Symbolic execution formally explained

Boer, F.S. de; Bonsangue, M.M.

https://doi.org/10.1007/s00165-020-00527-y

Formal Aspects of Computing (2021) 33: 617–636

Formal Aspects of Computing

Symbolic execution formally explained

Frank S. de Boer[1,2] and Marcello Bonsangue[2]

[1]Centrum Wiskunde and Informatica (CWI), Leiden, The Netherlands
[2]Leiden Institute Of Advanced Computer Science (LIACS), Leiden University, Leiden, The Netherlands

Abstract. In this paper, we provide a formal explanation of symbolic execution in terms of a symbolic transition system and prove its correctness and completeness with respect to an operational semantics which models the execution on concrete values. We first introduce a formal model for a basic programming language with a statically fixed number of programming variables. This model is extended to a programming language with recursive procedures which are called by a call-by-value parameter mechanism. Finally, we present a more general formal framework for proving the soundness and completeness of the symbolic execution of a basic object-oriented language which features dynamically allocated variables.

Keywords: Formal methods, Testing, Symbolic execution, Object-orientation

1. Introduction

Symbolic execution [Kin76] plays a crucial role in modern testing techniques, debugging, and automated program analysis. In particular, it is used for generating test cases [AAGR14, BCD+18]. Intuitively, its success is mainly due because one symbolic execution abstracts away a possibly infinite set of concrete executions, all having in common a similar execution path.

Although symbolic execution techniques have improved enormously in the last few years, not much effort has been spent on its formal justification. In fact, the symbolic execution community has concentrated most of the effort on effectiveness (improvement in speed-up) and significance (improvement in code coverage) and payed little attention to correctness so far [BCD+18].

Further, there exists a plethora of different techniques for one of the major problems in symbolic execution, namely the presence of dynamically allocated program variables, e.g., describing arrays and (object-oriented) pointer structures ("heaps"). For example, in [XMSN05] a heap is modeled as a graph, with nodes drawn from a set of objects and updated lazily, whereas [BDP15] introduces a constraint language for the specification of invariant properties of heap structures. In [DLR06] the symbolic state is extended with a heap configuration used to maintain objects which are initialized only when they are first accessed during execution. In the presence of aliasing, the uncertainty on the possible values of a symbolic pointer is treated either by forking the symbolic state or refining the generated path condition into several ones [TS14]. In [KC19] the expensive forking is avoided by using a segmented memory model. Powerful symbolic execution tools [CDE08, CGP+08, EGL09] handling arrays exploit various code pre-processing techniques, though formal correctness of the theory behind these tools is acknowledged as a potential problem that might limit the validity of the internal engine, and is validated only experimentally by testing [PMZC17]. The KeY theorem prover [ABB+16] supports symbolic execution of Java programs which is defined in terms of the underlying dynamic logic and which uses an explicit representation of the heap. In all of the above work no explicit formal account of the underlying model of the symbolic execution, and its correctness, is presented.

Correspondence to: Frank S. de Boer, e-mail: f.s.de.boer@cwi.nl

The main contribution of this paper is a formal explanation of symbolic execution in terms of a symbolic transition system and a general definition of its correctness and completeness with respect to an operational semantics which models the actual execution on concrete values. It extends the work presented in [dBB19] first of all by detailed proofs of correctness and completeness. It further describes two different approaches to symbolic execution, focusing on the generation of so-called *path conditions*. These are symbolic conditions on the initial states which ensure the concrete execution of the program along a given path, i.e., a particular "flow" of control described by the program.

In this paper, following [dBB19], we first formalize the standard approach to symbolic execution which consists of generating a path condition *on-the-fly* by maintaining during the symbolic execution a symbolic representation of the concrete program state, i.e., the assignment of values to program variables. This approach gives rise to what is usually called *forward* symbolic execution.

In this paper, we further introduce a new, more fundamental approach to the symbolic generation of path conditions. This approach is based on the application of a *weakest precondition* calculus on *symbolic execution traces* which are generated by a static *unfolding* of the program. We describe this new approach, which we claim is applicable to *any* programming language, in terms of a basic object-oriented language.

In [dBB19] we briefly sketched how to extend the on-the-fly generation of a path condition to object-oriented languages. The forward symbolic execution of object-oriented programs however is complicated by the symbolic description of the concrete program state in terms of the *unbounded* number of *heap* (or *navigation*) expressions of the programming language. This requires a complicated treatment of *aliasing* between these expressions in the symbolic description of heap assignments. We show that this new approach overcomes the above complications of the standard forward symbolic execution of object-oriented languages.

In [SAB10] a forward symbolic execution model is introduced as an extension of the operational semantics of a low-level assembly like language which adds to the configuration the accumulated path condition. The operational semantics itself is extended by the inclusion of symbolic values, that is, partially evaluated expressions, in the set of concrete values. In [SAB10] no formal justification of this hybrid approach is given, in fact, this hybrid approach does not allow for a formal statement of a (simulation) relation between the concrete and the symbolic semantics. The only other approach to a formal modeling and justification of symbolic execution, we are aware of, is the work presented in [LRA17]. A major difference with our approach is that in [LRA17] symbolic execution is defined in terms of a general logic (called "Reachability Logic") for the description of transition systems which abstracts from the specific characteristics of the programming language. A symbolic execution then consists basically of a sequence of logical specifications of the consecutive transitions. On the other hand, a model of the logic defines a concrete transition system. Thus correctness basically follows from the semantics of the logic. In our approach we both model symbolic execution and the concrete semantics (of any language) independently as transition systems. However, in both cases the transitions are directly defined in terms of the program to be executed. This allows to address the specific characteristics of the programming language (like dynamically allocated variables) still in a general manner. In [LRA17], however, these specific characteristics (like arrays) need to be imported in the general framework by corresponding logical theories which require an additional justification.

Detailed plan of the paper. In Section 2 we introduce a formal model of symbolic execution for a basic programming language with a statically fixed number of programming variables. A configuration of the symbolic transition system consists of the program statement to be executed, a substitution, and a path condition. Correctness then states that for every reachable symbolic configuration and state which satisfies the path condition, there exists a corresponding concrete execution. Conversely, completeness states that for every concrete execution there exists a corresponding symbolic configuration such that the initial state of the concrete execution satisfies the path condition and its final state can be obtained as a composition of the initial state and the generated substitution. In Subsection 2.1 we describe an extension of the basic theory with arrays symbolically modeled as mathematical functions.

In Section 3, we extend the basic theory of symbolic execution to a programming language with recursive procedures which are called by a call-by-value parameter mechanism. This extension requires a formal treatment of local variables stored on the stack of procedure calls.

In Section 4 we introduce a different, and more fundamental, approach to symbolic execution and its application to a basic object-oriented language. We conclude this section with a brief discussion of how to symbolically execute this language in the on-the-fly generation of path conditions, using the symbolic interpretation of fields as arrays.

In the final Section 5 we conclude with a brief discussion how multi-threading, and concurrent objects can be treated, showing the generality of our theory of symbolic execution.

2. Basic symbolic execution

We assume a set of Var of program variables $x, y, u, \ldots$, and a set Ops of operations $op, \ldots$. We abstract from typing information, but we assume Ops includes standard Boolean operators. The set $Expr$ of *programming expressions* e is defined by the following grammar.

$$e ::= x \mid op(e_1, \ldots, e_n)$$

where $x \in Var$ and $op \in Ops$. Expressions e consist of program variables x and operators op applied to expressions (as a special case, we include values v as nullary operators).

Statements S of the basic programming language are then defined by the grammar:

$$\begin{array}{lll} S ::= & x := e & \text{assignment} \\ \mid & S;\ S & \text{sequential composition} \\ \mid & \mathit{if}\ b\ \{S_1\}\{S_2\} & \text{choice} \\ \mid & \mathit{while}\ b\ \{S\} & \text{iteration} \end{array}$$

This basic language thus consists of (side-effect free) assignments $x := e$, and the usual control structures of sequential composition, choice, and iteration. In the latter two constructs b denotes a Boolean expression. We assume associativity of the sequential composition operator, and use the "empty" statement ϵ, e.g., $x := x$, which acts as the neutral element with respect to sequential composition. It will be used to denote termination. As a consequence, every statement is equivalent to a statement of the form $A;\ S$, where A is either an assignment, a choice or an iteration construct. Finally, we work only with programs that are well-typed, so operators are recursively applied to the correct number of correctly typed sub-expressions.

A *substitution* σ is a function $Var \to Expr$ which assigns to each variable an expression. By $e\sigma$ we denote the application of the substitution σ to the expression e, defined inductively by

$$\begin{aligned} x\sigma &= \sigma(x) \\ op(e_1, \ldots, e_n)\sigma &= op(e_1\sigma, \ldots, e_n\sigma) \end{aligned}$$

A *symbolic configuration* is a triple $\langle S, \sigma, \phi \rangle$ where S denotes the statement to be executed, σ denotes the current substitution, and the Boolean condition ϕ denotes the path condition. Next we describe a transition system for the symbolic execution of our basic programming language defined above.

Symbolic assignment

- $\langle x := e;\ S, \sigma, \phi \rangle \to \langle S, \sigma[x := e\sigma], \phi \rangle$

where $\sigma[x := e]$ is the update of a substitution defined by $\sigma[x := e](y) = \sigma(y)$ if x and y are syntactically distinct variables, and $\sigma[x := e](x) = e$ otherwise.

Symbolic choice

- $\langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S, \sigma, \phi \rangle \to \langle S_1;\ S, \sigma, \phi \wedge b\sigma \rangle$
- $\langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S, \sigma, \phi \rangle \to \langle S_2;\ S, \sigma, \phi \wedge \neg b\sigma \rangle$

Symbolic iteration

- $\langle \mathit{while}\ b\ \{S\};\ S', \sigma, \phi \rangle \to \langle S;\ \mathit{while}\ b\ \{S\};\ S', \sigma, \phi \wedge b\sigma \rangle$
- $\langle \mathit{while}\ b\ \{S\};\ S', \sigma, \phi \rangle \to \langle S', \sigma, \phi \wedge \neg b\sigma \rangle$

We illustrate the symbolic semantics by the following simple example of the symbolic execution of a while statement.

