On the Nature of Symbolic Execution

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

[1] Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
f.s.de.boer@cwi.nl
[2] Leiden Institute Of Advanced Computer Science (LIACS), Leiden, The Netherlands
m.m.bonsangue@liacs.leidenuniv.nl

Abstract. In this paper, we provide a formal definition of symbolic execution in terms of a symbolic transition system and prove its correctness with respect to an operational semantics which models the execution on concrete values. We first introduce such 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 show how to extend this latter model of symbolic execution to arrays and object-oriented languages which feature dynamically allocated variables.

1 Introduction

Symbolic execution [1] plays a crucial role in modern testing techniques, debugging, and automated program analysis. In particular, it is used for generating test cases [2,3].

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 [3].

Further, there exists a pletora 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 [15] a heap is modeled as a graph, with nodes drawn from a set of objects and updated lazily, whereas [5] introduces a constraint language for the specification of invariant properties of heap structures. In [11] 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 [6]. Powerful symbolic execution tools [7–9] 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

M. H. ter Beek et al. (Eds.): FM 2019, LNCS 11800, pp. 64–80, 2019.
https://doi.org/10.1007/978-3-030-30942-8_6

experimentally by testing [10]. The KeY theorem prover [18] 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.

The main contribution of this paper is a formal definition of symbolic execution in terms of a symbolic transition system and a general definition of its correctness with respect to an operational semantics which models the actual execution on concrete values. Our general starting point is that the basic idea of symbolic execution is to represent the program state, i.e., the assignment of values to program variables, by a corresponding substitution which assigns to each program variable an expression denoting its current value. Further, symbolic execution by its very nature is *syntax-directed* which implies that the *abstraction level* of the symbolic transition system should coincide with that of the programming language. This general requirement implies that symbolic execution operates on substitutions which (only) assign *programming* expressions to the variables (and no other expressions which express properties of the run-time).

The only other approach to a formal modeling of symbolic execution, we are aware of, is the work presented in [4]. A major difference with our approach is that in [4] 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 [4], 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 Sect. 2 we introduce a formal model of symbolic execution for a basic programming language with a statically fixed number of programming variables. The concrete transition system for this basic language is standard (and therefore omitted). 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 Sect. 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 Sect. 4 we show how to extend symbolic execution in a strictly syntax-directed manner to an object-oriented language which features dynamically allocated variables. These dynamically allocated variables give rise to an *infinite* number of program variables and corresponding substitutions with an infinite domain. We show how to extend our theory of symbolic execution to such infinite substitutions. Moreover, we introduce for a correct implementation a finite representation of these substitutions, and discuss different strategies for managing aliasing.

In the final technical Sect. 5 (unbounded) arrays, multithreading, and concurrent objects are discussed as a further illustration of the generality of our theory of symbolic execution.

Because of space limitations, in this paper we do not introduce all syntactic details of the programming languages we use, which however should be clear via their transition system semantics.

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. 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 $op \in Ops$. 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 Boolean condition ϕ denotes the path condition.

Next we describe a transition system for the symbolic execution of a simple programming language which features assignments, sequential composition, a choice and iteration statement.

Assignment

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

where $\sigma[x = e](y) = \sigma(y)$ if x and y are distinct variables, and $\sigma[x = e](x) = e$ otherwise.

Choice

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

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 formalize and prove correctness with respect to a concrete semantics. A *valuation* V is a function $\mathit{Var} \to \mathit{Val}$, where Val is a set of *values*. 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 $\overline{op}$ denotes the interpretation of the operation op as provided by the implicitly assumed underlying model. Composition is as usual: $(V \circ \sigma)(x) = V(\sigma(x))$[1].

Lemma 1 (Substitution). $V \circ \sigma(e) = V(e\sigma)$.

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

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

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 and therefore omitted.

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

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

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

Proof. Induction on the length of $\langle S, id, \mathit{true}\rangle \to^* \langle S', \sigma, \phi\rangle$ 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:

$$\langle S, id, \mathit{true}\rangle \to^* \langle x = e; S', \sigma, \phi\rangle \to \langle S', \sigma[x = e\sigma], \phi\rangle$$

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

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

[1] In the sequel we omit the parentheses and write $V \circ \sigma(e)$ for the application valuation $V \circ \sigma$ to the expression e (as defined above).

