Program Specialization Via a Software Verification Tool *

Richard Bubel, Reiner Hähnle, and Ran Ji

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

Abstract. Partial evaluation is a program specialization technique that allows to optimize a program for which partial input is known. We propose a new approach to generate specialized programs for a Java-like language via the software verification tool KeY. This is achieved by symbolically executing source programs interleaved with calls to a simple partial evaluator. In a second phase the specialized programs are synthesized from the symbolic execution tree. The correctness of this approach is guaranteed by a bisimulation relation on the source and specialized programs.

1 Introduction

Symbolic execution [13] and partial evaluation [12] are both generalizations of standard interpretation of programs 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.

Our previous work [5] showed how to speed up the symbolic execution engine by interleaving with partial evaluation. On the other hand, an important question that can be asked is the possibility of achieving a more sophisticated program specializer via symbolic execution interleaved with simple partial evaluation operations. Another interesting question is whether the specialized program will behave the same as the source program with respect to the observable output, i.e., the soundness of the program specialization procedure. This paper tries to give an answer.

We propose a new approach to specialize Java-like programs via the software verification tool KeY, in which a symbolic execution engine is used. It is a two-phase procedure that first symbolically executes the program interleaved with

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

a simple partial evaluator, and then synthesizes the specialized program in the second phase. The soundness of the approach is proved.

The paper is organized as follows: In Sect. 2 we introduce a Java-like programming language PL as the working language in this paper and define its program logic. Sect. 3 presents the sequent calculus for PL. In Sect. 4 we integrate a simple partial evaluator into the symbolic execution engine. In Sect. 5 we introduce the bisimulation modality and define its sequent calculus. Sect. 6 shows how to generate specialized programs using our approach. Sect. 7 discusses related work. Sect. 8 concludes and addresses future work.

2 Dynamic Logic

Dynamic logic (DL) [9] is a representative of multi-modal logic tailored towards program verification. The programs to be verified against their specification occur in unencoded form as first class citizens of dynamic logic similar to Hoare logic [10] and avoid the encoding of programs.

Sorted first-order dynamic logic is sorted first-order logic plus two additional kinds of modalities: $[\cdot]\cdot$ (box) and $\langle\cdot\rangle\cdot$ (diamond). The first parameter takes a program and the second parameter a dynamic logic formula. Let $\mathtt{p}$ denote a program and ϕ a dynamic logic formula then

- $[\mathtt{p}]\phi$ is a DL-formula and, informally, expresses that if $\mathtt{p}$ is executed and terminates *then* in all reached final states ϕ holds;
- $\langle\mathtt{p}\rangle\phi$ is a DL-formula. Informally, it means that if $\mathtt{p}$ is executed then it terminates *and* in at least one of the reached final states ϕ holds.

We consider from now on only deterministic programs: Hence, a program $\mathtt{p}$ executed in a given state s *either* terminates and reaches exactly *one* final state *or* it does not terminate and there are no final states reachable from s upon execution of program $\mathtt{p}$. In this setting, the box modality expresses *partial correctness* of a program, while the diamond modality coincides with *total correctness*. With the above statements we can see that Hoare logic is subsumed by dynamic logic. The Hoare triple $\{pre\}\ \mathtt{p}\ \{post\}$ can be expressed as the DL formula $pre \rightarrow [\mathtt{p}]post$.

In the remainder of this section we introduce some basic concepts of dynamic logic. We follow thereby closely the KeY-approach [3].

Programming Language. We consider a simple Java-like programming language called PL. It provides features like interfaces, classes, attributes, method polymorphism (but not method overloading). PL does not support multi-threading, floating points or garbage collection. For ease of presentation we omit also class and object initialization, exceptions as well as `break` or `continue` statements and require non-void methods to have a single point of return at the end of the method. Further, the only supported primitive types are `boolean` and `int`.

Fig. 1 shows a PL program which we use as a running example throughout the paper.

```
public class OnLineShopping {
 boolean cpn;
 public int read() { /* read price of item */ }
 public int sum(int n) {
  int i = 0;
  int count = n;
  int tot = 0;
  while(i <= count) {
   int m = read();
   if(i >=2 && cpn) { tot = tot + m * 9 / 10; i++; }
   else { tot = tot + m; i++; }
  }
  return tot;
 }
}
```

Fig. 1. Example PL program.

This PL program could be used in an online shopping session. The `read()` method collects the price for each item. The `sum()` method calculates the total amount to be paid when purchasing n items. If the customer provides a coupon and purchases at least 2 items, then a 10% discount will apply from the second item onwards.

Dynamic Logic. Given a PL program $\mathcal{C}$ including interface and class declarations. Program $\mathcal{C}$ is in the following referred to as *context program*. We define our dynamic logic PL-DL($\mathcal{C}$) as follows:

Definition 1 (PL-Signature $\Sigma_{\mathcal{C}}$). *The signature* $\Sigma_{\mathcal{C}} = (S, \preceq, \mathsf{Pred}, \mathsf{Func}, \mathsf{LVar})$ *consists of*

- *a set of names* S *called* sorts *containing at least one sort for each primitive type and for each interface* I *and class* C *declared in* $\mathcal{C}$ *as well as the* `Null` *sort:*

  $S \supseteq \{\texttt{int}, \texttt{boolean}, \texttt{Null}\} \cup \{T \mid \textit{f.a. interfaces or classes } T \textit{ declared in } \mathcal{C}\}$

- *The partial subtyping order* $\preceq$*:* $S \times S$ *models the subtype hierarchy of* $\mathcal{C}$ *faithfully.*
- *An infinite set of function symbols* $\mathsf{Func} := \{f : T_1 \times \ldots \times T_n \to T \mid T_i, T \in S\}$*. We call* $\alpha(f) = T_1 \times \ldots \times T_n \to T$ *the arity of the function symbol.* $\mathsf{Func} := \mathsf{Func}_r \cup \mathsf{PV} \cup \mathsf{Attr}$ *is further divided into disjoint subsets:*
  - Func_r *rigid function symbols,*
  - PV *program variables* `i`, `j` *(non-rigid constants), and*
  - *attribute function symbols* Attr*, where for each attribute* a *of type* T *declared in class* C *an attribute function* $a@C : C \to T \in \mathsf{Attr}$ *exists. We omit the* $@C$ *from attribute names if no ambiguity arises.*

– *An infinite set of predicate symbols* $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in S\}$.
– *A set of logical variables* $\mathsf{LVar} := \{x : T | T \in S\}$.