We formalize and prove correctness with respect to a concrete semantics. A *valuation* V is a function $Var \to Val$, where *Val* is a set of *values* (including the Boolean values *true* and *false*). By $V(e)$ we denote the value of the expression e with respect to the valuation V, defined inductively by $V(op(e_1, \ldots, e_n)) = \overline{op}(V(e_1), \ldots, V(e_n))$ where $\overline{op} : Val^n \to Val$ denotes the interpretation of the operation *op* as provided by the implicitly assumed underlying model. For technical convenience we assume that the interpretation of the operations are total functions, thus avoiding errors as generated by division by zero, e.g., we stipulate that x div $0 = 0$ (using the infix notation). A valuation obtained as a composition $V \circ \sigma$ of a valuation V and a substitution σ is defined as usual: $(V \circ \sigma)(x) = V(\sigma(x))$. In the sequel we omit the parentheses and write $V \circ \sigma(e)$ for the application of a valuation $V \circ \sigma$ to the expression e (as defined above). We have the following basic substitution lemma which states that evaluating an expression e in a composition $V \circ \sigma$, as defined above, gives the same result as evaluating in V the expression $e\sigma$ which results from first applying the substitution.

Lemma 2.1 (Substitution). $V \circ \sigma(e) = V(e\sigma)$.

Proof The proof of the lemma proceeds by induction on e. We have the following main case:

$$\begin{array}{lll} V \circ \sigma(op(e_1, \ldots, e_n)) = \overline{op}(V \circ \sigma(e_1), \ldots, V \circ \sigma(e_n)) & \text{(semantics expressions)} \\ = \overline{op}(V(e_1\sigma), \ldots, V((e_n\sigma)) & \text{(induction hypothesis)} \\ = V(op(e_1\sigma, \ldots, e_n\sigma)) & \text{(semantics expressions)} \\ = V(op(e_1, \ldots, e_n)\sigma) & \text{(substitution application)} \end{array}$$

□

The concrete semantics of our basic programming language is defined in terms of transitions $\langle S, V\rangle \to \langle S', V'\rangle$. The definition of this transition system is standard:

Concrete assignment

- $\langle x := e;\ S, V\rangle \to \langle S, V[x := V(e)]\rangle$

where the valuation $V[x := v]$ denotes the state update. Such a valuation is defined by $V[x := v](y) = V(y)$ if x and y are syntactically distinct variables, and $V[x := v](x) = v$ otherwise. Note that because assignments $x := e$ are assumed *side-effect free*, the valuation $V[x := V(e)]$ affects only the value of the variable x. In other words, we can define the semantics of side-effect free assignments in terms of such updates because of the absence of *aliasing*, i.e., absence of two distinct variables x and y which intuitively refer to the same memory location.

Concrete choice

- $\langle$*if* b $\{S_1\}\{S_2\};\ S, V\rangle \to \langle S_1;\ S, V\rangle$ if $V(b)$ is true
- $\langle$*if* b $\{S_1\}\{S_2\};\ S, V\rangle \to \langle S_2;\ S, V\rangle$ if $V(b)$ is false

Concrete iteration

- $\langle$*while* b $\{S\};\ S', V\rangle \to \langle S;$ *while* b $\{S\};\ S', V\rangle$ if $V(b)$ is true
- $\langle$*while* b $\{S\};\ S', V\rangle \to \langle S', V\rangle$ if $V(b)$ is false

From the above substitution lemma we derive the following corollary.

Corollary 2.2 (Soundness assignment). For $V' = V \circ \sigma$ we have that $V \circ (\sigma[x := e\sigma]) = V'[x := V'(e)]$.

Proof We treat the main case:

$$\begin{array}{lll} (V \circ (\sigma[x := e\sigma]))(x) = V(\sigma[x := e\sigma](x)) & \text{(def. } \circ) \\ = V(e\sigma) & \text{(def. } \sigma[x := e\sigma]) \\ = V \circ \sigma(e) & \text{(substitution lemma)} \\ = V'(e) & (V' = V \circ \sigma) \\ = V'[x := V'(e)](x) & \text{(def. } V'[x := V'(e)]) \end{array}$$

□

Let id be the identity substitution, i.e., $id(x) = x$, for every variable x. We have the following main correctness theorem.

Theorem 2.3 (Correctness). If $\langle S, id, true\rangle \rightarrow^* \langle S', \sigma, \phi\rangle$[1] and $V(\phi) = \text{true}$ then

$$\langle S,\ V\rangle \rightarrow^* \langle S',\ V \circ \sigma\rangle$$

Proof Induction on the length of $\langle S, id, true\rangle \rightarrow^* \langle S', \sigma, \phi\rangle$ and a case analysis of the last execution step. We consider the following cases.

First, we consider the case of an assignment as the last execution step:

$$\langle S, id, true\rangle \rightarrow^* \langle x := e;\ S', \sigma, \phi\rangle \rightarrow \langle S', \sigma[x := e\sigma], \phi\rangle$$

Induction hypothesis (note that $V(\phi) = \text{true}$):

$$\langle S,\ V\rangle \rightarrow^* \langle x := e;\ S',\ V \circ \sigma\rangle$$

Let $V' = V \circ \sigma$. By the concrete semantics we have

$$\langle S,\ V\rangle \rightarrow^* \langle x := e;\ S',\ V'\rangle \rightarrow \langle S',\ V'[x := V'(e)]\rangle$$

By the above Corollary 2.2 it then suffices to observe that $V \circ (\sigma[x := e\sigma]) = V'[x := V'(e)]$.

Next we consider the selection of the then-branch of a choice construct:

$$\langle S, id, true\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S, \sigma, \phi\rangle \rightarrow \langle S_1;\ S, \sigma, \phi \wedge b\sigma\rangle$$

We have that $V(\phi \wedge b\sigma) = \text{true}$ implies $V(\phi) = \text{true}$, so by the induction hypothesis we obtain the concrete computation

$$\langle S,\ V\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S,\ V \circ \sigma\rangle$$

By the above substitution lemma $V \circ \sigma(b) = V(b\sigma) = \text{true}$, we derive

$$\langle S,\ V\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S,\ V \circ \sigma\rangle \rightarrow \langle S_1;\ S,\ V \circ \sigma\rangle$$

All other cases are treated similarly. □

Theorem 2.3 guarantees that all possible inputs satisfying a path condition lead to a concrete state with variables conform to the substitution of the corresponding symbolic configuration. Correctness, however, is about coverage [LRA17], meaning that *satisfiable* symbolic execution paths can be simulated by concrete executions. The converse of correctness is completeness and is about precision [LRA17]: every concrete execution can be simulated by a symbolic one.

Theorem 2.4 (Completeness). For any concrete computation $\langle S,\ V_0\rangle \rightarrow^* \langle S',\ V\rangle$ there exists a symbolic computation

$$\langle S, id, true\rangle \rightarrow^* \langle S', \sigma, \phi\rangle$$

for some path condition ϕ and a substitution σ such that $V_0(\phi) = \text{true}$ and $V = V_0 \circ \sigma$.

Proof As above, the proof of this theorem proceeds by induction on the length of the concrete computation and a case analysis of the last concrete execution step. We consider the following cases.

First, we consider the case of an assignment as the last execution step:

$$\langle S,\ V_0\rangle \rightarrow^* \langle x := e;\ S',\ V\rangle \rightarrow \langle S',\ V[x := V(e)]\rangle$$

By the induction hypothesis there exists a symbolic computation

$$\langle S, id, true\rangle \rightarrow^* \langle x := e;\ S', \sigma, \phi\rangle$$

for some path condition ϕ and a substitution σ such that $V_0(\phi) = \text{true}$ and $V = V_0 \circ \sigma$. By the symbolic semantics we have that

$$\langle S, id, true\rangle \rightarrow^* \langle x := e;\ S', \sigma, \phi\rangle \rightarrow \langle S', \sigma[x := e\sigma], \phi\rangle$$

By the above Corollary 2.2 again it then suffices to observe that $V_0 \circ (\sigma[x := e\sigma]) = V[x := V(e)]$.

[1] For any transition relation $\rightarrow$ its reflexive, transitive closure is denoted by $\rightarrow^*$

Next we consider the case when the Boolean guard of a choice construct evaluates to true:

$$\langle S,\ V_0\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S,\ V\rangle \rightarrow \langle S_1;\ S,\ V\rangle$$

where $V(b) = \text{true}$. By the induction hypothesis there exists a symbolic computation

$$\langle S, \mathit{id}, \mathit{true}\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S, \sigma, \phi\rangle$$

for some path condition ϕ and a substitution σ such that $V_0(\phi) = \text{true}$ and $V = V_0 \circ \sigma$. By the symbolic semantics we have that

$$\langle S, \mathit{id}, \mathit{true}\rangle \rightarrow^* \langle \mathit{if}\ b\ \{S_1\}\{S_2\};\ S, \sigma, \phi\rangle \rightarrow \langle S_1;\ S, \sigma, \phi \wedge b\sigma\rangle$$

We then can conclude this case by again an application of the above substitution lemma from which we derive that $V_0(b\sigma) = V_0 \circ \sigma(b) = V(b) = \text{true}$.

All other cases are treated similarly. □

The above correctness and completeness theorems establish a correspondence between *reachable* symbolic and concrete states. It is straightforward to generalize these theorems to computations represented by sequences of (symbolic or concrete) states.

2.1. Extension to arrays

In this subsection we briefly discuss the symbolic execution of our basic programming language extended with arrays. For notational convenience we restrict to one-dimensional arrays. Following [AdBO09] and [Gri81] we view such arrays semantically as (mathematical) functions, i.e., an array variable has a type $T \rightarrow T'$, where the basic types T and T' denote the type of its domain and co-domain, respectively. Thus the domain of an array can be unbounded (below we discuss the extension of our theory to bounded arrays). Given an array variable a of type $T \rightarrow T'$, the expression $a[e]$ of type T' denotes the result of applying the function associated with a to the value of e, where the expression e is of type T. We extend the basic language with expressions $a[e]$ and assignments $a[e] := e'$, following the approach initially proposed in [Gri81] to the Hoare logic of assignments to sub-scripted variables: symbolically an assignment $a[e] := e'$ is viewed as an assignment $a := (a[e] := e')$, where the *expression* $(a[e] := e')$ denotes an *update* of the array a defined by

$$(a[e] := e')[e''] = \mathit{if}\ e = e''\ \mathit{then}\ e'\ \mathit{else}\ a[e'']\ \mathit{fi}$$

(assuming ternary operators for the description of conditional expressions). Formally, *array expressions* are defined inductively as follows: every array variable is an array expression, if a is an array expression of type $T \rightarrow T'$ so is $(a[e] := e')$, where e and e' are of type T and T', respectively.

