Science of Computer Programming 16 (1991) 207-249
Elsevier

Symbolic execution: a semantic approach

Ralf Kneuper*
Wilhelm-Leuschner-Strasse 2, 6100 Darmstadt, Germany

Communicated by M. Sintzoff
Received January 1989
Revised December 1990

Abstract

Kneuper, R., Symbolic execution: a semantic approach, Science of Computer Programming 16 (1991) 207-249.

This paper discusses symbolic execution from a semantic point of view, covering both programs and specifications. It defines the denotational semantics of symbolic execution of specifications and programs, and thus introduces a notion of correctness of symbolic execution which applies not just to an individual language but to a wide class of languages, namely those whose semantics can be described in terms of states and state transformations. Also described are the operational semantics of a language as used for symbolic execution.

This work also provided the basis of the prototype symbolic execution system SymbEx which was developed at the University of Manchester as part of the *mural* project. However, this paper only covers the theoretical foundations used by SymbEx, but not the system itself.

1. Introduction

1.1. *Symbolic execution*

Symbolic execution is a technique that allows the user to handle a whole range of input values at a time, rather than just a single value as in actual execution. This is done by allowing identifiers (symbols) as input in addition to the usual "actual" values. It was first introduced by King (cf. [22]) who was mainly interested in its use for program validation and verification [14]. Others later used it for a variety of purposes, such as test case generation [6, 17] and specification validation (Gist project, cf. [2, 7]). With the exception of Gist and the work done by Kemmerer [21], symbolic execution has so far only been applied to *programs*, as opposed to specifications.

* Most of the work described here was done while the author was working on the Ipse 2.5 project at the University of Manchester, England.

Techniques related to symbolic execution are partial evaluation (e.g. [4]) and abstract interpretation (e.g. [10, 9, 25]). They differ from symbolic execution in that the result of partial evaluation of a program, given some input data, is a new program. In abstract interpretation, one analyses the relationship between sets (or domains) of input and output data, abstracting away from the relationships between the individual values within those sets. In this sense, abstract interpretation is more abstract than symbolic execution, since symbolic execution tries to capture this information as well.

Complexity analysis can also be done using techniques similar to symbolic execution, see for example [30]. For a more detailed survey of the different approaches to symbolic execution and related techniques see [24, Chapter 2].

The purpose of this paper is to define precisely what is meant by symbolic execution, independent of the language used, and provide a semantic framework for it. As a result, it is not concerned with using symbolic execution for any of the practical purposes described above, or with any particular system for symbolic execution.

The framework described was, however, introduced as one step towards the overall aim of developing a system for symbolic execution of specifications which can help to validate them, and thus support the first step in a formal software development process. How such a system could be used is described in [23].

Symbolic execution can be considered as a technique for "executing" programs or specifications when some of the information normally needed is not available. In this sense, symbolic execution allows one to handle partial information about:

- *Input data*: the input values are not determined (or at least not uniquely); this means one has to handle a whole range of input values at once, not just a single value.
- *Algorithm*: the algorithm for computing the output value for any given input value is not provided (or is at least incomplete). In this case one usually talks about a *specification* rather than a program.
- *Output data*: the output values are not determined uniquely by the input values, i.e. the specification is nondeterministic.

In order to describe this variation of execution, one first has to define what exactly is meant by "interpretation", "execution" and "executable". Unless otherwise stated, a program will in future be considered as a special kind of specification. The differences between the two are discussed in Section 2.2. Furthermore, I shall distinguish between *interpreting* a specification and *executing* it. Interpreting a specification transforms one state into another according to the meaning of the specification. Executing it additionally requires that one has an *algorithm* for performing this transformation. When interpreting or executing a specification, one interprets or executes *a term* in the appropriate language. Obviously, it does not make sense to talk about interpretation or execution if only the syntax of this language is known, e.g. if one only knows its grammar from some production rules.

Interpretation and execution of a language clearly depend on its semantics. Therefore, Section 2.1 will discuss the relevant aspects of language semantics that will be needed later. Section 2.2 briefly discusses the notions of execution and interpretation, and compares specification and programming languages.

The main body of this paper starts in Section 3, which investigates the denotational semantics of symbolic execution. These will be expressed in terms of the denotational semantics of the specification language used. This is followed in Section 4 with a discussion of the operational semantics of symbolic execution.

The notation used in this paper for describing functions, data types, etc., is based on VDM [20]. A short summary of it is given in Appendix A.

1.2. Scope and structure of this paper

This paper defines the denotational and operational semantics of symbolic execution in a formal way independent of the specific language under consideration. This distinguishes it from other papers on symbolic execution which only consider one particular language and only give an intuitive description of the notion of symbolic execution itself.

The work described in this paper provided the framework for a language-generic symbolic execution system called SymbEx, a prototype of which was developed as part of the *mural*/Ipse 2.5 project. However, this paper concentrates on the semantic aspects of symbolic execution and does not try to describe SymbEx. For more information on the system SymbEx itself see [18, 24].

The ideas in this paper are intended to apply to any specification or programming language that is based on the notions of states and state transitions or, more precisely, whose semantics can be expressed in terms of states and state transitions. Therefore, most of the ideas described are not appropriate for *algebraic* specification languages, for *functional* or *logic* programming languages, or for languages such as CCS or CSP which are based on the concept of processes. However, within these restrictions the ideas described are intended to be fully generic. Note that on a very low level this implies that the concepts apply to all programming languages, since programs are eventually translated into state transformations in a computer. However, this very low-level view will in general not be very useful.

The concept of symbolic execution in the sense described here and in the previous literature relies heavily on the concept of variables and their (changing) values, so that it is difficult to imagine what symbolic execution of a language not using states and state transitions should be. Languages such as Prolog or Lisp support a considerable amount of symbol processing in ordinary execution, so that an additional concept of symbolic execution might not be needed.

For a more detailed discussion of the range of languages covered see [24, Sect. 4.3].

Furthermore, this paper ignores the problems arising from rounding errors in floating point arithmetic and from over- and underflow on computers with bounded storage capacity. The latter could be dealt with by introducing parameters expressing these bounds into the semantics of the language, using the "clean termination" approach described in [8].

As for the structure of this paper, Section 2 provides some of the theoretical background that will be used later. In particular, it discusses the semantics of specification or programming languages with an emphasis on denotational semantics. Based on that, the notions of execution and executability and the differences between specification and programming languages are examined.

The main body of the paper starts with Section 3, which gives a semantic definition of symbolic execution. By expressing it in terms of the denotational semantics of the language used, symbolic execution is defined generically over languages.

This denotational description is followed in Section 4 by a description of symbolic execution from an operational semantics point of view, and a discussion of the relationship between the two. Section 4 includes a number of rules that can be used to describe symbolic execution of some common language constructs.

Finally, Section 5 provides a short summary of the ideas discussed in this paper and assesses the achievements and limitations of this approach.

The appendices contain, apart from a few proofs that were too long to be included in the text itself, a short summary of some of the VDM-notation used.

2. Specification and programming languages

2.1. Semantics

Consider a specification or programming language $\mathscr{L}_1$. Unless otherwise stated, I shall in future consider a program as a special kind of specification. The differences between the two are discussed below. Specifications are a certain class of $\mathscr{L}_1$-terms, usually containing free variables called input and output variables and state variables.

For simplicity, I shall from now on assume that specifications only use a single state variable, but no other input or output variables. Since the state variable might be of arbitrarily complex type, this is no real restriction of generality.

We now need to define the concept of "language":

Definition 2.1 (*Language*). A language is defined by
- its syntax, expressed as a (usually context-free) grammar;
- well-formedness conditions (often expressed as static semantics or as context-sensitive syntax);
- its semantics, which may be given in any of the styles described below.

The semantics of a programming or specification language describe the "meaning" of terms of the language in some way. There are a number of different ways of describing the semantics of a language, the most common ones are the following (cf. [28, Section 2] or [11]):

- *Operational semantics.* The meaning of a construct of the language is given by explicitly stating its effect, the operation that it evokes (see for example [26]).

Given an input state for a specification, the operational semantics of the language provide an algorithm to find the appropriate output state.

Another way of describing operational semantics views only input variables as free variables of a specification. In this case one substitutes the input data for the free variables of the specification term, and then rewrites the resulting ground term into normal form in a rewrite system which is given by the semantics of the programming language. This normal form is then the output from executing the specification. For example, the semantics of λ-calculus can be given this way, using β-reduction, etc. [11, Chapter 5].

- *Denotational semantics.* The meaning of a construct of the language is described by giving it a "denotation", i.e. by translating it into a different structure which is considered to be understood (usually but not necessarily a formalized structure) and modelling the effect of statements of the language there. This different structure is often based on domain theory as introduced by Scott [27, 28]. One possible alternative is to express the denotations in the language of predicate calculus, this is called *predicative* semantics [15].
- *Axiomatic semantics.* The meaning of a language is described by axioms that can be used to prove theorems about (specification) terms in the language. These axioms act as constraints on the relation between input and output variables. The usual style for such axioms is Hoare logic [1, 16], using input and output assertions. A similar approach is the use of predicate transformers and weakest preconditions, as introduced by Dijkstra [13].

The following is mainly based on denotational semantics. By giving a denotational semantics to a language $\mathscr{L}_1$, one translates it into another language $\mathscr{L}_2$, called semantic language, which is considered to be "understood", i.e. the meaning of its constructs is known. In other words, one explains the semantics of $\mathscr{L}_1$ in terms of the semantics of $\mathscr{L}_2$. The translation is given by a recursive function from terms in the language $\mathscr{L}_1$ to terms in $\mathscr{L}_2$. This translation is called *valuation* function. Usually, $\mathscr{L}_2$ will have some theory associated with it, in that case it will be more adequate to say that we understand the *theory* associated with $\mathscr{L}_2$, rather than the language itself. Common choices for $\mathscr{L}_2$ with an associated theory are the languages of Scott's domain theory, of predicate calculus, of partial recursive functions, or of λ-calculus.

Definition 2.2 (*Valuation functions*). A valuation function $\mathscr{M}$ maps terms of a language $\mathscr{L}$ to their meaning (denotation), an element or set of elements of the abstract or semantic domain. We require that valuation functions are defined structurally, i.e. the meaning of a term is defined in terms of the meaning of its subterms. This property is sometimes called the "denotational rule".

The valuation function may map to a *set* of elements of the abstract domain in order to handle nondeterminism and underdeterminedness. Alternatively, power domains may be used instead.

A valuation function may also take additional arguments such as the environment or continuations, in order to handle more complicated language constructs. This will in the following be handled by "currying" the valuation function and turning the denotation of the construct itself into a function. In particular, the denotation of a program term is usually defined as a function from states to states. Variations are used for nondeterministic programs, whose meaning may be given as a binary relation between states or, equivalently, a function from states to sets of states, and for underdetermined programs, whose meaning may be given as a set of functions from states to states.

Usually, one introduces several different valuation functions for different classes of terms, such as commands, Boolean expressions, etc. The valuation function for terms in class C will be written as $\mathcal{M}_C$. Figure 1 describes the valuation functions on predicates and specifications. Specifications are terms in the language that denote a binary relation on states. The definitions in Fig. 1 only give those properties of $\mathcal{M}_{Pred}$ and $\mathcal{M}_{Spec}$ that will be needed later. Obviously, for any given language one will want to define these functions in much more detail, and *Pred* and *Spec* should probably allow expression of nonrecursive functions as well. The conditions on these two valuation functions ensure that the languages of predicates and specifications are "reasonably expressive", at least they allow one to express all (partial) recursive functions of the appropriate type, for example by expressing a suitable recursion scheme for defining the function.

Given a set *Name* of identifiers (names) where each identifier has a type associated with it, and a set *Val* of values, a state is a map of type

$$\Sigma = Name \xrightarrow{m} Val_{\perp}$$

Define

$$\Sigma_{\perp} = \Sigma \cup \{\perp\}$$

The valuation function on predicates (over states) is some function

$$\mathcal{M}_{Pred}: Pred \to \Sigma_{\perp} \to \mathbb{B}$$

such that

$$\forall f: \{\text{partial recursive functions } \Sigma_{\perp} \to \mathbb{B}\} \cdot \exists [\![\varphi]\!]: Pred \cdot \mathcal{M}_{Pred}[\![\varphi]\!] = f$$

The valuation function on specifications *Spec* is some function

$$\mathcal{M}_{Spec}: [\![spec]\!] \mapsto R;\ Spec \to (\Sigma_{\perp} \times \Sigma_{\perp} \to \mathbb{B})$$

where R is such that

$$\forall \sigma: \Sigma_{\perp} \cdot [R(\perp, \sigma) \ \Rightarrow \ \sigma = \perp] \wedge \exists \sigma': \Sigma_{\perp} \cdot R(\sigma, \sigma')$$

and where

$$\forall f: \{\text{partial recursive functions } \Sigma_{\perp} \times \Sigma_{\perp} \to \mathbb{B}\} \cdot \exists [\![spec]\!]: Spec \cdot \mathcal{M}_{Spec}[\![spec]\!] = f$$

Fig. 1. Valuation functions for specifications and predicates.

The valuation function $\mathcal{M}_{Spec}$ for specifications maps a specification to a binary relation on states that is interpreted as the input/output relation induced by the specification. Since the investigations in this paper are restricted to specification languages that are based on the notions of state and state transitions (cf. Section 1.2), this seems the most appropriate approach.

Note that the specification of Σ with function $\mathcal{M}_{Spec}$ is ***sufficiently abstract*** in the sense of [20, p. 218]. A model is sufficiently abstract if, for any two different states, one can find a sequence of operations, in this case, of elements $\mathcal{M}_{Spec}[\![spec]\!]$,[1] that distinguishes them. This is very similar to full abstraction of the denotational semantics of a language in relation to its operational semantics.

According to Fig. 1, the meaning of a specification $[\![spec]\!]$ is a relation R between input and output states.

Nontermination or abortion is described by the output state $\perp$, i.e. $R(\sigma, \perp)$ describes the fact that, starting from state σ, execution of $[\![spec]\!]$ may not terminate, or it may abort. We require that $R(\perp, \perp)$, and $R(\perp, \sigma)$ only if $\sigma = \perp$. This is done in order to ease the description of composition. It essentially describes the fact that if a specification never starts to be interpreted because the one interpreted previously fails to terminate, then interpretation of this specification also will not terminate. Equivalently this could of course be expressed as

$$R(\perp, \sigma) \Leftrightarrow \sigma = \perp.$$

It is also required that for every state σ_1 there exists at least one state σ_2 (possibly $\perp$) such that $R(\sigma_1, \sigma_2)$.

For deterministic specifications, R will actually be a *function* from states to states rather than a relation.