Terms and formulas are defined as usual. The grammar below together with the canonical well-typedness conditions defines the syntax:

$$\begin{aligned} t &::= x \mid \mathtt{i} \mid t.\mathtt{a} \mid f(t,\ldots,t) \mid (\phi \ ?\ t\ :\ t) \mid \\ &\quad\ \mathbb{Z} \mid \mathtt{TRUE} \mid \mathtt{FALSE} \mid \mathtt{null} \mid \{u\}t \\ u &::= \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \\ \phi &::= \neg\phi \mid \phi \circ \phi\ (\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}) \mid (\phi\ ?\ \phi\ :\ \phi) \mid \\ &\quad\ \forall x : T.\phi \mid \exists x : T.\phi \mid [\mathtt{p}]\phi \mid \langle \mathtt{p} \rangle \phi \mid \{u\}\phi \end{aligned}$$

where $\mathtt{a} \in \mathsf{Attr}$, $f \in \mathsf{Func}$, $\mathtt{i} \in \mathsf{PV}$, $x : T \in \mathsf{LVar}$, and $\mathtt{p}$ is a sequence of executable statements in PL. The elements of category u are called *updates* and used to describe state changes. An *elementary update* $\mathtt{i} := t$ or $t.\mathtt{a} := t$ is a pair of location and term. Its intended meaning is that of an assignment. Updates applied on terms or formulas are again terms or formulas. They can be composed to *parallel updates* playing then the role of simultaneous assignments.

In the remaining paper, we use the notion of a program to refer to a sequence of executable PL-statements. If we want to include class, interface or method declarations, we either include them explicitly or make a reference to the context program $\mathcal{C}$.

The next step is to assign meaning to PL-DL terms and formulas. A formula in dynamic logic is evaluated with respect to a Kripke structure, in our case a PL-DL Kripke structure:

Definition 2 (Kripke structure $\mathcal{K}_{\Sigma_{PL}}$). *A PL-DL Kripke structure is a triple* $\mathcal{K}_{\Sigma_{PL}} = (\mathcal{D}, I, St)$ *with*

– *a set of elements* $\mathcal{D}$ *called domain,*
– *an interpretation* I *with*
  • $I(s) = \mathcal{D}_s,\ s \in S$ *assigning each sort its non-empty domain* $\mathcal{D}_s$*. It adheres to the restrictions imposed by the subtype order* $\preceq$*;* `Null` *is always interpreted as a singleton set and subtype of all reference (class and interface) types.*
  • $I(f) : \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T$ *for each rigid function symbol* $f : T_1 \times \ldots \times T_n \rightarrow T \in \mathsf{Func}_r$.
  • $I(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n}$ *for each rigid predicate symbol* $p : T_1 \times \ldots \times T_n \in \mathsf{Pred}$.
– *a set of states* St *assigning non-rigid function symbols a meaning: Let* $s \in St$ *then* $s(\mathtt{a@}C) : \mathcal{D}_C \rightarrow \mathcal{D}_T, \mathtt{a@}C \in \mathsf{Attr}$ *and* $s(\mathtt{i}) : \mathcal{D}_T, \mathtt{i} \in \mathsf{PV}$.

The pair $D = (\mathcal{D}, I)$ *is called a first-order structure.*

Finally, a *variable assignment* $\beta : \mathsf{LVar} \rightarrow \mathcal{D}_T$ maps a logic variable $x : T$ to its domain $\mathcal{D}_T$. An update, program, term or formula ξ is evaluated with respect to a given first-order structure $D = (\mathcal{D}, I)$, a state $s \in St$ and a variable assignment β as $val_{D,I,s,\beta}(\xi)$. The evaluation function *val* is defined recursively on the term and formula structure. Fig. 2 shows an excerpt of its definition.

$$val_{D,s,\beta}(x) = \beta(x), x \in \mathsf{LVar}$$
$$val_{D,s,\beta}(f(t_1,\ldots,t_n)) = D(f)(val_{D,s,\beta}(t_1),\ldots,val_{D,s,\beta}(t_n))$$
$$val_{D,s,\beta}(\mathtt{x}) = s(\mathtt{x}), \mathtt{x} \in \mathsf{PV}$$
$$val_{D,s,\beta}(o.\mathtt{a}) = s(\mathtt{a})(val_{D,s,\beta}(o)), \mathtt{a} \in \mathsf{Attr}$$
$$val_{D,s,\beta}(\neg\phi) = tt \text{ iff } val_{D,s,\beta}(\phi) = ff$$
$$val_{D,s,\beta}(\psi \wedge \phi) = tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ and } val_{D,s,\beta}(\psi) = tt$$
$$val_{D,s,\beta}(\psi \vee \phi) = tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ or } val_{D,s,\beta}(\psi) = tt$$
$$val_{D,s,\beta}(\psi \rightarrow \phi) = val_{D,s,\beta}(\neg\psi \vee \phi)$$

$$val_{D,s,\beta}((\psi \ ? \ \xi_1 \ : \ \xi_2)) = \begin{cases} val_{D,s,\beta}(\xi_1) \text{ if } val_{D,s,\beta}(\psi) \\ val_{D,s,\beta}(\xi_2) \ otherwise \end{cases}$$
$$val_{D,s,\beta}(\mathtt{x} := v) = s', \text{ with } \begin{cases} s'(x) = val_{D,s,\beta}(v) \\ s'(y) = s(y) \qquad y \neq x \end{cases}$$
$$val_{D,s,\beta}(o.\mathtt{a} := v) = \{s'\}, s = s' \text{ except } s'(a)(val_{D,s,\beta}(o)) = val_{D,s,\beta}(v)$$
$$val_{D,s,\beta}([s_1;s_2]\phi) = \begin{cases} val_{D,s',\beta}([s_2]\phi), \{s'\} = val_{D,s,\beta}(s_1) \\ tt, s_1 \uparrow \end{cases}$$
$$val_{D,s,\beta}([\mathtt{if}(e) \ \{p\} \ \mathtt{else} \ \{q\}]\phi) = val_{D,s,\beta}([\mathtt{T\ b}; \mathtt{b} = \mathtt{e}; \ \mathtt{if}(e) \ \{p\} \ \mathtt{else} \ \{q\}]\phi)$$
$$val_{D,s,\beta}([\mathtt{if(b)} \ \{p\} \ \mathtt{else} \ \{q\}]\phi) = \begin{cases} val_{D,s,\beta}([p]\phi), \ s(b) = val_{D,s,\beta}(\mathtt{TRUE}) \\ val_{D,s,\beta}([q]\phi), \ \text{otherwise} \end{cases} (\mathtt{b} \in \mathsf{PV})$$

Fig. 2. Definition of evaluation function *val* (excerpt).

Some words on the semantics of updates. While elementary updates have the same meaning as assignments, the semantics of parallel updates is slightly more complicated. We explain them by example:

Example 1 (Update semantics).

- Evaluating $\{\mathtt{i} := \mathtt{j}+1\}\mathtt{i} \geq \mathtt{j}$ in a state s is identical to evaluate the formula $\mathtt{i} \geq \mathtt{j}$ in a state s' which coincides with s except for the value of $\mathtt{i}$ which is evaluated to the value of $val_{D,s,\beta}(\mathtt{j}+1)$.
- Evaluation of the parallel update $\mathtt{i} := \mathtt{j} \| \mathtt{j} := \mathtt{i}$ in a state s leads to the successor state s' identical to s except that the values of $\mathtt{i}$ and $\mathtt{j}$ are swapped.
- The parallel update $\{\mathtt{i} := 3 \| \mathtt{i} := 4\}$ has a conflict as $\mathtt{i}$ is assigned different values. In such a case the conflict is resolved by using a last-one-wins semantics. Last-one-wins semantics means that the textually last occurring assignment overrides all previous ones of the same location.