A substitution σ representing a concrete state then assigns to all the program variables a corresponding expression. For any array variable a, $\sigma(a)$ thus denotes an array expression. Given this symbolic interpretation of arrays as mathematical functions it is straightforward to extend the symbolic execution of our basic programming language to arrays, and generalize the above correctness and completeness theorems.

There are various ways to symbolically execute bounded arrays (see for example [FLP17]). One possible way to extend our approach to bounded arrays (of type $\mathbb{N} \rightarrow T$, for some T, where $\mathbb{N}$ denotes the type of the natural numbers) consists of adding the expression $\mid a \mid$ which denotes the length of the array a. The symbolic execution of (initially) setting the bound of an array, described by the statement $\mid a \mid = e$, then updates the path condition with the information $\mid a \mid = e\sigma$, where σ is the current substitution. We describe the *absence* of an *array-out-of-bound* error by a predicate $\delta(e)$ defined inductively by

$$\begin{array}{ll} \delta(x) & = \mathit{true} \\ \delta(a[e]) & = 0 \leq e \leq \mid a \mid \wedge \delta(e) \\ \delta(\mathit{op}(e_1, \ldots, e_n)) & = \delta(e_1) \wedge \cdots \wedge \delta(e_n) \end{array}$$

We indicate the *occurrence* of an array-out-of-bound error by a statement *array-out-of-bound*. This statement then can be further evaluated in the context of error-handling constructs. The symbolic execution of such constructs is out of scope of this paper though.

It is then straightforward to update the above symbolic transition system to account for array-out-of-bound errors. For example, for the symbolic execiution of an assignment $x := e$ we have the following two transitions:

- $\langle x := e;\ S, \sigma, \phi\rangle \rightarrow \langle S, \sigma[x := e\sigma], \phi \wedge \delta(e\sigma)\rangle$
- $\langle x := e;\ S, \sigma, \phi\rangle \rightarrow \langle \textit{array-out-of-bound}, \sigma, \phi \wedge \neg\delta(e\sigma)\rangle$

As another example, for the symbolic execution of the choice construct we have four transitions, of which we present the following two:

- $\langle \textit{if } b\ \{S_1\}\{S_2\};\ S, \sigma, \phi\rangle \rightarrow \langle S_1;\ S, \sigma, \phi \wedge \delta(b\sigma) \wedge b\sigma\rangle$
- $\langle \textit{if } b\ \{S_1\}\{S_2\};\ S, \sigma, \phi\rangle \rightarrow \langle \textit{array-out-of-bound}, \sigma, \phi \wedge \neg\delta(b\sigma)\rangle$

For an array assignment $a[e] := e'$ we have the following symbolic transitons.

- $\langle a[e] := e';\ S, \sigma, \phi\rangle \rightarrow \langle S, \sigma[a := (a[e\sigma] := e'\sigma)], \phi \wedge \delta(a[e\sigma)] \wedge \delta(e'\sigma)\rangle$
- $\langle a[e] := e';\ S, \sigma, \phi\rangle \rightarrow \langle \textit{array-out-of-bound}, \sigma, \phi \wedge \neg(\delta(a[e\sigma]) \wedge \delta(e'\sigma))\rangle$

Note that, as defined above, $\delta(a[e\sigma])$ equals $0 \leq e\sigma \leq \mid a \mid \wedge\delta(e\sigma)$. We do not need to apply the current substitution σ to a in the expression $\mid a \mid$ because the bound of an array is not affected by any of its assignments.

We conclude this discussion with the observation that alternatively we can include instead of the array variables the expressions $a[e]$ themselves in the domain of a substitution representing the concrete state. This however gives rise to *aliasing* between an *unbounded* number of such expressions: to execute symbolically an assignment $a[e] := e'$ we need to update in the given substitution *all* possible aliases of $a[e]$, namely all those expressions $a[e'']$ such that the value of e'' *after* the assignment $a[e] := e'$ equals that of e. We therefore recommend the above approach.

3. Recursion

In this section we extend our basic programming language with procedures. We assume a finite set of Var of program variables $x, y, u, \ldots$ to be partitioned in global variables $GVar$ and local variables $LVar$, without name clashes between them. Global variables are visible within the entire program while local variables are used as formal parameters of the procedure declarations, and their scope lie within the procedure body itself. We denote by $\bar{x}$ a tuple of variables.

The set $Expr$ of *programming expressions* e is defined as in the previous section, except that (Boolean) operators involve now both local and global variables. A *program* consists of set of procedure declarations of the form $P(\bar{u}) :: S$ and a main statement S'. Every procedure name P had a unique declaration. For simplicity we assume here that $\bar{u}$ consists of all local variables $LVar$. Statements of the basic programming language with procedures are defined by extending the grammar of the previous section with procedure calls:

$S ::=$	$x := e$	(global and local) assignment
$\mid$	$P(\bar{e})$	procedure call
$\mid$	$S;\ S$	sequential composition
$\mid$	$\textit{if } b\ \{S_1\}\{S_2\}$	choice
$\mid$	$\textit{while } b\ \{S\}$	iteration

Beside (side-effect free) assignments to global and local variables (here $x \in Var$), sequential composition, choice, and iteration, we have procedure calls $P(\bar{e})$, assuming a call by value parameter passing mechanism. Again, we consider only programs that are well-typed, meaning, among other things, that the length of $\bar{e}$ in the call $P(\bar{e})$ is the same as that $\bar{u}$ in the declaration $P(\bar{u}) :: S$. For technical convenience we do not consider the introduction of local variables by block statements here, that could be easily modelled using procedures.

A *symbolic configuration* is of the form $\langle \Delta, \sigma, \phi\rangle$, where

- Δ denotes a stack of *closures* of the form (τ, S), where $\tau : LVar \rightarrow Expr$ is a *local* substitution (assigning expressions to local variables, including the formal parameters). The top of a stack Δ is the leftmost closure, if it exists. In the sequel, we denote by $(\tau, S) \cdot \Delta$ the result of pushing the closure (τ, S) unto the stack Δ.
- $\sigma : GVar \rightarrow Expr$ is the current *global* substitution mapping expressions to global variables,
- ϕ is a Boolean expression denoting the path condition.

A configuration $\langle(id, S), id, \phi\rangle$ is called *initial* if no local variable occurs in S, that, in this case, denotes the main statement.

We denote by $\tau \cup \sigma : \mathit{Var} \to \mathit{Val}$ the union of a local substitution τ and a global substitution σ (defined in terms of their representations as sets of pairs argument/value). This is well-defined and total because of the absence of name clashes between local and global variables of a programs. Further, $e(\tau \cup \sigma)$ represents the expression e in which all occurrences of local and global variables have been substituted as dictated by their respective substitutions. We have the following symbolic transitions.

Symbolic assignment global variable Let x be a global variable.

- $\langle(\tau, x := e;\ S) \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau, S) \cdot \Delta, \sigma[x := e\theta], \phi\rangle$,

where $\theta = \tau \cup \sigma$. Note that expressions may contain both global and local variables, and that is why we need θ in the update global substitution $\sigma[x := e\theta]$. Also, the topmost local substitution τ in Δ is not affected by a global assignment because it is side-effect free.

Symbolic assignment local variable Let u be a local variable.

- $\langle(\tau, u := e;\ S) \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau[u := e\theta], S) \cdot \Delta, \sigma, \phi\rangle$,

where $\theta = \tau \cup \sigma$. As for the case of assignment global variable, expressions may contain any variables, and that is why we need θ in the update local substitution $\tau[u := e\theta]$. Only the topmost substitution in the stack is affected by a local assignment.

Symbolic procedure call Given a procedure declaration $P(\bar{u}) :: S'$, we have

- $\langle(\tau, P(\bar{e});\ S) \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau', S') \cdot (\tau, S) \cdot \Delta, \sigma, \phi\rangle$,

where $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$ (defined component-wise). A procedure call thus pushes a new closure unto the stack. This closure consists of the body of the procedure and a new local substitution which assigns the actual parameters to the formal parameters, implementing a call by value parameter passing mechanism. This mechanism thus avoids name clashes between the formal parameters of different procedures because each procedure call is symbolically executed with respect to its own local substitution. Termination of a procedure call is indicated by the empty statement ϵ, so that the return of a procedure call can be described simply by "popping" the top closure of the stack. Execution then can continue with executing the closure (τ, S) that generated the call. This is formalized by the following transition.

Symbolic procedure return

- $\langle(\tau, \epsilon) \cdot \Delta, \sigma, \phi\rangle \to \langle\Delta, \sigma, \phi\rangle$,

Recall that ϵ denotes the empty statement, so when there is nothing more to be currently executed, a pop operation on the stack of closure ensures that the control goes back to the procedure caller, restoring the local substitution but continuing with the global one.

Symbolic choice

- $\langle(\tau, \mathit{if}\ b\ \{S_1\}\{S_2\};\ S) \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau, S_1;\ S) \cdot \Delta, \sigma, \phi \wedge b(\tau \cup \sigma)\rangle$
- $\langle(\tau, \mathit{if}\ b\ \{S_1\}\{S_2\};\ S) \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau, S_2;\ S) \cdot \Delta, \sigma, \phi \wedge \neg b(\tau \cup \sigma)\rangle$

Symbolic iteration

- $\langle(\tau, \mathit{while}\ b\ \{S\};\ S') \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau, S;\ \mathit{while}\ b\ \{S\};\ S') \cdot \Delta, \sigma, \phi \wedge b(\tau \cup \sigma)\rangle$
- $\langle(\tau, \mathit{while}\ b\ \{S\};\ S') \cdot \Delta, \sigma, \phi\rangle \to \langle(\tau, S') \cdot \Delta, \sigma, \phi \wedge \neg b(\tau \cup \sigma)\rangle$

Because the main statement does not contain local variables, we have that local variables will never appear in global and local substitution of any reachable configuration.

Proposition 3.1 For any computation $\langle(\mathit{id}, S), \mathit{id}, \text{true}\rangle \to^n \langle(\tau, S') \cdot \Delta, \sigma, \phi\rangle$ of $n > 0$ steps, where S denotes the main statement, we have that both $\tau(u)$, for any local variable u in its domain, and $\sigma(x)$, for any global variable x, do *not* contain local variables.