The reason for using $\perp$ as a state in its own right, rather than for example introducing a termination set T of states (as done in VDM), is that this will make it easier to describe symbolic execution and to distinguish between nontermination of symbolic execution itself and termination with the result "execution does not terminate". Additionally, composition is easier to describe this way. However, provided two relations R are considered equivalent if they agree on all pairs of states whose first element is in T (i.e. whenever they are guaranteed to terminate), then this is only a matter of taste and the two models are isomorphic: given R as above, (R_1, T_1) can be defined as

$$R_1(\sigma_1, \sigma_2) = R(\sigma_1, \sigma_2) \wedge \sigma_2 \neq \perp \quad \text{and} \quad T_1 = \{\sigma \mid \neg R(\sigma, \perp)\}.$$

Conversely, given (R_1, T_1) we can define

$$R(\sigma_1, \sigma_2) = R_1(\sigma_1, \sigma_2) \vee (\sigma_1 \notin T_1 \wedge \sigma_2 = \perp).$$

[1] Terms in the object (or specification or programming) language are written in Strachey brackets $[\![\ldots]\!]$, in order to distinguish them from terms in the (meta-) language used for describing the semantics of the term.

The definition of $\mathcal{M}_{Spec}$ in Figure 1 can easily be extended to cover *sequences* of specification:

$$\mathcal{M}_{Spec}[\![[spec]]\!] \triangleq \mathcal{M}_{Spec}[\![spec]\!],$$

$$\mathcal{M}_{Spec}[\![[spec_1, \ldots, spec_{n-1}, spec_n]]\!] \triangleq$$

$$\mathcal{M}_{Spec}[\![[spec_1, \ldots, spec_{n-1}]]\!] \circ \mathcal{M}_{Spec}[\![spec_n]\!].$$

using overloading of $\mathcal{M}_{Spec}$.

2.2. Execution and executability

We first discuss the difference between execution and interpretation of a program.

Definition 2.3 (*Execution and interpretation*). Given a specification $[\![spec]\!]$, interpretation of $[\![spec]\!]$ is a state transformation from a state σ to a state σ' such that $\mathcal{M}_{Spec}[\![spec]\!](\sigma, \sigma')$. *interpret* is defined as an arbitrary function

$$interpret: Spec \to \Sigma_\perp \to \Sigma_\perp$$

that satisfies

$$\mathcal{M}_{Spec}[\![spec]\!](\sigma, interpret[\![spec]\!]\sigma).$$

If this state transformation is given by a (partial) recursive algorithm, then interpretation $[\![spec]\!]$ is called *execution.*

Note that in general the result $interpret[\![spec]\!]\sigma$ is not defined uniquely by this definition, since $[\![spec]\!]$ may be underdetermined or nondeterministic. If this is the case, one has to force the interpretation of $[\![spec]\!]$ into choosing a particular result state out of the set of possible result states.

Definition 2.4 (*Executable languages*). A language $\mathcal{L}$ is executable if every specification term in $\mathcal{L}$ can be executed in the sense of Definition 2.3, i.e. if all specification terms in the language are recursive.

A special case of this is a language $\mathcal{L}$ with operational semantics. In this case, the recursive algorithm is given explicitly by the operational semantics, and the language is therefore executable. It follows that a language is recursive if and only if its operational semantics can be given. Executability of a language or term does not (or at least not only) depend on its denotational semantics: two functions may denote the same input/output relationship, but only one of them is executable, since the other one is defined using properties of the result, without providing an algorithm for it. E.g. sorting of a list could be defined as a procedure that takes a list of elements and returns a list containing the same elements but where each element is smaller than or equal to any following element. In this case, sorting would be non-executable. Alternatively, sorting could be defined using a suitable algorithm, such as bubble sort, in which case sorting would be executable.

Similarly, termination of a term clearly depends on its operational semantics, since it refers to the length of the process of computation. It therefore does not make sense to talk about termination of non-executable terms or languages.

A difficulty that arises in this context is that even though recursive functions do model the hardware operations within a computer to a certain extent, they do not take into account time and space restrictions that apply to any computer in the real world. Almost all language constructs are in some sense non-executable on real computers, since for sufficiently large arguments, the capacity of any computer will be exceeded. However, Definition 2.3 ignores such restrictions and says that a specification is executable if it is executable given a large enough computer and unlimited (but finite) time.

To some extent, executability can be used to differentiate between specification and programming languages: programming languages are always executable, while specification languages in general are not. This implies that specification languages can be more expressive. In particular one can use more abstract concepts for specifications, such as more abstract types, or describing *what* the result of an operation should be, as opposed to describing *how* it should be computed. From this point of view, "executable specification languages" are programming languages that usually allow more abstract constructs than the more common programming languages, but nevertheless the fact that they are executable implies that they can never be as abstract as a genuine specification language, and it therefore is at least questionable whether they are suitable for actually specifying a system.

When one specifies a system before implementing it, the question of course arises whether the implementation is correct, whether it "satisfies" the specification. There are a number of different definitions of satisfaction, which differ mainly in their treatment of undefinedness. These are discussed in more detail in [5] or [24, Section 3.3].

In the following a specification $[\![Spec_2]\!]$ will be called an implementation of $[\![Spec_1]\!]$ if it satisfies $[\![Spec_1]\!]$ and is executable. $[\![Spec_1]\!]$ is implementable if there exists an implementation of it, i.e. if it can be defined as a recursive function.

3. Denotational semantics of symbolic execution

3.1. *The semantic model*

As a first attempt at a formal description of symbolic execution, one might try to base it on the observation that in symbolic execution, the input to a specification $[\![spec]\!]$ can be considered as a set $S \subseteq \Sigma_\perp$ of states. As output, it returns the set of states that can be reached from a state in S via $\mathcal{M}_{Spec}[\![spec]\!]$. However, for describing the denotational semantics of symbolic execution this is not sufficient, since it would lose all the information about the relationship between input and output states themselves, as opposed to the relationship between the *sets* of these states.(For other purposes it can still be very useful to consider only these sets; this is essentially what is done in *abstract interpretation.*) For example, given the specification

$$x = 0 \vee x = \bar{x} + 1$$

(where $\bar{x}$ denotes the "old" value of x before running the operation specified), symbolic execution would map $\mathbb{N}$ to $\mathbb{N}$ and not really provide any information. To get more useful information, one would have to restrict the set S, in this case $\{\sigma \mid \sigma(x) \in \mathbb{N}\}$, to a small subset, which would be contrary to the ideas behind symbolic execution and lead towards "testing" of specifications.

In addition to the requirement that the semantic model should support the use of (fairly arbitrary) sets of input states, we therefore need that the semantic model describes the relationship between individual input and output states (and not just the relationship between the set of *all* input states and the set of *all* output states). Other requirements on the model are:

- It should allow composition of two (or more) symbolic execution steps. In particular, this implies that input and output must be of the same type.
- It should be possible to make assumptions on the set of input states (as described above) not only at the beginning of a sequence of symbolic execution steps, but also at an intermediate stage. In this case, assumptions may be expressed in terms of the values of variables in earlier states. (Cf. the **assume** command in SymbEx [24, pp. 40f, 85, 110].)

As a result, the model of symbolic execution used is based on a "symbolic execution state" called *SEStateDen* which contains sets of *sequences* of states. The definition of *SEStateDen* is given in Fig. 2. The name *SEStateDen* stands for *S*ymbolic *E*xecution *State* as used for *Den*otational semantics. Similarly, Section 4 will introduce *SEStateOp* for states in operational semantics.

In addition to the set of sequences of states, *SEStateDen* contains a field *LEN* which stores the number of symbolic execution steps performed, plus 1 for the

A state as used in symbolic execution is given by

$$\begin{aligned} \mathit{SEStateDen} :: \mathit{SEQS} &: \mathcal{P}((\Sigma_\perp)^*) \\ \mathit{LEN} &: \mathbb{N} \end{aligned}$$

where

$$\mathit{inv\text{-}SEStateDen}(\mathit{mk\text{-}SEStateDen}(\mathit{set}, l)) \triangleq$$
$$\forall \sigma\text{-}\mathit{seq} \in \mathit{set} \cdot \operatorname{len} \sigma\text{-}\mathit{seq} \le l$$
$$\wedge \forall \sigma\text{-}\mathit{seq}_1, \sigma\text{-}\mathit{seq}_2 \in \mathit{set} \cdot \forall \sigma\text{-}\mathit{seq}: (\Sigma_\perp)^* \cdot \sigma\text{-}\mathit{seq}_1 = \sigma\text{-}\mathit{seq}_2 \curvearrowright \sigma\text{-}\mathit{seq} \Rightarrow \sigma\text{-}\mathit{seq} = [\,]$$

A set $S \subseteq \Sigma_\perp$ of states (or, similarly, a predicate on states) can be represented by the *SEStateDen*

$$\tau(S) \triangleq \mathit{mk\text{-}SEStateDen}(\{[\sigma] \mid \sigma \in S\}, 1)$$

The function *yield* extracts the input/output relationship from the sequences in *SEStateDen*.

$$\mathit{yield}(\tau) \triangleq \lambda\sigma: \Sigma_\perp \cdot \{\sigma': \Sigma_\perp \mid \exists \sigma\text{-}\mathit{seq} \in \mathit{SEQS}(\tau) \cdot$$
$$\operatorname{hd} \sigma\text{-}\mathit{seq} = \sigma \wedge \operatorname{last} \sigma\text{-}\mathit{seq} = \sigma' \wedge \operatorname{len} \sigma\text{-}\mathit{seq} = \mathit{LEN}(\tau)\}$$

Fig. 2. Denotational semantics of symbolic execution: state.

initial state (see Fig. 2). At the same time, this is the number of *actual* execution steps modelled in any sequence of states in the field *SEQS* plus 1, which leads to the first conjunct in the invariant. In this model, assumed restrictions are modelled by "cutting off" as much as necessary from the end of all sequences of states until the condition is satisfied. This intuition explains the second conjunct of the invariant on *SEStateDen*, which demands that no sequence in *SEStateDen* is an initial segment of another such sequence. The *LEN* field is then needed to recognize if all sequences in *SEQS* have been cut off.

As a convention, τ will be used to denote elements of *SEStateDen*, while σ denotes elements of $\Sigma_\perp$, as before.

Symbolic execution of a specification is modelled by adding another state to all those sequences that have not been "cut off", see Fig. 3. Just as interpretation or execution, given a specification, maps states to states, so symbolic execution, given a specification, maps *SEStateDen*s to *SEStateDen*s.

The functions front and last used in Figs. 2 and 3 are defined as

$$\mathsf{front} = \mathsf{rev} \circ \mathsf{tl} \circ \mathsf{rev},$$

$$\mathsf{last} = \mathsf{hd} \circ \mathsf{rev},$$

where the function rev reverses a list.

Doing symbolic execution in the way described here and storing *all* possible sequences of states allowed by a sequence of specifications requires a fairly rich language for expressing the results of symbolic execution, which might not always be available. For example, the result of executing a **while**-loop will often not be expressible in the language available. Therefore, in addition to such *full* symbolic execution Fig. 3 also defines *weak* symbolic execution, where the result *includes* the

(Full) symbolic execution is given by the function

$$symbolic\text{-}ex : Spec \to SEStateDen \to SEStateDen$$

$$\begin{aligned}
&symbolic\text{-}ex[\![spec]\!]\tau \;\triangleq\\
&\quad mk\text{-}SEStateDen(\\
&\qquad \{\sigma\text{-}seq \mid \mathsf{len}\, \sigma\text{-}seq = LEN(\tau) + 1 \wedge \mathsf{front}\, \sigma\text{-}seq \in SEQS(\tau)\\
&\qquad\qquad \wedge\, \mathcal{M}_{Spec}[\![spec]\!](\mathsf{last\,front}\, \sigma\text{-}seq, \mathsf{last}\, \sigma\text{-}seq)\\
&\qquad\quad \vee\, \mathsf{len}\, \sigma\text{-}seq < LEN(\tau) \wedge \sigma\text{-}seq \in SEQS(\tau)\},\\
&\qquad LEN(\tau) + 1)
\end{aligned}$$

Weak symbolic execution is any function

$$w\text{-}symbolic\text{-}ex : Spec \times SEStateDen \to SEStateDen;\ ([\![spec]\!], \tau_1) \mapsto \tau_2$$

which satisfies

$$SEQS(\tau_2) \supseteq SEQS(symbolic\text{-}ex[\![spec]\!]\tau_1) \wedge LEN(\tau_2) = LEN(symbolic\text{-}ex[\![spec]\!]\tau_1)$$

Fig. 3. Denotational semantics of symbolic execution: functions.

set of all possible sequences of states. This ensures that the properties one gets as a result of weak symbolic execution still hold for the denotation of the full result, they just do not in general give a complete description.

Since in many cases one is really interested in the relationship between input and output states and less in the intermediate states, a function called *yield* for extracting this relationship from an *SEStateDen* is also provided (in Fig. 2). This can be considered as extracting from an *SEStateDen* the map from initial states to possible resulting states, the possible data model for symbolic execution rejected above. It thus is quite similar to the *yield* operator $^+$ introduced in [12].

Note that there is a distinction between symbolic execution of the composition of specifications and the composition of symbolic execution steps. As Lemma 3.4 will show, they give rise to *SEStateDen*s that describe the same relationship between initial and final states, but the *SEStateDen*s themselves are different. They lead to *SEStateDen*s of different lengths, since symbolic execution of the composition of specifications is considered as a single step, while a sequence of symbolic executions in general consists of several steps.

3.2. Some properties of symbolic execution

It is not immediately obvious that *symbolic-ex* as defined is a *total* function. Although a result is constructed for any input value, this result might not be of type *SEStateDen*. The following lemma shows that this case does not arise.

Lemma 3.1. *symbolic-ex is total.*

Proof. We have to show that for any $[\![spec]\!]$: *Spec* and τ: *SEStateDen*

$$\textit{inv-SEStateDen}(\textit{symbolic-ex}[\![spec]\!]\tau).$$

The first condition of the invariant is obviously true. Now let

$$\sigma\textit{-seq}_1, \sigma\textit{-seq}_2 \in \textit{SEQS}(\textit{symbolic-ex}[\![spec]\!]\tau)$$

and $\sigma\textit{-seq}$: $(\Sigma_\perp)^*$, $\sigma\textit{-seq} \neq [\,]$ be such that

$$\sigma\textit{-seq}_1 = \sigma\textit{-seq}_2 \frown \sigma\textit{-seq}.$$

The definition of *symbolic-ex* then implies that $\sigma\textit{-seq}_2 \in \textit{SEQS}(\tau)$.

Case 1: $\mathsf{len}\ \sigma\textit{-seq}_1 = \textit{LEN}(\tau)+1$. Then

$$\sigma\textit{-seq}_2 \frown \mathsf{front}\ \sigma\textit{-seq} = \mathsf{front}\ \sigma\textit{-seq}_1 \in \textit{SEQS}(\tau)$$

and $\textit{inv-SEStateDen}(\tau)$ implies that $\mathsf{front}\ \sigma\textit{-seq} = [\,]$, i.e. $\mathsf{len}\ \sigma\textit{-seq} = 1$. But then $\mathsf{len}\ \sigma\textit{-seq}_2 = \textit{LEN}(\tau)$, therefore $\sigma\textit{-seq}_2$ cannot be in $\textit{SEQS}(\textit{symbolic-ex}[\![spec]\!]\tau)$—contradiction.

