

Interleaving Symbolic Execution and Partial Evaluation^{*}

Richard Bubel, Reiner Hähnle, and Ran Ji

Department of Computer Science and Engineering
Chalmers University, 41296 Gothenburg, Sweden
`bubel|reiner|ran.ji@chalmers.se`

Abstract. Partial evaluation is a program specialization technique that allows to optimize programs for which partial input is known. We show that partial evaluation can be used with advantage to speed up as well symbolic execution of programs. Interestingly, the input required for partial evaluation comes from symbolic execution itself which makes it natural to interleave partial evaluation and symbolic execution steps in a software verification setup.

1 Introduction

Symbolic execution [15] and partial evaluation [14] both are generalizations of standard interpretation of programs, however, they generalize in different ways: while symbolic execution permits interpretation of a program with symbolic (i.e., unspecified) initial values, the aim of partial evaluation is to transform a program with partially specified input values into a (hopefully, more efficient) program that has only the unspecified arguments as input. For fully specified input arguments the result of both mechanisms is standard program interpretation.

In this paper we show that both technologies not only are compatible with each other, but that there is considerable potential for synergies. Specifically, we integrate a simple partial evaluator for a JAVA-like language into the logic-based symbolic execution engine of the software verification tool KeY [4]. This allows to interleave symbolic execution and partial evaluation steps within a uniform (logic-based) framework in a sound way. Intermittent partial evaluation during symbolic execution has the effect that the remaining program that is yet to be executed is continuously simplified relative to the current path conditions and the current symbolic state in each symbolic execution trace.

This paper is organized as follows: in the next section we introduce a small object-oriented programming language which is used for the formal definitions (the actual system is implemented for nearly full-fledged sequential JAVA); we also provide background on symbolic execution and partial evaluation. Sect. 3 defines the program logic and deduction system that we use as a framework for

^{*} This work has been partially supported by the EU project FP7-ICT-2007-3 HATS *Highly Adaptable and Trustworthy Software using Formal Methods* and the EU COST Action IC0701 *Formal Verification of Object-Oriented Software*.

the integration. In Sect. 4 we introduce a version of a program specialization operator that is suitable for logic-based verification and we extend the symbolic execution calculus with sound rules that permit intermittent partial evaluation. In Sect. 5 we show the context in which the resulting calculus is applied, and in Sect. 6 we evaluate the integrated system using formal verification tasks for a number of JAVA programs. This is followed by a discussion of related work (Sect. 7). We stress that the particular combination of symbolic execution and partial evaluation explored in the present paper is by far not the only possible one. We sketch further possibilities in the final section on future work.

2 Background

2.1 A Simple Programming Language

The object-oriented programming language PL described in this section is basically a simplified JAVA variant and closely related to the language defined in [5]. We briefly sketch the differences to JAVA:

Unsupported Features. Multi-threading, graphics, dynamic class loading, generic types or floating point datatypes are *not* supported by PL nor by the actual implementation in the KeY tool. Formal specification and verification of these features is a topic of ongoing research, therefore, left out completely.

Restricted Features. For ease of presentation PL imposes some additional restrictions compared to JAVA. The KeY tool and the prototype implementation of our ideas evaluated in Sect. 6 do not impose these restrictions, but model and respect the JAVA semantics faithfully. The following restrictions apply to PL:

Inheritance and Polymorphism. For the sake of a simple semantics for dynamic dispatch of method invocations PL abstains from JAVA-like interfaces and method overloading. Likewise, with exception of the `Null` type, the type hierarchy induced by user-defined class types has a tree structure with class `Object` as root.

Prohibiting method overloading allows to identify a method within a class unambiguously by its name and number of parameters. We allow polymorphism (i.e. methods can be overwritten in subclasses) but require that their signature must be exactly the same, otherwise it is a compile-time error.

Visibility. All classes, methods and fields are publicly visible. This restriction contributes also to a simpler dynamic dispatch semantics.

No Exceptions. PL has no support for exceptions. Instead of runtime exceptions like `NullPointerException` the program will simply not terminate in these cases.

No class/object Initialization. In JAVA the first active usage of a type or creation of a new instance triggers complex initialization. PL supports only instance creation, but does not initialize fields upon creation. In particular, PL does not support static or instance initializers. Constructors are also missing in PL, a new instance is simply created by the expression `new T()`.

Primitive Types. Only `boolean` and `int` are available. To keep the semantics of standard arithmetic operators simple, `int` is an unlimited datatype representing the whole numbers \mathbb{Z} rather than a finite datatype with overflow.

A PL program p is a non-empty set of class declarations with at least one class of name `Object`. The class hierarchy is a tree with class `Object` as root. A class $Cl := (cname, sname_{opt}, fld, mtd)$ consists of (i) a classname $cname$ unique in p , (ii) the name of its superclass $sname$ (only omitted for $cname = \text{Object}$), and (iii) a list of field fld and method mtd declarations.

The syntax for class declaration is the same as in JAVA. The only lacking features are constructors and static/instance initialization blocks. PL knows also the special reference type `Null` which is a singleton with `null` as the only element. It may be used in place of any reference type and is the only type that is a subtype of all class types.

To keep examples short we agree on the following convention: if not explicitly stated otherwise, any given sequence of statements is seen as if it would be the body of a static, void method declared in a class `Default` with no fields declared.

The syntax of the executable fragment needed for the purpose of this paper as follows:

Statements

```

stmt ::= stmt stmt | lvarDecl | locExp '=' exp ';' | cond | loop
loop ::= while '(' exp ')' '{' stmt '}'
lvarDecl ::= Type IDENT ('=' exp)opt ';'
cond ::= if '(' exp ')' '{' stmt '}' else '{' stmt '}'

```

Expressions

```

exp ::= (exp.)opt mthdCall | opExp | locExp
mthdCall ::= mthdName '(' expopt '(' , exp )* ')'
opExp ::= f(expopt ( , exp )*) |  $\mathbb{Z}$  | TRUE | FALSE | null
f ::= ! | - | < | <= | >= | > | == | & | | | * | / | % | + | -

```

Locations

```

locExp ::= IDENT | exp.IDENT

```

Dynamic dispatch works in PL as follows: we need to determine the implementation of a method on encountering a method invocation such as `o.m(a)`. To do so, first look up the dynamic type T of the object referenced by `o`. Then scan all classes between T and the static type of o for an implementation of a method named m and the correct number of parameters. The first match is taken.

2.2 Symbolic Execution

Symbolic execution is an idea from the 1960s [15], but it has only recently been realized efficiently for industrially relevant programming languages. Symbolic execution is a central, very versatile program analysis technique that is used for formal program verification [4, 12, 16], extended static checking and verification [2], debugging [3], and automatic test case generation [7, 9].

In the last decade a number of efficient symbolic execution engines for real heap-based programming and intermediate languages were created including KeY (for JAVA, C, Creol, see [4]), KIV (for JAVA, see [19]), Bogor/Kiasan (for BIR, see [8]), Pex (for MSIL, see [7]), and VeriFast (for C, JAVA, see [13]).

In symbolic execution one permits either uninitialized program locations or, more generally, program locations that are initialized with symbolic expressions. The following PL program orders the values of x and y : after its execution x contains the maximum of x_0 , y_0 and y their minimum.

```
int x = x0; int y = y0; int z = max(x,y);
if (x < z) {y = x; x = z;}
```

We use location-value pairs to represent states in symbolic execution. The expression $\{l_1 := t_1 \parallel \dots \parallel l_n := t_n\}$ denotes a symbolic state in which each program location of the form l_i has the expression t_i as its symbolic value.