Proof We prove it by course of value induction on the length of the computation. For the case when S begins with a global assignment $x := e$ it is enough to notice that $\theta = id \cup id$, and that e does not contain local variables because they do not occur in the main statement S.

Assignment to local variables cannot occur in S, so the only remaining interesting base case of our induction is when S begin with a procedure call $P(\bar{e})$. In this case the new local substitution τ' assigning expressions $\bar{e}(id \cup id) = \bar{e}$ to the formal parameter (i.e. local variables) $\bar{u}$. But every expression in $\bar{e}$ does contain local variables, so they will not occur in the local substitution in the closure of the new top of the stack.

All other base cases are trivially true. So next we consider the case of a global assignment after a computation of length $n > 0$:

$$\langle (id, S), id, \text{true}\rangle \rightarrow^n \langle (\tau, x := e;\ S') \cdot \Delta, \sigma, \phi\rangle \rightarrow \langle (\tau, S') \cdot \Delta, \sigma[x := e\theta], \phi\rangle$$

where $\theta = \tau \cup \sigma$. By induction hypothesis, both $\tau(u)$, for every local variable u, and $\sigma(x)$, for every global variable x, do not contain local variables. So $e\theta$ cannot contain local variables too. A similar reasoning can be applied to the cases of procedure call and local assignment.

For the case of procedure return we use the the stronger hypothesis of our course of value induction: If $\langle (id, S), id, \text{true}\rangle \rightarrow^n \langle (\tau, \epsilon) \cdot \Delta, \sigma, \phi\rangle \rightarrow \langle \Delta, \sigma, \phi\rangle$ then $\sigma(x)$ does not contain local variables by induction hypothesis. If Δ is the empty string then there is nothing more to prove, and otherwise $\Delta = (\tau', S') \cdot \Delta'$ which means that there must exist an intermediate configuration $\langle (\tau', S') \cdot \Delta', \sigma', \phi'\rangle$ such that

$$\langle (id, S), id, \text{true}\rangle \rightarrow^m \langle (\tau', S') \cdot \Delta', \sigma', \phi'\rangle \rightarrow^l \langle (\tau, \epsilon) \cdot (\tau', S') \cdot \Delta', \sigma, \phi\rangle$$

with $m > 0$ and such that the same intermediate configuration does not occur in the last l steps of rightmost part of the computation. By course of value induction, for all local variables u, $\tau'(u)$ does not contain local variables. □

Since path conditions are constructed by local and global substitutions applied to Boolean expressions, we have the following immediate corollary.

Corollary 3.2 For any computation $\langle (id, S), id, \text{true}\rangle \rightarrow^* \langle (\tau, S') \cdot \Delta, \sigma, \phi\rangle$ where S denotes the main statement, the generated path condition ϕ does not contain local variables.

Proof The case for a computation of length $n > 0$ is a consequence of Proposition 3.1. The case when a computation is of length 0 is immediate. □

A *concrete configuration* is of the form $\langle \Gamma, G\rangle$, where $G : GVar \rightarrow Val$ is an assignment of values to global variables, and Γ denotes the stack of *concrete closures*, i.e. a finite string of element of the form (L, S). Here $L : LVar \rightarrow Val$ is an assignment of values to local variables and S denotes the statement to be executed.

As in the previous section, we have a substitution lemma which states that evaluating an expression e in the composition of a valuation V after a substitution θ gives the same value as evaluating in V the expression $e\theta$.

Lemma 3.3 (Substitution). Let $V = G \cup L$ and $\theta = \sigma \cup \tau$. Then $V \circ \theta(e) = V(e\theta)$ for every expression e.

Proof The proof of the lemma proceeds by induction on e, exactly as in the substitution Lemma in the previous section. □

Next we present a concrete transition system semantics which instead of substitutions is defined in terms of valuations both for the local and the global variables.

Concrete assignment global variable Let x be a global variable.

- $\langle (L, x := e;\ S) \cdot \Gamma, G\rangle \rightarrow \langle (L, S) \cdot \Gamma, G[x := V(e)]\rangle$, where $V(e) = (L \cup G)(e)$.

Concrete assignment local variable Let u be a local variable.

- $\langle (L, u := e;\ S) \cdot \Gamma, G\rangle \rightarrow \langle (L[u := V(e)], S) \cdot \Gamma, G\rangle$, where $V(e) = (L \cup G)(e)$.

Concrete procedure call Given a procedure declaration $P(\bar{u}) :: S'$, we have

- $\langle (L, P(\bar{e});\ S) \cdot \Gamma, G\rangle \rightarrow \langle (L', S') \cdot (L, S) \cdot \Gamma, G\rangle$,where $L'(\bar{u}) = (L \cup G)(\bar{e})$ (defined component-wise).

Concrete procedure return

- $\langle (L, \epsilon) \cdot \Gamma, G\rangle \to \langle \Gamma, G\rangle$.

Concrete choice

- $\langle (L, \mathit{if}\ b\ \{S_1\}\{S_2\};\ S) \cdot \Gamma, G\rangle \to \langle (\tau, S_1;\ S) \cdot \Gamma, G\rangle$, if $(L \cup G)(b)$ is true;
- $\langle (L, \mathit{if}\ b\ \{S_1\}\{S_2\};\ S) \cdot \Gamma, G\rangle \to \langle (\tau, S_2;\ S) \cdot \Gamma, G\rangle$, if $(L \cup G)(b)$ is false.

Concrete iteration

- $\langle (L, \mathit{while}\ b\ \{S\};\ S') \cdot \Gamma, G\rangle \to \langle (L, S;\ \mathit{while}\ b\ \{S\};\ S') \cdot \Gamma, G\rangle$, if $(L \cup G)(b)$ is true;
- $\langle (L, \mathit{while}\ b\ \{S\};\ S') \cdot \Gamma, G\rangle \to \langle (L, S') \cdot \Gamma, G\rangle$, if $(L \cup G)(b)$ is false.

From the above substitution lemma we derive the following corollary.

Corollary 3.4 (Soundness assignment). Let $V = G \cup L$ and $\theta = \sigma \cup \tau$. For $V' = V \circ \theta$, let G' be the restriction of V' to global variables, and L' the restriction of V' to local variables. For every expression e, global variable x, and local variable u we have that $V \circ (\sigma[x := e\theta]) = G'[x := V'(e)]$ and $V \circ (\tau[u := e\theta]) = L'[u := V'(e)]$.

Proof We treat the case of global variables, the one for local variables is analogous:

$$\begin{array}{lll}
(V \circ \sigma[x := e\theta])(x) & = V(\sigma[x := e\theta](x)) & (\text{def. } \circ)\\
& = V(e\theta) & (\text{def. } \sigma[x := e\theta])\\
& = V \circ \theta(e) & (\text{substitution lemma})\\
& = V'(e) & (V' = V \circ \theta)\\
& = V'[x := V'(e)](x) & (\text{def. } V'[x := V'(e)])\\
& = G'[x := V'(e)](x) & (\text{def. } G' \text{ and } x \text{ global variable})
\end{array}$$

□

We have the following correctness theorem of the symbolic execution of recursive programs: for every reachable configuration of a symbolic execution there exists an analogous concrete configuration reachable from every valuations satisfying the last path condition.

Theorem 3.5 (Correctness). For a main statement S with no occurrences of local variables and $V = G \cup L$ if $\langle (\mathit{id}, S), \mathit{id}, \text{true}\rangle \to^* \langle (\tau, S') \cdot \Delta, \sigma, \phi\rangle$ and $V(\phi) = \mathit{true}$ then

$$\langle (L, S), G\rangle \to^* \langle (V \circ \tau, S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle$$

where $(V \circ \sigma)(x) = V(\sigma(x))$ for any global variable x, $V \circ \tau'(x) = V(\tau'(u))$ for any local substitution τ' and local variable u, and $V \bullet \Delta$ denotes the concrete stack resulting from replacing every local environment τ' in Δ by $V \circ \tau'$.

Proof As in Theorem 2.3, we proceed by induction on the length of the symbolic computation and a case analysis of the last execution step.

First, we consider the case of a global assignment as the last execution step:

$$\langle (\mathit{id}, S), \mathit{id}, \text{true}\rangle \to^* \langle (\tau, x := e;\ S') \cdot \Delta, \sigma, \phi\rangle \to \langle (\tau, S') \cdot \Delta, \sigma[x := e\theta], \phi\rangle\,,$$

where $\theta = \tau \cup \sigma$. Let $V = G \cup L$ such that $V(\phi)$ is true. By induction hypothesis there exists a concrete computation

$$\langle (L, S), G\rangle \to^* \langle (V \circ \tau, x := e;\ S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle$$

Let $V' = V \circ \sigma$ and G' its restriction to global variables. By the concrete semantics of global assignment we have

$$\langle (V \circ \tau, x := e;\ S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle \to \langle (V \circ \tau, S) \cdot V \bullet \Delta,\ G'[x := V'(e)]\rangle$$

By Corollary 3.4 it then suffices to observe that $V \circ (\sigma[x := e\theta]) = G'[x := V'(e)]$. The case of a local assignment is similar, using the local variable part of Corollary 3.4.

Given the procedure declaration $P(\bar{u}) :: S'$, next we consider the case of a procedure call as last step of a symbolic execution

$$\langle (\mathit{id}, S), \mathit{id}, \text{true}\rangle \to^* \langle (\tau, P(\bar{e});\ S'') \cdot \Delta, \sigma, \phi\rangle \to \langle (\tau', S') \cdot (\tau, ;\ S'') \cdot \Delta, \sigma, \phi\rangle$$

where $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$. Let $V = G \cup L$ be such that $V(\phi)$ is true. By induction hypothesis there exists a concrete computation

$$\langle (L, S),\ V\rangle \to^* \langle (V \circ \tau,\ P(\bar{e});\ S'') \cdot V \bullet \Delta,\ V \circ \sigma\rangle\,.$$

By the concrete semantics of procedure call we have

$$\langle (V \circ \tau,\ P(\bar{e});\ S'') \cdot V \bullet \Delta,\ V \circ \sigma\rangle \to \langle (\tau'', S') \cdot (V \circ \tau;\ S'') \cdot V \bullet \Delta,\ V \circ \sigma\rangle\,,$$

where $\tau''(\bar{u}) = (V \circ \tau \cup V \circ \sigma)(\bar{e}) = V \circ (\tau \cup \sigma)(\bar{e}) = V(\bar{e}(\tau \cup \sigma)) = V \circ \tau'(\bar{u})$, concluding the proof for this case.