Case 2: $\mathsf{len}\ \sigma\textit{-seq}_1 < \textit{LEN}(\tau)$. In this case $\sigma\textit{-seq} = [\,]$ follows immediately from $\textit{inv-SEStateDen}(\tau)$. $\square$

Example 3.2. Let $Name = \{x, y\}$. We want to symbolically execute the VDM operation

$$\begin{array}{l} OP_1 \\ \textsf{ext wr } x\colon \mathbb{Z} \\ \quad \textsf{wr } y\colon \mathbb{N} \\ \textsf{pre } x \geq 0 \\ \textsf{post } y^2 \leq \overleftarrow{x} \wedge x = \overleftarrow{x} + 1 \end{array}$$

Then

$$\mathcal{M}_{Spec}[\![OP_1]\!](\sigma, \sigma_1) \Leftrightarrow$$
$$\textsf{if } \sigma(x) \geq o \textsf{ then } \sigma_1(y)^2 \leq \sigma(x) \wedge \sigma_1(x) = \sigma(x) + 1 \textsf{ else true}$$

Now the user **assume** s that the precondition of OP_1 is true. This means that OP_1 is to be symbolically executed in the *SEStateDen* τ_1 which represents the predicate $x \geq 0$:

$$\begin{aligned} \tau_1 &= \textit{mk-SEStateDen}(\{[\sigma] \mid \mathcal{M}_{Pred}[\![x \geq 0]\!]\sigma\}, 1) \\ &= \textit{mk-SEStateDen}(\{[\sigma] \mid \sigma(x) \geq 0\}, 1). \end{aligned}$$

Then symbolic execution of the specification OP_1 starting in the *SEStateDen* τ_1 results in the *SEStateDen*

$$\begin{aligned} &\textit{symbolic-ex}[\![OP_1]\!]\tau_1 \\ &= \textit{mk-SEStateDen}(\{\sigma\textit{-seq} \mid \textsf{len } \sigma\textit{-seq} = LEN(\tau_1) + 1 \\ &\qquad\qquad \wedge \textsf{front } \sigma\textit{-seq} \in SEQS(\tau_1) \\ &\qquad\qquad \wedge \mathcal{M}_{Spec}[\![OP_1]\!](\textsf{last front } \sigma\textit{-seq}, \textsf{last } \sigma\textit{-seq}) \\ &\qquad\quad \vee \textsf{len } \sigma\textit{-seq} < LEN(\tau_1) \\ &\qquad\qquad \wedge \sigma\textit{-seq} \in SEQS(\tau_1)\}, \\ &\qquad LEN(\tau_1) + 1) \\ &= \textit{mk-SEStateDen}(\{\sigma\textit{-seq} \mid \textsf{len } \sigma\textit{-seq} = 2 \\ &\qquad\qquad \wedge \sigma\textit{-seq}[1](x) \geq 0 \\ &\qquad\qquad \wedge \mathcal{M}_{Spec}[\![OP_1]\!](\sigma\textit{-seq}[1], \sigma\textit{-seq}[2])\}, 2) \\ &= \textit{mk-SEStateDen}(\{[\sigma_1, \sigma_2] \mid \sigma_1(x) \geq 0 \\ &\qquad\qquad \wedge \sigma_2(y)^2 \leq \sigma_1(x) \\ &\qquad\qquad \wedge \sigma_2(x) = \sigma_1(x) + 1\}, 2). \end{aligned}$$

Strictly speaking, Op_1 is the *name* of the operation (or specification) rather than the operation itself. For the time being, I shall use names of specifications to denote both the name itself and the specification referred to by it, until in Section 4.3 a mapping from specification names to specifications is introduced.

The following lemma states that the result of interpreting a specification in a state σ can also be achieved by symbolically executing the specification in a *SEStateDen* τ which represents a set of states including σ, and then selecting a sequence starting with σ in the result. This is a property that one would "obviously" want to hold, and it thus serves to validate the model.

Lemma 3.3. *Let* $[\![spec]\!]$: *Spec, let* τ_1, τ_2: *SEStateDen be such that*

$$\textit{symbolic-ex}[\![spec]\!]\tau_1 = \tau_2.$$

Then for all states $\sigma, \sigma_1: \Sigma_\perp$

$$\sigma_1 \in \textit{yield}(\tau_1)(\sigma) \Rightarrow \textit{interpret}[\![spec]\!]\sigma_1 \in \textit{yield}(\tau_2)(\sigma).$$

In particular if τ_1 *represents a set S of states, i.e.*

$$\tau_1 = \textit{mk-SEStateDen}(\{[\sigma] \mid \sigma \in S\}, 1),$$

then for all $\sigma \in S$

$$[\sigma, \textit{interpret}[\![spec]\!]\sigma] \in \textit{SEQS}(\tau_2).$$

3.3. Assignment statement

The denotational semantics of the assignment statement can be described as follows:

$$\mathcal{M}_{Spec}[\![x := e]\!](\sigma_1, \sigma_2) \Leftrightarrow$$
$$\sigma_2(x) = \mathcal{M}[\![e]\!] \wedge \forall y \in \mathrm{dom}\, \sigma_1 \cdot y \neq x \Rightarrow \sigma_2(y) = \sigma_1(y).$$

The denotational semantics of symbolic execution of the assignment statement can now be calculated as

$$\begin{aligned}
&\textit{symbolic-ex}[\![x := e]\!]\tau \\
&\quad = \textit{mk-SEStateDen}(\{\sigma\textit{-seq} \mid \mathrm{len}\ \sigma\textit{-seq} = \textit{LEN}(\tau) + 1 \\
&\qquad\qquad \wedge \mathrm{front}\ \sigma\textit{-seq} \in \textit{SEQS}(\tau) \\
&\qquad\qquad \wedge \mathrm{last}\ \sigma\textit{-seq}(x) = \mathcal{M}[\![e]\!] \\
&\qquad\qquad \wedge \forall y \cdot y \neq x \Rightarrow \mathrm{last}\ \sigma\textit{-seq}(y) = \mathrm{last}\ \mathrm{front}\ \sigma\textit{-seq}(y) \\
&\qquad \vee \mathrm{len}\ \sigma\textit{-seq} < \textit{LEN}(\tau) \wedge \sigma\textit{-seq} \in \textit{SEQS}(\tau)\}, \\
&\qquad 2).
\end{aligned}$$

As a simple example, consider the case where we start with the *SEStateDen* that represents $\Sigma_\perp$, that is, no restriction has been placed on the starting state and no other statement has been symbolically executed:

$$\tau = \textit{mk-SEStateDen}(\{[\sigma] \mid \sigma \in \Sigma_\perp\}, 1).$$

We then get

$$\begin{aligned}&\textit{symbolic-ex}[\![x := e]\!]\tau\\&\quad = \textit{mk-SEStateDen}(\{\sigma\text{-}seq \mid \mathsf{len}\ \sigma\text{-}seq = 2 \wedge \sigma\text{-}seq[1] \in \textit{SEQS}(\tau)\\&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \sigma\text{-}seq[2](x) = \mathcal{M}[\![e]\!]\\&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \forall y \cdot y \neq x\ \Rightarrow\ \sigma\text{-}seq[2](y) = \sigma\text{-}seq[1](y)\}, 2)\\&\quad = \textit{mk-SEStateDen}(\{[\sigma_1, \sigma_2] \mid \sigma_1 \in \Sigma_\perp \wedge \sigma_2(x) = \mathcal{M}[\![e]\!]\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \forall y \cdot y \neq x\ \Rightarrow\ \sigma_2(y) = \sigma_1(y)\}, 2)\end{aligned}$$

which expresses, as one would expect, that all sequences of states of length 2 are included for which the value in the second state of x is (the denotation of) e, and the values of all other variables are the same as in the first state.

3.4. Composition of specifications

Let ; denote sequential composition of specifications, and let *symbolic-ex-s* be the obvious generalisation of *symbolic-ex* that symbolically executes a sequence of specifications instead of just a single one. Then:

Lemma 3.4 (Composition). *For all specifications* $[\![spec_1]\!], [\![spec_2]\!]$: *Spec*,

$$\textit{yield}(\textit{symbolic-ex}[\![spec_1 ; spec_2]\!]) = \textit{yield}(\textit{symbolic-ex-s}[\![[spec_1, spec_2]]\!]).$$

Proof. See Appendix B. □

Note that we do *not* have

$$\textit{symbolic-ex}[\![spec_1 ; spec_2]\!] = \textit{symbolic-ex-s}[\![[spec_1, spec_2]]\!]$$

since $[\![spec_1 ; spec_2]\!]$ is regarded as a single specification, while $[\![[spec_1, spec_2]]\!]$ is a sequence of two specifications. Therefore symbolic execution of the two leads to *SEStateDen*s of different lengths.

Example 3.5. Given the operation specification

```
$OP_2$
ext wr $x$: $\mathbb{Z}$
    rd $y$: $\mathbb{N}$
pre $-100 \leq x \leq +100$
post $\exists z: \mathbb{Z} \cdot y * z + x = \overleftarrow{x} \wedge 0 \leq x < y$
```

we want to symbolically execute OP_2 starting in the *SEStateDen* τ_2 resulting from symbolically executing OP_1, as given in Example 3.2. From the specification it follows that

$$\begin{aligned}
&\mathcal{M}_{Spec}[\![OP_2]\!](\sigma, \sigma_1)\\
&\quad\Leftrightarrow \text{if } -100 \leqslant \sigma(x) \leqslant +100\\
&\qquad \text{then } \exists z\colon \mathbb{Z} \cdot \sigma_1(y) * z + \sigma_1(x) = \sigma(x)\\
&\qquad\qquad \wedge\, 0 \leqslant \sigma_1(x) < \sigma_1(y) \wedge \sigma_1(y) = \sigma(y)\\
&\qquad \text{else true.}
\end{aligned}$$

Then symbolic execution of OP_2 starting in τ_2 results in the τ_3: *SEStateDen* with $LEN(\tau_3) = 3$ and

$$\begin{aligned}
&SEQS(symbolic\text{-}ex[\![OP_2]\!]\tau_2)\\
&\quad = \{\sigma\text{-}seq \mid \text{len } \sigma\text{-}seq = LEN(\tau_2) + 1 \wedge \text{front } \sigma\text{-}seq \in SEQS(\tau_2)\\
&\qquad\qquad \wedge\, \mathcal{M}_{Spec}[\![OP_2]\!](\text{last front } \sigma\text{-}seq, \text{ last } \sigma\text{-}seq)\\
&\qquad\quad \vee \text{len } \sigma\text{-}seq < LEN(\tau_2) \wedge \sigma\text{-}seq \in SEQS(\tau_2)\}\\
&\quad = \{\sigma\text{-}seq \mid \text{len } \sigma\text{-}seq = 3\\
&\qquad\qquad \wedge\, \sigma\text{-}seq[1](x) \geqslant 0 \wedge \sigma\text{-}seq[2](y)^2 \leqslant \sigma\text{-}seq[1](x)\\
&\qquad\qquad \wedge\, \sigma\text{-}seq[2](x) = \sigma\text{-}seq[1](x) + 1\\
&\qquad\qquad \wedge\, \mathcal{M}_{Spec}[\![OP_2]\!](\sigma\text{-}seq[2], \sigma\text{-}seq[3])\}\\
&\quad = \{[\sigma_1, \sigma_2, \sigma_3] \mid \sigma_1(x) \geqslant 0 \wedge \sigma_2(y)^2 \leqslant \sigma_1(x) \wedge \sigma_2(x) = \sigma_1(x) + 1\\
&\qquad\qquad\qquad \wedge \text{ if } -100 \leqslant \sigma_2(x) \leqslant +100\\
&\qquad\qquad\qquad\quad \text{then } \exists z\colon \mathbb{Z} \cdot \sigma_3(y) * z + \sigma_3(x) = \sigma_2(x)\\
&\qquad\qquad\qquad\qquad \wedge\, 0 \leqslant \sigma_3(x) < \sigma_3(y) \wedge \sigma_3(y) = \sigma_2(y)\\
&\qquad\qquad\qquad\quad \text{else true}\}.
\end{aligned}$$

Note that the restriction on the set of starting states for the resulting set of state sequences (i.e. $\sigma_1(x) \geqslant 0$) was explicitly introduced by the user, before symbolically executing OP_1. This is the reason why, in spite of the second precondition, $-100 \leqslant x \leqslant +100$, the result still considers all σ-*seq* such that $\sigma_1(x) \geqslant 0$. Instead, the result itself contains a conditional. It is only for practical reasons that the user will often assume that the precondition is true, so as to keep the resulting expression reasonably simple.

3.5. Nondeterminism and underdeterminedness

In symbolic execution, the effects of underdeterminedness and nondeterminism are captured by the *state* rather than by making symbolic execution itself nondeterministic—the reason being that one wants to check that *all* outputs allowed by the specification or program are correct, and not just one of them.

As an example, consider the command (from Dijkstra's language of guarded commands [13])

$$IF \triangleq \textbf{if } b_1 \to spec_1 \, [\!] \, b_2 \to spec_2 \textbf{ fi}$$

The meaning of IF is given by

$$\begin{aligned}&\mathcal{M}_{Spec}[\![IF]\!](\sigma_1, \sigma_2)\\ &\quad\Leftrightarrow \mathcal{M}_{Pred}[\![b_1]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_1]\!](\sigma_1, \sigma_2)\\ &\qquad \vee \mathcal{M}_{Pred}[\![b_2]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_2]\!](\sigma_1, \sigma_2).\end{aligned}$$

Since we are interested in the nondeterministic case, we let τ_1: *SEStateDen* represent

$$\{\sigma\colon \Sigma_\perp \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\},$$

i.e.

$$\tau_1 = mk\text{-}SEStateDen(\{[\sigma] \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma\}, 1).$$

Then $LEN(symbolic\text{-}ex[\![IF]\!]\tau_1) = 2$ and

$$\begin{aligned}&SEQS(symbolic\text{-}ex[\![IF]\!]\tau_1)\\ &\quad = \{\sigma\text{-}seq \mid \mathsf{len}\ \sigma\text{-}seq = 2 \wedge \mathsf{front}\ \sigma\text{-}seq \in SEQS(\tau_1)\\ &\qquad\qquad \wedge \mathcal{M}_{Spec}[\![IF]\!](\sigma\text{-}seq[1], \sigma\text{-}seq[2])\}\\ &\quad = \{[\sigma_1, \sigma_2] \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma_1 \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma_1\\ &\qquad\qquad \wedge (\mathcal{M}_{Pred}[\![b_1]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_1]\!](\sigma_1, \sigma_2)\\ &\qquad\qquad\quad \vee \mathcal{M}_{Pred}[\![b_2]\!]\sigma_1 \wedge \mathcal{M}_{Spec}[\![spec_2]\!](\sigma_1, \sigma_2))\}\\ &\quad = \{[\sigma_1, \sigma_2] \mid \mathcal{M}_{Pred}[\![b_1]\!]\sigma_1 \wedge \mathcal{M}_{Pred}[\![b_2]\!]\sigma_1\\ &\qquad\qquad \wedge (\mathcal{M}_{Spec}[\![spec_1]\!](\sigma_1, \sigma_2) \vee \mathcal{M}_{Spec}[\![spec_2]\!](\sigma_1, \sigma_2))\}\end{aligned}$$

and the nondeterminism has been transferred inside the *SEStateDen* τ_2.