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

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

where $V'[x = V'(e)](x) = V'(e)$ and $V'[x = V'(e)](y) = V'(y)$, for any other variable y. Suffices to show $V \circ (\sigma[x = e\sigma]) = V'[x = V'(e)]$. We treat the main case:

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

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

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

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

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

Since $V \circ \sigma(B) = V(B\sigma) = \text{true}$, 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 1 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 [4], meaning that *satisfiable* symbolic execution paths can be simulated by concrete executions. The converse of correctness is completeness and is about precision [4]: every concrete execution can be simulated by a symbolic one. To this end we introduce the following relation between symbolic and the concrete transition systems: Let $\langle S, V\rangle \simeq \langle S, \sigma, \phi\rangle$ denote that $V = V_0 \circ \sigma$ and $V_0(\phi) = \mathit{true}$, for some valuation V_0.

Theorem 2 (Completeness). *The relation $\simeq$ between symbolic and concrete configurations is a* simulation *relation, i.e., if $\langle S, V\rangle \simeq \langle S, \sigma, \phi\rangle$ then $\langle S, V\rangle \rightarrow \langle S', V'\rangle$ implies the existence of a corresponding symbolic transition $\langle S, \sigma, \phi\rangle \rightarrow \langle S', \sigma', \phi'\rangle$ such that $\langle S', V'\rangle \simeq \langle S', \sigma', \phi'\rangle$.*

The proof of this theorem proceeds by a straightforward case analysis of the concrete execution steps.

3 Recursion

We extend the basic programming language with procedure declarations $P(\bar{u}) :: S$ and procedure calls $P(\bar{e})$, assuming a call by value parameter passing mechanism. A program then consists of set of procedure declarations and a main statement. We assume absence of name clashes between the global variables of a program and its local variables (e.g., the formal parameters of the procedure declarations). A *symbolic configuration* is of the form $\langle \Sigma, \sigma, \phi \rangle$, where

- Σ denotes the stack of *closures* (τ, S), where τ is a *local* substitution (assigning expressions to formal parameters),
- σ is the current *global* substitution (mapping expressions to global variables),
- ϕ is a Boolean condition denoting the path condition.

In the sequel we indicate by $\perp$ the absence of local variables in a closure $(\perp, S)$ which represents a continuation of the execution of the main statement which does not contain local variables (for technical convenience we do not consider the introduction of local variables by block statements). By $\tau \cup \sigma$ we denote the union of the substitutions τ and σ (defined in terms of their graphs). This is well-defined because of the absence of name clashes between local and global variables of a programs. We have the following symbolic transitions.

Procedure Call. Given the procedure declaration $P(\bar{u}) :: S'$, we have

- $\langle (\tau, P(\bar{e}); S) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau', S') \cdot (\tau, S) \cdot \Sigma, \sigma, \phi \rangle$, where $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$.

Procedure Return

- $\langle (\tau, \epsilon) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle \Sigma, \sigma, \phi \rangle$, where ϵ denotes the empty statement.

Choice

- $\langle (\tau, \mathit{if}\ B\ \{S_1\}\{S_2\}; S) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau, S_1; S,) \cdot \Sigma, \sigma, \phi \wedge B(\tau \cup \sigma) \rangle$
- $\langle (\tau, \mathit{if}\ B\ \{S_1\}\{S_2\}; S) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau, S_2; S) \cdot \Sigma, \sigma, \phi \wedge \neg B(\tau \cup \sigma) \rangle$

Iteration

- $\langle (\tau, \mathit{while}\ B\ \{S\}; S') \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau, S; \mathit{while}\ B\ \{S\}; S') \cdot \Sigma, \sigma, \phi \wedge B(\tau \cup \sigma) \rangle$
- $\langle (\tau, \mathit{while}\ B\ \{S\}; S') \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau, S') \cdot \Sigma, \sigma, \phi \wedge \neg B(\tau \cup \sigma) \rangle$

Assignment Global Variable Let x be a global variable.

- $\langle (\tau, x = e; S) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau, S) \cdot \Sigma, \sigma[x = e\theta], \phi \rangle$, where $\theta = \tau \cup \sigma$.

Assignment Local Variable Let x be a local variable.

- $\langle (\tau, x = e; S) \cdot \Sigma, \sigma, \phi \rangle \rightarrow \langle (\tau[x = e\theta], S) \cdot \Sigma, \sigma, \phi \rangle$, where $\theta = \tau \cup \sigma$.

Proposition 1. *For any computation* $\langle(\bot, S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ *where* S *denotes the main statement, we have that* $\tau(x)$*, for every local variable* x *in its domain, and* $\sigma(x)$*, for every global variable* x*, does* not *contain local variables.*

Proof. By induction on the length of the computation, using that $e\theta$, for any programming expression e, does not contain local variables, where $\theta = \tau \cup \sigma$ and $\tau(x)$, for every local variable x, and $\sigma(x)$, for every global variable x, does not contain local variables.