The remaining interesting case of procedure return as last step of a symbolic computation is easier:

$$\langle (id, S), id, \text{true}\rangle \to^* \langle (\tau, \epsilon) \cdot (\tau', S') \cdot \Delta, \sigma, \phi\rangle \to \langle (\tau', S') \cdot \Delta, \sigma, \phi\rangle\,.$$

Let $V = G \cup L$ be such that $V(\phi)$ is true. By induction hypothesis there exists a concrete computation

$$\langle (L, S),\ V\rangle \to^* \langle (V \circ \tau, \epsilon) \cdot (V \circ \tau', S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle\,.$$

By the concrete semantics of procedure call we have

$$\langle (V \circ \tau, \epsilon) \cdot (V \circ \tau', S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle \to \langle (V \circ \tau', S') \cdot V \bullet \Delta,\ V \circ \sigma\rangle\,.$$

The choice construct and the iterations can be treated in a similar way as in Theorem 2.3. □

As in the basic case, we have a similar completeness result for recursive procedures expressed in terms reachable concrete configurations: for every reachable concrete configuration starting from a global valuation G there exists an analogous reachable symbolic configuration with a path condition satisfied by G.

Theorem 3.6 (Completeness). Let S be a statement with no occurrence of local variables. For any concrete computation $\langle (L, S), G\rangle \to^* \langle\langle (L', S') \cdot \Gamma, G'\rangle$ there exists a symbolic computation

$$\langle (id, S), id, true\rangle \to^* \langle (\tau, S') \cdot \Delta, \sigma, \phi\rangle$$

for some path condition ϕ and local substitution τ and global substitution σ such that $V(\phi)$ is true, $\Gamma = V \bullet \Delta$, $L' = V \circ \tau$, and $G' = V \circ \sigma$, for $V = G \cup L$.

Proof The proof proceeds by induction on the length of the concrete computation and a case analysis of the last concrete execution step. First, we consider the case of a local assignment as the last execution step:

$$\langle (L, S), G\rangle \to^* \langle (L', u := e;\ S') \cdot \Gamma, G'\rangle \to \langle (L'[u := V'(e)]) \cdot \Gamma, G'\rangle\,,$$

where $V' = L' \cup G'$. By induction hypothesis there exists a symbolic computation

$$\langle (id, S), id, true\rangle \to^* \langle (\tau, u := e;\ S') \cdot \Delta, \sigma, \phi\rangle$$

for some path condition ϕ and a substitutions τ and σ such that $V(\phi)$ is true, $\Gamma = V \bullet \Delta$, $G' = V \circ \sigma$ and $L' = V \circ \tau$.

By the symbolic semantics of local assignment we have that

$$\langle (\tau, u := e;\ S') \cdot \Delta, \sigma, \phi\rangle \to \langle (\tau[u := e\theta], S') \cdot \Delta, \sigma, \phi\rangle$$

with $\theta = \tau \cup \sigma$. It now suffices to see that by Corollary 3.4 we have $V \circ (\tau[u := e\theta]) = L'[u := V'(e)]$.

The case of global assignment is analogous. Next we treat another case, the one of a procedure call as the last execution step:

$$\langle (L, S), G\rangle \to^* \langle (L', P(\bar{e});\ S') \cdot \Gamma, G'\rangle \to \langle (L'', S'') \cdot (L', S') \cdot \Gamma, G'\rangle\,,$$

where $P(\bar{u}) :: S''$ and $L''(\bar{u}) = V'(\bar{e})$, with $V' = L' \cup G'$. By induction hypothesis there exists a symbolic computation

$$\langle (id, S), id, true\rangle \to^* \langle (\tau, P(\bar{e});\ S') \cdot \Delta, \sigma, \phi\rangle$$

for some path condition ϕ and a substitutions τ and σ such that $V(\phi)$ is true, $\Gamma = V \bullet \Delta$, $G' = V \circ \sigma$ and $L' = V \circ \tau$.

By the symbolic semantics of procedure call we have that

$$\langle (\tau, P(\bar{e});\ S') \cdot \Delta, \sigma, \phi\rangle \to \langle (\tau', S'') \cdot (\tau, S') \cdot \Delta, \sigma, \phi\rangle$$

where $\tau'(\bar{u}) = \bar{e}\theta$, with $\theta = \tau \cup \sigma$. By the above we can conclude the proof with the following series of equality $L''(\bar{u}) = V'(\bar{u}) = (L' \cup G')(\bar{u}) = ((V \circ \tau) \cup (V \circ \sigma))(\bar{u}) = (V \circ (\tau \cup \sigma))(\bar{u}) = (V \circ \tau')(\bar{u})$.

The case of choice and iteration constructs can be essentially treated in the same way as the completeness theorem of the previous section. All other cases are treated similarly. □

4. Object orientation

We next extend the language of the previous section with object-oriented features. We first introduce a distinction between variables $x, y, \ldots$, e.g., the formal parameters of methods, including the keyword *this*, and *fields* $f, f', \ldots$ (of the classes of a program). In contrast to the previous sections, because of dynamically allocated objects, we need to assume here an infinite number of variables. We abstract again from the typing information of the variables and fields. We have the following syntax of *side-effect free* programming expressions e:

$$e ::= \mathit{nil} \mid x \mid e.f \mid \mathit{op}(e_1, \ldots, e_n)$$

where *nil* stands for "undefined' (used to initialize the fields of newly created objects)', x is a variable, and in the expression $e.f$ we implicitly assume that f is a field of the class of the object denoted by e. For notational convenience, in the sequel h denotes a variable x or an expression $e.f$. An expression h is called a *heap variable*. As in the previous section *op* denotes a built-in operation (which includes for example equality '='). We assume a ternary operation which describes conditional expressions written as *if* b *then* e_1 *else* e_2 *fi*, where b denotes a Boolean expression. Without loss of generality we assume that expressions which denote objects can only be used for dereferencing, for checking equality (denoted by '='), or as argument of a conditional expression. More specifically, we do not have, for example, built-in operations for constructing sets or sequences of objects.

Statements S of the object-oriented programming language are then defined by the grammar:

$$\begin{array}{lll} S ::= & h := e & \text{assignment} \\ \mid & h := \mathit{new}\ C & \text{object creation} \\ \mid & h := e_0.m(\bar{e}) & \text{method call} \\ \mid & \mathit{return}\ e & \text{return statement} \\ \mid & S;\ S & \text{sequential composition} \\ \mid & \mathit{if}\ B\ \{S_1\}\{S_2\} & \text{choice} \\ \mid & \mathit{while}\ B\ \{S\} & \text{iteration} \end{array}$$

A statement $h := \mathit{new}\ C$ creates an instance of class C (we model a call $x := \mathit{new}\ C(\bar{e})$ of a constructor method by the object creation statement $x := \mathit{new}\ C$ followed by a method call $x.C(\bar{e})$). A method call $h := e_0.m(\bar{e})$ specifies the called object e_0. The execution of a method terminates with a statement *return* e which returns the value of e. Each class definition consists of a set of method definitions. A method definition specifies its formal parameters and a statement (its "body"). Finally, a program consists of some class definitions and a main statement which only involves (global) variables.

The approach of the previous sections is naturally extended by interpreting fields as arrays, i.e., a heap expression $e.f$ is then interpreted as $f[e]$. This is briefly discussed in Section 4.1. Here we proceed with a more fundamental approach to symbolic execution which is based on *symbolic execution traces* and a weakest preconditon calculus. In this new approach the symbolic execution of a program is described in terms of a transition relation between configurations of the form $\langle S, \rho \rangle$, where ρ denotes a symbolic execution trace consisting of Boolean expressions and assignments. We will define the path condition of a symbolic trace by means of a weakest precondition calculus. Since our language does not include error-handling constructs, it suffices that the resulting path condition will ensure absence of so-called *nil-pointer* errors, which are generated by expressions $e.f$ in case the expression e denotes the *nil* pointer. Note that such path conditions still can be used to test *nil* pointer errors occurring at a specific control point. The symbolic execution of constructs for handling *nil* pointer errors can be modeled following the basic approach described in subsection 2.1 for array-out-of-bound errors, but is out of scope of this paper, as are other object-oriented features like dynamic method dispatch. Most of such additional features however we expect can be integrated by a symbolic representation of their standard operational semantics (similar to the symbolic representation of the standard operational semantics of the choice and iteration constructs).

For our object-oriented language we have the following transition rules for generating such traces by statically unfolding a statement.

Symbolic assignment

- $\langle h := e;\ S, \rho\rangle \to \langle S, \rho \cdot h := e\rangle$

Symbolic object creation

- $\langle h := \mathit{new}\ C;\ S, \rho\rangle \to \langle S, \rho \cdot x := \mathit{new}\ C \cdot h := x\rangle$,

where x is a *fresh* variable not appearing in $\langle h := \mathit{new}\ C;\ S, \rho\rangle$.The introduction of a fresh variable in the above transition allows to disentangle aliasing and object creation in the definition of a path condition (as defined below).

Symbolic method call Given a method declaration $m(\bar{u})\{S\}$, we have

- $\langle h := e_0.m(\bar{e});\ S'), \rho\rangle \to \langle S[\mathit{this}, \bar{u} := e_0, \bar{y}];\ h := r;\ S', \rho \cdot e_0 \neq \mathit{nil} \cdot \bar{y} := \bar{e}\rangle$,

where application of the *simultaneous* substitution $[\mathit{this}, \bar{u} := e_0, \bar{y}]$ replaces in S all occurrences of the distinguished variable *this* by the expression e_0 and all the formal parameters $\bar{u}$ by *fresh* variables (i.e., not appearing in the configuration $\langle h := e_0.m(\bar{e});\ S'), \rho\rangle$). The condition $e_0 \neq \mathit{nil}$ records that the callee is not *nil*. By $\bar{y} := \bar{e}$ we denote a sequence of assignments of the expressions $\bar{e}$ to the variables $\bar{y}$ (note that the order does not matter, because the variables $\bar{y}$ do not appear in $\bar{e}$). The variable r is used to denote the return value (see below). Note that thus a method call symbolically is described by inlining the method body, replacing the formal parameters by fresh variables thus avoiding name clashes between different method invocations. Note that in the semantics of (recursive) procedures (described in the previous section) such name clashes are avoided because of the use of an explicit stack of closures so that each call is executed in its own local environment (which assigns values to its local variables).