4. Operational semantics of specifications as used for symbolic execution

This section describes a model of symbolic execution based on the operational semantics approach. The style of operational semantics used is based on that of Plotkin's "Structured Operational Semantics" [26], but of course the transitions themselves are rather different since they describe *symbolic* rather than actual execution. However, if there is no danger of confusion, I shall in future not explicitly mention that I am dealing with the particular version of operational semantics used for symbolic execution, but just talk about operational semantics.

In Section 4.1, the data structure (or state) used is defined. Sections 4.2-4.5 introduce some general ideas about symbolic execution. After that, the state transitions used for symbolic execution of some specific language constructs are introduced. These include block structures, variable declarations, operation definitions in terms of pre- and postconditions, deterministic and nondeterministic conditionals, and loops. Note that this paper does not try to provide the complete operational semantics for any one language, but it does show the rules for a number of important language constructs.

There is an important difference between the descriptions of the denotational and operational semantics of symbolic execution. While it is possible to explicitly define the denotational semantics of symbolic execution itself by expressing them in terms of the denotational semantics of the language used, this is not possible for the operational semantics. Instead, one here has to provide a different version of the operational semantics of the language, specifically for symbolic execution.

There is a similarity here between operational semantics as used for symbolic execution, and the axiomatic semantics of the same language. Both are essentially concerned with what properties are provable about a given specification term. However, in axiomatic semantics such properties are expressed as a logical expression in a suitable theory, usually making reference to the values of variables before and after interpreting the specification. In operational semantics as used for symbolic execution a similar logical expression is constructed. However, it is expressed as a predicate on a sequence of states (called a *PredS*, see below) or "description value", and used as the value of the appropriate variable in a suitably defined "symbolic execution state" (*SEStateOp*, see below).

4.1. The data structure

States as used on the operational level will be called *SEStateOps*—*S*ymbolic *E*xecution *States* as used for *Op*erational semantics. In *SEStateOp*s, the information derived using symbolic execution should get associated with those identifiers whose values are described by it. For this reason, *SEStateOp*s use maps from *Name* to the relevant information. The easiest way to model this relevant information seems to be as predicates. These predicates must be predicates on *sequences* of states rather than single states, since they should model the relationship between different states. These are the predicates the user should actually get to see in description values of variables at any stage in the symbolic execution. A *PredS* then is any expression whose semantics can be given as

$$\mathcal{M}_{PredS} : PredS \to StateSeq \to \mathbb{B}$$

where *StateSeq* is defined as

$$StateSeq = (\Sigma_\perp \,|\, StateSeq)^*.$$

StateSeq is defined recursively rather than just as a sequence of states in order to be able to handle blocks and loops, as described below.

The language of *PredS* has to include constant symbols true and false, and operator symbols $\wedge$, $\Rightarrow$, and $\Leftrightarrow$ (all with their standard interpretation).

The only condition on the internal structure of *PredS* is that it must be possible to define a function

$$mention : PredS \to \mathcal{P}(Name)$$

which collects the identifiers mentioned in a given *PredS* into a set. No other conditions are needed since symbolic execution itself makes almost no use of the information contained in the *PredS*, it mainly stores it in a suitable way. Only *simplification* will need to know about the syntax and semantics of *PredS*. (In particular, it needs to know when two *PredS* are equivalent.) The definitions of the syntax and semantics of *PredS* are therefore given in a theory which is used to instantiate symbolic execution for a particular specification language (and thus for a particular language of *PredS*), but not as part of the model of symbolic execution itself. Simplification theories are described in Section 4.5.

We now need to define the structure of the states *SEStateOp* which should store the information contained in the *PredS*. Since allowing *sets of PredS* rather than only individual *PredS* as description values makes it easier to combine different *PredS* and, when needed (for example for simplification), split the result again to get its components, *SEStateOp*s are modelled using maps from *Name* to $\mathcal{P}(PredS)$.

Each symbolic execution step gives rise to a new predicate (or set of predicates) on sequences of states, and obviously each such predicate may provide valuable information that should be associated with the appropriate identifier and the appropriate execution step. Therefore, *SEStateOp*s will be defined as *sequences* of maps from identifiers to sets of predicates on sequences of states. An *SEStateOp* thus stores a *history* of the results of symbolic execution.

In this history a loop should be considered as a single step, even though it may really consist of any number of steps (including 0). Therefore, the result of the loop is modelled as an *SEStateOp* itself, which is then considered as one step in the original *SEStateOp*. Similarly, blocks should be considered as a single step and are therefore also modelled as an *SEStateOp* themselves. This leads to the recursive definition of *SEStateOp* given in Fig. 4. One might thus consider an *SEStateOp* as a tree, where the leaves of the tree are maps and the inner nodes are *SEStateOp*s. Pre-order traversal of this tree describes the execution sequence modelled by the (root) *SEStateOp*.

In addition to the sequence described above, *SEStateOp* contains a field *INDEX* which stores the index or position of this *SEStateOp* in the recursive definition. Without this, one would not be able to recognize whether an *SEStateOp* is itself an element in another *SEStateOp*, or whether it is a top-level state. However, one needs to know this in order to get the right description values in the *SEStateOp*. Since these description values express properties of sequences of states, they need to know which sequence of states they should refer to.

The invariant on *SEStateOp* ensures that every $SEQ(S)$ has a first element which defines the allowed parameter states. An *SEStateOp* itself would not be allowed as first element because it should only arise *as a result of* symbolically executing a

Define

$Index = \mathbb{N}_1^*$

A state as used for describing the operational semantics of a language for symbolic execution is defined recursively as

$SE\text{-}map = Name \xrightarrow{m} \mathcal{P}(PredS)$

$SE\text{-}elem = SE\text{-}map \mid SEStateOp$

$SEStateOp :: \quad SEQ : SE\text{-}elem^*$
$\qquad\qquad\qquad INDEX : Index$

where

$inv\text{-}SEStateOp(mk\text{-}SEStateOp(Seq, ix)) \triangleq$
$\quad Seq \neq [\,]$
$\quad \wedge\ \mathsf{hd}\, Seq : SE\text{-}map$
$\quad \wedge\ \forall k \leq \mathsf{len}\, Seq \cdot$
$\qquad Seq[k] : SEStateOp \Rightarrow INDEX(Seq[k]) = \mathsf{cons}(k, ix)$

Fig. 4. Operational semantics of symbolic execution: state.

specification (usually a loop or block). Additionally, the invariant ensures that *SEStateOp* describes the intuition behind *INDEX* as described above—the *INDEX* of any *SEStateOp* which is the *k*th element of *SEQ* of the *SEStateOp S* is the *INDEX ix* of *S* with *k* added at the front, or cons(*k*, *ix*).

The valuation function $\mathcal{M}_{SEStateOp}$ in Fig. 5 maps an *SEStateOp* to an *SEStateDen*, where the resulting *SEStateDen* contains those sequences of states that satisfy all the predicates in the *SEStateOp*. This is expressed using the following notation and auxiliary functions:

- *satisfies-restriction* takes a sequence of states *σ-seq* and a *PredS ps* and checks whether *σ-seq* satisfies *ps*. Any restriction on a state *σ-seq*[*k*], where len *σ-seq* < *k*, is considered as satisfied. The detailed definition of this function depends on the language of *PredS* under discussion, it would have to be defined formally by recursion over the syntax of *PredS*. $\tilde{\sigma}$ returns a name for the value of an identifier *nm* at some stage *k* in an actual execution sequence and is used to refer to that value in a *PredS* (cf. Section 4.5).

  $satisfies\text{-}restriction : (\Sigma_\perp)^* \times PredS \times Index \to \mathbb{B}.$

  $satisfies\text{-}restriction(\sigma\text{-}seq, ps, ix) \triangleq$
  (1) replace any $\tilde{\sigma}(\mathsf{cons}(k, ix), nm)$ in *ps* by $\sigma\text{-}seq[k](nm)(k \leq \mathsf{len}\ \sigma\text{-}seq)$,
  (2) in the result, replace any atomic formula still containing $\tilde{\sigma}$ by true and evaluate.

The denotation of an *SEStateOp* is given by

$\mathcal{M}_{SEStateOp}$: *SEStateOp* $\rightarrow$ *SEStateDen*

$\mathcal{M}_{SEStateOp}[\![S]\!] \ \triangleq$

$\quad$ *mk-SEStateDen*$(\{[\sigma] \mid$ *satisfies-all-restrictions*$([\sigma], S, 1)\}, 1)$

pre len $SEQ(S) = 1$

$\mathcal{M}_{SEStateOp}[\![$*mk-SEStateOp*$(Seq \oplus e, ind)]\!] \ \triangleq$

$\quad$ let $S_1 =$ *mk-SEStateOp*$(Seq \oplus e, ind)$ in

$\quad$ let $S_2 =$ *mk-SEStateOp*(Seq, ind) in

$\quad$ *mk-SEStateDen*$\Big($

$\quad\quad \{\sigma\text{-}seq: (\Sigma_\perp)^* \mid$ *satisfies-all-restrictions*$(\sigma\text{-}seq, S_1, \text{len}\, Seq + 1)$

$\quad\quad\quad \wedge\ (\text{front}\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!]) \wedge \text{len}\ \sigma\text{-}seq = \text{len}\, Seq + 1$

$\quad\quad\quad\quad \vee\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!])$

$\quad\quad\quad\quad\quad \wedge\ \text{len}\ \sigma\text{-}seq = \text{len}\, Seq$

$\quad\quad\quad\quad\quad \wedge\ \neg\exists \sigma: \Sigma_\perp \cdot$ *satisfies-all-restrictions*$(\sigma\text{-}seq \oplus \sigma, S_1, \text{len}\, Seq + 1)$

$\quad\quad\quad\quad \vee\ \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S_2]\!]) \wedge \text{len}\ \sigma\text{-}seq < \text{len}\, Seq)\},$

$\quad\quad \text{len}\, Seq + 1\Big)$

Fig. 5. Denotation of *SEStateOp*.

- The function satisfies-restrictions checks that σ-*seq* satisfies all the restrictions imposed by S at level i:

$\quad$ *satisfies-restrictions* : $(\Sigma_\perp)^* \times$ *SEStateOp* $\times \mathbb{N}_1 \rightarrow \mathbb{B}$.

$\quad$ *satisfies-restrictions*$(\sigma\text{-}seq, S, i) \ \triangleq$

$\quad\quad$ if $SEQ(S)[i]$: *SE-map*

$\quad\quad$ then $\displaystyle\bigwedge_{n \in \text{dom}\, SEQ(S)[i]} \ \bigwedge_{ps \in SEQ(S)[i](n)}$ *satisfies-restriction*$(\sigma\text{-}seq, ps, INDEX(S))$

$\quad\quad$ else $\exists \sigma\text{-}seq' \cdot$ *satisfies-all-restrictions*$(\sigma\text{-}seq', SEQ(S)[i], \text{len}\ SEQ(SEQ(S)[i]))$

$\quad\quad\quad \wedge\ \text{len}\ \sigma\text{-}seq' = \text{len}\ SEQ(SEQ(S)[i])$

$\quad\quad\quad \wedge\ \sigma\text{-}seq[i-1] = \text{hd}\ \sigma\text{-}seq' \wedge \sigma\text{-}seq[i] = \text{last}\ \sigma\text{-}seq'$

$\quad\quad$ pre $i \leq \text{len}\ SEQ(S)$.

The function *satisfies-all-restrictions* is defined below. As will be seen, the two functions are mutually recursive.

- The function *is-legal-sequence* arises from the conditions on $\mathcal{M}_{Spec}$ and is a (very weak) check that σ-*seq* can actually arise from a sequence of executions.

$\quad$ *is-legal-sequence*$(\sigma\text{-}seq) \ \triangleq$

$\quad\quad \forall i < \text{len}\ \sigma\text{-}seq \cdot \sigma\text{-}seq[i] = \perp \ \Rightarrow\ \sigma\text{-}seq[i+1] = \perp.$

- Finally, *satisfies-all-restrictions* checks that *all* restrictions imposed by S up to level j are satisfied and the sequence "is legal":

$$satisfies\text{-}all\text{-}restrictions : (\Sigma_\perp)^* \times SEStateOp \times \mathbb{N}_1 \to \mathbb{B}.$$

$$satisfies\text{-}all\text{-}restrictions(\sigma\text{-}seq, S, j) \ \triangleq$$

$$\bigwedge_{i=1}^{j} satisfies\text{-}restrictions(\sigma\text{-}seq, S, i)$$

$$\wedge\ is\text{-}legal\text{-}sequence(\sigma\text{-}seq).$$

$$\mathsf{pre}\ j \leq \mathsf{lens}\ SEQ(S).$$

We now discuss some of the properties of $\mathcal{M}_{SEStateOp}$. The first one follows immediately from the definitions:

Lemma 4.1. *For all S: SEStateOp*

$$LEN(\mathcal{M}_{SEStateOp}[\![S]\!]) = \mathsf{len}\ SEQ(S).$$

Theorem 4.2. *The valuation function $\mathcal{M}_{SEStateOp}$ is total.*

Proof. One needs to show that, for any S: *SEStateOp*, $\mathcal{M}_{SEStateOp}[\![S]\!]$: *SEStateDen* exists. To do so, one needs to show that $\mathcal{M}_{SEStateOp}[\![S]\!]$ satisfies the invariant *inv-SEStateDen*. This is done in Appendix B. □

A *SEStateDen* can represent a predicate on states. Similarly, one can represent such predicates by *SEStateOps*. Given φ: *Pred*, let Φ be the *PredS*

$$\mathcal{M}_{Pred}[\![\varphi]\!]\{n \mapsto \tilde{\sigma}([1], n) \mid n: Name\}$$

and let

$$S(\varphi) \ \triangleq \ mk\text{-}SEStateOp([\{n \mapsto \{\Phi\} \mid n: Name\}], [\,]).$$

Then $\mathcal{M}_{SEStateOp}[\![S(\varphi)]\!]$ is the *SEStateDen* that represents φ, and we say that $S(\varphi)$ is the *SEStateOp* that represents φ. Of course, Φ does not have to be associated with each *Name n*, one could alternatively only associate it with those n that are mentioned in Φ, or even only with one arbitrary n.