We conclude the presentation of PL-DL by defining the notions of satisfiability, model and validity.

Definition 3 (Satisfiability, model and validity). *A formula ϕ*

- *is called* satisfiable *if there exists a first-order structure D, a state $s \in St$ and a variable assignment β with $val_{D,s,\beta}(\phi) = tt$ (short: $D, s, \beta \models \phi$).*
- *has a* model *if there exists a first-order structure D, a state $s \in St$, such that for all variable assignments β: $val_{D,s,\beta}(\phi) = tt$ holds (short: $D, s \models \phi$).*
- *is* valid *if for all first-order structures D, states $s \in St$ and for all variable assignments β: $val_{D,s,\beta}(\phi) = tt$ holds (short: $\models \phi$).*

3 Sequent Calculus

To analyze a PL-DL formula for validity, we use a Gentzen style sequent calculus. A sequent

$$\underbrace{\phi_1, \ldots, \phi_n}_{\Gamma} \Longrightarrow \underbrace{\psi_1, \ldots, \psi_m}_{\Delta}$$

is a pair of sets of formulas Γ (antecedent) and Δ (succedent). Its meaning is identical to the meaning of the formula

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

A sequent calculus rule

$$\mathsf{rule}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \Delta_n}^{premises}}{\underbrace{\Gamma \Longrightarrow \Delta}_{conclusion}}$$

consists of one conclusion and possibly many premises. One example of a sequent calculus rule is the rule andRight:

$$\mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma \Longrightarrow \psi,\ \Delta}{\Gamma \Longrightarrow \phi \wedge \psi,\ \Delta}$$

We call ϕ and ψ (formula) schema variables which match here any arbitrary formula. A rule is applied on a sequent s by matching it conclusion against s. The instantiated premises are then added as children of s. For example, when applying andRight to the sequent $\Longrightarrow \mathtt{i} \geq 0 \wedge \neg \mathtt{o.a} = \mathtt{null}$ we instantiate ϕ with $\mathtt{i} \geq 0$ and ψ with $\neg \mathtt{o.a} = \mathtt{null}$. The instantiated sequents are then added as children to the sequent and the resulting partial proof tree becomes:

$$\frac{\Longrightarrow \mathtt{i} \geq 0 \qquad \Longrightarrow \neg \mathtt{o.a} = \mathtt{null}}{\Longrightarrow \mathtt{i} \geq 0 \wedge \neg \mathtt{o.a} = \mathtt{null}}$$

Fig. 3 shows a selection of first-order sequent calculus rules. A proof of the validity of a formula ϕ in a sequent calculus is a tree where

- each node is annotated with a sequent,
- the root is labeled with $\Longrightarrow \phi$,
- for each inner node n: there is a sequent rule whose conclusion matches the sequent of n and there is a bijection between the rule's premises and the children of n, and,
- the last rule application on each branch is the application of a close rule (axiom).

Axioms and Propositional Rules

$$\mathsf{close}\ \frac{*}{\phi \Longrightarrow \phi} \qquad \mathsf{closeTrue}\ \frac{*}{\Longrightarrow true} \qquad \mathsf{closeFalse}\ \frac{*}{false \Longrightarrow}$$

$$\mathsf{andLeft}\ \frac{\Gamma, \psi, \phi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta} \qquad \mathsf{orRight}\ \frac{\Gamma \Longrightarrow \phi, \psi, \Delta}{\Gamma \Longrightarrow \phi \vee \psi,\ \Delta} \qquad \mathsf{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi,\ \Delta}$$

$$\mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma \Longrightarrow \psi,\ \Delta}{\Gamma \Longrightarrow \phi \wedge \psi,\ \Delta} \qquad \mathsf{orLeft}\ \frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi,\ \Delta \Longrightarrow}$$

First-Order Rules

$$\mathsf{allLeft}\ \frac{\Gamma, \phi[x/t] \Longrightarrow \Delta}{\Gamma, \forall x : T.\phi \Longrightarrow \Delta} \qquad \mathsf{exRight}\ \frac{\Gamma \Longrightarrow \phi[x/t], \Delta}{\Gamma \Longrightarrow \exists x : T.\phi\ \Delta}$$

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow \phi[x/c], \Delta}{\Gamma \Longrightarrow \forall x : T.\phi,\ \Delta} \qquad \mathsf{exLeft}\ \frac{\Gamma, \phi[x/c] \Longrightarrow \Delta}{\Gamma, \exists x : T.\phi \Longrightarrow \Delta}$$

$$c \text{ new},\ \text{freeVars}(\phi) = \emptyset$$

Fig. 3. First-order calculus rules (excerpt).

So far the considered rules were pure first-order reasoning rules. The calculus design regarding rules for formulas with programs is discussed next. We consider only the box modality variant of these rules.

Our sequent calculus variant is designed to stepwise symbolically execute a program. It behaves for most parts as a symbolic program interpreter. Symbolic execution as a means for program verification goes back to King [13]. Symbolic execution means that upon program execution the initial values of the input variables, fields etc., are symbolic values (terms) instead of concrete ones. The program then performs algebraic computations on those terms instead of actually computing concrete values.

We explain the core concepts along a few selected rules. Starting with the assignment rule:

$$\mathsf{assignLocalVariable}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := litVar\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x} = litVar; \omega]\phi, \Delta}$$

where $x \in \mathsf{PV}$, and $litVar$ is either a boolean/integer literal or a program variable, and ω the rest of the program. The assignment rule works as most program rules on the first active statement ignoring the rest of the program (collapsed into ω). Its effect is the movement of the elementary program assignment into an update.

The assignment rule for an elementary addition is similar and looks like

$$\mathsf{assignLocalVariable}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := litVar1 + litVar2\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x} = litVar1 + litVar2; \omega]\phi, \Delta}$$

There is a number of other assignment rules for the different program expressions. All of the assignment rules have in common that they operate on elementary (pure) expressions. This is necessary to reduce the number of rules and also as expressions may have side-effects that need to be "computed" first. Our calculus works in two phases: first complex statements and expressions are decomposed into a sequence of simpler statements, then they are moved to an assignment or are handled by other kinds of rules (e.g., a loop invariant rule). The decomposition phase consist mostly of so called unfolding rules such as:

unfoldAssignmentAddition

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{int v1} = exp1; \texttt{int v2} = exp2;\ \texttt{x} = \texttt{v1} + \texttt{v2}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{x} = exp1 + exp2; \omega]\phi, \Delta}$$

where $exp1, exp2$ are arbitrary (nested) expressions and $\texttt{v1}, \texttt{v2}$ new program variables not yet used in the proof or in ω.