After symbolic execution of the first three statements of the program above we obtain the symbolic state $\mathcal{U} = \{x := x_0 \parallel y := y_0 \parallel z := \max(x_0, y_0)\}$. Symbolic execution of the conditional splits the execution into two branches, because the value $x_0 < \max(x_0, y_0)$ of the guard expression is symbolic and cannot be reduced. The (negated) value of the guard becomes a *path condition* relative to which symbolic execution continues. Under the path condition $P_1 \equiv x_0 < \max(x_0, y_0)$ the body of the conditional is executed which results in the final symbolic state $\mathcal{U}' = \{x := \max(x_0, y_0) \parallel y := x_0 \parallel z := \max(x_0, y_0)\}$. From P_1 and properties of \max one can infer $\max(x_0, y_0) \doteq y_0$ which simplifies \mathcal{U}' to $\{x := y_0 \parallel y := x_0 \parallel z := y_0\}$. The other branch terminates immediately in state \mathcal{U} under path condition $P_2 \equiv x_0 \geq \max(x_0, y_0)$ ($\equiv x_0 \doteq \max(x_0, y_0)$).

It is obvious already from this small example that simplification of intermediate states wrt first-order theories is essential for efficiency and to obtain intuitive results. Modern symbolic execution engines use SMT solvers [7, 13] and also powerful built-in theorem provers [4, 19] for this purpose.

The example suggests that a single state during symbolic execution of a program p consists of the following three components:

1. A program pointer to the next executable statement of the remaining statements in p that have to be executed.
2. A path condition P relative to which the remaining statements are executed.
3. A symbolic state \mathcal{U} relative to which the remaining statements are executed.

Symbolic execution of a program is then arranged as a *symbolic execution tree* whose nodes are triples consisting of program pointer, path condition, and symbolic state.

In general it is not possible to symbolically execute a program fully, because unbounded loops give rise to infinitely many branches with differing symbolic path conditions. Loop invariants or induction are required to turn symbolic execution into a complete method for computing strongest post-states of programs.

2.3 Partial Evaluation

The ideas behind partial evaluation go back in time even further than those behind symbolic execution: Kleene's well-known s_{mn} theorem from 1943 states that for each computable function $f(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} = x_1, \dots, x_m$, $\mathbf{y} = y_1, \dots, y_n$ there

is an $m+1$ -ary primitive recursive function s_n^m such that $\phi_{s_n^m(f, \mathbf{x})} = \lambda \mathbf{y}. f(\mathbf{x}, \mathbf{y})$. Partial evaluation can be characterized as the research programme to prove Kleene's theorem under the following conditions:

1. $\phi_{s_n^m(f, \mathbf{x})}$ is supposed to run more efficiently than f .
2. f is a program from a non-trivial programming language, not merely a recursive function.
3. The construction of $\phi_{s_n^m(f, \mathbf{x})}$ is efficient, i.e., its runtime should be comparable to compilation of f -programs.

In contrast to symbolic execution the result of a partial evaluator is not the value of output variables, but another program. The known input (named \mathbf{x} above) is also called *static input* while the general part \mathbf{y} is called *dynamic input*. The partial evaluator or *program specializer* is often named *mix*. Fig. 1 gives a schematic overview of partial evaluation.

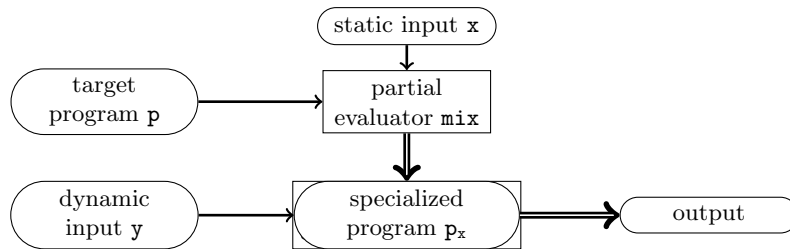


Fig. 1. Partial evaluation schema [14].

The first efforts in partial evaluation date from the mid 1960s and were targeted towards Lisp. Due to the rise in popularity of functional and logic programming languages the 1980s saw a large amount of research in partial evaluation of such languages. A seminal text on partial evaluation is the book by Jones et al. [14].

There has been relatively little research on partial evaluation of JAVA. The paper [18] summarizes the state-of-art until 2002 and discusses the JAVA specializer JSPEC which worked by cross-translation to C as an intermediate language. JSPEC seems to be no longer maintained. We found only one other (commercial) JAVA partial evaluator called JPE¹, but its capabilities and underlying theory is not documented.

The application context of partial evaluation is rather different from that of symbolic execution: in practice, partial evaluation is not only employed to boost the efficiency of individual programs, but often used in meta-applications such as parser/compiler generation.

We illustrate the main principles of partial evaluation by a small control circuit PL program depicted in Fig. 2 on the left. The program approximates

¹ http://www.gradsoft.ua/products/jpe_eng.html

the value of variable `y` to a given `threshold` with accuracy `eps` by repeatedly increasing or decreasing it as appropriate.

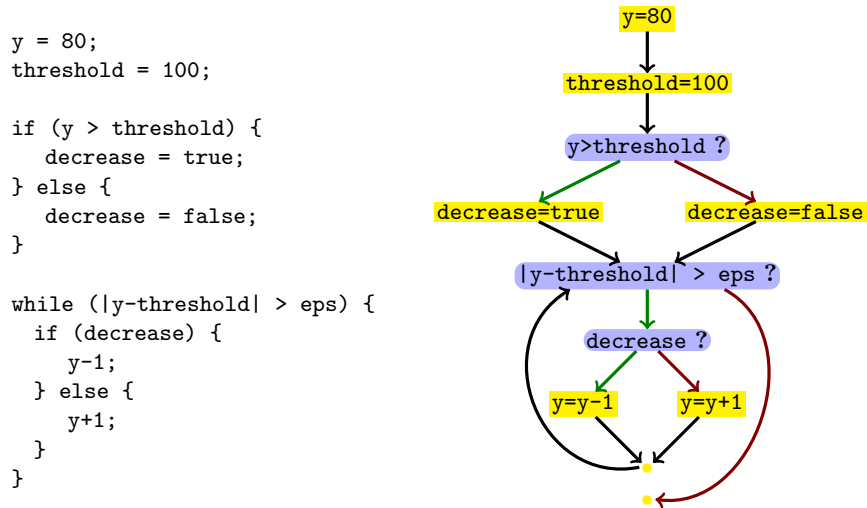


Fig. 2. A simple control circuit PL program and its control flow graph.

We can imagine to walk a partial evaluator through the control flow graph (for the example on the right of Fig. 2) while maintaining a table of concrete (i.e., constant) values for the program locations. In the example, that table is empty at first. After processing the two initial assignments it contains $\mathcal{U} = \{y := 80 \parallel \text{threshold} := 100\}$ (using the update notation of Section 2.2).

Whenever a new constant value becomes known, the partial evaluator attempts to propagate it throughout the current control flow graph (CFG). For the example, this *constant propagation* results in the CFG depicted in Fig. 3 on the left. Note that the occurrences of `y` that are part of the loop have *not* been replaced. The reason is that `y` might be updated in the loop so that these latter occurrences of `y` cannot be considered to be static. Likewise, the value of `decrease` after the first conditional is not static either. The check whether the value of a given program location can be considered to be static with respect to a given node in the CFG is called *binding time analysis* (BTA) in partial evaluation.

Partial evaluation of our example proceeds now until the guard of the first conditional. This guard became a *constant expression* which can be evaluated to `false`. As a consequence, one can perform *dead code elimination* on the left branch of the conditional. The result is depicted in Fig. 3 in the middle. Now the value of `decrease` is static and can be propagated into the loop (note that `decrease` is not changed inside the loop). After further dead code elimination, the final result of partial evaluation is the CFG on the right of Fig. 3.