The valuation function of *SEStateOp*, like the others defined before, could also be considered as a *retrieve* function [20, pp. 181ff]. In this case, it has an adequacy proof obligation associated with it: a representation *Rep* is *adequate* with respect to a retrieve function $retr: Rep \to Abs$ iff $\forall a \in Abs \cdot \exists r \in Rep \cdot retr(r) = a$. If *Val* is finite, then it depends on the expressiveness of *PredS* whether $\mathcal{M}_{SEStateOp}$ satisfies this obligation. For infinite *Val*, however, there are uncountably many sets of state sequences and therefore uncountably many *SEStateDen*. On the other hand, there are only countably many *SEStateOp* and therefore *SEStateOp* cannot be adequate with respect to $\mathcal{M}_{SEStateOp}$.

So far we have always allowed the *PredS*-conditions inside an *SEStateOp* to refer to *any* element of a state sequence, including future ones. This will cause some problems when adding another element to the sequence *SEStateOp*, for example in the transition for VDM-operations in Section 4.8. Earlier conditions on the current state σ may destroy the faithfulness (see Definition 4.6) of that transition. We therefore define the following property which ensures that this problem does not arise. The definition uses the auxiliary function

$$\textit{highest-index}: \textit{PredS} \to \textit{Index}$$

which finds the highest index *ix* such that for some n: *Name* and some *ixseq*: $\mathbb{N}^*$ with hd $ixseq = ix$, $\tilde{\sigma}(ixseq, n)$ occurs in a *PredS*. This function has to be defined recursively over the syntax of *PredS*.

Definition 4.3. *mk-SEStateOp*(*Seq*, *ind*) is *well-behaved* iff

$$\forall i \leq \text{len } Seq \cdot Seq[i]: \textit{SE-map} \Rightarrow \bigwedge_{n \in \text{dom } Seq[i]} \bigwedge_{ps \in Seq[i](n)} \textit{highest-index}(ps) \leq i.$$

The main motivation for the definition of well-behaviour is captured by the following lemma:

Lemma 4.4. *Let* len σ-*seq* $\geq i$. *If* S: *SEStateOp is well-behaved, then for all* σ: $\Sigma_\perp$

$$\textit{satisfies-restrictions}(\sigma\text{-}seq, S, i) \Leftrightarrow \textit{satisfies-restrictions}(\sigma\text{-}seq \oplus \sigma, S, i).$$

Proof. ($\Rightarrow$) Follows from the well-behaviour of S.
($\Leftarrow$) Follows directly from the definition of *satisfies-restrictions*. □

4.2. A syntactic view of symbolic execution

It is not immediately clear from the above how *SEStateOps* relate to the conventional concept of symbolic execution, where identifiers take symbolic values. Consider an identifier **int** x. Possible kinds of values of x include:

- *Actual values* (ground terms). These are the "usual" values as used in actual execution. The identifier x has value c for some $c \in Val$ at stage i of the *SEStateDen* τ iff

  $$\forall \sigma\text{-}seq \in SEQS(\tau) \cdot \sigma\text{-}seq[i](x) = c.$$

  Accordingly, this is represented by the *PredS* $\tilde{\sigma}([i], x) = c$. In the appropriate S: *SEStateOp* we then get that the *PredS* $\tilde{\sigma}([i], x) = c$ is in $S[i](x)$.
- *Symbolic values* (terms containing symbols denoting identifiers). For example $x = 2 * y - 1$ is a possible symbolic value of the identifier x. Symbolic values denote a whole range of input values but possibly restricted to those of a particular form (odd numbers for x in the above example). They are distingusished by the fact that they express the value of an identifier (x in the above

example) as an explicit function of the values of other identifiers (y in the example). The identifier x has value $f(y)$ at stage i of the *SEStateDen* τ iff

$$\forall \sigma\text{-}seq \in SEQS(\tau) \cdot \sigma\text{-}seq[i](x) = f(\sigma\text{-}seq[i](y)).$$

These two kinds of values in symbolic execution are the ones used in most symbolic execution systems. However, they are too restricted for dealing with specifications, since they cannot deal with values that are defined implicitly, or underdefined. Therefore we introduce:

- *Description values.* A variable, and in particular the output variable, may have a *predicate* as a value, which describes the value implicitly, rather than a term describing it explicitly. A set of such predicates is called a description value, it may describe a set of states as associated with an identifier in a *SEStateDen.*

These description values are general predicates of type *PredS,* while both actual values and symbolic values can be considered as special cases of description values.

The most general results would be achieved by letting S: *SEStateOp* describe the results of (actual) execution starting with the set $\Sigma_\perp$ of all states. For practical reasons, however, one will usually have to cut down the complexity of the output term by (interactively) restricting the admissible universes of the variables used, in extreme cases even restricting it to just one element, i.e. to mix symbolic and actual execution. In S, such a restriction just has the effect of adding another constraint ps: *PredS* at the last element of S. Although in theory it does not matter for which n: *Name* ps is added to $S[i](n)$, in practice one will probably want to add it to all those n which are mentioned in the constraint ps.

In some cases, it might be more useful to show only part of the information gained from symbolic execution, for example to ignore a more general description such as an invariant and only show those parts of the information about the output that arise from the execution itself. In this case, the information that is not shown should be hidden behind "...", so that the user can always get to it again and "unhide" it. Eliminating the information rather than just hiding it would lead to *weak* symbolic execution.

4.3. Transitions and rules

In the following I am going to define the kind of transitions and rules used for describing the operational semantics of language constructs in general, and then give the appropriate transitions and rules for various constructs. In many cases (e.g. the rule for **if-then-else**), the transitions and rules of the operational semantics of various language constructs are defined by translating them into an equivalent construct in the language used for describing the results (the language of *PredS*), and then simplying the result whenever possible. This simplification will hopefully help to eliminate the construct from the description.

From the point of view of their purpose, one can therefore distinguish three different kinds of transitions:

- Transitions describing (state-changing) specifications, like the one in Section 4.8. describing the effect of VDM-operations. Since such operations actually lead to a new state, they are described by transitions that extend a *S*: *SEStateOp* by adding another element to the sequence *SEQ*(*S*).
- Transitions that eliminate combinators for specifications by translating them into equivalent constructs used inside *PredS* expressions. As an example, consider the rule for **if-then-else** given in Section 4.9 (Rule 6).
- Simplification transitions derived from the theory for *PredS*, as discussed in Section 4.5. The transition $S_1 \hookrightarrow S_2$ is allowed if S_2 can be derived from S_1 by simplification of *PredS* only.

We now define the various components that are needed to express transitions. *SpecName* is the type of specification names, and *SpecMap* associates specification names with specifications:

$$SpecMap = SpecName \xrightarrow{m} Spec.$$

Configurations consist of a sequence of *SpecNames* (which may be empty) and an *SEStateOp*:

$$\begin{aligned} Conf :: \; & SNSEQ : SpecName^* \\ & STATE : SEStateOp \end{aligned}$$

A configuration *mk-Conf*(*snseq*, *S*) will be written as $\langle snseq, S\rangle$.

The configuration $\langle snseq, S\rangle$: *Conf* describes the fact that the sequence of specifications given by *snseq* is to be applied to *S*. Given some *sm*: *SpecMap*, the denotation of a configuration is therefore defined as below, using the auxiliary function *evalseq* which, given a sequence *aseq* and a function *f* on its elements, applies *f* to all the elements of *aseq*:

$$\mathcal{M}_{Conf} : Conf \rightarrow SEStateDen,$$

$$\mathcal{M}_{Conf}[\![\langle snseq, S\rangle]\!] \;\triangleq$$

$$symbolic\text{-}ex\text{-}s[\![evalseq(snseq, sm)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]).$$

Transitions are defined as

$$Trans = \biguplus_{E} E \times E$$

where $\biguplus$ denotes disjoint union, and *E* ranges over *Conf* and the different syntactic categories of the specification language such as *Expr*. A transition *mk-Trans*(e_1, e_2) will be written as $e_1 \hookrightarrow e_2$.

$\langle Op_1, S_1\rangle \hookrightarrow \langle Op_2, S_2\rangle$ denotes the fact that one symbolic execution step transforms $\langle Op_1, S_1\rangle$ into $\langle Op_2, S_2\rangle$, but $\hookrightarrow$ will also be used to denote its transitive-reflexive closure.

Rules take the form

$$\begin{aligned} \mathit{Rule} :: \mathit{hyps} &: \mathcal{P}(\mathit{Trans} \cup \mathit{PredS}),\\ \mathit{conc} &: \mathit{Trans}. \end{aligned}$$

This fits with the definition of rules (or rule statements) in *mural* [18], sinc; both *Trans* and *PredS* are special forms of *Assertions*.

Definition 4.5 (*Operational semantics*). The operational semantics of a language $\mathcal{L}$ are given by a transition relation $t \subseteq \mathit{Trans}$ (usually written as $\_\hookrightarrow\_$). This relation is often given as a (fairly small) set T of transitions and transition schemata together with a set R of rules and rule schemata. t is then the smallest set of transitions containing all transitions and instantiations of transition schemata from T, which is closed under application of (instantiations of) rules in R.

4.4. The relationship between denotational and operational semantics

There are a number of important properties describing the relationship between denotational and operational semantics, for example faithfulness, full abstraction or termination. Faithfulness is a property of *individual* statements or transitions, while the other properties mentioned deal with the semantics of the language as a whole. This paper does not give the complete operational semantics of any language, but only a few transitions describing some important language constructs. Therefore these other properties are not relevant in this context and we will only deal with faithfulness.

The following definition is based on [27, Section 10.7].

Definition 4.6 (*Faithfulness*).

(a) A transition $e_1 \hookrightarrow e_2$ is faithful with respect to the denotational semantics $\mathcal{M}$ if it implies $\mathcal{M}[\![e_1]\!] = \mathcal{M}[\![e_2]\!]$, or $\mathcal{M}[\![e_1]\!] \supseteq \mathcal{M}[\![e_2]\!]$ if $\mathcal{M}$ returns a *set* of valuations.

(b) An operational semantics $\_\hookrightarrow\_$ is faithful with respect to the denotational semantics $\mathcal{M}$ if for all expressions e_1 and e_2, $e_1 \hookrightarrow e_2$ implies $\mathcal{M}[\![e_1]\!] = \mathcal{M}[\![e_2]\!]$, or $\mathcal{M}[\![e_1]\!] \supseteq \mathcal{M}[\![e_2]\!]$ if $\mathcal{M}$ returns a *set* of valuations.

All the transitions given in the rest of the paper can be shown to be faithful with respect to the standard denotational semantics of the relevant statement of the language, although the proofs tend to be lengthy and tedious. Therefore only one is given below, since it is quite short, and a second one is given elsewhere (Theorem 4.10).

An important general rule that shows how symbolic execution of a sequence of specifications can be split up into symbolic execution of its elements is the following:

Rule 1.

$$\frac{\langle [sn], S\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle \mathsf{cons}(sn, snseq), S\rangle \hookrightarrow \langle snseq, S'\rangle}$$

Lemma 4.7. *Rule* 1 *preserves faithfulness: if the hypothesis transition is faithful, then so is the conclusion.*

Proof. Let sm: *SpecMap* be given. Assume that

$$\langle [\![sn]\!], S\rangle \hookrightarrow \langle [\,], S'\rangle$$

is faithful. This implies that

$$\textit{symbolic-ex}[\![sm(sn)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]) = \mathcal{M}_{SEStateOp}[\![S']\!].$$

Then

$$\begin{aligned}
&\mathcal{M}_{Conf}[\![\langle \mathsf{cons}(sn, snseq), S\rangle]\!]\\
&\quad = \textit{symbolic-ex-s}[\![\textit{evalseq}(\mathsf{cons}(sn, snseq), sm)]\!](\mathcal{M}_{SEStateOp}[\![S]\!])\\
&\quad = \textit{symbolic-ex-s}[\![\textit{evalseq}(snseq, sm)]\!](\textit{symbolic-ex}[\![sm(sn)]\!](\mathcal{M}_{SEStateOp}[\![S]\!]))\\
&\quad = \textit{symbolic-ex-s}[\![\textit{evalseq}(snseq, sm)]\!](\mathcal{M}_{SEStateOp}[\![S']\!])\\
&\quad = \mathcal{M}_{Conf}[\![\langle snseq, S'\rangle]\!]
\end{aligned}$$

as required. □

4.5. Simplification

An important aspect of symbolic execution is the simplification of result terms. In general, it depends very much on the user and what he wants to do whether a given terms is "simpler" than another. On the other hand, a term cannot be "simplified" into an arbitrary other term, both terms need to be equivalent in a suitable theory.

Assume we are given a specification language $\mathcal{L}$. To reason about *PredS*, for example to decide whether a *PredS* ps_1 can be simplified to ps_2, one needs a suitable theory of *PredS*. This theory, which will be called $Th(\mathcal{L})$, is based on the theory used to reason about terms in $\mathcal{L}$. Additionally an indexing mechanism is needed to differentiate between the values of program variables (identifiers or names) at different stages in the execution sequence. To do so, sequences $(\sigma_i)_i$ of states are introduced, where $\sigma_i : \Sigma_\perp$. Since the definition of *SEStateOp* is recursive, simple sequences are not enough—we actually need iterated sequences where σ_i might be a sequence of states itself. This is modelled by introducing a function $\tilde{\sigma}$, which returns the name of the value of the identifier n at a given stage in the execution, with the signature

$$\tilde{\sigma} : \textit{Index} \times \textit{Name} \to \textit{Val-ref}.$$

For simplicity, we shall in the following identify the element i: $\mathbb{N}_1$ with the index $[i]$.

Now a *PredS* can be defined as a predicate that contains names of values of identifiers *at some stage*, instead of the identifiers themselves. The resulting theory of *PredS* is the theory used for simplification: ps_1: *PredS* inside some *SEStateOp* can be simplified to ps_2: *PredS* if they are equivalent in $Th(\mathscr{L})$. Weak simplification, as used in weak symbolic execution, requires that ps_1 implies ps_2 in $Th(\mathscr{L})$.

4.6. Assignment statement

Consider again the assignment statement $x := e$. We now have to provide a transition that describes the effect of this statement in an *SEStateOp* such that the transition is faithful with respect to the denotational semantics as given in Section 3.3.

Rule 2.

$$
\begin{aligned}
&\vdash \langle [x := e], \textit{mk-SEStateOp}(Seq, ind)\rangle\\
&\quad\hookrightarrow \langle [\,], \textit{mk-SEStateOp}(\\
&\qquad Seq \oplus \{n \mapsto \text{if } n = x\\
&\qquad\qquad \text{then } \{\tilde{\sigma}(\text{cons}(\text{len } Seq + 1, ind), x)\\
&\qquad\qquad\qquad = e[nm/\tilde{\sigma}(\text{cons}(\text{len } Seq + 1, ind), nm) \mid nm \in Name]\}\\
&\qquad\qquad \text{else } \{\sigma(\text{cons}(\text{len } Seq + 1, ind), n)\\
&\qquad\qquad\qquad = \tilde{\sigma}(\text{cons}(\text{len } Seq, ind), n)\}\},\\
&\qquad ind)\rangle
\end{aligned}
$$

Lemma 4.8. *The transition given in Rule* 2 *is faithful with respect to the denotation given in Section* 3.3.