Symbolic method return

- $\langle \mathit{return}\ e;\ S, \rho\rangle \to \langle S, \rho \cdot r := e\rangle$,

where r is the distinguished variable used to denote the return value. The following symbolic transitions of the choice and iteration statements simply record the (negation of) the Boolean condition.

Symbolic choice

- $\langle \mathit{if}\ b\ \{S_1\}\ \{S_2\};\ S, \rho\rangle \to \langle S_1;\ S, \rho \cdot b\rangle$
- $\langle \mathit{if}\ b\ \{S_1\}\ \{S_2\}, \rho\rangle \to \langle S_2;\ S, \rho \cdot \neg b\rangle$

Symbolic iteration

- $\langle \mathit{while}\ b\ \{S\};\ S', \rho\rangle \to \langle S;\ \mathit{while}\ b\ \{S\};\ S', \rho \cdot b\rangle$
- $\langle \mathit{while}\ b\ \{S\};\ S', \rho\rangle \to \langle S', \rho \cdot \neg b\rangle$

Note that the above symbolic transition system abstracts from *nil-pointer* errors, e.g., for the symbolic execution of the Boolean conditions in the choice and iteration statements we do *not* consider the third possibility that the evaluation of the condition generates a *nil-pointer* error. We emphasize here again that for the purpose of generating path conditions this is not needed because our language does not include error-handling constructs and thus a path condition should only ensure *absence* of such errors.

To define the path condition and the substitution of a symbolic execution trace, we first need the following basic substitutions. By $e'[x := e]$ we denote the result of replacing all occurrences of the variable x in e' by e. On the other hand, we define $e'[h.f := e]$ by

$$
\begin{array}{ll}
\mathit{nil}[h.f := e] & = \mathit{nil} \\
x[h.f := e] & = x \\
(h'.f')[h.f := e] & = h'[h.f := e].f' \\
(h'.f)[h.f := e] & = \mathit{if}\ h'[h.f := e] = h\ \mathit{then}\ e\ \mathit{else}\ h'[h.f := e].f\ \mathit{fi} \\
\mathit{op}(e_1, \ldots, e_n)[h.x := e] & = \mathit{op}(e_1[h.x := e], \ldots, e_n[h.x := e])
\end{array}
$$

where in the third clause f' is different (syntactically) from f. The conditional expression in the fourth clause captures aliasing. The overall idea of the substitution $e'[h.f := e]$ is that it generates an expression the value of which equals that of e' *after* the execution of the assignment $h.f := e$. In case of an expression $h'.f$ we have to check whether h' *after* the assignment $h.f := e$ equals h, in which case the value of $h'.f$ after the assignment $h.f := e$ equals that of e. Otherwise, the value of $h'.f$ after the assignment $h.f := e$ equals the value of the field of the object denoted by the value of h' after the assignment $h.f := e$. The last clause, instead, captures basic substitution of Boolean expressions too. Simultaneous substitutions $e'[\bar{h} := \bar{e}]$ are defined by extending the above as expected, thus denoting the result of replacing all occurrence of variables $h_i \in \bar{h}$ in e' by the correspondent $e_i \in \bar{e}$.

Next we define $e[x := \mathit{new}\ C]$, for expressions e (syntactically) different from x. First we observe that we can assume without loss of generality that e also does not contain conditional (sub)expressions with the variable x as argument, because we can transform every equation *if* b *then* e_1 *else* e_2 *fi* $= e$ between object expressions to *if* b *then* $e_1 = e$ *else* $e_2 = e$ *fi* (and similarly for $e =$ *if* b *then* e_1 *else* e_2 *fi*). Further, any expression *if* b *then* e_1 *else* e_2 *fi*.f can be transformed into *if* b *then* $e_1.f$ *else* $e_2.f$ *fi*. Consequently, we may assume that the variable x can only be dereferenced or appear as an argument of an equation (in the expression e). We thus can restrict the definition of $e[x := \mathit{new}\ C]$ to the following cases:

- $\mathit{nil}[x := \mathit{new}\ C] = \mathit{nil}$;
- $y[x := \mathit{new}\ C] = y$, y a variable distinct from x;
- $(x.f)[x := \mathit{new}\ C] = \mathit{nil}$;
- $(e.f)[x := \mathit{new}\ C] = e[x := \mathit{new}\ C].f$, where e is different from x;
- $(x = e)[x := \mathit{new}\ C] = \mathit{false}$, where e is different from x;
- $(e = x)[x := \mathit{new}\ C] = \mathit{false}$, where e is different from x;
- $(x = x)[x := \mathit{new}\ C] = \mathit{true}$.
- $\mathit{op}(e_1, \ldots, e_n)[x := \mathit{new}\ C] = \mathit{op}(e_1[x := \mathit{new}\ C], \ldots, e_n[x := \mathit{new}\ C])$.

The last clause also covers the case of conditional expressions.

Further, following subsection 2.1, we define the *absence* of a *nil-pointer* errors by a predicate $\delta(e)$.

Definition 4.1 (Absence of nil-pointer errors). We define $\delta(e)$ inductively by

$$\begin{array}{ll} \delta(\mathit{nil}) & = \mathit{true} \\ \delta(x) & = \mathit{true} \\ \delta(e.f) & = e \neq \mathit{nil} \wedge \delta(e) \\ \delta(\mathit{op}(e_1, \ldots, e_n)) & = \delta(e_1) \wedge \cdots \wedge \delta(e_n) \end{array}$$

We can now define the path condition of an execution trace by moving forward (from left to right) through it. Recall that until now, we always appended the last "action" to the right of the execution trace.

Definition 4.2 (Path condition). We define the path condition $\mathit{path}(\rho)$ of a symbolic execution trace inductively by (here ϵ denotes the empty path):

$$\begin{array}{ll} \mathit{path}(\epsilon) & = \mathit{true} \\ \mathit{path}(b \cdot \rho) & = \mathit{path}(\rho) \wedge b \\ \mathit{path}(h := e \cdot \rho) & = \mathit{path}(\rho)[h := e] \wedge \delta(h, e) \\ \mathit{path}(x := \mathit{new}\ C \cdot \rho) & = \mathit{path}(\rho)[x := \mathit{new}\ C] \end{array}$$

where in the third clause $\delta(h, e)$ abbreviates the conjunction $\delta(h) \wedge \delta(e)$.

Note that we do not include the predicate $\delta(b)$ in the path condition in case of a Boolean expression, because, as we will see below, if a Boolean expression evaluates to true then clearly its evaluation does not generate a *nil-pointer* error. In general, the above definition of a path condition thus ensures absence of *nil-pointer* errors.

Next we discuss the concrete semantics. Given a program, for each of its classes we assume an unbounded set of object identities (which for different classes are disjoint). Such identities we denote in the sequel by o, o', etc.. A concrete configuration $V = (G, H)$ consists of an assignment G of values to the variables and a *heap* H which assigns to each object o in its finite domain a local state $H(o)$. Such a local state assigns to every field f of the object o its value denoted by $H(o.f)$. As a special case, we identify the expression *nil* with its value. The objects

in the domain of H represent the current set of *existing* objects. We restrict to *consistent* concrete configurations $V = (G, H)$: if $G(x) = o$ then the object o exists, i.e., the domain of H includes o, and any field f of an existing object o used to denote objects denotes either an existing object or has the value *nil*.

The concrete semantics then can be defined by a translation relation between configurations $\langle S, V\rangle$, where the semantics of method calls is defined as above in terms of body replacement (replacing the local variables by fresh variables). The details of this semantics are standard (see for example [AdBO09]) and therefore omitted. *Here it suffices to observe that in general the standard concrete semantics of a program in any (imperative) programming language can be defined also in terms of the concrete semantics of its symbolic execution traces*. Below we illustrate this general approach by means of our object-oriented language.

First, we define the value $V(e)$ of an expression e in the configuration V. By '$V(e) = \bot$' we denote that the evaluation of e gives rise to a *nil-pointer* error. Let $V = (G, H)$.

$$\begin{array}{ll} V(\mathit{nil}) & = \mathit{nil} \\ V(x) & = G(x) \\ V(e.f) & = \begin{cases} H(V(e).f)) & \text{if } V(e) \neq \bot \text{ and } V(e) \neq \mathit{nil} \\ \bot & \text{otherwise} \end{cases} \\ V(\mathit{op}(e_1, \ldots, e_n)) & = \overline{\mathit{op}}(V(e_1), \ldots, V(e_n)) \end{array}$$

where $\overline{op}$ again denotes the given semantic interpretation of *op*, which is strict in the sense that it yields $\bot$ if one if its arguments is $\bot$.

The following lemma proves a semantic justification of the predicate $\delta(e)$.

Lemma 4.3 For any expression e we have that

$$V(e) \neq \bot \text{ iff } V(\delta(e)) = \mathit{true}$$

Proof The proof involves a straightforward induction on the structure of e. □

Note that for any Boolean expression b we have that $V(b) = \mathit{true}$ thus implies $V(\delta(b)) = \mathit{true}$. In words, if a Boolean expression evaluates to true then clearly its evaluation does not generate a *nil-pointer* error. This explains the above definition of a path condition in case of Boolean expressions.

We further define for $V = (G, H)$ and value v the update $V[x := v]$ by $(G[x := v], H)$ and $V[o.f := v]$ by $(G, H[o.f := v])$], where $H[o.f := v]$ is a shorthand for updating the field f of the object o. For notational convenience, we abbreviate $V[V(e).f := v]$, where $V(e) = o$, for some object o (that is, $V(e) \neq \bot$ and $V(e) \neq \mathit{nil}$), by $V[e.f := v]$. Finally, for $V = (G, H)$ we define the update $V[x := \mathit{new}\ C]$ by $(G[x := o], H')$], where the object identifies o of type C does not belong to the domain of H and H' results from H by adding o to the domain of H and assigning to every field of o the value it nil, i.e., $H'(o.f) =$ it nil, for every field f.

We are now sufficiently equipped to define the concrete semantics of symbolic execution traces as a transition relation between configurations $\langle V, \rho\rangle$.