The conditional rule is a typical representative of a program rule to show how splits in control flows are treated:

conditionalSplit

$$\frac{\Gamma, \{\mathcal{U}\}\texttt{b} = \texttt{TRUE} \Longrightarrow \{\mathcal{U}\}[\texttt{p}; \omega]\phi, \Delta \qquad \Gamma, \{\mathcal{U}\}\neg\texttt{b} = \texttt{TRUE} \Longrightarrow \{\mathcal{U}\}[\texttt{q}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{if (b) \{p\} else \{q\}}\ \omega]\phi, \Delta}$$

where $\texttt{b}$ is a program variable.

The calculus provides two different kinds of rules to treat loops. The first one realizes—as one would expect from a program interpreter—a simple unwinding of the loop:

loopUnwind $$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{if (b) \{}\bar{\texttt{p}}\texttt{; while (b) \{p\}\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (b) \{p\}}\ \omega]\phi, \Delta}$$

where $\bar{\texttt{p}}$ is identical to $\texttt{p}$ except for renaming of the newly declared variables in $\texttt{p}$ to avoid name collisions.

The major drawback of the rule is that except for cases where the loop has a fixed and known number of iterations, the rule can be applied arbitrarily often. Instead of unwinding the loop, one often used alternative is the loop invariant rule whileInv:

whileInv

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta & \text{(init)}\\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\texttt{b} = \texttt{TRUE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\texttt{p}]inv, \Delta & \text{(preserves)}\\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\texttt{b} = \texttt{FALSE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\omega]\phi, \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (b) \{p\}}\ \omega]\phi, \Delta}$$

The loop invariant rule requires the user to provide a sufficiently strong formula inv capturing the functionality of the loop. The formula needs to be valid before the loop is executed (init branch) and must not be invalidated by any loop iteration started from a state satisfying the loop condition (preserves branch). Finally, in the third branch the symbolic execution continues with the remaining program after the loop.

The *anonymizing* update $\mathcal{V}_{mod}$ requires further explanation: We have to show that inv is preserved by an arbitrary iteration of the loop body as long as the loop

condition is satisfied. But in an arbitrary iteration, values of program variables may have changed and outdated the information provided by Γ, Δ and $\mathcal{U}$. In traditional loop invariant rules, this context information is removed completely and the still valid portions have to be added to the invariant formula inv. We use the approach described in [3] and avoid to invalidate all previous knowledge. For this we require the user to provide a superset of all locations mod that are potentially changed by the loop. The anonymizing update $\mathcal{V}_{mod}$ erases all knowledge about these locations by setting them to a fixed, but unknown value. An overapproximation of mod can be computed automatically.

The last rule we want to introduce is about method contracts and it is a necessity to achieve modularity in program verification. More important for this paper is that it allows to achieve a modular program specializer. Given a method $T\ \mathtt{m}(T\ param_1, \ldots, T_n\ param_n)$ and a method contract

$$C(m) = (pre(param_1, \ldots, param_n), post(param_1, \ldots, param_n, res), mod)$$

The formulas pre and $post$ are the precondition and postcondition of the method with access to the parameters and to the result variable res (the latter only in $post$). The location set mod describes the locations (fields) that may be changed by the method. When we encounter a method invocation, the calculus first unfolds all method arguments. After that the method contract rule is applicable:

methodContract

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{param_1 := \mathtt{v1} \| \ldots \| param_n := \mathtt{vn}\}pre, \Delta \quad \Gamma \Longrightarrow \{\mathcal{U}\}\{param_1 := \mathtt{v1} \| \ldots \| param_n := \mathtt{vn}\}\{\mathcal{V}_{mod}\}(post \rightarrow [\mathtt{r} = res; \omega]\phi), \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{r} = \mathtt{m}(\mathtt{v1}, \ldots, \mathtt{vn});\ \omega]\phi, \Delta}$$

In the first branch we have to show that the precondition of the method is satisfied. The second branch then allows us to assume that the postcondition is valid and we can continue to symbolically execute the remaining program. The anonymizing update $\mathcal{V}_{mod}$ erases again all information about the locations that may have been changed by the method. About the values of these locations, the information encoded in the postcondition is the only knowledge that is available and on which we can rely in the remaining proof.

Definition 4 (Soundness). *A rule is* sound *if and only if the validity of the premises implies the validity of the conclusion.*

Theorem 1. *The sequent calculus rules for PL-DL are sound.*

4 Integrated Simple Partial Evaluator

In this section, we show how to integrate a simple partial evaluator into the symbolic execution engine to perform some basic partial evaluation when symbolically executing the program. The operations defined here were also introduced in our previous paper [5], where we showed how to speed up the symbolic execution engine by interleaving these partial evaluation operations.

The basic idea is to introduce a partial evaluation operator $\mathtt{p} \downarrow (\mathcal{U}, \phi)$ that can be attached to any program statement or expression p. The partial evaluation operator then specializes the program construct p with respect to the knowledge accumulated in update $\mathcal{U}$ and formula ϕ. The integration is achieved by introducing special sequent calculus rules that trigger the specialization on the program under consideration.

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 $(\mathcal{U}, \phi)$, 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

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

This rule is sound under a number of restrictions of $\mathcal{U}'$, ϕ', see [5] for details.

Constant propagation and constant expression evaluation. 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. Note that c could also be another variable and in this case usages of v can be replaced by c until v is reassigned or c is changed. 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 (within the given region). The above rule can be easily modified to include constant expression evaluation.

For example, in Fig. 1, when executing `int count = n;` in method `sum()`, `count` will be replaced by `n` in the loop guard because both `count` and `n` are unchanged, therefore, the loop guard becomes `i <= n`. However, `int i = 0;` could not propagate `0` into the loop guard since `i` could potentially be reassigned in the loop. One interesting point is that, if we unwind the loop once according to loopUnwind rule introduced in Section 3, it will become `if(i <= n) ... if(i >= 2 && cpn) ... while(i <= n) ...`, where `i` is ok to be replaced by `0` in both conditional guards (because `i` is not reassigned), but not in the loop guard. The result looks like `if(0 <= n) ... if(0 >= 2 && cpn) ... while(i <= n) ....`

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

$$(\mathtt{if}\ (\mathtt{b})\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}) \downarrow (\mathcal{U}, \phi) \qquad \rightsquigarrow \qquad \mathtt{p} \downarrow (\mathcal{U}, \phi)$$

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: $\mathcal{U}(\phi \rightarrow \mathtt{b} \doteq \mathtt{TRUE})$.

Continuing the example above, we can perform further specialization with the dead-code elimination rule. Since in the second conditional guard `0 >= 2` is evaluated to $false$, the `then`-branch is pruned. The result is `if(0 <= n) int m = read(); tot = tot + m; i++; while(i <= n) ....`

Some other partial evaluation operations such as *Safe Field Access* and *Type Inference* are also integrated. Please refer to [5] for more details.

5 A Sequent Calculus for Bisimulation

In the previous sections we introduced a dynamic logic based on symbolic execution. In section 4 we reported about our previous work on speeding up symbolic execution by interleaving the symbolic execution with partial evaluation steps.

In this section we present how to extend the existing framework in a natural way to extract a specialized version for the verified program.

In Sect. 5.1 we introduce a bisimulation modality which allows us to relate two programs that behave indistinguishably on a given set of locations. The programs being related to each other are the original program and its specialized version. Sect. 5.2 defines the calculus rules for the newly added modal operator.