Proof. We have to show that both sides of the transition have the same denotation. Identifying *SpecName* and *Spec*, we get for the left-hand side

$$
\begin{aligned}
&\mathcal{M}_{Conf}[\![\langle [x := e], \textit{mk-SEStateOp}(Seq, ind)\rangle]\!]\\
&\quad = \textit{symbolic-ex}[\![x := e]\!](\mathcal{M}_{SEStateOp}[\![\textit{mk-SEStateOp}(Seq, ind)]\!])
\end{aligned}
$$

while for the right-hand side of the transition

$$\mathcal{M}_{Conf}[\![\langle [\,], \textit{mk-SEStateOp}(\ldots)\rangle]\!] = \mathcal{M}_{SEStateOp}[\![\textit{mk-SEStateOp}(\ldots)]\!].$$

The proof that both are equal now amounts to a very lengthy and tedious calculation which will not be given here. □

4.7. Block structures, variable declarations and scoping

We start off the description of operational semantics of language constructs with some rules describing block structures. The approach taken by, for example, Plotkin [26] for operational semantics of *actual* execution of blocks and local variable declarations is not possible here, since it discards information about earlier states, only the current values of variables being stored. In symbolic execution, this is not sufficient since the predicates describing a current value of a variable in general refer to earlier values, therefore the whole history needs to be preserved.

Therefore blocks are modelled by *SEStateOps* that are *elements* of the sequence *SEQ* of the original *SEStateOp*. In order to be able to describe how this is done, the following auxiliary functions will be needed:

- The function *current-names* is defined as follows

```
current-names: $SEStateOp \to \mathscr{P}(Name)$.

current-names($S$) $\triangleq$ if last $SEQ(S)$: $SE\text{-}map$
                    then dom last $SEQ(S)$
                    else dom hd $SEQ(\text{last } SEQ(S))$.
```

- The function *current-index* finds the current or last index in an *SEStateOp*:

```
current-index: $SEStateOp \to Index$.

current-index($S$) $\triangleq$
  if last $SEQ(S)$: $SE\text{-}map$
  then $[\text{len } SEQ(S)] \curvearrowright INDEX(S)$
  else $(current\text{-}index(\text{last } SEQ(S)) \oplus \text{len } SEQ(S)) \curvearrowright INDEX(S)$.
```

current-index(S) is always the index of a *SE-map*.

- *previous* takes as input the index *ix* of an element of some *SEStateOp* and returns the index of the previous element:

```
previous: $Index \to Index$.

previous($ix$) $\triangleq$ if hd $ix = 1$
               then tl $ix$
               else cons(hd $ix - 1$, tl $ix$).

pre $ix \neq [\,]$.
```

- The function *add-to-SEStateOp* adds an element to the sequence in an *SEStateOp*.

```
add-to-SEStateOp: $SEStateOp \times SE\text{-}elem \to SEStateOp$.

add-to-SEStateOp($S, e$) $\triangleq$ $mk\text{-}SEStateOp(SEQ(S) \oplus e, INDEX(S))$.
```

- The function *start-block* starts a new block by creating a new *SEStateOp* which is then added as a new element to the sequence *SEQ* of the current one. *SEQ* of the new *SEStateOp* only consists of one element which describes that "nothing changes"—all identifiers keep the same value that they had before.

$$\textit{start-block}: \textit{SEStateOp} \to \textit{SEStateOp}.$$

$$\begin{aligned}
&\textit{start-block}(S) \triangleq\\
&\quad \text{let } S' = \textit{mk-SEStateOp}([\{n \mapsto \{\tilde{\sigma}([1, \text{len } SEQ(S)+1] \frown INDEX(S), n)\\
&\qquad\qquad\qquad = \tilde{\sigma}([\text{len } SEQ(S)] \frown INDEX(S), n)\}\\
&\qquad\qquad\qquad | \, n \in \textit{current-names}(S)\}],\\
&\qquad\qquad \text{cons}(\text{len } SEQ(S)+1, INDEX(S))) \text{ in}\\
&\quad \textit{add-to-SEStateOp}(S, S').
\end{aligned}$$

- The function *finish-block* is defined as:

$$\textit{finish-block}: \textit{SEStateOp} \to \textit{SEStateOp}.$$

$$\begin{aligned}
&\textit{finish-block}(S) \triangleq\\
&\quad \text{let } m = \{n \mapsto \{\tilde{\sigma}([\text{len } SEQ(S)] \frown INDEX(S), n) = \tilde{\sigma}(INDEX(S), n)\}\\
&\qquad\qquad | \, n \in \text{dom hd } (SEQ(S))\} \text{ in}\\
&\quad \textit{add-to-SEStateOp}(S, m).
\end{aligned}$$

The rule for describing the operational semantics of a block is then given by.

Rule 3.

$$\frac{\langle \textit{snseq}, \text{last } SEQ(\textit{start-block}(S))\rangle \hookrightarrow \langle [\,], S'\rangle}{\langle \textbf{begin } \textit{snseq} \textbf{ end}, S\rangle \hookrightarrow \langle [\,], \textit{add-to-SEStateOp}(S, \textit{finish-block}(S'))\rangle}$$

where **begin** *snseq* **end** is used as the *name* of the appropriate sequence of specifications. A similar convention will be used for other constructs below.

Declarations of local variables are handled by mapping them to the empty set of restrictions and keeping all other variables equal:

Rule 4.

$$\begin{aligned}
&\vdash \langle [\textbf{var } x\text{: } \textit{Type}], \textit{mk-SEStateOp}(\textit{Seq}, \textit{ind})\rangle\\
&\quad \hookrightarrow \langle [\,], \textit{mk-SEStateOp}(\\
&\qquad \textit{Seq} \oplus \{n \mapsto \text{if } n = x\\
&\qquad\qquad\qquad \text{then } \{\tilde{\sigma}(\text{cons}(\text{len } \textit{Seq}+1, \textit{ind}), x)\text{: } \textit{Type}\}\\
&\qquad\qquad\qquad \text{else } \{\tilde{\sigma}(\text{cons}(\text{len } \textit{Seq}+1, \textit{ind}), n)\\
&\qquad\qquad\qquad\qquad = \tilde{\sigma}(\text{cons}(\text{len } \textit{Seq}, \textit{ind}), n)\}\},\\
&\qquad \textit{ind})\rangle
\end{aligned}$$

Additionally one needs to express that if a variable has a certain type at some stage of the execution, it will keep that type until the end of the current block or until a new declaration of that variable is encountered. (Most languages do have restrictions as to when a new declaration of the same variable is allowed to occur, but since we here assume that all specifications handled are syntactically correct we do not need to deal with such restrictions.) This can be done using the following rule:

Rule 5.

$$\frac{\tilde{\sigma}(\mathit{current\text{-}index}(S), x)\colon T; \quad \mathit{declares\text{-}var}(sn, x); \quad \langle [\![sn]\!], S\rangle \hookrightarrow \langle [\,], S'\rangle}{\tilde{\sigma}(\mathit{current\text{-}index}(S'), x)\colon T}$$

where *declares-var* is a predicate that returns true if the specification $[\![spec]\!]$ declares the variable x to be of a certain type. E.g. *declares-var*($[\![x\colon T]\!], x$) is true. The exact definition of *declares-var* must be defined based on the syntax of the language, in particular it needs to recognize *implicit* variable declarations as allowed by some languages, such as FORTRAN.

Define the map $m\colon \mathit{SE\text{-}map}$ as

$$m(n) \triangleq \begin{cases} \text{let } \mathit{rest} = \mathit{current\text{-}names}(S) - \{a, ew, r\} \text{ in} & \\ \text{let } \mathit{oldseq} = \text{cons}(\text{len}\,\mathit{Seq}, \mathit{index}) \text{ in} & \\ \text{let } \mathit{newseq} = \text{cons}(\text{len}\,\mathit{Seq} + 1, \mathit{index}) \text{ in} & \\ \{\mathit{inv\text{-}T_1}[nm/\tilde{\sigma}(\mathit{newseq}, nm) \mid nm\colon \mathit{Name}]\} & \text{if } n = a \\ & \\ \{\text{if } \mathit{pre}(\mathit{new}(a), \mathit{old}(er), \mathit{old}(ew)) & \text{if } n = ew \\ \text{then } \mathit{post}(\mathit{new}(a), \mathit{old}(er), \mathit{old}(ew), \mathit{new}(er), \mathit{new}(ew)), & \\ \mathit{inv\text{-}T_4}[nm/\tilde{\sigma}(\mathit{newseq}, nm) \mid nm\colon \mathit{Name}]\} & \\ & \\ \{\text{if } \mathit{pre}(\mathit{new}(a), \mathit{old}(er), \mathit{old}(ew)) & \text{if } n = r \\ \text{then } \mathit{post}(\mathit{new}(a), \mathit{old}(er), \mathit{old}(ew), \mathit{new}(er), \mathit{new}(ew)), & \\ \mathit{inv\text{-}T_2}[nm/\tilde{\sigma}(\mathit{newseq}, nm) \mid nm\colon \mathit{Name}]\} & \\ & \\ \{\text{if } \mathit{pre}(\mathit{new}(a), \mathit{old}(er), \mathit{old}(ew)) & \text{if } n \in \mathit{rest} \\ \text{then } \tilde{\sigma}(\mathit{newseq}, n) = \tilde{\sigma}(\mathit{oldseq}, n)\} & \end{cases}$$

where $\mathit{old}(x)$ denotes the value of x before, and $\mathit{new}(x)$ the value of x after interpretation of Op.

Then

$$\langle [Op], S\rangle \hookrightarrow \langle [\,], \mathit{add\text{-}to\text{-}SEStateOp}(S, m)\rangle$$

Fig. 6. Transition for VDM-operations.

4.8. *Operations defined in terms of pre- and postconditions*

Given a VDM-specification of an operation

$$\begin{array}{l} Op\,(a\colon T_1)\; r\colon T_2 \\ \textsf{ext rd}\; er\colon T_3 \\ \quad \textsf{wr}\; ew\colon T_4 \\ \textsf{pre}\; \varphi(a, er, ew) \\ \textsf{post}\; \psi(a, \overleftarrow{ew}, r, er, ew) \end{array}$$

Here a, er, and ew may each denote a sequence of variable names, where T_1, T_3, and T_4 are sequences of types of the same length.

If the language of the theory $Th(\mathscr{L})$ of *PredS* was not rich enough to express these predicates, one would have to be content with *weak* symbolic execution and use predicates which are *implied* by the ones above. However, this language should be derived from LPF in the way described in Section 4.5, in which case it is expressive enough.

Instead of using the conditional if-then in Fig. 6, it could be useful for practical purposes to use an alternative conditional with the same denotational semantics but which result in a warning message to the user if the condition is not satisfied, since it really is the precondition of an operation.

Example 4.9. Given the specification OP_1 from Example 3.2. As before, we assume that the precondition holds. Since x and y are the only identifier used, we start with the *SEStateOp*

$$S = mk\text{-}SEStateOp([x \mapsto \{\tilde{\sigma}([1], x) \geq 0\}, y \mapsto \{\tilde{\sigma}([1], x) \geq 0\}], [\,]).$$

The appropriate instantiation of the rule giving the operational semantics of VDM-operations is then given by (after some simplification)

$$\text{let } m = \left\{\begin{array}{l} x \mapsto \{\tilde{\sigma}([2], y)^2 \leq \tilde{\sigma}([1], x) \\ \qquad \wedge\, \tilde{\sigma}([2], x) = \tilde{\sigma}([1], x) + 1\} \\ y \mapsto \{\tilde{\sigma}([2], y) \leq \tilde{\sigma}([1], x)\} \end{array}\right\} \text{ in}$$

$$\vdash \langle [OP_1], S\rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S) \oplus m, [\,])\rangle$$

Theorem 4.10. *The transition scheme in Fig. 6 giving the operational semantics of VDM-operations is faithful, provided that S is well-behaved.*

Proof. See [24, Appendix A.3]. □

4.9. *Operational semantics of* ***if-then-else***

Unfortunately, the rule describing the operational semantics of the **if-then-else** combinator as used for symbolic execution turns out to be far more complicated than those used for actual execution as given by Plotkin [26]. This is due to the fact that, as mentioned before, in symbolic execution one has to store the whole *history* of results, not just the current ones, and the recursive definition of states *SEStateOp* needed accordingly. The rule is therefore expressed using a (recursive) auxiliary function that "merges" two *SEStateOps*, and at the same time turns each *ps*: *PredS* in either of the two *SEStateOps* into the appropriate conditional. The latter is done by *ITE-merge-map*, which is then called by the general function *ITE-merge*. *ITE-merge* has to distinguish nine different cases, since either of the two sequences to be merged may be empty or start with a *SE-map* or start with a *SEStateOp*.