Concrete assignment

- $\langle V, h := e \cdot \rho\rangle \rightarrow \langle V[V(h) := V(e)], \rho\rangle$, provided $V(h) \neq \bot$ and $V(e) \neq \bot$

Concrete object creation

- $\langle V, x := \mathit{new}\ C \cdot \rho\rangle \rightarrow \langle V[x := \mathit{new}\ C], \rho\rangle$

Concrete Boolean test

- $\langle V, b \cdot \rho\rangle \rightarrow \langle V, \rho\rangle$, provided $V(b) = \mathit{true}$.

In order to formally relate the semantics of a path condition of a symbolic execution trace and its concrete semantics as defined above, we need the following substitution lemma (see also [dB99]).

Lemma 4.4 (Substitution). Let $V(h) \neq \bot$ and $V(e) = v(\neq \bot)$. If $V[h := v](e') \neq \bot$ and $V(e'[h := e]) \neq \bot$ then

$$V[h := v](e') = V(e'[h := e])$$

Further, if $V[x := new\ C](e) \neq \bot$ and $V(e[x := new\ C]]) \neq \bot$ then if

$$V[x := new\ C](e) = V(e[x := new\ C]])$$

where the expression e is different from x and also does not contain conditional expressions with the variable x as argument.

Proof Let $V(e) = v$. We first prove for a variable x that $V[x := v](e') = V(e'[x := e])$ for $V = (G, H)$, by induction on e'. Let $V' = V[x := v]$, we have four cases e':

$e' = x$	$e' = y$
$V'(x)$ $= G[x := v](x)$ $= V(e)$ $= V(x[x := e])$	$V'(y)$ $= V(y)$ $= V(y[x := e])$
$e' = h.f$	$e' = op(e_1, \ldots, e_n)$
$V'(h.f)$ $= H(V'(h).f))$ $= H(V(h[x := e]).f)$ $= H(V((h.f)[x := e]))$	$V'(op(e_1, \ldots, e_n))$ $= \overline{op}(V'(e_1), \ldots, V'(e_n))$ $= \overline{op}(V((e_1[x := e]), \ldots, V](e_n[x := e]))$ $= V(op(e_1[x := e], \ldots, e_n[x := e]))$ $= V(op(e_1, \ldots, e_n)[x := e])$

Next we prove that $V[h.f := v](e') = V(e'[h.f := e])$ for $V = (G, H)$, by induction on e'.
Let $V' = V[h.f := v]$. Also here we have four cases for e':

$e' = x$
$V'(x)$ $= V(x)$ $= V(x[h.f := e])$
$e' = h'.f'$
$V'(h'.f')$ $= H(V'(h').f')$ $= H(V(h'[h.f := v]).f')$ $= V(h'[h.f := v].f')$ $= V((h'.f')[h.f := v])$
$e' = h'.f$
$V'(h'.f)$ $= H[V(h).f := v](V'(h').f)$ $=$ *if* $V(h) = V'(h')$ *then* v *else* $H(V'(h').f)$ *fi* $=$ *if* $V(h) = V(h'[h.f := e])$ *then* v *else* $H(V(h'[h.f := e]).f)$ *fi* $=$ *if* $V(h) = V(h'[h.f := e])$ *then* v *else* $V(h'[h.f := e].f)$ *fi* $= V($*if* $h = h'[h.f := e]$ *then* e *else* $h'[h.f := e].f$ *fi*$)$ $= V((h'.f)[h.f := e])$
$e' = op(e_1, \ldots, e_n)$
$V'(op(e_1, \ldots, e_n))$ $= \overline{op}(V'(e_1), \ldots, V'(e_n))$ $= \overline{op}(V((e_1[h.f := e]), \ldots, V](e_n[h.f := e]))$ $= V(op(e_1[h.f := e], \ldots, e_n[h.f := e]))$ $= V(op(e_1, \ldots, e_n)[h.f := e])$.

We continue with a proof of $V[x := new\ C](e) = V(e[x := new\ C])$, where the e, distinct from x, does not contain conditional expressions with the variable x as argument. Let $V = (G, H)$, and $V[x := new\ C]$ be $(G[x := o], H')$, where o does not belong to the domain of H (o of type C) and H' results from H by assigning *nil* to every field of o, i.e., $H'(o.f) = nil$, for every field f.

First we show that $V[x := new\ C](e) = nil$ or $V[x := new\ C](e)$ denotes an "old" object, i.e., $H(V[x := new\ C](e)) \neq \bot$, for every object expression e different from x (and which does not contain conditional subexpressions with x as argument). Let $V' = V[x := new\ C]$. It suffices to consider the following cases.

- $V'(y) = G[x := o](y) = G(y)$. From the consistency of V we conclude that $G(y) \neq nil$ implies $H(G(y)) \neq \bot$.
- $V'(\mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{fi})$ equals either $V'(e_1)$ or $V'(e_2)$. By assumption both e_1 and e_2 are different from x, and so by induction both $V'(e_1)$ and $V'(e_2)$ are either *nil* or an old object.
- $V'(x.f) = nil$.
- $V'(e.f) = H'(V'(e).f)$, where e is different from x. By induction $V'(e)$ is either *nil* or an old object, and so by the consistency of V and the So $H'(V'(e).f) = H(V'(e).f))$ denotes an old object.

We can now proceed with the following case analysis.

- $V(nil[x := new\ C]) = nil = V'(nil)$.
- $V(y[x := new\ C]) = V(y) = V'(y)$, y a variable distinct from x.
- $V((x.f)[x := new\ C]) = nil = V'(x.f)$.
- $V((e.f)[x := new\ C]) = H(V(e[x := new\ C]).f$, where e is different from x. By induction $V(e[x := new\ C]) = V'(e)$. Further, by the above $V'(e)$ denotes an "old" object, and so by definition of V', we have that $H(V'(e).f) = H'(V'(e).f) = V'(e.f)$.
- $V((x = e)[x := new\ C]) = false$, where e is different from x. By the above again, $V'(e)$ denotes an "old" object, so $V'(x = e) = false$. (Similarly, $V((e = x)[x := new\ C]) = V'(e = x))$.
- $V((x = x)[x := new\ C]) = true = V'(x = x)$.
- $$\begin{aligned} V(op(e_1, \ldots, e_n)[x := new\ C]) &= \overline{op}(V(e_1[x := new\ C]), \ldots, V(e_n[x := new\ C])) \\ &= \overline{op}(V'(e_1), \ldots, V'(e_n)) \\ &= V'(op(e_1, \ldots, e_n)) \end{aligned}$$

The last clause also covers the case of conditional expressions. □

The following soundness and completeness theorem abstracts from programs and is stated directly in terms of symbolic execution traces (which we only assume to be well-typed). For notational convenience, let $\langle V, \rho\rangle \rightarrow^* \epsilon$ denotes the existence of a concrete configuration V' such that $\langle V, \rho\rangle \rightarrow^* \langle V', \epsilon\rangle$.

Theorem 4.5 (Soundness and completeness). For any symbolic execution trace ρ we have that

$\langle V, \rho\rangle \rightarrow^* \epsilon$ iff $V(path(\rho)) = true$

Proof The proof proceeds by induction on ρ.

First, let ρ be of the form $h := e \cdot \rho'$ (the base case $\rho = \epsilon$ is trivial). We have $\langle V, h := e \cdot \rho'\rangle \rightarrow^* \epsilon$ iff $V(h) \neq \bot$, $V(e) \neq \bot$, $\langle V, h := e \cdot \rho'\rangle \rightarrow \langle V[h := V(e)], \rho'\rangle$, and $\langle V[h := V(e)], \rho'\rangle \rightarrow^* \epsilon$. By Lemma 4.3 we have that $V(h) \neq \bot$ and $V(e) \neq \bot$ iff $V(\delta(h, e)) = true$. By the induction hypothesis we have that $\langle V[h := V(e)], \rho'\rangle \rightarrow^* \epsilon$ iff $V[h := V(e)](path(\rho')) = true$. By definition $path(\rho) = path(\rho')[h := e] \wedge \delta(h, e)$. It then suffices to observe that by the substitution lemma 4.4 we have $V(path(\rho')[h := e]) = V[h := V(e)(path(\rho'))$.

Next , let ρ be of the form $y := new\ C \cdot \rho'$. By definition $path(\rho) = path(\rho')[y := new\ C]$. We have $\langle V, y := new\ C \cdot \rho'\rangle \rightarrow^* \epsilon$ iff $\langle V, y := new\ C \cdot \rho'\rangle \rightarrow \langle V[y := new\ C], \rho'\rangle$ and $\langle V[y := new\ C], \rho'\rangle \rightarrow^* \epsilon$. From the induction hypothesis we infer that $\langle V[y := new\ C], \rho'\rangle \rightarrow^* \epsilon$ iff $V[y := new\ C](path(\rho')) = true$. By definition of $path(\rho)$ and the above substitution lemma 4.4 we have $V(path(\rho)) = V(path(\rho')[y := new\ C]) = VV[y := new\ C]path(\rho'))$.

We are left with the following case: $\rho = b \cdot \rho'$. By definition of the concrete semantics, $\langle V, \rho\rangle \rightarrow^* \epsilon$ iff $V(b) = true$ and $\langle V, \rho'\rangle \rightarrow^* \epsilon$. By the induction hypothesis we have that $\langle V, \rho'\rangle \rightarrow^* \epsilon$ iff $V(path(\rho')) = true$. Suffices then to observe that by definition $path(\rho) = b \wedge path(\rho')$. □

Note that the above theorem is thus applicable to any symbolic trace generated by the symbolic transitions system, that is, to any symbolic execution $\langle S, \epsilon\rangle \rightarrow^* \langle S', \rho\rangle$. Differently from the approach of the previous sections however we do not have a direct symbolic representation by a substitution of the state V' for which there exists a computation $\langle V, \rho\rangle \rightarrow^* \langle V', \epsilon\rangle$. This state V' is only implicitly given by the symbolic trace ρ itself. In fact, the development of our new approach is primarily motivated by avoiding the computation of such a substitution in the construction of the path condition.

4.1. Fields as arrays

In the above approach to the generation of a path condition no substitution representing a concrete state is needed. Instead, a weakest precondition calculus is used to calculate the path condition as the weakest precondition of the given symbolic trace (seen as a basic straight-line program). As such the path condition is calculated in a *backward* manner whereas the trace is generated by a basic *forward* symbolic execution of the given program.