Corollary 1. *For any computation* $\langle(\bot\ S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ *where* S *denotes the main statement, the generated path condition* ϕ *does not contain local variables.*

We omit the details of the standard concrete semantics which instead of substitutions is defined in terms of valuations both for the local variables and the global variables, where $V \cup L(e)$ denotes the result of the evaluation of the expression e in the global valuation V and the local valuation L. We have the following correctness theorem of the symbolic execution of recursive programs.

Theorem 3 (Correctness). *If* $\langle(\bot, S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ *and* $V(\phi) = true$ *then*

$$\langle(\bot, S), V\rangle \rightarrow^* \langle(V \circ \tau, S') \cdot V \circ \Sigma, V \circ \sigma\rangle$$

where $V \circ \tau'(x) = V(\tau'(x))$*, for any local environment* τ' *and local variable* x*, and* $V \circ \Sigma$ *denotes the result of replacing every local environment* τ' *in* Σ *by* $V \circ \tau'$ *(and by* $\bot$*, in case of the empty local environment* $\bot$*). Note that by the above proposition and corollary* $\tau(x)$ *and* ϕ *do not contain local variables.*

Proof. As above, we proceed by induction on the length of the symbolic computation and a case analysis of the last execution step. Given the procedure declaration $P(\bar{u}) :: S'$, we consider the case of a procedure call:

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

where $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$. By the induction hypothesis we obtain the concrete computation

$$\langle(\bot, S), V\rangle \rightarrow^* \langle(V \circ \tau, P(\bar{e}); S'') \cdot V \circ \Sigma, V \circ \sigma\rangle$$

We observe that $V \circ \tau'(\bar{u}) = V(\bar{e}(\tau \cup \sigma)) = V \circ (\tau \cup \sigma)(\bar{e}) = (V \circ \tau \cup V \circ \sigma)(\bar{e})$. So we obtain that

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

As in the basic case, we have a similar completeness result for recursive procedures also expressed in terms of a simulation relation between the symbolic and the concrete transition system.

4 Object Orientation

We distinguish between the global variables appearing in the main statement, the local variables (i.e., the formal parameters of methods, including the keyword *this*), and the instance variables (of the classes) of the given program. For modeling symbolically the dynamic creation of new objects, we assume a (countable) infinite set of global variables. We abstract from the typing information of the variables. We have the following syntax of programming expressions e in class definitions

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

where x is a local or instance variable[2] and op denotes a built-in operation.

The syntax of *heap variables* H and *heap expressions* E is defined by the following grammar:

$$\begin{aligned} H &:= x \mid H.y \\ E &:= H \mid op(E_1, \ldots, E_n)\,, \end{aligned}$$

where x is a global variable and op an operation. In the last clause defining heap variables we implicitly assume that y is an instance variable of the class of the object represented by H.

A *symbolic heap* σ is a substitution which assigns to each heap variable a heap expression. A *local environment* (of a given method) τ is a substitution which assigns to each formal parameter a general heap expression. Note that thus $\tau(x)$ does not contain local variables.

We have the following inductive definition of the application of a substitution θ which consists of the union $\tau \cup \sigma$ of a symbolic heap σ and a local environment τ to a programming expression e (as above, assuming absence of name clashes between the formal parameters, on the one hand, and the instance and global variables, on the other hand).

$$\begin{array}{lll} x\theta & = \tau(x) & \text{local variable} \\ x\theta & = \sigma(\tau(this).x) & \text{instance variable} \\ op(E_1, \ldots, E_n)\theta & = op(E_1\theta, \ldots, E_n\theta) & \end{array}$$

A *heap update* $\sigma[x = E]$, where x is a global variable, is defined by $\sigma[x = E](x) = E$ and $\sigma[x = E](H) = \sigma(H)$, for any other heap variable H. Next we define a *symbolic heap update* $\sigma[H.x = E]$ by

- $\sigma[H.x = E](H'.x) = \mathit{if}\ \sigma(H') = \sigma(H)\ \mathit{then}\ E\ \mathit{else}\ \sigma(H'.x)\ \mathit{fi}$,
- $\sigma[H.x = E](H') = \sigma(H')$, *for any other heap variable H'.*

It is important to note that the resulting expression in the first clause is a *conditional* heap expression which captures possible aliases. Note further that the case $\sigma[H.x = E](H.x)$ simplifies to E.