First define the auxiliary functions:

$$ITE\text{-}merge\text{-}map: SE\text{-}map \times SE\text{-}map \times PredS \to SE\text{-}map.$$

$$ITE\text{-}merge\text{-}map(m_1, m_2, ps) \triangleq$$

$$\{n \mapsto \left\{\begin{array}{ll} \{\text{if } ps \text{ then } ps_1 \text{ else } p_2 & \\ \quad \mid ps_i \in m_i(n), i = 1, 2\} & \text{if } n \in \text{dom } m_1 \cap \text{dom } m_2 \\ \{\text{if } ps \text{ then } ps_1 \text{ else true} & \\ \quad \mid ps_1 \in m_1(n)\} & \text{if } n \in \text{dom } m_1 - \text{dom } m_2 \\ \{\text{if } ps \text{ then true else } ps_2 & \\ \quad \mid ps_2 \in m_2(n)\} & \text{if } n \in \text{dom } m_2 - \text{dom } m_1 \end{array}\right\}\}$$

If the two sequences to be merged have different length, then, as defined in *ITE-merge* below, one will eventually get into the position where one of the sequences starts with a map, and the other one is empty. This case is handled by:

$$ITE\text{-}merge\text{-}empty: SE\text{-}map \times PredS \times Index \to SE\text{-}map.$$

$$ITE\text{-}merge\text{-}empty(m, ps, ix) \triangleq$$

$$\{n \mapsto \{\text{if } ps \text{ then } ps_1 \text{ else } `\tilde{\sigma}(ix, n) = \tilde{\sigma}(previous(ix), n)' \mid ps_1 \in m(n)\}\}.$$

Now define

$$kind: SE\text{-}elem^* \to \{\text{EMPTY}, \text{MAP}, \text{SES}\},$$

```
kind(seq) ≜ if seq = [ ]
            then EMPTY
            else if hd seq: SE-map
                 then MAP
                 else SES.
```

Then

$$ITE\text{-}merge: SE\text{-}elem^* \times SE\text{-}elem^* \times PredS \times SEStateOp \to SEStateOp.$$

$$ITE\text{-}merge(seq_1, seq_2, ps, S) \triangleq$$

```
cases $kind(seq_1)$, $kind(seq_2)$ of
  EMPTY, EMPTY → $S$
  EMPTY, MAP →
    let $S_1$ = $add\text{-}to\text{-}SEStateOp($
            $S$, $ITE\text{-}merge\text{-}empty($hd $seq_2$, $ps$, $INDEX(S)))$ in
    $ITE\text{-}merge([\,]$, tl $seq_1$, $ps$, $S_1)$
  EMPTY, SES →
    let $S_1$ = $add\text{-}to\text{-}SEStateOp(S$, $ITE\text{-}merge([\,]$, hd $seq_2$, $ps$, $S))$ in
    $ITE\text{-}merge([\,]$, tl $seq_2$, $ps$, $S_1)$
  MAP, EMPTY →
    let $S_1$ = $add\text{-}to\text{-}SEStateOp($
            $S$, $ITE\text{-}merge\text{-}empty($hd $seq_1$, $ps$, $INDEX(S)))$ in
    $ITE\text{-}merge($tl $seq_1$ $[\,]$, $ps$, $S_1)$
  MAP, MAP →
    let $S_1$ = $add\text{-}to\text{-}SEStateOp($
            $S$, $ITE\text{-}merge\text{-}map$ (hd $seq_1$, hd $seq_2$, $ps))$ in
    $ITE\text{-}merge($tl $seq_1$, tl $seq_2$, $ps$, $S_1)$
  MAP, SES →
    let $S_1$ = last($SEQ(start\text{-}block(S)))$ in
    let $S_2$ = $finish\text{-}block(add\text{-}to\text{-}SEStateOp(S_1$, hd $seq_1))$ in
    let $S_3$ = $ITE\text{-}merge(S_2$, hd $seq_2$, $ps$, $S)$ in
    $ITE\text{-}merge($tl $seq_1$, tl $seq_2$ $ps$, $S_3)$
  SES, EMPTY →
    let $S_1$ = $add\text{-}to\text{-}SEStateOp(S$, $ITE\text{-}merge($hd $seq_1$, $[\,]$, $ps$, $S))$ in
    $ITE\text{-}merge($tl $seq_1$, $[\,]$, $ps$, $S_1)$
  SES, MAP →
    let $S_1$ = last($SEQ(start\text{-}block(S)))$ $in$
    let $S_2$ = $finish\text{-}block(add\text{-}to\text{-}SEStateOp(S_1$, hd $seq_2))$ in
    let $S_3$ = $ITE\text{-}merge($hd $seq_1$, $S_2$, $ps$, $S)$ in
    $ITE\text{-}merge($tl $seq_1$, tl $seq_2$, $ps$, $S_3)$
  SES, SES →
    let $S_1$ = $ITE\text{-}merge($hd $seq_1$, hd $seq_2$, $ps$, $S)$ in
    $ITE\text{-}merge($tl $seq_1$, tl $seq_2$, $ps$, $S_1)$
end.
```

Note that this definition implies

$$\text{len } SEQ(ITE\text{-}merge(seq_1, seq_2, ps, S)) = \text{len } SEQ(S) + \max(\text{len } seq_1, \text{len } seq_2)$$

(proof by double induction over len seq_1 and len seq_2).

Now the rule describing the operational semantics of **if-then-else** can be given as:

Rule 6. let $ps(S) = \varphi[n/\bar{\sigma}(cons(\text{len } SEQ(S), INDEX(S)), n) \mid n\colon Name]$ in

$$\frac{\begin{array}{c} ps(S) \vdash \langle snseq_1, S\rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S) \frown seq_1, INDEX(S))\rangle \\ \neg ps(S) \vdash \langle snseq_2, S\rangle \hookrightarrow \langle [\,], mk\text{-}SEStateOp(SEQ(S) \frown seq_2, INDEX(S))\rangle \end{array}}{\langle [\textbf{if } \varphi \textbf{ then } sn_1 \textbf{ else } sn_2], S\rangle \hookrightarrow \langle [\,], ITE\text{-}merge(seq_1, seq_2, ps(S), S)\rangle}$$

assuming that the language of the simplification theory $Th(\mathscr{L})$ (cf. Section 4.5) has the connective if-then-else.

Here "**if** φ **then** sn_1 **else** sn_2" is the *name* of the appropriate specification. Note that the combinator **if-then-else** and the connective if-then-else are different constructs, of different types.

The simplification theory $Th(\mathscr{L})$ should then contain some rules for handling if-then-else, for example

$$\text{if true then } \psi_1 \text{ else } \psi_2 \Leftrightarrow \psi_1$$

$$\text{if false then } \psi_1 \text{ else } \psi_2 \Leftrightarrow \psi_2$$

$$\text{if } ps \text{ then } \psi \text{ else } \psi \Leftrightarrow \psi$$

Of course one could additionally introduce two rules that handle the case when either $ps(S)$ or $\neg ps(S)$ is known to hold. Although these rules are not strictly necessary since they can be derived from the above (assuming that the operational semantics given always allow one to find seq_1 and seq_2), they would save a lot of simplification work.

*4.10. Operational semantics of **while**-loops*

Similar to block structures, loops are considered as a single step even though their execution may consist of any number of steps. This is achieved by describing the results of this execution sequence in a different *SEStateOp* or block which is then considered as a single step in the original *SEStateOp*. However, there is an additional complication in that with the usual approach to operational semantics, using a rule like

$$\begin{array}{ll} \langle \textbf{while } \varphi \textbf{ do } [\![spec]\!] \textbf{ od}, S\rangle \hookrightarrow \langle \textbf{if } \varphi & \\ & \textbf{then } ([\![spec]\!]; \textbf{while } \varphi \textbf{ do } [\![spec]\!] \textbf{ od}) \\ & \textbf{else skip}, S\rangle \end{array}$$

it is not clear when encountering a **while**-statement whether to start a new block (because it is a new **while**-statement) or continue the current one (because it is a new iteration of a previously encountered statement). Therefore we introduce two different versions of the **while**-statement that allow one to distinguish the two.

while-do is the "proper" statement that starts a new block, and **auxwhile-do** is an auxiliary version that is used to continue the current block.

This leads to the rules (again identifying the names of specifications with the specifications they name):

Rule 7.

$$\frac{\langle[\textbf{auxwhile}\ \varphi\ \textbf{do}\ [sn]\ \textbf{od}],\ \text{last}\ \mathit{start\text{-}block}(S)\rangle \hookrightarrow \langle[\,],S'\rangle}{\langle[\textbf{while}\ \varphi\ \textbf{do}\ [sn]\ \textbf{od}],S\rangle \hookrightarrow \langle[\,],\mathit{add\text{-}to\text{-}SEStateOp}(S,\mathit{finish\text{-}block}(S'))\rangle}$$

Rule 8.

$$\vdash \langle[\textbf{auxwhile}\ \varphi\ \textbf{do}\ [sn]\ \textbf{od}],S\rangle \hookrightarrow \langle \textbf{if}\ \varphi\ \textbf{then}\ \text{cons}(sn,\textbf{auxwhile}\ \varphi\ \textbf{do}\ sn\ \textbf{od})\ \textbf{else}\ [\,],S\rangle.$$

4.11. Handling nondeterminism

As mentioned before (in Section 3.5), in symbolic execution the effects of nondeterminism should be captured by the *state* rather than by supplying different transitions that apply to the same configuration. As an example, consider the following rule describing the operational semantics of the command *IF* (as defined in Section 3.5). Since it is quite similar to the rule for **if-then-else** as given in Section 4.9, only the analogue of *ITE-merge-map* is given here, the other cases are completely analogous to **if-then-else**.

$$\mathit{IF\text{-}merge\text{-}map}: \mathit{SE\text{-}map} \times \mathit{SE\text{-}map} \times \mathit{PredS} \times \mathit{PredS} \to \mathit{SE\text{-}map}.$$

$$\mathit{IF\text{-}merge\text{-}map}(m_1,m_2,ps_1,ps_2) \triangleq$$

$$\left\{n \mapsto \begin{cases} \{\text{if } ps_1 \to ps_1' \,[\!]\, ps_2 \to ps_2' \text{ fi} \\ \quad \mid ps_i' \in m_i(n), i=1,2\} & \text{if } n \in \text{dom } m_1 \cap \text{dom } m_2 \\ \{\text{if } ps_1 \to ps_1' \,[\!]\, ps_2 \to \text{true fi} \\ \quad \mid ps_1' \in m_1(n)\} & \text{if } n \in \text{dom } m_1 - \text{dom } m_2 \\ \{\text{if } ps_1 \to \text{true} \,[\!]\, ps_2 \to ps_2' \text{ fi} \\ \quad \mid ps_2' \in m_2(n)\} & \text{if } n \in \text{dom } m_2 - \text{dom } m_1. \end{cases}\right\}$$

Rule 9. let $ps_i(S) = \varphi_i[n/\tilde{\sigma}(\mathit{cons}(\text{len } \mathit{SEQ}(S), \mathit{INDEX}(S)), n) \mid n: \mathit{Name}]$ in

$$\frac{\begin{array}{c} m_1, m_2: \mathit{SE\text{-}map} \\ ps_1(S) \vdash \langle[sn_1],S\rangle \hookrightarrow \langle[\,],\mathit{add\text{-}to\text{-}SEStateOp}(S,m_1)\rangle \\ ps_2(S) \vdash \langle[sn_2],S\rangle \hookrightarrow \langle[\,],\mathit{add\text{-}to\text{-}SEStateOp}(S,m_2)\rangle \end{array}}{\begin{array}{l} \langle[\textbf{if}\ \varphi_1 \to sn_1 \,[\!]\, \varphi_2 \to sn_2 \textbf{fi}],S\rangle \\ \quad \hookrightarrow \langle[\,],\mathit{add\text{-}to\text{-}SEStateOp}(S,\mathit{IF\text{-}merge\text{-}map}(m_1,m_2,ps_1(S),ps_2(S)))\rangle \end{array}}$$

Again, as for **if-then-else**, the consequent transition of this rule is used to transform the combinator **if-fi** into the connective if-fi. This connective is then dealt with in $Th(\mathscr{L})$ by rules such as those below. These simplification rules are slightly more complicated than those for if-then-else since they have to consider the different alternatives in parallel—in symbolic execution it is not enough to know if *one* of the guards is true.

$$\frac{\varphi_1 \wedge \varphi_2}{\text{if } \varphi_1 \rightarrow \psi_1 [\!] \varphi_2 \rightarrow \psi_2 \text{ fi} \Leftrightarrow \psi_1 \vee \psi_2}$$

$$\frac{\varphi_1 \wedge \neg\varphi_2}{\text{if } \varphi_1 \rightarrow \psi_1 [\!] \varphi_2 \rightarrow \psi_2 \text{ fi} \Leftrightarrow \psi_1}$$

$$\frac{\neg\varphi_1 \wedge \varphi_2}{\text{if } \varphi_1 \rightarrow \psi_1 [\!] \varphi_2 \rightarrow \psi_2 \text{ fi} \Leftrightarrow \psi_2}$$

5. Conclusions

This paper describes a formal definition of the denotational semantics of symbolic execution for a wide class of specification and programming languages, expressed in terms of the denotational semantics of the language being executed. Another way to view this would be as a language-generic notion of *correctness* for symbolic execution. This is believed to be the main contribution of the work described here. Until now, the concept of symbolic execution had not been defined on a general, semantic level, but only on the syntactic level for a number of specific programming languages (with the exception of GIST [2, 7] which is a specification and not a programming language).

In the next step, the paper introduced the notion of operational semantics of symbolic execution. Based on a concept of state adapted for this purpose, rules are given which describe the operational semantics as used for symbolic execution for a small language including block structures, variable declarations, operation definitions in terms of pre- and postconditions, deterministic and non-deterministic conditionals, and loops.

In a further step not discussed in this paper, the work described here provided the basis for the development of a *language-generic* tool for symbolic execution (called SYMBEX and described in detail in [24]) which can handle specifications as well as programs and is intended to support the user in validating specifications. It is based on (a particular version of) the operational semantics of the language being executed. The next steps in the development of this tool were the specification based on these ideas, and the implementation of a first prototype written in SMALLTALK-80 and integrated into the *mural* system.

This prototype interacted with the formal reasoning tools provided by *mural*. The operational semantics of a language were expressed as a collection of theories in *mural*, and symbolic execution was then based on that theory. Likewise, simplification was based on the theory of the language expressed in *mural*.

Work was not continued beyond this first prototype since there was not enough time left for the project, and it was decided to concentrate on other aspects of *mural*.

Therefore, not much can be said about the efficiency and complexity of SymbEx. Certainly, the prototype was very inefficient, but this was largely due to a naïve implementation. In particular the simplication and theorem proving algorithms provided by *mural* were since improved considerably.

As mentioned before, the work described here applies to state-based languages. It has to remain open for now whether and how these concepts can be extended to cover other classes of languages, such as algebraic or logic programming languages. As described in [24, Section 4.3], these languages certainly do not fit easily into the framework described.

Appendix A. A short summary of the notation used

The following is a (very) short summary of the notation used in this paper and which is not standard mathematical notation. It is essentially based on the VDM specification language (see [20] for more details on VDM).

VDM is based on the notions of states and state transformations called operations. It supports a number of primitive data types such as functions, finite sets, finite maps, and sequences, and also allows product types and defined types. A type definition takes the form

type-name = *type-expression*

where *type-name* is defined to be *type-expression*. Additionally one can provide a type invariant in order to define subtypes:

type-name = *type-expression*
where
inv-type-name(*t*) $\triangleq$ ...

Definitions of record types take the form

type-name::*field-name*1:*field-type*1
*field-name*2:*field-type*2
...:...

With each record type T we associate a constructor function *mk-T* that takes as arguments objects of the component types of T and returns the object of type T consisting of these components. Essentially, a record is a Cartesian product with names for its components and a constructor function associated with it.

Operations that access the state of a system are specified in the form

$OP\,(a\colon T_1)\; r\colon T_2$
ext rd $er\colon T_3$
wr $ew\colon T_4$
pre $\varphi(a, er, ew)$
post $\psi(a, \overleftarrow{ew}, r, er, ew)$.

Here a: T_1 denotes the arguments of the operation, r: T_2 denotes the result, er: T_3 the external or state variables to which the operation has got read access, and ew: T_3 the external or state variables to which the operation has got write access. φ denotes a precondition, ψ a postcondition. The semantics of *OP* is defined as: if the precondition holds before the operation, then the postcondition will hold afterwards. Since the operation may have changed the state, the postcondition refers both to the values before ($\overleftarrow{ew}$) and after (ew) the operation.