5.1 The Bisimulation Modality

Please note that several of the definitions given in this section assume that in program specialization the source language is the same as the target language. The definitions are generalizable to specialization (and finally compilation) between different languages (see Sect. 8).

Definition 5 (Location Sets). *A* location set *is a set of*

- *program variables* `x` *or*
- *attribute expressions* $o.\mathtt{a}$ *with* $\mathtt{a} \in \mathsf{Attr}$ *and* o *being a term of appropriate sort.*

We are often not interested whether two states are identical, but rather whether they coincide on a given set of locations:

Definition 6. *Let* s_1, s_2 *denote two states and loc a location set. We write* $s_1 \sim_{loc} s_2$ *if and only if for all* $l \in loc$ *it holds that* $val_{D,s_1,\beta}(l) = val_{D,s_2,\beta}(l)$ *where* D *denotes a first-order structure and* β *a variable assignment.*

Specialized programs behave indistinguishably from the original program for (externally) observable locations. The set of observable locations includes usually all output variables and the part of the heap reachable from input and output variables. The formal definition bears a close relationship to the definition of non-interference:

Definition 7 (Observable Locations). *Let D denote a first-order structure and β a variable assignment. A location loc is called* observable *by a program* $\mathtt{p}$ *if there are two states s_0, s_1 differing only in the evaluation of loc and either*

- *program* $\mathtt{p}$ *terminates for s_i, but not for s_{1-i} ($i \in \{0,1\}$), or*
- *program* $\mathtt{p}$ *terminates for both states in final states $\{s_i'\} = val_{D,s_i,\beta}(\mathtt{p})$ ($i \in \{0,1\}$) and there is a location loc' (not necessarily the same as loc) with $val_{D,s_i,\beta}(loc') \neq val_{D,s_i',\beta}(loc')$ and $val_{D,s_i,\beta}(loc') \neq val_{D,s_{1-i}',\beta}(loc')$*

A location loc is observable *by a formula ϕ if there are two states s_1, s_2 differing only in loc with $val_{D,s_1,\beta}(\phi) \neq val_{D,s_2,\beta}(\phi)$.*

Definition 8 (Bisimulates Relationship). *Let obs be a location set and s_1, s_2 two states with $s_1 \sim_{obs} s_2$. Two programs $\mathtt{p}, \mathtt{q}$ are in a* obs-bisimulation relation with respect to s_1 and s_2 *if and only if for all first-order structures D and variable assignments β*

$val_{D,s_1,\beta}(\mathtt{p}) \sim_{obs} val_{D,s_2,\beta}(\mathtt{q})$

holds. We write $s_1, s_2 \models \mathtt{p} \sim_{obs} \mathtt{q}$.

If for all states s_1, s_2 with $s_1 \sim_{obs} s_2$ the statement $s_1, s_2 \models \mathtt{p} \sim_{obs} \mathtt{q}$ holds we simply write $\mathtt{p} \sim_{obs} \mathtt{q}$ and say $\mathtt{p}$ obs-bisimulates $\mathtt{q}$

In this paper we restrict ourselves to program specialization and can use the same state s for s_1, s_2 ($s \sim_{obs} s$ holds trivially). The more general definition above is necessary when extending our approach to compilation (see Sect. 8).

Lemma 1. *Let obs be the set of all locations observable by formula ϕ and let $\mathtt{p}, \mathtt{q}$ be programs. If $s \models \mathtt{p} \sim_{obs} \mathtt{q}$ then for all first-order structures D and variable assignments β $D, s, \beta \models [\mathtt{p}]\phi \leftrightarrow [\mathtt{q}]\phi$ holds.*

Definition 9 (Bisimulation Modality—Syntax). *The* bisimulation modality $[\ p_{src} \sim p_{target}\ ]@(obs, use)$ *is a modal operator providing compartments for the source program p_{src}, the target (or specialized) program p_{target}, and two location sets obs and use.*

We extend our definition of formulas: Let ϕ be a PL-DL formula and $\mathtt{p}, \mathtt{q}$ two programs and obs, use two location sets, then $[\ \mathtt{p} \sim \mathtt{q}\ ]@(obs, use)\phi$ is also a PL-DL formula.

Remark 1. The intended meaning of the location set *use* is to keep track of use-definition chains and contains roughly all locations that are read by program $\mathtt{p}$ before they are redefined. The intent of set *obs* is to capture the locations observable by p and ϕ.

Remark 2. The definition above is tailored to the presentation of this paper, but in its general setting the modality can accommodate locations sets that represent arbitrary (local) analysis information.

We formalize our intuition by defining the semantics of the bisimulation modality:

Definition 10 (Bisimulation Modality—Semantics). *Let D, s, β denote a first-order structure, state and variable assignment, respectively. Further,* $\mathtt{p}, \mathtt{q}$ *are programs and obs and use location sets.*

$val_{D,s,\beta}([\ \mathtt{p}\ \sim\ \mathtt{q}\ ]@(obs, use)\phi) = tt$ *if and only if*

(i) $val_{D,s,\beta}([\mathtt{p}]\phi) = tt$
(ii) $s \models \mathtt{p} \sim_{obs} \mathtt{q}$
(iii) obs is a superset of all locations observable by $\mathtt{p}$ *and* ϕ
(iv) $usedVar(s, \mathtt{p}, \phi) \subseteq use$ *where* $usedVar$ *returns the set of variables read by* $\mathtt{p}$ *or observed by* ϕ *before any redefinition (when executing* $\mathtt{p}$ *in state* s*).*

5.2 Sequent Calculus Rules for the Bisimulation Modality

The general sequent calculus rules for the bisimulation modality are of the following form:

ruleName

$$\frac{\Gamma_1 \Longrightarrow \{\mathcal{U}_1\}[\ p_1\ \sim\ q_1\ ]@(obs_1, use_1)\phi_1, \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \{\mathcal{U}_n\}[\ p_n\ \sim\ q_n\ ]@(obs_n, use_n)\phi_n, \Delta_n}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ p\ \sim\ q\ ]@(obs, use)\phi, \Delta}$$

Unlike the normal sequent calculus rules which are executed in a bottom-up manner, the application of sequent calculus rules for the bisimulation modality consists of two phases.

1. Symbolic execution of source program p. It is performed bottom-up as usual in sequent calculus rules. In addition, the observable location sets obs_i are also propagated since they contain the locations observable by $\mathtt{p_i}$ and ϕ_i that will be used in the second phase to synthesize the specialized program. Normally *obs* could contain the return variables of a method and the locations used in the continuation of the program.
2. We synthesize the target program $\mathtt{q_i}$ and $\mathtt{use_i}$ by applying the rules in a top-down manner.