[2] In the main statement only global variables are used.

Given a program, i.e., a set of class definitions and a main statement, a symbolic configuration is defined as above. We have the following symbolic transitions (the transitions for the assigning local variables, the choice and iteration constructs are as above).

Assignment Global Variable. As above, absence of local variables in the main statement is indicated by $\bot$.

- $\langle(\bot, x = e; S), \sigma, \phi\rangle \rightarrow \langle(\bot, S), \sigma[x = e\sigma], \phi\rangle$
  Note that a closure representing the execution of the main statement is always at the bottom of the stack.

Assignment Instance Variable

- $\langle(\tau, x = e; S) \cdot \Sigma, \sigma, \phi\rangle \rightarrow \langle(\tau, S) \cdot \Sigma, \sigma[\tau(this).x = e\theta], \phi\rangle$, where $\theta = \tau \cup \sigma$.

Object Creation. We describe the assignment of a new object to a local variable[3].

- $\langle(\tau, x = \text{new } C; S) \cdot \Sigma, \sigma, \phi\rangle \rightarrow \langle(\tau[x = y], S) \cdot \Sigma, \sigma', \phi\rangle$

The global variable y (of type C) is *fresh* and σ' results from σ by assigning nil to $y.x$, x an instance variable of C. Freshness is defined with respect to the computation (which thus requires recording the set of new global variables introduced so far, the details of which are straightforward and therefore omitted).

Method Call. Given a method declaration $m(\bar{u})\{S\}$, we have

- $\langle(\tau, y = e_0.m(\bar{e}); S') \cdot \Sigma, \sigma, \phi\rangle \rightarrow \langle(\tau'.S) \cdot (\tau, y =?; S') \cdot \Sigma, \sigma, \phi'\rangle$,
  where $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$ and $\tau'(this) = e_0(\tau \cup \sigma)$. The question mark in the assignment $y =?$ serves as a placeholder of the return expression (see below). Further, ϕ' denotes the path condition $\phi \wedge e_0(\tau \cup \sigma) \neq nil$.

Method Return

- $\langle(\tau, \text{return e}) \cdot (\tau', x =?; S) \cdot \Sigma, \sigma, \phi\rangle \rightarrow \langle(\tau'[x = e\theta], S) \cdot \Sigma, \sigma, \phi\rangle$,
  where $\theta = (\tau \cup \sigma)$. For an assignment of the return expression to an instance variable we have a similar transition.

We have the following basic proposition about the expressions generated by a symbolic computation.

Proposition 2. *For any computation* $\langle(\bot, S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ *where* S *denotes the main statement, we have that* $\tau(x)$ *and* $\sigma(x)$ *do not contain local variables and do not dereference a global variable that does not occur in the main statement (i.e., a global variable used to denote a newly created object).*

Proof. The proof proceeds by a straightforward induction on the length of the symbolic computation.

[3] We model a call $x = \text{new } C(\bar{e})$ of a constructor method by the object creation statement $x = \text{new } C$ followed by a method call $x.C(\bar{e})$.

Corollary 2. *For any computation* $\langle(\bot\ S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ *where* S *denotes the main statement, the generated path condition* ϕ *does not contain local variables and derefences only global variables appearing in the main statement (in other words, global variables which do* not *appear in the main statement are* not *dereferenced).*

In order to define and prove the correctness of the above symbolic transition system for object-oriented programs, we first introduce the notion of a global valuation V which assigns to each heap variable a value (of the corresponding type) and which satisfies the following two conditions:

- $V(H) = V(H')$ implies $V(H.x) = V(H'.x)$, for every heap variables H and H' and instance variable x (belonging to the class of the object).
- $V(x) \neq V(x')$, for any two distinct global variables x and x' which do not appear in the main statement (*unique name assumption*).

A concrete local valuation L assigns to every formal parameter x of the corresponding method a value $L(x)$ (of the appropriate type).

An update $V[x = v]$ of a global valuation V, where x is a global variable and v a value of corresponding type, is defined by $V[x = v](x) = v$ and $V[x = v](H) = V(H)$, for any other heap variable H. On the other hand, an update $V[H.x = v]$ is defined by

- $V[H.x = v](H'.x) = \begin{cases} v & \text{if } V(H') = V(H) \\ V(H'.x) & \text{otherwise} \end{cases}$
- $V[H.x = v](H') = V(H')$, for any other heap variable H'.