All the parameters in the definition of an operation are optional. In particular, this notation can be used for implicit function definitions, which do not have external read and write variables.

Maps $A \xrightarrow{m} B$ are functions from A to B with a finite domain. T^* is the type of finite sequences with elements from T.

$seq_1 \frown seq_2$ denotes the concatenation of two sequences seq_1 and seq_2, while $seq_1 \oplus e$ denotes appending an element e at the end of a sequence seq_1:

$$seq \oplus e \;\triangleq\; seq \frown [e].$$

Appendix B. Proofs of Lemma 3.4 and Theorem 4.2

Proof of Lemma 3.4. For all τ: *SEStateDen*

$$\begin{aligned}
&\mathit{yield}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1;\mathit{spec}_2]\!]\tau)\\
&= \lambda\sigma\cdot\{\sigma' \mid \exists \sigma\text{-}seq \in \mathit{SEQS}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1;\mathit{spec}_2]\!]\tau)\cdot\\
&\qquad \text{hd } \sigma\text{-}seq = \sigma \wedge \text{last } \sigma\text{-}seq = \sigma'\\
&\qquad \wedge \text{len } \sigma\text{-}seq = \mathit{LEN}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1;\mathit{spec}_2]\!]\tau)\}\\
&= \lambda\sigma\cdot\{\sigma_1 \mid \exists \sigma\text{-}seq\cdot \text{front } \sigma\text{-}seq \in \mathit{SEQS}(\tau)\\
&\qquad \wedge \text{len } \sigma\text{-}seq = \mathit{LEN}(\tau)+1 \wedge \text{last } \sigma\text{-}seq = \sigma_1 \wedge \text{hd } \sigma\text{-}seq = \sigma\\
&\qquad \wedge \mathcal{M}_{Spec}[\![\mathit{spec}_1;\mathit{spec}_2]\!](\text{last front } \sigma\text{-}seq, \text{last } \sigma\text{-}seq)\}\\
&= \lambda\sigma\cdot\{\sigma_1 \mid \exists \sigma\text{-}seq\cdot \text{front } \sigma\text{-}seq \in \mathit{SEQS}(\tau)\\
&\qquad \wedge \text{len } \sigma\text{-}seq = \mathit{LEN}(\tau)+1 \wedge \text{last } \sigma\text{-}seq = \sigma_1 \wedge \text{hd } \sigma\text{-}seq = \sigma\\
&\qquad \wedge \exists \sigma_2\cdot \mathcal{M}_{Spec}[\![\mathit{spec}_1]\!](\text{last front } \sigma\text{-}seq, \sigma_2)\\
&\qquad\qquad \wedge \mathcal{M}_{Spec}[\![\mathit{spec}_2]\!](\sigma_2, \text{last } \sigma\text{-}seq)\}\\
&= \lambda\sigma\cdot\{\sigma_1 \mid \exists \sigma\text{-}seq'\cdot \text{front front } \sigma\text{-}seq \in \mathit{SEQS}(\tau)\\
&\qquad \wedge \text{len } \sigma\text{-}seq' = \mathit{LEN}(\tau)+2 \wedge \text{last } \sigma\text{-}seq = \sigma_1 \wedge \text{hd } \sigma\text{-}seq = \sigma\\
&\qquad \wedge \mathcal{M}_{Spec}[\![\mathit{spec}_1]\!](\text{last front front } \sigma\text{-}seq', \text{last front } \sigma\text{-}seq')\\
&\qquad \wedge \mathcal{M}_{Spec}[\![\mathit{spec}_2]\!](\text{last front } \sigma\text{-}seq', \text{last } \sigma\text{-}seq')\}\\
&= \lambda\sigma\cdot\{\sigma_1 \mid \exists \sigma\text{-}seq'\cdot \text{front } \sigma\text{-}seq' \in \mathit{SEQS}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1]\!]\tau)\\
&\qquad \wedge \text{len } \sigma\text{-}seq' = \mathit{LEN}(\mathit{symbolic\text{-}ex}[\![\mathit{spec}_1]\!]\tau)+1\\
&\qquad \wedge \text{last } \sigma\text{-}seq' = \sigma_1 \wedge \text{hd } \sigma\text{-}seq' = \sigma\\
&\qquad \wedge \mathcal{M}_{Spec}[\![\mathit{spec}_2]\!](\text{last front } \sigma\text{-}seq', \text{last } \sigma\text{-}seq')\}
\end{aligned}$$

$$= \lambda\sigma\cdot\{\sigma_1 \mid \exists \sigma\text{-}seq' \cdot \sigma\text{-}seq' \in SEQS(symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]\tau)$$

$$\wedge \text{ len } \sigma\text{-}seq' = LEN(symbolic\text{-}ex\text{-}s[\![[spec_1, spec_2]]\!]\tau)$$

$$\wedge \text{ last } \sigma\text{-}seq' = \sigma_1 \wedge \text{hd } \sigma\text{-}seq' = \sigma\}$$

$$= yield(symbolic\text{-}ex\text{-}s[\![spec_1, spec_2]\!]\tau). \quad \square$$

Proof of Theorem 4.2. We show, by induction on len $SEQ(S)$, that

$$\forall S: SEStateOp \cdot inv\text{-}SEStateDen(\mathcal{M}_{SEStateOp}[\![S]\!]).$$

Base case: len $SEQ(S) = 1$. We have to show that

$$\text{let } set = \{[\sigma] \mid satisfies\text{-}all\text{-}restrictions([\sigma], S, 1)\} \text{ in}$$

$$\forall \sigma\text{-}seq \in set \cdot \text{len } \sigma\text{-}seq \leq 1$$

$$\wedge \forall \sigma\text{-}seq_1, \sigma\text{-}seq_2 \in set \cdot \forall \sigma\text{-}seq: (\Sigma_\perp)^* \cdot \sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq$$

$$\Rightarrow \sigma\text{-}seq = [\,]$$

which is trivially true.

Induction step. Now assume that, for some S,

$$inv\text{-}SEStateDen(\mathcal{M}_{SEStateOp}[\![S]\!])$$

holds and consider

$$S' = mk\text{-}SEStateOp(SEQ(S) \oplus e, DEPTH(S)), \quad \text{for some } e.$$

We first have to show that

$$\forall \sigma\text{-}seq \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!] \cdot \text{len } \sigma\text{-}seq \leq LEN(\mathcal{M}_{SEStateOp}[\![S']\!]).$$

This follows immediately from the definition of $\mathcal{M}_{SEStateOp}[\![S']\!]$.

For the second part of the proof assume that

$$\sigma\text{-}seq_1, \sigma\text{-}seq_2 \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!]),$$

and that for some $\sigma\text{-}seq: (\Sigma_\perp)^*$

$$\sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \sigma\text{-}seq.$$

We distinguish three cases:

Case 1: len $\sigma\text{-}seq_2 = LEN(\mathcal{M}_{SEStateOp}[\![S']\!])$. Then $\sigma\text{-}seq = [\,]$ follows immediately, since there are no sequences in $SEQS(\mathcal{M}_{SEStateOp}[\![S']\!])$ that are longer than $LEN(\mathcal{M}_{SEStateOp}[\![S']\!])$.

Case 2: len $\sigma\text{-}seq_2 = LEN(\mathcal{M}_{SEStateOp}[\![S']\!]) - 1$. Then, by definition of $\mathcal{M}_{SEStateOp}$,

$$\neg\exists \sigma: \Sigma_\perp \cdot satisfies\text{-}all\text{-}restrictions(\sigma\text{-}seq_2 \oplus \sigma, S, \text{len } S + 1),$$

therefore $\sigma\text{-}seq$ cannot have length 1. It cannot be longer either, since then $\sigma\text{-}seq_1$ would be too long to be in $SEQS(\mathcal{M}_{SEStateOp}[\![S \oplus e]\!])$. This only leaves $\sigma\text{-}seq = [\,]$, as required.

Case 3: $\text{len}\ \sigma\text{-}seq_2 < LEN(\mathcal{M}_{SEStateOp}[\![S']\!] - 1$. In this case

$$\sigma\text{-}seq_2 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!]),$$

and we have to distinguish two further cases:

(3.1) $\sigma\text{-}seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])$. Then $\sigma\text{-}seq = [\,]$ follows by induction hypothesis.

(3.2) $\sigma\text{-}seq_1 \notin SEQS(\mathcal{M}_{SEStateOp}[\![S]\!])$. In this case, since

$$\sigma\text{-}seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S']\!],$$

we have

$$\text{front}\ \sigma\text{-}seq_1 \in SEQS(\mathcal{M}_{SEStateOp}[\![S]\!]) \wedge \text{len}\ \sigma\text{-}seq_1 = LEN(\mathcal{M}_{SEStateOp}[\![S]\!]) + 1$$

Now assume $\sigma\text{-}seq \neq [\,]$. Then

$$\text{front}\ \sigma\text{-}seq_1 = \sigma\text{-}seq_2 \frown \text{front}\ \sigma\text{-}seq,$$

and we can apply the induction hypothesis to get $\text{front}\ \sigma\text{-}seq = [\,]$, or $\text{len}\ \sigma\text{-}seq = 1$. Then we get

$$\begin{aligned}\text{len}\ \sigma\text{-}seq_1 &= \text{len}\ \sigma\text{-}seq_2 + \text{len}\ \sigma\text{-}seq \\ &< (LEN(\mathcal{M}_{SEStateOp}[\![S']\!] - 1) + 1 \\ &= \text{len}\ \sigma\text{-}seq_1\end{aligned}$$

which shows that our assumption $\sigma\text{-}seq \neq [\,]$ must have been false. □

Acknowledgement

Most of the work reported here has been done as part of the IPSE 2.5 project, the aim of which was to build an "Integrated Project Support Environment".

Major innovative aspects of this project were genericity with respect to languages and development methods supported, and an emphasis on support for formal methods. The part of IPSE 2.5 dealing with support for formal methods was called *mural*. See [18, 19] for more information on *mural*.

IPSE 2.5 was developed under an Alvey/SERC project by a consortium of the following member organisations: STC plc, ICL plc, University of Manchester, Dowty Defence & Air Systems Ltd., Plessey Research Roke Manor Ltd., SERC Rutherford Appleton Laboratory, British Gas plc.

At the same time, the work described here formed a major part of the author's Ph.D. thesis [24]. My thanks for many fruitful discussions go to the other members of the IPSE 2.5 group here in Manchester and at Rutherford Labs, in particular to Cliff Jones, my supervisor, and also to Roy Simpson and John Fitzgerald.

Minor parts of this paper are taken from the author's contribution to [18].

References

[1] K.R. Apt, Ten years of Hoare's logic: a survey—part I, *ACM Trans. Programming Languages Syst.* 4 (1981) 431-483.

[2] R.M. Balzer, N.M. Goldman and D.S. Wile, Operational specification as the basis for rapid prototyping, *ACM SIGSOFT Software Engrg. Notes* 5 (1982) 3-16.

[3] H. Barringer, J.H. Cheng and C.B. Jones, A logic covering undefinedness in program proofs, *Acta Inform.* 21 (1984) 251-269.

[4] D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation*, Gl. Avernaes, Ebberup, Denmark (1987).

[5] M. Broy, Extensional behaviour of concurrent, nondeterministic, communicating systems, in: M. Broy, ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer, Berlin, 1985).

[6] L.A. Clarke and D.J. Richardson, Symbolic evaluation—an aid to testing and verification, in: H.-L. Hausen, ed., *Softeware Validation—Proceedings Symposium on Software Validation, Darmstadt, W. Germany* (North-Holland, Amsterdam, 1984) 141-166.

[7] D. Cohen, W. Swartout and R. Balzer, Using symbolic execution to characterize behavior, *ACM SIGSOFT Software Engrg. Notes* 5 (1982) 25-32.

[8] D. Coleman and J.W. Hughes, The clean termination of Pascal programs, *Acta Inform.* 11 (1979) 195-210.

[9] P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: *Proceedings ACM Symposium on Principles of Programming Languages*, Los Angeles, CA (1977) 238-252.

[10] P. Cousot and R. Cousot, Static determination of dynamic properties of generalized type unions, in: *Proceedings ACM Conference on Language Design for Reliable Software*, Raleigh, SC (1977) 77-94.

[11] R.E. Davis, *Truth, Deduction and Computation. Logic and Semantics for Computer Science* (Computer Science Press, Rockville, MD, 1989).

[12] J.W. de Bakker and J.I. Zucker, Processes and the denotational semantics of concurrency. *Inform. Control* 1/2 (1982) 70-120.

[13] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[14] S.L. Hantler and J.C. King, An introduction to proving the correctness of programs, in: *Proceedings ACM Computer Science Conference* (1976) 331-353.

[15] E.C.R. Hehner, Predicative programming. *Comm. ACM* 27 (1984) 134-151.

[16] C.A.R. Hoare, An axiomatic basis for computer programming, *Comm. ACM* 12 (1969) 576-583.

[17] D.C. Ince, The automatic generation of test data, *Computer J.* 1 (1987) 63-69.

[18] C.B. Jones, K.D. Jones, P.A. Lindsay and R.C. Moore, eds., *mural—A Formal Development Support System* (Springer, Berlin, 1991).

[19] C.B. Jones and P.A. Lindsay, A support system for formal reasoning: requirements and status, in: R. Bloomfield, L. Marshall and R. Jones, eds., *VDM'88—The Way Ahead. Proceedings 2nd VDM-Europe Symposium,Dublin, Ireland*, Lecture Notes Computer Science 328 (Springer, Berlin, 1988) 139-152.

[20] C.B. Jones, *Systematic Software Development Using VDM* (Prenctice-Hall, Englewood Cliffs, NJ, 2nd ed., 1990).

[21] R.A. Kemmerer, Testing formal specifications to detect design errors, *IEEE Trans. Software Engrg.* 11 (1985) 32-43.

[22] J.C. King, Symbolic execution and program testing, *Comm. ACM* 19 (1976) 385-394.

[23] R. Kneuper, Symbolic execution of specifications: user interface and scenarios, Tech. Report. Department of Computer Science, University of Manchester (1987).

[24] R. Kneuper, Symbolic execution as a tool for validation of specifications, Ph.D. Thesis, Department of Computer Science, University of Manchester (1989).
[25] A. Mycroft, Abstract interpretation and optimising transformations for applicative programs, Ph.D. Thesis, University of Edinburgh (1981).
[26] G.D. Plotkin, A structural approach to operational semantics, Tech. Report, Computer Science Department, Aarhus University, Denmark (1981).
[27] D.A. Schmidt, *Denotational Semantics—a Methodology for Language Development* (Allyn and Bacon, Newton, MA, 1986).
[28] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).
[29] D. Talbot and R.W. Witty, Alvey programme for software engineering, Alvey Directorate (1983).
[30] W. Zimmermann, How to mechanize complexity analysis (Submitted).