Based on the application of sequent calculus rules for the bisimulation modality, the process of synthesizing specialized programs is a two-phase procedure. The first phase is symbolic execution of the source program while keeping track of the observable location set *obs*. In the second phase, when the program is fully symbolically executed, the specialized program is synthesized by applying the rules in the other direction, starting with the emptyBox rule.

emptyBox

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}@(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathtt{nop}\ \sim\ \mathtt{nop}\ ]@(obs, obs)\phi, \Delta}$$

where $\_$ is an anonymous placeholder, and $\mathtt{nop}$ explicitly denotes that the program is empty (no operation). The interesting aspect of this rule is that the

location set *use* tracking read access to variables is set to *obs*, ensuring that observable locations are accessible in the specialized program.

Here are some examples of sequent calculus rules for the bisimulation modality. For convenience, we use $\overline{p}$ to denote the specialized version of p.

assignLocalVariable

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{l} := \mathtt{r}\}[\ \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \{\mathcal{U}\}[\ l = r; \omega\ \sim\ l = r; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \{\mathcal{U}\}[\ l = r; \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

The *use* set contains all program variables on which a read access might occur in the remaining program before being overwritten. In the first case, when the left side l of the assignment is among those variables, we have to update the *use* set by removing the newly assigned program variable l and adding the variable r which is read by the assignment. The second case makes use of the knowledge that the value of l is not accessed in the remaining program and skips the specialization of the assignment.

conditionalSplit

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}b \Longrightarrow \{\mathcal{U}\}[\ p; \omega\ \sim\ \overline{p; \omega}\ ]@(obs, use_{p;\omega})\phi, \Delta \\ \Gamma, \{\mathcal{U}\}\neg b \Longrightarrow \{\mathcal{U}\}[\ q; \omega\ \sim\ \overline{q; \omega}\ ]@(obs, use_{q;\omega})\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}[\ \mathtt{if}\ (b)\ \{p\}\ \mathtt{else}\ \{q\}; \omega\ \sim \\ \qquad \mathtt{if}\ (b)\ \{\overline{\mathtt{p};\omega}\}\ \mathtt{else}\ \{\overline{\mathtt{q};\omega}\}\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi, \Delta\end{array}}$$

(with b boolean variable.)

On encountering a conditional statement, symbolic execution splits into two branches, namely the `then`-branch and `else`-branch. The specialization of the conditional statement will result in a conditional. The guard is the same as used in the source program, `then`-branch is the specialization of the source `then`-branch continued with the rest of the program after the conditional, and the `else`-branch is analogous to the `then`-branch.

Note that the statements following the conditional statement are symbolically executed on both branches. This leads to duplicated code in the specialized program, and, potentially to code size duplication at each occurrence of a conditional statement. One note in advance: code duplication can be avoided when applying a similar technique as presented later in connection with the loop translation rule. However, it is noteworthy that the application of this rule might have also advantages: as discussed in [5], symbolic execution and partial evaluation can be interleaved resulting in (considerably) smaller execution trace. Interleaving symbolic execution and partial evaluation is orthogonal to the approach presented here and can be combined easily. In several cases this can lead to different and drastically specialized and therefore smaller versions of the remainder program ω and its specialization $\overline{\omega}$. The *use* set is extended canonically by joining the *use* sets of the different branches and the guard variable.

loopUnwind

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \texttt{if (b) \{p; while (b) \{p\}\}}\ \omega \sim \overline{\texttt{if (b) \{p; while (b) \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \texttt{while(b) \{p\}}\ \omega \sim \overline{\texttt{if (b) \{p; while(b) \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta}$$

whileInv

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}inv, \Delta \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\texttt{b} = \texttt{TRUE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\} \\ \qquad [\ p \sim \overline{p}\ ]@(obs \cup use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \{\mathcal{U}\}\{\mathcal{V}_{mod}\}(\texttt{b} = \texttt{FALSE} \wedge inv) \Longrightarrow \{\mathcal{U}\}\{\mathcal{V}_{mod}\}[\ \omega \sim \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \texttt{while(b)\{p\}}\,\omega \sim \texttt{while(b)\{}\overline{\texttt{p}}\texttt{\}}\,\overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

On the logical side the loop invariant rule is as expected and has three premises. Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use *true* as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onwards (if *true* is used as invariant then it holds trivially).

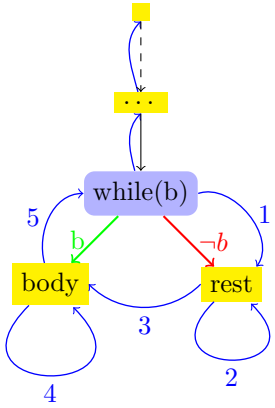

Fig. 4. Work flow of synthesizing loop.

Two things are of importance: the third premise executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, we must add to the variable set *obs* of the second premise the used variables of the third premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the *use case* premise of the loop invariant rule and then using the resulting use_1 set in the second premise. The work flow of the synthesizing loop is shown in Figure 4.

methodContract

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}\{param_1 := \texttt{v1} \| \ldots \| param_n := \texttt{vn}\}pre, \Delta \\ \Gamma \Longrightarrow \{\mathcal{U}\}\{param_1 := \texttt{v1} \| \ldots \| param_n := \texttt{vn}\}\{\mathcal{V}_{mod}\} \\ \qquad (post \rightarrow [\ \texttt{r} = res; \omega \sim \overline{\texttt{r} = res; \omega}\ ]@(obs, use)\phi), \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\ \texttt{r} = \texttt{m(v1, \ldots, vn)};\ \omega \sim \overline{\texttt{r} = res; \omega}\ ]@(obs, use)\phi, \Delta}$$

Theorem 2 (Soundness Sequent Calculus Rules). *The rules for the bisimulation modality are sound.*

A proof sketch of the theorem is given in the appendix.

Theorem 3 (Soundness Procedure). *The procedure of program specialization by application of the sequent calculus for the bisimulation modality is sound.*

6 Application

In this section, we show the application of our framework to generate specialized programs for a Java-like language. Consider the program in Fig. 1. Our purpose is to specialize the `sum()` method which consists of non-trivial constructs such as attributes, a conditional, loop and method call. To achieve a clearer presentation we omit the postcondition ϕ following the bisimulation modality throughout the example as well as other unnecessary formulas in the sequents.

The first phase of our approach starts symbolically executing method `sum`() with the return value `tot` as the only observable location, i.e., $obs = \{\texttt{tot}\}$. The first statements of the method declare and initialize variables. These statements are executed similar to assignments. Altogether the assignLocalVariable rule is applied three times, where each assignment rule application is immediatley followed by a partial evaluation step. We end up with

$$\dfrac{\dfrac{\dfrac{\Longrightarrow \{\ldots||\texttt{tot} := 0\}[\ \texttt{while(i <= n)}\ldots \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}{\Longrightarrow \{\ldots||\texttt{count} := \texttt{n}\}[\ \texttt{tot} = 0; \texttt{while(i <= n)}\ldots \sim sp_2\ ]@(\{\texttt{tot}\}, use_2)}}{\Longrightarrow \{\texttt{i} := 0\}[\ \texttt{count} = \texttt{n}; \ldots \sim sp_1\ ]@(\{\texttt{tot}\}, use_1)}}{\Longrightarrow [\ \texttt{i} = 0; \ldots \sim sp_0\ ]@(\{\texttt{tot}\}, use_0)}$$

where sp_i denotes the corresponding specialized program.