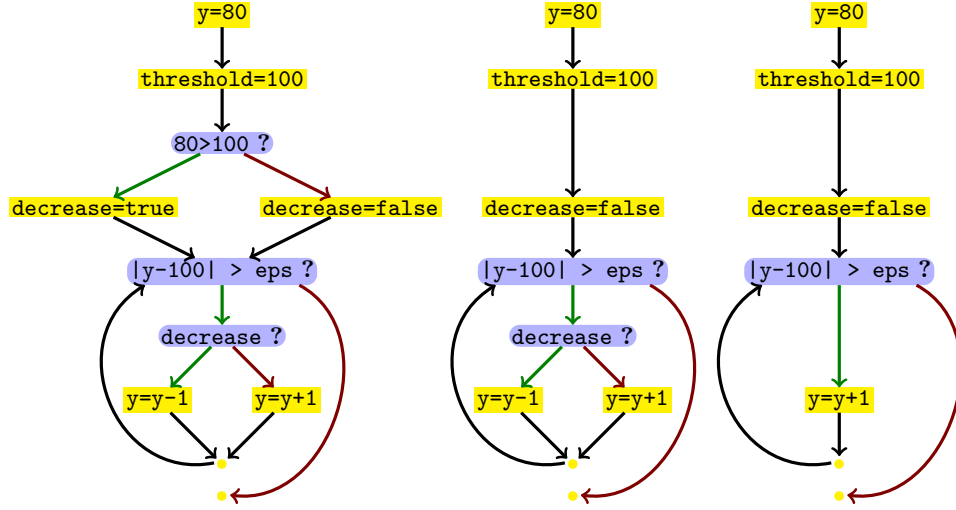


Fig. 3. Partial evaluation example.

Partial evaluators necessarily approximate the target programming language semantics, because they are supposed to run fast and automatic. In the presence of such programming language features as exceptions, inheritance with complex localization rules (as in JAVA), and aliasing (e.g., references, array entries) BTA becomes very complex [18].

3 Dynamic Logic with Updates

3.1 Program Logic

As program logic for PL we use a sorted first-order dynamic logic instantiated by a given PL program p . We define formally the family of first-order dynamic logics DPL used to reason about PL programs. Each concrete instance of this family is associated to exactly one PL program which is then referred to as the *context program* or sometimes the *program context* of that logic.

Definition 1 (Signature). For any PL program p a DPL signature Σ_p is defined as a tuple $(Types, FSym, PSym, VSym)$, where $Types$ is a set of sort names that contains at least $\{\top, \text{boolean}, \text{int}, \text{Object}, \text{Null}\} \cup \text{classes}(p)$. Further, $FSym$ is a set of function symbols, $PSym$ a set of predicate symbols, and $VSym$ a set of logic variable symbols (we omit the subscript p in Σ_p whenever it can be unambiguously derived from the context). Function, predicate, and logic variable symbols have a fixed sorted signature. Sorts are ordered wrt a sort hierarchy \preceq . The order \preceq models p 's type hierarchy with maximum element \top .

We distinguish between *rigid* and *non-rigid* function and predicate symbols. Intuitively, the semantics of rigid symbols does not depend on the current state of

program execution while non-rigid symbols are state-dependent. (Local) program variables, arrays, static, and instance fields are modeled as non-rigid function symbols and together form a separate class of non-rigid symbols called *location* symbols. Specifically, local program variables and static fields are modeled as non-rigid constants, instance fields as unary non-rigid functions, and array access as a binary non-rigid function. For example, an instance field `size` of type `int` declared in a class `List` is modeled as a unary non-rigid function $size@List : List \rightarrow int$. For terms representing field accesses, such as $size@List(head)$, we use the more readable short form $head.size$, if no ambiguities arise (and similar for array accesses). Π_Σ denotes the set of all executable PL programs (i.e., sequence of statements) with locations over signature Σ .

The inductive definition of terms and formulas is standard, but we introduce a new syntactic category called *update* to represent state updates with symbolic expressions. An *elementary update* has the general shape $l := t$ with terms l , t and l being a *location term* (i.e., a program variable, field or array access). It has the same semantics as an assignment. Updates can be composed into *parallel updates* $l_1 := t_1 \parallel l_2 := t_2$ or *quantified updates* `for` T $x; \phi; l(x) := t(x)$.

Definition 2 (Terms, Updates and Formulas). Terms t , updates u and formulas ϕ are well-sorted *first-order expressions of the following kind*:

$$\begin{aligned}
t &:= x \mid f(t_1, \dots, t_n) \mid \text{if } (\phi) \text{ then } (t) \text{ else } (t) \mid \{u\}t \\
u &:= l := t \mid u \parallel u \mid \text{for } T \ x; \phi; u \\
\phi &:= q(t_1, \dots, t_n) \mid \neg\phi \mid \phi \circ \phi \ (\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}) \mid \\
&\quad \{u\}t \mid \mathcal{Q}x; \phi \ (\mathcal{Q} \in \{\exists, \forall\}) \mid \text{if } (\phi) \text{ then } (\phi) \text{ else } (\phi) \\
&\quad [s]\phi \mid \langle s \rangle \phi \\
s &:= \text{any element of } \Pi_\Sigma
\end{aligned}$$

The formula $[p]\phi$ has the intuitive meaning that *if* the program p terminates *then* in its final state the formula ϕ must hold (*partial correctness*). The formula $\langle p \rangle \phi$ means that p terminates *and* in its final state ϕ holds (*total correctness*).

All formulas, terms and updates are evaluated with respect to a DPL-Kripke structure whose states correspond to program states.

Definition 3 (DPL-Kripke structure). A DPL-Kripke structure is a tuple $\mathcal{K} = (D, I, \mathcal{S}, \rho)$ where:

- D is a non-empty domain together with a domain function $\delta : D \rightarrow \text{Types}$ mapping each domain element to its (run-time) type.
- $D_T = \{d \in D \mid \delta(d) \preceq T\}$ denotes the projection of D to elements of sort T or any subsort of T . We ensure $D_T \neq \emptyset$, for all $T \in \text{Types}$ by setting $D_{\text{Null}} = \{\text{null}\}$, $D_{\text{int}} = \mathbb{Z}$, $D_{\text{boolean}} = \{\text{true}, \text{false}\}$.
- I is an interpretation mapping each rigid function symbol $f : T_1 \times \dots \times T_n \rightarrow S$ to a total function $I(f) : D_{T_1} \times \dots \times D_{T_n} \rightarrow D_S$ and each rigid predicate symbol $p : T_1 \times \dots \times T_n$ to a relation $I(p) \subseteq D_{T_1} \times \dots \times D_{T_n}$.
- \mathcal{S} is a set of states. Each state $s \in \mathcal{S}$ is an interpretation of the non-rigid function and predicate symbols.

- $\rho : \Pi \times \mathcal{S} \times \mathcal{S}$ is a state transition relation relating two states s, t by a program p iff p started in state s terminates in the final state t . Any set of final states $\rho(p)(s)$ is either a singleton set or empty as PL is deterministic.