An initial configuration $\langle(\bot, S), V\rangle$ of the concrete semantics of a given program consists of the main statement S (as above, $\bot$ indicates the absence of local variables) and an *initial* global valuation V such that for any global variable x which does not appear in the main statement and heap variable H rooted in a global variable we have that $V(x) \neq V(H)$. Any global variable is a heap variable rooted in a global variable, and if H is such a heap variable, so is $H.x$, for any instance variable x. We thus can use these initially unreachable objects in the concrete semantics as a repository of fresh object identities (which are selected non-deterministically, as the fresh global variables in the symbolic semantics). Since every executing object is reachable (from a global variable) we can define the concrete semantics of an assignment $x = e$ to an instance variable e as follows. Given the above update of a global valuation V and a local environment L (which assigns values to the local variables of the executing method), we can define the resulting global valuation of the execution of the assignment $x = e$ by the object $L(\mathit{this})$ by $V[H.x = v]$, where H is such that $V(H) = L(\mathit{this})$, and v is the result of evaluating the expression e in the local environment L and the valuation V.

We omit the further details of the standard concrete semantics (which thus, instead of substitutions, is defined in terms of valuations both for the local variables and the heap variables).

We have the following correctness theorem of the symbolic execution of object-oriented programs.

Theorem 4 (Correctness). *Given an object-oriented program with main statement S we have that if $\langle(\bot, S), id, \text{true}\rangle \rightarrow^* \langle(\tau, S') \cdot \Sigma, \sigma, \phi\rangle$ and $V(\phi) =$ true, where V is an initial valuation, then*

$$\langle(\bot, S), V\rangle \rightarrow^* \langle(V \circ \tau, S') \cdot V \circ \Sigma, V \circ \sigma\rangle$$

where $V \circ \tau$ and $V \circ \Sigma$ are defined as above.

Proof. As above, the proof proceeds by induction on the length of the symbolic computation and a case analysis of the last execution step. For the case of an assignment $x = e$ to an instance variable, it suffices to show that

$$V \circ (\sigma[\tau(\mathit{this}).x = e\theta]) = (V \circ \sigma)[\tau(\mathit{this}).x = V(e\theta)]$$

where $\theta = \tau \cup \sigma$. Here we go: First, for any heap variable H not of the form $H'.x$, for some H', we have

$$\begin{array}{lll} V \circ (\sigma[\tau(\mathit{this}).x = e\theta])(H) & = & V(\sigma[\tau(\mathit{this}).x = e\theta](H)) \quad (\text{def.} \circ) \\ & = & V(\sigma(H)) \quad (\text{def. update } \sigma[\tau(\mathit{this}).x = e\theta]) \\ & = & (V \circ \sigma)(H) \quad (\text{def.} \circ) \\ & = & (V \circ \sigma)[\tau(\mathit{this}).x = V(e\theta)](H) \quad (\text{def. update}(\mathrm{V} \circ \sigma)[\tau(\mathit{this}).x = V(e\theta)]) \,. \end{array}$$

Further,

$$\begin{array}{l} V \circ (\sigma[\tau(\mathit{this}).x = e\theta])(H.x) \\ = \quad (\text{def.} \circ) \\ V(\sigma[\tau(\mathit{this}).x = e\theta](H.x)) \\ = \quad (\text{def. update } \sigma[\tau(\mathit{this}).x = e\theta]) \\ V(\mathit{if}\ \sigma(H) = \sigma(\tau(\mathit{this}))\ \mathit{then}\ e\theta\ \mathit{else}\ \sigma(H.x)\ \mathit{fi}) \\ = \quad (\text{semantics of conditional heap expression}) \\ \text{if}\ V(\sigma(H)) = V(\sigma(\tau(\mathit{this})))\ \mathit{then}\ V(e\theta)\ \mathit{else}\ V(\sigma(H.x)) \\ = \quad (\text{def. update } (V \circ \sigma)[\tau(\mathit{this}).x = V(e\theta)]) \\ (V \circ \sigma)[\tau(\mathit{this}).x = V(e\theta)](H.x) \,. \end{array}$$

Again, completeness can be established by means of a simulation relation between the symbolic transition system and the concrete one.

Implementation

An implementation of the above symbolic execution of object-oriented programs requires a finite representation of the generated substitutions (note that we have a countable infinite set of heap variables). As an example, we can represent the generated substitutions by (possibly empty) sequences $\{H_1.x_1 = E_1\}, \ldots, \{H_n.x_n = E_1\}$ of updates of instance variables. Such a sequence ρ simply represents the substitution $\bar{\rho}$ which results from the identity substitution id by applying the updates $[H_1, x_1 = E_1], \ldots, [H_n.x_n = E_n]$, as defined above, consecutively. For such a sequence of updates ρ we define $\rho(H)$ inductively by

- $id(H) = H$,
- $\rho\{H.x = E\}(H'.x) = \mathit{if}\ \rho(H') = \rho(H)\ \mathit{then}\ E\ \mathit{else}\ \rho(H'.x)\ \mathit{fi}$,
- $\rho\{H.x = E\}(H') = \rho(H')$, for any other heap variable H'.