We briefly describe here how to generate the path condition *in one pass* by a standard forward symbolic execution of the given program, using a substitution to represent symbolically the concrete state. As already stated above, we can do so simply by modeling symbolically fields as arrays of type $O \to T$, where O is the type of all objects, and process field updates symbolically as array updates, as described in Subsection 2.1). More specifically, any expression e is processed symbolically as the expression $A(e)$ defined inductively, e.g., $A(e.f) = f[A(e)]$. A heap update $h := e$ then is symbolically processed as the array assignment $A(h) = A(e)$. Thus we have reduced the object-oriented language to the language with recursive procedures (as introduced in Section 3) extended with arrays. The main question then remains how to update the substitution representing a concrete state in case of an assignment $h := new\ C$. We can do so by simply processing such an assignment symbolically in terms of a statement

$$h := x;\ x.f_1 := nil;\ \ldots x.f_n := nil$$

where x is a *fresh* variable and $f_1, \ldots, f_n$ are all the fields declared in class C. We can remove again occurrences of these fresh variables in the generated path condition by applying the substitutions $[x := new\ C]$, as defined above but now reformulated in terms of arrays, e.g., the clause for $(x.f)[x := new\ C]$ becomes

$$(f[x])[x := new\ C] = nil.$$

5. Conclusion

Despite the popularity and success of symbolic execution techniques, to the best of our knowledge, a general theory of symbolic execution is missing which covers in an uniform manner mainstream programming features like arrays and (object-oriented) pointer structures, as well as local scoping as it arises in the passing of parameters in recursive procedure calls. In fact, most existing tools for symbolic execution lack an explicit formal specification and justification.

In this paper we provided a detailed discussion of the on-the-fly (or forward) symbolic generation of path conditions, using a symbolic representation of the concrete program state. We discussed main (object-oriented) programming features, including arrays. In fact, we showed how to model symbolically arrays as mathematical functions and object structures as arrays.

We further introduced a new, more fundamental approach which computes the path condition by means of a weakest pre-conditon calculus ([BK94]), and as such avoids the use of a symbolic representation of the concrete program state. This approach (introduced in Section 4) is based on symbolic execution traces which are obtained from unfolding statically the control flow of a program, accumulating the basic instructions , e.g., assignments and Boolean tests. As such these traces abstract from the specific control structures of the programming language. In the second phase, the application of a weakest precondition calculus for these basic instructions allows to generate a path condition from a symbolic execution trace.

It should be noted that the notion of a path condition as a Boolean condition on the initial state which ensures the concrete execution of the program along a given path is only applicable to *deterministic* languages. However, our approach based on symbolic execution traces is applicable also to non-deterministic languages, e.g., multi-threaded shared variable programs. For example, in [dBBJ+20] information about the non-deterministic choices, e.g., scheduling information, is naturally integrated in such traces for multi-threaded shared variable programs. In [dBBJ+20] this information is further used for partial order reduction of the exploration of the symbolic execution traces.

In general, symbolic execution traces can be subject to different kinds of analysis: e.g., the path condition, the symbolic representation of the concrete program states, and partial-order reduction. Its abstraction from control structures also allows to simulate the actual execution of a program by executing instead the symbolic representation of the final concrete program state encoded in such a trace. This also allows to combine deductive

verification and testing: for a path condition ϕ and a substitution σ representing the final concrete program state of a symbolic execution trace, instead of executing the program for a given initial state which satisfies ϕ and testing whether the final state satisfies a condition ψ, we can also *verify* the implication $\phi \rightarrow (\psi\sigma)$.

A further illustration of the generality of our approach is its application to concurrent objects as, for example, in the Abstract Behavioral Specification (ABS) language [JHS+12] which describes systems of objects that interact via asynchronous method calls. Such a call spawns a corresponding process associated with the called object. Return values are communicated via futures [dBCJ07]. Each object cooperatively schedules its processes one at a time. The processes of an object can only access their local variables and the instance variables (fields) of the object. Symbolically, the semantics can be described analogously to the one in Section 4. To model the communication of the return values by futures we can introduce for each process a distinguished local variable.

The major potential practical implication of computing path conditions as the weakest precondition of a symbolic trace is that it supports a clear separation of concerns between the generation of symbolic traces by unfolding the program and the computation of the path condition by means of a weakest precondition calculus. The use of a backward calculation of path conditions by a weakest precondition calculus further avoids the (symbolic) state-space explosion due to various forms of aliasing. To investigate the practical implications of our work, we will develop prototype implementations to compare the performance between the two different formal models of symbolic execution discussed in this paper. These prototype implementations will further be used to compare performance with other tools, and investigate optimizations.

Another interesting research direction is the development of a further extension of our theory for concolic execution, mixing symbolic and concrete executions [GKS05], and the symbolic backward execution [CFS09].

Acknowledgements

This work arose out of our *Foundation of Testing* master course (LIACS) in 2018, and we thank the master students for their valuable comments. We thank the anonymous reviewers for their valuable comments.

References

[AAGR14] Albert Elvira, Arenas Puri, Gómez-Zamalloa Miguel, Rojas José Miguel (2014) Test case generation by symbolic execution: Basic concepts, a clp-based instance, and actor-based concurrency. In: Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer (eds.), Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures, volume 8483 of Lecture Notes in Computer Science. Springer, pp 263–309

[ABB+16] Ahrendt Wolfgang, Beckert Bernhard, Bubel Richard, Hähnle Reiner, Schmitt Peter H, Ulbrich Mattias, (eds.) (2016) Deductive Software Verification - The KeY Book - From Theory to Practice, volume 10001 of Lecture Notes in Computer Science. Springer

[AdBO09] Apt Krzysztof R, de Boer Frank, Olderog Ernst-Rdiger (2009) Verification of Sequential and Concurrent Programs, 3rd edition. Springer

[BCD+18] Baldoni Roberto, Coppa Emilio, D'elia Daniele Cono, Demetrescu Camil, Finocchi Irene (2018) A survey of symbolic execution techniques. ACM Computing Surveys 51(3)

[BDP15] Braione Pietro, Denaro Giovanni, Pezzè Mauro (2015) Symbolic execution of programs with heap inputs. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), pp 602–613

[BK94] Bonsangue Marcello M, Kok Joost N (1994) The weakest precondition calculus: Recursion and duality. Formal Aspect of Computing 6(1):788–800

[CDE08] Cadar Cristian, Dunbar Daniel, Engler Dawson (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008). USENIX Association, pp 209–224

[CFS09] Chandra Satish, Fink Stephen J, Sridharan Manu (2009) Snugglebug: a powerful approach to weakest preconditions. In: Hind Michael, Diwan Amer (eds.), Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009). ACM, pp 363–374

[CGP+08] Cadar Cristian, Ganesh Vijay, Pawlowski Peter M, Dill David L, Engler Dawson R (2008) Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security 12(2):10:1–10:38

[dB99] de Boer Frank S (1999) A wp-calculus for OO. In: Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings, pp 135–149

[dBB19] de Boer Frank S, Bonsangue Marcello (2019) On the nature of symbolic execution. In: ter Beek Maurice H, McIver Annabelle, Oliveira José N (eds.) Formal Methods – The Next 30 Years, Lecture Notes in Computer Science. Springer, pp 64–80

[dBBJ+20] de Boer Frank, Bonsangue Marcello, Johnsen Einar Broch, Pun Violet Ka I, Tarifa S Lizeth Tapia, Tveito Lars (2020) Sympaths: Symbolic execution meets partial order reduction. In: Bubel R, Hähnle R, et al. (eds.), Deductive Verification: The Next 70 Years. Springer

[dBCJ07] de Boer Frank S, Clarke Dave, Johnsen Einar Broch (2007) A complete guide to the future. In: De Nicola Rocco (ed.), Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP 2007), volume 4421 of Lecture Notes in Computer Science, pp 316–330. Springer

[DLR06] Deng Xianghua, Lee Jooyong, and Robby (2006) Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). IEEE Computer Society, pp 157–166

[EGL09] Elkarablieh Bassem, Godefroid Patrice, Levin Michael Y (2009) Precise pointer reasoning for dynamic test generation. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA 2009). ACM, pp 129–140

[FLP17] Fromherz Aymeric, Luckow Kasper S, Păsăreanu Corina S (2017) Symbolic arrays in symbolic pathfinder. SIGSOFT Softw. Eng. Notes 41(6):1–5

[GKS05] Godefroid Patrice, Klarlund Nils, Sen Koushik (2005) DART: directed automated random testing. In: Sarkar Vivek, Hall Mary W (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005). ACM, pp 213–223

[Gri81] Gries David (1981) The Science of Programming. Texts and Monographs in Computer Science. Springer

[JHS+12] Johnsen Einar Broch, Hähnle Reiner, Schäfer Jan, Schlatte Rudolf, Steffen Martin (2012) ABS: A core language for abstract behavioral specification. In: Aichernig Bernhard K, de Boer Frank S, Bonsangue Marcello M (eds.) Formal Methods for Components and Objects - 9th International Symposium (FMCO 2010), volume 6957 of Lecture Notes in Computer Science. Springer, pp 142–164

[KC19] Kapus Timotej, Cadar Cristian (2019) A segmented memory model for symbolic execution. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, pp 774–784

[Kin76] King James C (1976) Symbolic execution and program testing. Communications of ACM 19(7):385–394

[LRA17] Lucanu Dorel, Rusu Vlad, Arusoaie Andrei (2017) A generic framework for symbolic execution. Journal of Symbolic Computation 80(1):125–163

[PMZC17] Perry David M, Mattavelli Andrea, Zhang Xiangyu, Cadar Cristian (2017) Accelerating array constraints in symbolic execution. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017). ACM, pp 68–78

[SAB10] Schwartz Edward J, Avgerinos Thanassis, Brumley David (2010) All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, page 317–331, USA, 2010. IEEE Computer Society

[TS14] Trtík Marek, Strejček Jan (2014) Symbolic memory with pointers. In: Cassez Franck, Raskin Jean-François (eds.) Proc. Symp. on Automated Technology for Verification and Analysis (ATVA 2014), volume 8837 of Lecture Notes in Computer Science. Springer, pp 380–395

[XMSN05] Xie Tao, Marinov Darko, Schulte Wolfram, Notkin David (2005) Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Proceeding of the international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), volume 3440 of Lecture Notes in Computer Science. Springer, pp 365–381

Received 10 March 2020

Accepted in revised form 11 December 2020 by Annabelle McIver, Maurice ter Beek and Cliff Jones

Published online 2 February 2021