As usual in first-order logic, to define evaluation of terms and formulas in addition to a structure we need the notion of a *variable assignment*. This is a function $\beta : \text{VSym} \rightarrow D$ assigning to logical variables a value in D . The evaluation function $val_{\mathcal{K},s,\beta}$ is then defined as usual and summarized in Fig. 4. Due to space

$$\begin{aligned}
val_{\mathcal{K},s,\beta}(f(t_1, \dots, t_n)) &= I(f)(val_{\mathcal{K},s,\beta}(t_1), \dots, val_{\mathcal{K},s,\beta}(t_n)) \\
val_{\mathcal{K},s,\beta}(q(t_1, \dots, t_n)) &= tt \text{ iff } (val_{\mathcal{K},s,\beta}(t_1), \dots, val_{\mathcal{K},s,\beta}(t_n)) \in I(q) \\
val_{\mathcal{K},s,\beta}(\phi \wedge \psi) &= \begin{cases} tt, & \text{if } val_{\mathcal{K},s,\beta}(\phi) = tt \text{ and } val_{\mathcal{K},s,\beta}(\psi) = tt \\ ff, & \text{otherwise.} \end{cases} \\
\dots & \\
val_{\mathcal{K},s,\beta}([\mathbf{s}](\phi)) &= \begin{cases} val_{\mathcal{K},s',\beta}(\phi), & \text{if } \exists s' \in \mathcal{S} \text{ such that } \rho(p)(s, s') \\ tt, & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 4. Definition (excerpt) of evaluation function val .

reasons we do not give a formal semantics of updates and refer to [4] for details on updates. Instead we explain the meaning intuitively along some examples:

- Elementary updates $\mathbf{i} := \mathbf{j}$ have exactly the same meaning as assignments: in a DPL-Kripke structure \mathcal{K} and state s , an update application $\{\mathbf{i} := \mathbf{j}\} \xi$ on a term/formula ξ yields the same value as if evaluating ξ in \mathcal{K}, s' where s' is identical to s except at \mathbf{i} which is evaluated to $val_{\mathcal{K},s,\beta}(\mathbf{j})$ in s' .
- Parallel updates $u_1 \parallel u_2$ are evaluated simultaneously and do not interfere with each other. Content swapping of two program variables can thus be expressed by $\mathbf{i} := \mathbf{j} \parallel \mathbf{j} := \mathbf{i}$.
- Quantified updates **for** $T x; \phi; u$ allow to update arbitrarily many locations simultaneously. The update “**for int** $i; i \geq 0 \wedge i < a.length; a[i] := 0$ ”, for example, assigns all array components the value 0.
- In case of parallel and quantified updates conflicts may arise when the same location is assigned different values as in $\mathbf{i} := 0 \parallel \mathbf{i} := 1$. Conflict resolution for parallel updates utilizes a last-wins semantics where the previous update is equivalent to $\mathbf{i} := 1$. Conflict resolution for quantified updates requires a well-founded order on T and the update with the smallest value for the quantified variable wins [4].

To summarize, updates are similar to explicit substitutions and allow to express state changes concisely at the syntactic level.

Definition 4 (Satisfiability and Validity). A DPL-formula ϕ is

- satisfiable iff there exists a DPL-Kripke structure $\mathcal{K} = (D, I, \mathcal{S}, \rho)$, a state $s \in \mathcal{S}$ and a variable assignment β such that $val_{D,I,s,\beta}(\phi) = tt$ (or in short: $\mathcal{K}, s, \beta \models \phi$);

- valid in a DPL-Kripke structure \mathcal{K} (we also say that \mathcal{K} is a model for ϕ and write $\mathcal{K} \models \phi$) iff for all states $s \in \mathcal{S}$ and variable assignments β we have $\mathcal{K}, s, \beta \models \phi$;
- logically valid iff all DPL-Kripke structures \mathcal{K} are models for ϕ .

We introduce two notions which we will need later on. For technical reasons we must have the possibility to extend a logic's signature.

Definition 5 (Signature Extension). Let Σ, Σ' denote two signatures. Σ' is called a signature extension of Σ if there is an embedding $\sigma(\Sigma) \subset \Sigma'$ that is unique up to isomorphism and enjoys the following properties:

- $\sigma(\text{Types}_\Sigma) = \text{Types}_{\Sigma'}$,
- $\sigma(\text{FSym}_\Sigma) \subseteq \text{FSym}_{\Sigma'}$ where for any arity countably infinite additional function symbols exist (analogously for predicates and logic variables)
- $\sigma(\Pi_\Sigma) \subseteq \Pi_{\Sigma'}$

An important property of signature extensions is the following:

Lemma 1. Let $\Sigma' \supseteq \Sigma$ denote a signature extension in the sense of Def. 5.

If a DPL-formula ϕ over Σ has a counter example, i.e., a DPL-Kripke structure \mathcal{K}_Σ , $s \in \mathcal{S}_\Sigma$ with $\mathcal{K}, s \not\models \phi$ then $\sigma(\mathcal{K}, s) \not\models \phi$. In words, signature extensions are counter example preserving.

Finally, we define the notion of an *anonymizing update*. The motivation behind anonymizing updates is to erase knowledge about the values of the fields included in the set *mod* of locations that can be modified by a program. This is achieved by assigning fresh constant or function symbols to those locations. For example, the anonymizing update for the modifier set $\text{mod}_\Sigma = \{i, j\}$ is $i := c_i \parallel j := c_j$ where c_i, c_j are constants freshly introduced in the extended signature Σ' .

Definition 6 (Anonymizing Update). Let *mod* denote a set of terms built from location symbols in Σ . An anonymizing update for *mod* is an update \mathcal{V}_{mod} over an extended signature Σ' assigning each location $l(t_1, \dots, t_n) \in \text{mod}$ a term $f'_l(t_1, \dots, t_n)$ where $f'_l \in \Sigma' \setminus \Sigma$.

3.2 Sequent Calculus

The calculus for reasoning in DPL is a *sequent calculus*. A sequent is an expression of the form $\Gamma \Rightarrow \Delta$ with Γ, Δ being sets of DPL-formulas. We call Γ the *antecedent* and Δ the *succedent* of the sequent. A sequent has the same meaning as the formula

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi .$$

Sequent rules have the general form

$$\text{name} \frac{s_1 \quad \cdots \quad s_n}{s}$$

where s, s_1, \dots, s_n are sequents. The sequents above the line are the rule's *premises* while sequent s is called the rule's *conclusion*. A sequent without any premises is an *axiom*.

A *sequent proof* is a tree whose nodes are labelled with sequents and with a sequent whose validity is to be proven at its root. This *proof tree* is constructed by applying sequent rules r to leaf nodes n whose sequent matches the conclusion r . The premises of r are then added as children of n . A branch of a proof tree is *closed* iff it contains an application of an axiom. A proof tree is closed iff all its branches are closed.

As usual, sequent rules are written in schematic form using schema variables (pattern variables with matching restrictions):

$$\text{andLeft} \frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta} \quad \text{close} \frac{*}{\Gamma, \phi \Rightarrow \phi, \Delta}$$

Here, ϕ, ψ (Γ, Δ) are schema variables that can be instantiated with any formula (set of formulas). The sequent rule **andLeft** is applicable at any leaf sequent that contains a disjunctively connected formula in its antecedent.

To handle formulas containing programs within our sequent calculus we aim to model symbolic execution (see Sect. 2.2). Recall that a node in a symbolic execution tree contains a program pointer to the next active statement, path condition, and a symbolic state relative to which symbolic execution is executed. Accordingly, nearly all sequent rules for programs work on a *first active statement* s and a current *update* \mathcal{U} in the following general form of a conclusion:

$$\Gamma \Rightarrow \{\mathcal{U}\}[\pi \mathbf{s}; \omega]\phi, \Delta$$

In addition, π stands for an inactive prefix containing labels, opening braces or method-frames (see below) and ω for the remaining program. Path conditions are represented by suitable formulas and accumulate in the antecedent Γ .