It is easy to prove by induction on the length of the sequence ρ that $\rho(H) = \bar{\rho}(H)$, for every heap variable. We then can define $e(\tau \cup \rho)$ in the same manner as $e(\tau \cup \sigma)$ defined above. Thus we can now define the following transition for an assignment to an instance variable

$$\langle (\tau, x = e; S) \cdot \Sigma, \rho, \phi \rangle \rightarrow \langle (\tau, S) \cdot \Sigma, \rho\{\tau(\mathit{this}).x = e\theta\}, \phi \rangle$$

where ρ is a sequence of updates and $\theta = \tau \cup \rho$. In general, we abstract from the infinite number of heap variables by simply replacing in the symbolic transitions the substitution σ by the sequence of updates ρ. In particular, we have the following adaptation of the transition describing object creation

$$\langle (\tau, x = \text{new } C; S) \cdot \Sigma, \rho, \phi \rangle \rightarrow \langle (\tau[x = y], S) \cdot \Sigma, \rho', \phi \rangle$$

where, as above, the global variable y (of type C) is *fresh* and ρ' results from ρ by adding the updates $\{y.x = \text{nil}\}$, for every an instance variable x of C.

Other implementation issues concern the aliasing of heap variables. There are various ways to manage (resolve) aliasing. We briefly describe the following enhancements. First, we can import information from the path condition ϕ into the evaluation of $\rho(H)$:

$$\rho\{H.x = E\}(H'.x) = \begin{cases} E & \text{if } \phi \vdash \rho(\mathrm{H}) = \rho(\mathrm{H}') \\ \rho(H'.x) & \text{if } \phi \vdash \rho(\mathrm{H}) \neq \rho(\mathrm{H}') \\ \mathit{if}\ \rho(H') = \rho(H)\ \mathit{then}\ E\ \mathit{else}\ \rho(H'.x)\ \mathit{if} & \text{otherwise} \end{cases}$$

Here $\vdash$ denotes logical entailment.

Further, there are various ways of branching the symbolic execution by resolving aliasing of heap variables. For example, we can resolve aliasing in the symbolic transition of an assignment $x = e$ as follows:

$$\langle (\tau, x = e; S) \cdot \Sigma, \rho, \phi \rangle \rightarrow \langle (\tau, S) \cdot \Sigma, \rho\{\tau(\mathit{this}).x = E\}, \phi' \rangle$$

where $\langle e(\tau \cup \rho), \phi \rangle \Rightarrow^* \langle E, \phi' \rangle$ and $\Rightarrow^*$ denotes the reflexive, transitive closure of the rewrite system consisting of the rules which resolve conditional expression, like

- $\langle op(\ldots, \mathit{if}\ B\ \mathit{then}\ E_1\ \mathit{else}\ E_2 \mathit{fi}, \ldots), \phi \rangle \Rightarrow \langle op(\ldots, E_1, \ldots), B \wedge \phi \rangle$
- $\langle op(\ldots, \mathit{if}\ B\ \mathit{then}\ E_1\ \mathit{else}\ E_2\ \mathit{fi}, \ldots), \phi \rangle \Rightarrow \langle op(\ldots, E_2, \ldots), \neg B \wedge \phi \rangle$

In a similar manner, we can resolve aliasing which results from the symbolic evaluation of the Boolean condition of the choice and iteration constructs. For example, we have the following symbolic transition for the choice construct.

$$\langle (\tau, \mathit{if}\ B\ \{S_1\}\{S_2\}; S) \cdot \Sigma, \rho, \phi \rangle \rightarrow \langle (\tau, S_1; S,) \cdot \Sigma, \rho, B' \wedge \phi' \rangle$$

where $\langle B(\tau \cup \rho), \phi \rangle \Rightarrow^* \langle B', \phi' \rangle$.

5 Arrays, Multithreading, and Concurrent Objects

To illustrate the generality of our theory of symbolic execution we discuss the following extensions and applications.

Arrays. Arrays and object structures (i.e., heaps) are similar because both give rise to a (countable) infinite number of program variables. Instead of an infinite number of heap variables, arrays give rise to an infinite number of so-called *subscripted* variables.