The next statement to be symbolically executed is the while loop computing the total sum. Instead of immediately applying the loop invariant rule, we unwind the loop once using the loopUnwind rule. Partial evaluation allows to simplify the guard `i <= n` and `i >= 2 && cpn` of the introduced conditional to `i <= 2` and `0 >= 2 && cpn` by applying constant propagation. Furthermore, the `then`-branch is eliminated because the guard `0 >= 2 && cpn` can be evaluated to *false*. The result is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Longrightarrow \{\texttt{i} := 0||\ldots||\texttt{tot} := 0\}\ [\ \texttt{if(0 <= n)\{int m1 = read(); tot = m1; i = 1; while}\ldots\} \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}{\Longrightarrow \{\texttt{i} := 0||\ldots; \texttt{tot} := 0\}\ [\ \texttt{if(0 <= n)\{}\ldots\texttt{tot} = 0 + \texttt{m1}; \texttt{i} = 1; \texttt{while}\ldots\} \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}}{\Longrightarrow \{\texttt{i} := 0||\ldots\}\ [\ \texttt{if(0 <= n)\{}\ldots\texttt{if(0 >= 2\&\&cpn)}\ldots; \texttt{i} = 0 + 1; \texttt{while}\ldots\} \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}}{\Longrightarrow \{\texttt{i} := 0||\ldots\}\ [\ \texttt{if(i <= n)\{}\ldots\texttt{if(i >= 2\&\&cpn)}\ldots; \texttt{i++}; \texttt{while}\ldots\} \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}}{\Longrightarrow \{\texttt{i} := 0||\ldots||\texttt{tot} := 0\}[\ \texttt{while(i <= n)}\ldots \sim sp_3\ ]@(\{\texttt{tot}\}, use_3)}$$

Application of the conditionalSplit rule creates two branches. The `else`-branch contains no program so it is synthesized right away by applying the emptyBox rule. Symbolic execution of the `then`-branch, applies the assignLocalVariable rule three times until we reach the while loop again. We decide to unwind the loop a second time. The symbolic execution follows then the same pattern as before until we reach the loop for a third time. Fig. 5(a) shows the relevant part of the proof tree of the second loop unwinding.

Instead of unwinding the loop once more, we apply the loop invariant rule whileInv. The rule creates three new goals. The goal for the `init` premise is not of importance for the specialization itself, hence, we ignore it in the following.

The *used variables* set *use* of the `preserves` premise depends on the instantiation of the *use* set in the `use case` premise. To resolve the dependency we continue with the latter. In this case, the `use case` premise contains no program, so it is trivially synthesized by applying the emptyBox rule which results in `nop` as the specialized program and the only element `tot` in *obs* becomes the *use* set. Based on this, the *use* set of the `preserves` premise is the union of *obs*, {`tot`} and the locations used in the loop guard: {`tot`, `i`}. The program in the `preserves` premise is then symbolically executed by applying suitable rules until it is empty. This process is similar to that when executing the program in the `then`-branch of the conditional generated by loopUnwind. The proof tree resulting from the application of the loop invariant rule is shown in Fig. 5(b). After symbolic execution we enter the second phase of our approach in which the specialized program is synthesized. Recall that when applying the whileInv rule, the procedure of synthesizing the loop starts with the `use case` branch. In our example, we have already performed this step and could already determine the instantiation of the observable location set *obs* of the `preserves` premise.

We explain now how the loop body is synthesized using the `preserves` premise: applying the emptyBox rule instantiates the placeholders sp_{12} and use_{12} with `nop` and {`tot`, `i`}. Going backwards, the assignLocalVariable rule tells us how to derive the instantiations for sp_{11} = `i++;` and use_{11} = {`tot`, `i`}. The instantiations for sp_{10} and use_{10} can be derived as `tot=tot+m; i++;` and {`tot`, `i`}. Before we can continue, the instantiations of sp_9 and use_9 need to be determined. Similar to the derivation of sp_{10} and use_{10}, applying the assignLocalVariable rule two times, we get sp_9 = `tot=tot+m*9/10; i++;` and use_9={`tot`, `i`}.

We have now reached the node where we previously applied the conditionalSplit rule. This rule allows us to derive `if(cpn) {tot=tot+m*0.9; i++;} else {tot=tot+m; i++;}`, as instantiation for sp_8 and {`tot`, `i`, `cpn`} as instantiation for use_8. Applying suitable rules, we end up with the specialized program sp_6

```
while (i<=n) {
  int m = read();
  if (cpn) {tot=tot+m*9/10; i++;}
  else {tot=tot+m; i++; }
}
```

and the used variable set use_6 = {`tot`, `i`, `cpn`}.

$$\dfrac{\dfrac{\dfrac{1 \leq \texttt{n} \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\ \texttt{while(i <= n)} \ldots \sim\ sp_6\ ]@(\{\texttt{tot}\}, use_6)}{\ldots}}{0 \leq \texttt{n} \Longrightarrow \{\ldots\}[\ \texttt{if(i <= n)} \ldots ; \texttt{while} \ldots \sim\ sp_5\ ]@(\{\texttt{tot}\}, use_5)}}{0 \leq \texttt{n} \Longrightarrow \{\ldots \| \texttt{m2} := \texttt{read}() \| \texttt{tot} := \texttt{m2} \| \texttt{i} := 1\}[\ \texttt{while(i <= n)} \ldots \sim\ sp_5\ ]@(\{\texttt{tot}\}, use_5)}$$

$$\dfrac{\neg(0 \leq \texttt{n}) \Longrightarrow \{\ldots\}[\ \sim \texttt{nop}\ ]@(\{\texttt{tot}\}, \{\texttt{tot}\}) \qquad \dfrac{\dfrac{}{\ldots}}{0 \leq \texttt{n} \Longrightarrow \{\ldots\}[\ \texttt{int m2} = \texttt{read}(); \ldots \sim\ sp_4\ ]@(\{\texttt{tot}\}, use_4)}}{\Longrightarrow \{\ldots\}[\ \texttt{if(0 <= n)\{int m2} = \texttt{read}(); \texttt{tot} = \texttt{m2}; \texttt{i} = 1; \texttt{while} \ldots\} \sim\ sp_3\ ]@(\{\texttt{tot}\}, use_3)}$$