Symbolic execution in our DPL-calculus can be roughly organized into two phases. The first is the *rewriting phase* where the first active statement is replaced with an equivalent series of simpler statements. A typical rule is

$$\text{evalIfGuard} \frac{\Gamma \Rightarrow \{\mathcal{U}\}[\pi \text{ boolean } \mathbf{b} = \mathbf{nse}; \text{ if } (\mathbf{b}) \{s1\} \text{ else } \{s2\} \omega]\phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi \text{ if } (\mathbf{nse}) \{s1\} \text{ else } \{s2\} \omega]\phi, \Delta}$$

where \mathbf{nse} is a schema variable matching any non-simple PL-expression (basically, an expression that is neither a literal nor a program variable). As these kind of rules are pure rewrite rules that can be applied in any possible syntactic context (antecedent, succedent, box, diamond) we use the short form $\xi \rightsquigarrow \xi'$ to express that a term/program ξ is replaced with an equivalent term/program ξ' :

$$\text{if } (\mathbf{nse}) \{s1\} \text{ else } \{s2\} \rightsquigarrow \text{boolean } \mathbf{b} = \mathbf{nse}; \text{ if } (\mathbf{b}) \{s1\} \text{ else } \{s2\}$$

After the first active statement has been reduced to an elementary statement it is translated into a first-order representation of its semantics with the help of

rules belonging to the second phase. For instance, if the first active statement is a conditional whose guard is a simple expression (a program variable or a boolean literal) then the rule

$$\text{ifElseSplit} \frac{\Gamma, \{\mathcal{U}\}(b \doteq TRUE) \Rightarrow \{\mathcal{U}\}[\pi \{s1\} \omega] \phi, \Delta \quad \Gamma, \{\mathcal{U}\}(b \doteq FALSE) \Rightarrow \{\mathcal{U}\}[\pi \{s2\} \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi \text{ if } (b) \{s1\} \text{ else } \{s2\} \omega] \phi, \Delta}$$

splits the current proof branch into two branches, one for the case when the guard evaluates to true, and the other covering the `else` case. Further important representatives of the rules in this phase are assignment rules like

$$\text{writeAttribute} \frac{\Gamma, \{\mathcal{U}\} \neg(o \doteq \text{null}) \Rightarrow \{\mathcal{U}\}\{o.a := se\}[\pi \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi o.a = se; \omega] \phi, \Delta}$$

where o is a schema variable matching program variables, a matches fields and se matches simple expressions without side-effects that can be directly translated into a logic term. Fig. 5 shows a small excerpt of a sequent proof illustrating symbolic execution. Finally, we discuss how dynamic dispatch of a method is

$$\frac{\frac{\frac{\vdots \quad \vdots}{\Gamma, \{\mathcal{U}\} (b \doteq TRUE) \Rightarrow \{\mathcal{U}\} [s1] \phi, \Delta \quad \Gamma, \{\mathcal{U}\} (b \doteq FALSE) \Rightarrow \{\mathcal{U}\} [s2] \phi, \Delta}}{\Gamma \Rightarrow \{\mathcal{U}\} [\text{if } (b) \text{ then } s1 \text{ else } s2;] \phi, \Delta}}{\Gamma \Rightarrow [\text{boolean } b = (i >= 0); \text{ if } (b) \text{ then } s1 \text{ else } s2;] \phi, \Delta}}{\Gamma \Rightarrow [\text{if } (i >= 0) \text{ then } s1 \text{ else } s2;] \phi, \Delta}$$

where \mathcal{U} is the update $b := \text{if } (i \geq 0) \text{ then } (TRUE) \text{ else } (FALSE)$

Fig. 5. Excerpt of a proof demonstrating symbolic execution.

realized in the calculus. The rule for method invocation translates a dynamic dispatch into a cascade of concrete method calls:

$$\text{methodInvocation} \frac{\Gamma, \{\mathcal{U}\} \neg(o \doteq \text{null}) \Rightarrow \{\mathcal{U}\}[\pi \quad \text{if } (o \text{ instanceof } T_n) \text{ res} = o.m(se) @ T_n; \quad \text{else if } (o \text{ instanceof } T_{n-1}) \text{ res} = o.m(se) @ T_{n-1}; \quad \dots \quad \text{else res} = o.m(se) @ T_1; \quad \omega] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\} [\pi \text{ res} = o.m(se); \omega] \phi, \Delta}$$

- o, res are schema variables for program variables.
- $\text{res} = o.m(se) @ T$ are so called *method-body* statements. A method-body statement is a place holder for an actual method body namely exactly the method body of method m with the specified number of parameters as implemented in class T .

- T_1, \dots, T_n are all the subtypes of the static type of the program variable against which o is matched and that contain an actual implementation of the method m . As the most specific implementation has to be taken, the list T_1, \dots, T_n fulfills the condition that for all $0 < i < j \leq n : T_i \not\leq T_j$.

4 Interleaving Symbolic Execution and Partial Evaluation

4.1 General Idea

Recall from Section 2.2 that a symbolic execution tree unwinds a program’s control flow graph (CFG). As a consequence, identical code is (symbolically) executed in many branches, however, under differing path conditions and symbolic states. Merging back different nodes is usually not possible without approximation or abstraction [6, 21].

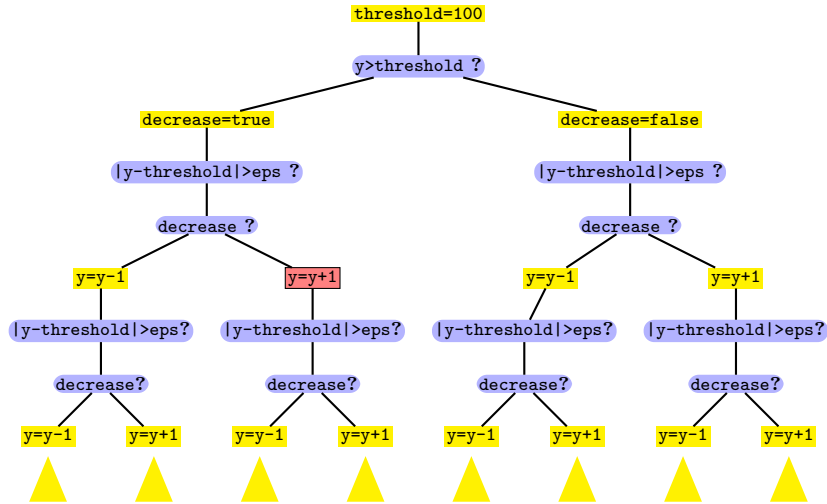


Fig. 6. Symbolic execution tree of the control circuit program.

The hope with employing partial evaluation is that it is possible to factor out common parts of computations in different branches by evaluating them partially *before* symbolic execution takes place. The naïve approach, however, to *first* evaluate partially and *then* perform symbolic execution fails miserably. The reason is that for partial evaluation to work well the input space dimension of a program must be significantly reducible by identifying certain input variables to have static values.

Typical usage scenarios for symbolic execution like program verification are not of this kind. For example, in the program of Fig. 2 in Sect. 2.3 it is unrealistic to classify the value of y as static. If we redo the example without the initial

assignment $y=80$ then partial evaluation can only perform one trivial constant propagation. The fact that input values for variables are not required to be static can even be considered to be one of the main advantages of symbolic execution and is the source of its generality: it is possible to cover all finite execution paths simultaneously and one can start execution at any given source code position without the need for initialization code.

The central observation that makes partial evaluation work in this context is that *during* symbolic execution static values are accumulated continuously as path conditions added to the current symbolic execution path. This suggests to perform partial evaluation *interleaved* with symbolic execution.

To be specific, we reconsider the example shown in Fig. 2, but we remove the first statement assigning the static value 80 to y . As observed above, no noteworthy simplification of the program’s CFG can be achieved by partial evaluation any longer. The structure of the CFG after partial evaluation remains exactly the same and only the occurrences of variable `threshold` are replaced by the constant value 100. If we perform symbolic execution on this program, then the resulting execution tree spanned by two executions of the loop is shown in Fig. 6. The first conditional divides the execution tree in two subtrees. The left subtree deals with the case that the value of y is too high and needs to be decreased. The right subtree with the complementary case.