To focus on the main ideas, we restrict this discussion to the extension of the basic programming language with one-dimensional arrays. We have the following syntax of *expressions* e in the basic programming language (abstracting from the typing information).

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

where $x \in Var$, a is an array variable, and op denotes a built-in operation.

A *substitution* then assigns to each (subscripted) variable an expression. An update $\sigma[x = e]$, where x is a program variable, is defined by $\sigma[x = e](x) = e$ and $\sigma[x = e](y) = \sigma(y)$, for any other (subscripted) variable y. Next we define a symbolic update $\sigma[a[e] = e']$ by

- $\sigma[a[e] = e'](a[e'']) = \mathit{if}\ \sigma(e) = \sigma(e'')\ \mathit{then}\ e'\ \mathit{else}\ \sigma(a[e'])\ \mathit{if}$,
- $\sigma[a[e] = e'](y) = \sigma(y)$, for any other (subscripted) variable y.

As above, it is important to note that the resulting expression in the first clause is a *conditional* expression which captures possible aliases.

Given this definition of a symbolic update we can define in a straightforward manner a symbolic transition system for the basic programming language extended with arrays (possibly taking into account symbolically array bounds). Correctness then is defined with respect to the notion of a global valuation V which assigns to each (subscripted) variable a value (of the corresponding type) and which satisfies the following condition:

- $V(e) = V(e')$ implies $V(a[e]) = V(a[e'])$.

It is straightforward to extend Theorem 1. In particular, correctness of a symbolic update of a subscripted variable then can be proved in a similar manner as that of a heap variable. Further, we can apply the same techniques as introduced for heap variables to obtain a finite representation of the generated substitutions and resolve aliasing.

We conclude this discussion on arrays with another approach which consists of a *functional view* of arrays (see [13]). In this view array variables themselves are expressions which denote functions, and a substitution assigns to each array variable an expression which denotes a function. Notably, an *expression* $(a[e] = e')$ denotes the function which results from updating the function denoted by a. Applying a substitution σ then amounts simply to subtituting every occurrence of an array variable a by the expression $\sigma(a)$. Similarly, object structures can

be viewed as a function h which symbolically represents the heap. However, the abstraction level of such a functional view does not coincide with that of the programming language (it extends the set of programming expressions).

Multithreading. It is straightforward to extend the symbolic transition system introduced above with multithreading: A symbolic configuration $\langle \mathit{Threads}, \sigma \rangle$ then consists of a set *Threads* of stacks of closures and local symbolic transitions of a single thread are extended to global transitions by the following rule:

$$\frac{\langle T, \sigma, \phi \rangle \to \langle T', \sigma', \phi' \rangle}{\langle \{T\} \cup \mathit{Threads}, \sigma, \phi \rangle \to \langle \{T'\} \cup \mathit{Threads}, \sigma', \phi' \rangle}$$

where T and T' denote stacks of closures. For a call of the run method of a thread class we need the following separate rule which spawns a new thread.

$$\begin{array}{c} \langle \{(\tau, e.\mathrm{run}; S) \cdot \mathit{Stack}\} \cup \mathit{Threads}, \sigma, \phi \rangle \\ \to \\ \langle \{(\tau, S) \cdot \mathit{Stack}, (\tau', S')\} \cup \mathit{Threads}, \sigma, \phi \rangle \end{array}$$

where $\tau'(\mathit{this}) = e(\tau \cup \sigma)$ and S' denotes the body of the run method. The proof of correctness is a straightforward extension of the correctness of the symbolic execution of sequential object-oriented programs (as stated by Theorem 4).

Concurrent Objects. We briefly sketch how to extend the symbolic execution of object-oriented programs to the Abstract Behavioral Specification (ABS) language [14] 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 [12]. 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 of the object. As in Sect. 4, we assume a main statement that only contains global variables.

Symbolically, a system of concurrent objects in ABS can be described by a configuration $\langle P, \sigma, \phi \rangle$, where P is simply a set of closures (τ, S) which represent the processes, and σ is a substitution, ϕ is a path condition, both as defined in Sect. 4. To model the communication of the return values by futures we introduce for each process a distinguished local variable *dest* which denotes its own future (see below).