All subsequent branches result from either the loop condition (omitted in Fig. 6) or the conditional expression inside the loop body testing the value of `decrease`. As `decrease` is not modified within the loop, some of these branches are infeasible. For example the branch below the boxed occurrence of $y=y+1$ (filled in red) is infeasible, because the value of `decrease` is true in that branch. Symbolic execution will not continue on these branches (at least for simple cases like that), but abandon them as infeasible by *proving* that the path condition is contradictory. Since the value of `decrease` is only tested *inside* the loop, however, the loop must still be first unwound and the proof that the current path condition is contradictory must be repeated. Partial evaluation can replace this potentially expensive proof search by *computation* which is drastically cheaper.

In the example, specializing the remaining program in each of the two subtrees after the first assignment to `decrease` eliminates the inner-loop conditional, see Fig. 7 (the partial evaluation steps are labelled with `mix`). Hence, interleaving symbolic execution and partial evaluation promises to achieve a significant speed-up by removing redundancy from subsequent symbolic execution.

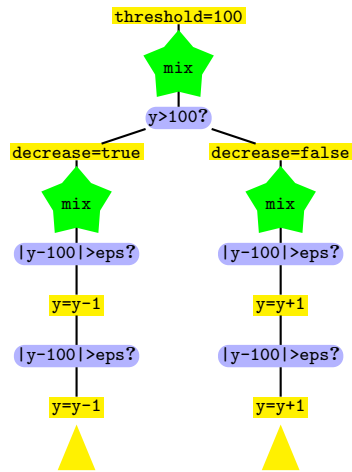


Fig. 7. Symbolic execution with interleaved partial evaluation.

4.2 The Program Specialization Operator

We define a program specialization operator suitable for interleaving with symbolic execution in DPL. A soundness condition ensures that the operator can be safely integrated into the sequent calculus. This approach avoids to formalize the partial evaluator in DPL which would be tedious and inefficient.

Definition 7 (Program Specialization Operator). *Let Σ be a signature and Σ' an extension of Σ as in Def. 5 containing countably infinite additional program variables and function symbols for any type and arity. Let σ be the embedding of Σ in Σ' ($\sigma(\Sigma) \subseteq \Sigma'$). The program specialization operator*

$$\downarrow_{\Sigma' \supseteq \Sigma}: ProgramElement \times Updates_{\Sigma'} \times For_{\Sigma'} \rightarrow ProgramElement$$

takes as arguments a PL-statement (-expression), an update and a DPL-formula and maps these to a PL-statement (-expression), where all arguments and the result are over Σ' .

The intention behind the above definition is that $\mathbf{p} \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi)$ denotes a “simpler” but semantically equivalent version of \mathbf{p} under the assumption that both are executed in a state coinciding with \mathcal{U} and satisfying φ . The signature extension allows the specialization operator to introduce new temporary variables or function symbols.

A program specialization operator is *sound* iff for all DPL-formulas $\psi \in For_{\Sigma}$, DPL-Kripke structures $\mathcal{K}_{\Sigma'}$, and states $s \in \mathcal{S}_{\Sigma'}$

$$\mathcal{K}_{\Sigma'}, s \models \langle \mathbf{p} \rangle \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi) \psi \Rightarrow \mathcal{K}_{\Sigma'}, s \models \mathcal{U}(\varphi \rightarrow \langle \mathbf{p} \rangle \psi) .$$

In words, the specialized program $\mathbf{p} \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi)$ must be able to reach at least the same post-states as the original program \mathbf{p} when started in a state coinciding with \mathcal{U} in which (path condition) φ holds.

Interleaving partial evaluation and symbolic execution is achieved by introduction rules for the specialization operator. The simplest possibility is:

$$\text{introPE} \frac{\Gamma \Rightarrow \{\mathcal{U}\} [\langle \mathbf{p} \rangle \downarrow (\mathcal{U}, true)] \phi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\} [\mathbf{p}] \phi, \Delta}$$

4.3 Specific Specialization Actions

We instantiate the generic program specialization operator of Def. 7 with some possible actions. In each case we derive soundness conditions.

Specialization Operator Propagation. The specialization operator needs to be propagated along the program as most of the different specialization operations work locally on single statements or expressions. During propagation of the operator, its knowledge base, the pair (\mathcal{U}, ϕ) , needs to be updated by additional knowledge learned from executed statements or by erasing invalid knowledge

about variables altered by the previous statement. Propagation of the specialization operator as well as updating the knowledge base is realized by the following rewrite rule

$$(\mathbf{p}; \mathbf{q}) \downarrow (\mathcal{U}, \phi) \rightsquigarrow \mathbf{p} \downarrow (\mathcal{U}, \phi); \mathbf{q} \downarrow (\mathcal{U}', \phi')$$

This rule is unsound for arbitrarily chosen \mathcal{U}' , ϕ' . Soundness is ensured under a number of restrictions:

1. Let mod denote the set of all program locations possibly changed by \mathbf{p} . Then we require that the DPL-formula “ $\{\mathcal{U}\} \text{respectStrongModifies}(\mathbf{p}, mod)$ ” is valid where the predicate `respectStrongModifies` abbreviates a formula that is valid iff \mathbf{p} changes at most locations included in mod . “Strong” means that mod must contain even locations whose values are only changed temporarily. Such a formula is expressible in DPL, see [10] for details.
2. Let \mathcal{V}_{mod} be the anonymizing update for mod (Def. 6). By fixing $\mathcal{U}' := \mathcal{U}\mathcal{V}_{mod}$ we ensure that the program state reached by executing \mathbf{p} is covered by at least one interpretation and variable assignment over the extended signature.²
3. ϕ' must be chosen in such a way that if $\mathcal{K}_{\Sigma} \models \{\mathcal{U}\}\langle \mathbf{p} \rangle \phi$ then there exists also an extended DPL-Kripke structure $\mathcal{K}_{\Sigma'}$ over an extended signature Σ' such that $\mathcal{K}_{\Sigma'} \models \{\mathcal{U}'\}\phi'$. This ensures that the post condition of \mathbf{p} is correctly represented by ϕ' . One possible heuristic to obtain ϕ' consists of symbolic execution of \mathbf{p} and applying the resulting update to ϕ . This yields a formula ϕ'' from which we obtain a candidate for ϕ' by “anonymizing” all occurrences of locations in it that occur in mod .

The first two soundness conditions can be expressed in DPL, the third one only in absence of quantified updates. In the latter case, the necessary proofs could be added as additional nodes that spawn side proofs. A more efficient (and generally necessary) approach is to show once and for all that the oracle used to determine mod and ϕ' is correct wrt the conditions.

Constant propagation and constant expression evaluation. Constant propagation is one of the most basic operations in partial evaluation and often a prerequisite for more complex rewrite operations. Constant propagation entails that if the value of a variable v is known to have a constant value c within a certain program region (typically, until the variable is potentially reassigned) then usages of v can be replaced by c . The rewrite rule

$$(v) \downarrow (\mathcal{U}, \varphi) \rightsquigarrow c$$

models the replacement operation. To ensure soundness the rather obvious condition $\mathcal{U}(\varphi \rightarrow v \doteq c)$ has to be proved where c is a rigid constant. The above rule can be easily modified to include constant expression evaluation.

² It is sufficient to let \mathcal{U}' be any update more general than $\mathcal{U}\mathcal{V}_{mod}$.

Dead-Code Elimination. Constant propagation and constant expression evaluation result often in specializations where the guard of a conditional (or loop) becomes constant. In this case, unreachable code in the current state and path condition can be easily located and pruned. A typical example for a specialization operation eliminating an infeasible symbolic execution branch is the rule

$$(\text{if } (\mathbf{b}) \{ \mathbf{p} \} \text{ else } \{ \mathbf{q} \}) \downarrow (\mathcal{U}, \phi) \quad \rightsquigarrow \quad \mathbf{p} \downarrow (\mathcal{U}, \phi)$$

which eliminates the `else` branch of a conditional if the guard can be proved true. The soundness condition of the rule is straightforward and self-explaining: $\mathcal{U}(\phi \rightarrow \mathbf{b} \doteq \text{TRUE})$.

Safe Field Access. Partial evaluation can be used to mark expressions as safe that contain field accesses or casts that may otherwise cause non-termination. We use the notation $\textcircled{\mathbf{e}}$ to mark an expression \mathbf{e} as safe, for example, if we can ensure that $\mathbf{o} \neq \text{null}$, then we can derive the annotation $\textcircled{\mathbf{o}.\mathbf{a}}$ for any field \mathbf{a} in the type of \mathbf{o} . The advantage of safe annotations is that symbolic execution can assume that safe expressions terminate normally and needs not to spawn side proofs that ensure it. The rewrite rule for safe field accesses is

$$\mathbf{o}.\mathbf{a} \downarrow (\mathcal{U}, \phi) \quad \rightsquigarrow \quad \textcircled{\mathbf{o}.\mathbf{a}} \downarrow (\mathcal{U}, \phi) .$$

Its soundness condition is $\mathcal{U}(\phi \rightarrow \neg(\mathbf{o} \doteq \text{null}))$.

Type Inference. For deep type hierarchies dynamic dispatch of method invocations may cause serious performance issues in symbolic execution, because a long cascade of method calls is created by the method invocation rule (Sect. 3.2, p. 12). To reduce the number of implementation candidates we use information from preceding symbolic execution to narrow the static type of the callee as far as possible and to (safely) cast the reference to that type. The method invocation rule can then determine the implementation candidates more precisely:

$$\begin{aligned} \text{res} = \mathbf{o}.m(a_1, \dots, a_n); \downarrow (\mathcal{U}, \phi) & \rightsquigarrow \\ \text{res} = \textcircled{((C)\mathbf{o}} \downarrow (\mathcal{U}, \phi)).m(a_1 \downarrow (\mathcal{U}, \phi), \dots, a_n \downarrow (\mathcal{U}, \phi)); \end{aligned}$$

The accompanying soundness condition $\mathcal{U}(\phi \rightarrow \exists C x; (\mathbf{o} \doteq x))$ ensures that the type of \mathbf{o} is compatible with C in any state specified by \mathcal{U}, ϕ .

5 Application

As an application of interleaving symbolic execution and partial evaluation, consider the verification of a GUI library. It includes standard visual elements such as `Window`, `Icon`, `Menu` and `Pointer`. An element has different implementations for different platforms or operating systems. Consider the following program snippet involving dynamic method dispatch:

```
framework.ui.Button button = radiobuttonX11;
button.paint();
```

The element `Button` is implemented in one way for Max OS X, while it is implemented in a different way for the X Window System. The method `paint()` is defined in `Button` which is extended by `CheckBox`, `Component`, and `Dialog`. Altogether, `paint()` is implemented in 16 different classes including `ButtonX11`, `ButtonMPC`, `RadioButtonX11`, `MenuItemX11`, etc. The complete type hierarchy is shown in Fig. 8. In the code above `button` is assigned an object with type `RadioButtonX11` which implements `paint()`. As a consequence, it should always terminate and the DPL-formula $\langle \text{gui} \rangle \text{true}$ should be provable where `gui` abbreviates the code above.

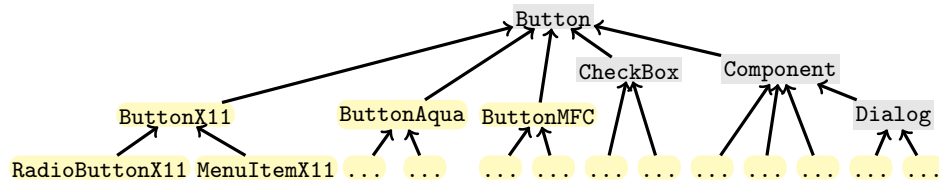


Fig. 8. Type hierarchy for the GUI example.

First, we employ symbolic execution alone to do the proof. During this process, `button.paint()` is unfolded into 16 different cases by the method invocation rule (Sect. 3.2, p. 12), each corresponding to a possible implementation of `button` in one of the subclasses of `Button`. The proof is constructed automatically in KeY with 161 nodes and 10 branches in the proof tree.

In a second experiment, we interleave symbolic execution and partial evaluation to prove the same claim. The partial evaluator propagates with the help of the *Type Inference* rule in the previous section the information that the runtime type of `button` is `RadioButtonX11` and the only possible implementation of `button.paint()` is `RadioButtonX11.paint()`. All other possible implementations are pruned. Only 24 nodes and 2 branches occur in the proof tree when running KeY integrated with a partial evaluator.

6 Evaluation

We implemented a simple partial evaluator for JAVA and interleaved it with symbolic execution in the KeY system as described above. We formally verified a number of JAVA programs with KeY with and without partial evaluation.

Table 1 shows the experimental results for a number of small JAVA programs which can be found in the KeY distribution. The column “Program” shows the name of the program we prove, the column “Strategy” shows the strategy we choose to perform the proof where “SE” means symbolic execution and “SE+PE” means interleaving symbolic execution and partial evaluation; the column “#Nodes” shows the total number of nodes in the proof; the column “#Branches” shows the total number of branches in the proof. The results show

that interleaving symbolic execution with partial evaluation significantly speeds up the proof for `complexEval`, `constantPropagation`, `dynamicDispatch`, `safeAccess`, and `safeTypeCast` which can all be considered to be amenable for partial evaluation. Table 2 shows the experimental results of verifying a larger

| Program | Strategy | #Nodes | #Branches |
|---------------------|----------|--------|-----------|
| complexEval | SE | 261 | 15 |
| | SE+PE | 158 | 3 |
| constantPropagation | SE | 65 | 1 |
| | SE+PE | 56 | 1 |
| dynamicDispatch | SE | 161 | 10 |
| | SE+PE | 24 | 2 |
| methodCall | SE | 113 | 4 |
| | SE+PE | 108 | 3 |
| safeAccess | SE | 28 | 4 |
| | SE+PE | 24 | 3 |
| safeTypeCast | SE | 73 | 5 |
| | SE+PE | 45 | 3 |

Table 1. Symbolic execution and partial evaluation for small JAVA programs.

and more realistic JAVA e-banking application used in [4, Ch. 10]. The column “Proof Obligation” shows which property we prove; the remaining columns are as in Table 1. The results show that symbolic execution interleaved with partial evaluation can speed up verification proofs even for larger applications. As is to be expected, depending on the structure of the program the benefit varies. It is noteworthy that none of the programs and proof obligations used in the present section have been changed in order to make them more amenable to partial evaluation. In no case we have to pay a significant performance penalty which seems to indicate that partial evaluation is a generally useful technology for symbolic execution and should generally be applied.

The case study in Sect. 5 suggests that it could pay off to take partial evaluation into account when designing programs, specifications, and proof obligations.

7 Related Work

Partial evaluation as a technique has been applied in a variety of areas including program optimization, compiler generation and meta-compilation. Partial evaluation has been applied successfully in logic programming [17] as well as for imperative and object-oriented languages like C [11] and JAVA [18]. A good overview including many references is given in [14]. As far as we know, the present paper is the first application of partial evaluation in formal verification.