We have the following symbolic transition for an asynchronous call $x = e_0!m(\bar{e})$ to a method m with body S:

$$\langle \{(\tau, x = e_0!m(\bar{e}); S)\} \cup P, \sigma, \phi \rangle \to \langle \{(\tau, x = y; S), (\tau', S')\} \cup P, \sigma[y = \mathit{nil}], \phi' \rangle$$

where the newly generated future is symbolically represented by a fresh global variable y which is initialized to *nil* (indicating that the return value has not yet been computed). Further, $\tau'(\bar{u}) = \bar{e}(\tau \cup \sigma)$, $\tau'(\mathit{this}) = e_0(\tau \cup \sigma)$, and $\tau'(\mathit{dest}) = y$. Finally, ϕ' denotes the path condition $\phi \wedge e_0(\tau \cup \sigma) \neq \mathit{nil}$.

For returning a value we have the transition

$$\langle \{(\tau, \mathrm{return}\ e)\} \cup P, \sigma, \phi \rangle \to \langle P, \sigma[\tau(\mathit{dest}) = e(\tau \cup \sigma)], \phi \rangle$$

Obtaining a returned value from a future by means of a "get" operation on a future variable is described by the transition

$$\langle\{(\tau, x = y.\text{get}); S)\} \cup P, \sigma, \phi\rangle \rightarrow \langle\{(\tau[x = \sigma(y)], S)\} \cup P, \sigma, \phi\rangle$$

where $\phi \vdash \sigma(y) \neq nil$ (that is, ϕ entails that $\sigma(y) \neq nil$). Note that this transition thus requires that the return value has been computed as recorded by the path condition.

Scheduling a process that is waiting on a Boolean condition is modeled by

$$\langle\{(\tau, \text{await } e; S)\} \cup P, \sigma, \phi\rangle \rightarrow \langle\{(\tau, S)\} \cup P, \sigma, \phi \wedge e\theta\rangle$$

where $\theta = \tau \cup \sigma$ and e is a Boolean condition. On the other hand, scheduling a process that is waiting on a future is modeled by

$$\langle\{(\tau, \text{await } x?; S)\} \cup P, \sigma, \phi\rangle \rightarrow \langle\{(\tau, S)\} \cup P, \sigma, \phi\rangle$$

where x is a future variable and $\phi \vdash \sigma(y) \neq nil$.

The transitions for the usual statements, e.g., that of assigning an instance variable of an object, are modeled in a straightforward manner after the corresponding transitions in Sect. 4. Assuming that all method bodies start with the awaittrue statement, we can globally constrain the scheduling of processes by the invariant that for each object there exists at most one process with an initial statement different from an await statement.

The concrete transition system for the ABS language can be defined as in Sect. 4 in terms of valuations. Here we use the initially unreachable objects in the concrete semantics additionally as a repository of fresh future identities. Again, the proof of correctness is a straightforward extension of the correctness of the symbolic execution of sequential object-oriented programs.

6 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 proposed such a general theory which covers the above mainstream programming features, and further illustrated the generality of our approach by its application to both multithreading and concurrent objects. From a practical point of view, we also illustrated how our theory sheds light on major implementation issues related to dynamically allocated variables and aliasing. This point of view we want to further explore by the development of proto-type implementations of the presented formal models of symbolic execution, 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 [16], and the symbolic backward execution [17].

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

1. King, C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
2. Albert, E., Arenas, P., Gómez-Zamalloa, M., Rojas, J.M.: Test case generation by symbolic execution: basic concepts, a CLP-based instance, and actor-based concurrency. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 263–309. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_7
3. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018)
4. Lucanu, D., Rusu, V., Arusoaie, A.: A generic framework for symbolic execution: a coinductive approach. J. Symbolic Comput. **80**(1), 125–163 (2017)
5. Braione, P., Denaro, G., Pezzè, M.: Symbolic execution of programs with heap inputs. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), pp. 602–613. ACM (2015)
6. Trtík, M., Strejček, J.: Symbolic memory with pointers. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 380–395. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_27
7. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006), pp. 322–335. ACM (2006)
8. Cadar, C., Dunbar, D., Engler, D.R.: 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), pp. 209–224, USENIX Association (2008)
9. Elkarablieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA 2009), pp. 129–140. ACM (2009)
10. Perry, D.M., Mattavelli, A., Zhang, X., Cadar, C.: Accelerating array constraints in symbolic execution. In Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA 2017), pp. 68–78. ACM (2017)
11. Deng, X., Lee, J.: 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), pp. 157–166 (2006)
12. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
13. Gries, D.: The Science of Programming. Texts and Monographs in Computer Science. Springer (1981)

14. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
15. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: a framework for generating object-oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 365–381. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_24
16. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 2005), pp. 213–223. ACM (2005)
17. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009), pp. 363–374. ACM (2009)
18. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbric, M.: Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6