Our approach is also related to supercompilation [20]. Supercompilation goes beyond partial evaluation by being able not only to specialize but also to gen-

| Proof Obligation | Strategy | #Nodes | #Branches |
|---|----------|--------|-----------|
| ATM.insertCard (EnsuresPost) | SE | 949 | 20 |
| | SE+PE | 805 | 13 |
| ATM.insertCard (PreservesInv) | SE | 2648 | 89 |
| | SE+PE | 2501 | 79 |
| ATM.enterPIN (EnsuresPost) | SE | 661 | 7 |
| | SE+PE | 654 | 8 |
| ATM.enterPIN (PreservesInv) | SE | 1524 | 45 |
| | SE+PE | 1501 | 44 |
| ATM.confiscateCard (EnsuresPost) | SE | 260 | 2 |
| | SE+PE | 255 | 2 |
| ATM.confiscateCard (PreservesInv) | SE | 739 | 19 |
| | SE+PE | 695 | 19 |
| ATM.accountBalance (EnsuresPost) | SE | 1337 | 35 |
| | SE+PE | 1271 | 29 |
| ATM.accountBalance (PreservesInv) | SE | 2233 | 57 |
| | SE+PE | 2223 | 59 |
| Account.checkAndWithdraw (EnsuresPost) | SE | 16174 | 136 |
| | SE+PE | 17023 | 135 |
| Account.checkAndWithdraw (PreservesInv) | SE | 14076 | 89 |
| | SE+PE | 10478 | 78 |

Table 2. Symbolic execution and partial evaluation for an e-banking application.

eralize a program to achieve a functionally equivalent but better performing program even in the absence of static input.

Partial evaluation is used in [1] to generate test cases and test case generators for given target programs. Instead of using a dedicated symbolic execution engine, they use partial evaluation to obtain an executable version of the implementation under test in the language CLP. CLP programs can then be executed on symbolic values returning a set of constraints on those input values. Partial evaluation is used as an approximation and replacement for a fully precise symbolic execution engine while we are interested in using partial evaluation to speed up symbolic execution in a dedicated symbolic execution engine.

There is a close relationship between the rule for specialization operator propagation (SOP) in Sect. 4.3 and what is known as *binding time analysis* (BTA) in partial evaluation. Partial evaluation techniques roughly categorize program variables into those which are known to have a constant value independent from any input and those whose value may vary. BTA in partial evaluation determines to which of these categories a variable belongs to. The precision of the analysis has a significant impact on the power of partial evaluation as too early binding prevents certain optimizations. The modifier set *mod* in the SOP rule influences directly the precision of the BTA performed by our specialization operator. If the oracle determining *mod* is too conservative (imprecise) too much knowledge of the current state \mathcal{U} will be lost and cannot be utilized in later specializations.

8 Conclusions and Future Work

In this paper we concentrated on deductive program verification as the main application scenario, however, as pointed out in Sect. 2.2, symbolic execution has other important usages, such as automatic test case generation [9, 7]. It would be interesting to investigate whether partial evaluation can lead to a reduction of redundant test cases.

We showed that a fairly naïve partial evaluator can be used to boost performance of a symbolic execution engine. In Sect. 7 we pointed out that symbolic execution in connection with assignable-clauses amounts to a relatively precise binding time analysis (BTA). As BTA becomes rather tricky for complex languages such as JAVA, it would be interesting to use symbolic execution and our simple partial evaluator to bootstrap a sophisticated partial evaluator for JAVA. It could also be interesting to use symbolic execution in addition to partial evaluation to improve precision, for example, in the test case generation approach of [1] discussed in the previous section.

The example in Sect. 5 shows that interleaving partial evaluation and symbolic execution has potential for speed-up especially for programs that are written generically. This is the case for two software development paradigms that gained much popularity in recent times: *model-driven development* (MDD) and *software product line* (SWPL) engineering. In both cases, development takes place as much as possible on a generic level: in MDD programs are modelled in abstract notations (the Platform Independent Model) and code generation is used to derive Platform-Specific Models and actual code; in SWPL one separates Domain Engineering which includes feature modeling and library development from Application Engineering where code is derived via instantiation and composition. In either case the executable code has been derived from generic artefacts and, therefore, verification is likely to benefit from the ability to partially evaluate specific information. We are currently experimenting with an SWPL scenario where we plan to use interleaved partial evaluation and symbolic execution.

References

1. E. Albert, M. Gomez-Zamalloa, et al. PET: a partial evaluation-based test case generation tool for Java bytecode. In *ACM SIGPLAN WS on Partial Evaluation and Semantics-based Program Manipulation*. ACM Press, 2010.
2. M. Barnett, K. R. M. Leino, et al. The Spec# programming system: an overview. In G. Barthe, L. Burdy, et al., eds., *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, vol. 3362 of *LNCS*, pp. 49–69. Springer, 2005.
3. M. Baum. *Debugging by Visualizing of Symbolic Execution*. Master’s thesis, Dept. of Computer Science, Institute for Theoretical Computer Science, Jun. 2007.
4. B. Beckert, R. Hähnle, et al., eds. *Verification of Object-Oriented Software: The KeY Approach*, vol. 4334 of *LNCS*. Springer, 2006.
5. B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, eds., *Proc.*

- Intl. Joint Conference on Automated Reasoning, Seattle, USA*, vol. 4130 of *LNCS*, pp. 266–280. Springer, 2006.
6. R. Bubel, R. Hähnle, et al. Abstract interpretation of symbolic execution with explicit state updates. In F. de Boer, M. M. Bonsangue, et al., eds., *Post Conf. Proc. 6th Intl. Symposium on Formal Methods for Components and Objects (FMCO)*, vol. 5751 of *LNCS*, pp. 247–277. Springer, 2009.
 7. J. de Halleux and N. Tillmann. Parameterized unit testing with Pex. In B. Beckert and R. Hähnle, eds., *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, vol. 4966 of *LNCS*, pp. 171–181. Springer, 2008.
 8. X. Deng, J. Lee, et al. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. 21st IEEE/ASM Intl. Conference on Automated Software Engineering, Tokyo, Japan*, pp. 157–166. IEEE Computer Society, 2006.
 9. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In B. Meyer and Y. Gurevich, eds., *Proc. Tests and Proofs (TAP), Zürich, Switzerland*, vol. 4454 of *LNCS*. Springer, 2007.
 10. C. Engel, A. Roth, et al. Verification of modifies clauses in dynamic logic with non-rigid functions. Tech. Rep. 2009-9, Department of Computer Science, University of Karlsruhe, 2009.
 11. A. J. Glenstrup, H. Makhholm, et al. C-mix: Specialization of c programs. In *Partial Evaluation*, pp. 108–154. 1998.
 12. M. Heisel, W. Reif, et al. Program verification by symbolic execution and induction. In K. Morik, ed., *Proc. 11th German Workshop on Artificial Intelligence*, vol. 152 of *Informatik Fachberichte*. Springer, 1987.
 13. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.
 14. N. D. Jones, C. K. Gomard, et al. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.
 15. J. C. King. *A program verifier*. Ph.D. thesis, Carnegie-Mellon University, 1969.
 16. C. S. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In S. Graf and L. Mounier, eds., *Proc. Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain*, vol. 2989 of *LNCS*, pp. 164–181. Springer, 2004.
 17. D. Sahlin. Mixtus: an automatic partial evaluator for full prolog. *New Gen. Comput.*, 12(1):7–51, 1993. ISSN 0288-3635.
 18. U. P. Schultz, J. L. Lawall, et al. Automatic program specialization for java. *ACM Transactions on Programming Languages and Systems*, 25, 2003.
 19. K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, et al., eds., *Proc. Algebraic Methodology and Software Technology, AMAST 2004, Stirling, Scotland, UK*, vol. 3116 of *Lecture Notes in Computer Science*, pp. 491–505. Springer, 2004.
 20. V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986. ISSN 0164-0925.
 21. B. Weiß. Predicate abstraction in a program logic calculus. In M. Leuschel and H. Wehrheim, eds., *Proc. 7th International Conference on integrated Formal Methods (iFM 2009)*, vol. 5423 of *LNCS*, pp. 136–150. Springer, 2009.