(a) Specialization of the `while` loop via unwinding

$$\dfrac{\dfrac{\dfrac{\dots, \texttt{cpn} \Longrightarrow \ldots[\ \ldots \sim\ sp_9\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_9), \ldots, \neg\texttt{cpn} \Longrightarrow \{\ldots\}[\ \texttt{tot} = \texttt{tot} + \texttt{m}; \ldots \sim\ sp_{10}\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_{10})}{\ldots \Longrightarrow \{\ldots \| \texttt{tot} := \texttt{tot} + \texttt{m}\}[\ \texttt{i++}; \sim\ sp_{11}\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_{11})}}{\ldots \Longrightarrow \{\ldots \| \texttt{i} := \texttt{i} + 1\}[\ \sim\ sp_{12}\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_{12})}$$

$$\dfrac{\dfrac{\dfrac{\ldots, \neg(\texttt{i} \leq \texttt{n}) \Longrightarrow [\ \sim \texttt{nop}\ ]@(\{\texttt{tot}\}, \{\texttt{tot}\}) \qquad \dfrac{\ldots, \texttt{i} \leq \texttt{n} \Longrightarrow [\ \texttt{int} \ \ldots \sim\ sp_7\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\} \cup \{\texttt{tot}\}, use_7)}{\ldots \Longrightarrow \{\texttt{m} := \texttt{read}()\}[\ \texttt{if(cpn)} \ldots \sim\ sp_8\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_8)}}{\dots, \neg\texttt{cpn} \Longrightarrow \{\ldots\}[\ \texttt{tot} = \texttt{tot} + \texttt{m}; \ldots \sim\ sp_{10}\ ]@(\{\texttt{tot}\} \cup \{\texttt{i}\}, use_{10})}}{1 \leq \texttt{n} \Longrightarrow \{\ldots \| \texttt{i} := 2\}[\ \texttt{while(i <= n)} \ldots \sim\ \{\texttt{tot}\}\ ]@(use_6,)}$$

(b) Specialization of the `while`-loop using the loop invariant rule

Fig. 5. Specialization of the `while`-loop by different means.

Following the symbolic execution tree backwards and applying the corresponding rules, we finally synthesize the specialized program for `sum()` as follows:

```
public int sum(int n) {
 int i; int tot; tot = 0;
 if (0 <= n) {
  int m1 = read(); tot = m1;
  if (1 <= n) {
   int m2 = read();
   tot = tot + m2; i = 2;
   while(i <= n) {
    int m = read();
    if (cpn) { tot = tot + m * 9 / 10; i++; }
    else { tot = tot + m; i++; }
   } } }
 return tot; }
```

7 Related Work

JSpec [15] is a program specializer for Java and, therefore, has the same goal as our approach. In fact, JSpec is not working with full Java but a subset without concurrency, dynamic loading, etc. In this sense it is similar to our work. However, they use an *offline* partial evaluation technique that depends on *binding time analysis*. Our work is based on symbolic execution to derive information on-the-fly, similar to *online* partial evaluation [14]. Our work is related to the latter, the main difference being that we do not generate the specialized program during the symbolic execution phase, but synthesize it in the second phase. In principle, our first phase can obtain as much information as *online* partial evaluation, and the second phase can generate a more precise specialized program.

Our approach is also related to the *Verifying Compiler* [11] project which aims at the development of a compiler that verifies the program during compilation. In contrast to this, our approach might be called instead the *Compiling Verifier*. Like our work, compiler verification [8] aims to guarantee the correctness of the target program. The difference is that compiler verification attempts to verify the compiling program which is very expensive and hardly scales to realistic target languages and sophisticated optimizations.

Our work is closely related to rule-based compilation [1,4]. It differs in the sense that to the best of our knowledge their inference machine is by far not as powerful as the mature simplification engine used in KeY. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [2]) which also use a theorem prover to discharge proof obligations. They work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates

the actual strategy from the translation rules. What distinguishes our work from most approaches that we know is that the starting point is a system for functional verification of Java which is used for program specialization in such a way that it becomes fully automatic.

8 Conclusion and Future Work

We presented a novel approach to specialize programs via a software verification tool in a two-phase manner. In the first phase, symbolic execution interleaved with simple partial evaluation is performed. Symbolic execution permits dynamic analysis at compile time which is similar to online partial evaluation. In the second phase, the specialized program is synthesized. A use-definition chain set is maintained to eliminate unused assignments and to avoid unnecessary statements occurring in the specialized program. The correctness of the specialization is guaranteed by the bisimulation relationship of the source and specialized programs, together with the soundness of the program logic. It is a new architecture to construct verified compilers by combining verification, partial evaluation and local transformation. The implementation is currently ongoing with KeY tool and more results will be reported later.

Although this approach is defined for a Java-like language, it will be interesting to see whether other features such as concurrency could be handled, going towards full-Java.

Orthogonally, there are still opportunities to optimize the procedure. For instance, on encounter of a loop, the heuristics that decide whether to unwind it or not have a strong influence on the resulting specialized programs. Importing information, e.g., loop invariants, from other tools could also be useful.

The idea of this paper is to generate specialized programs, however, the bisimulation modality is not restricted to source and target program being from the same language, but it can be generalized to other languages provided with corresponding observable locations. Consequentially, the approach is still sound for generating bytecode or other intermediate languages. We plan to apply our approach to the modeling language ABS developed in the context of the HATS project[6,7]

Furthermore, the close connection between the program logic and compilation allows to ensure the correctness of the compilation process as such. We see a great potential of our approach when encoding security or safety properties in terms of pre-/postconditions. This should allow to identify unsafe or unsecured execution paths during compilation and either to abort compilation or to wrap the undesired execution paths in a wrapper that at least ensures the safety or security property of interest. For example, execution paths that may leak information can be secured by omitting the assignments that violate secure information flow. Another possibility would be to ensure that if the program enters an unsecured execution path, then the program will not terminate. Exploring these avenues is future work.

Acknowledgments

We thank Wolfgang Ahrendt for fruitful discussions as well as for valuable comments on an earlier version of this paper.

References

1. L. Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 218–227, New York, NY, USA, 1984. ACM.
2. C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. TVOC: a translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer-Verlag, 2005.
3. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
4. L. Breebaart. *Rule-based compilation of data parallel programs*. PhD thesis, Delft University of Technology, 2003.
5. R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In *Post Conf. Proc. FMCO2009*, Lecture Notes in Computer Science. Springer-Verlag, 2010.
6. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schafer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer-Verlag, 2011.
7. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language, 2011. In this volume.
8. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28:2–2, November 2003.
9. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
10. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), Oct. 1969.
11. T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50:63–69, January 2003.
12. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
13. J. C. King. *A program verifier*. PhD thesis, Carnegie-Mellon University, 1969.
14. E. S. Ruf. *Topics in online partial evaluation*. PhD thesis, Stanford University, Stanford, CA, USA, 1993. UMI Order No. GAX93-26550.
15. U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

Appendix: Proof of Lemma 2

Proof (Sketch). We give here only the proof for conditionalSplit. The proofs for other rules are similar. To prove soundness of a rule we need to show that the validity of the conclusion is a consequence of the premises' validity.

Let D, s_a, β be arbitrary, but fixed. We assume that $val_{D,s_a,\beta}(\Gamma \wedge \neg\Delta) = tt$ otherwise we are trivially done. We have now to prove that

$$val_{D,s_a,\beta}(\{\mathcal{U}\}[\begin{array}{l}\texttt{if } (b)\ \{p\}\ \texttt{else}\ \{q\};\omega \sim \texttt{if } (b)\ \{\overline{\texttt{p;}\omega}\}\ \texttt{else}\ \{\overline{\texttt{q;}\omega}\}\\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi)\end{array}$$

holds or, equivalently, that

$$val_{D,s,\beta}([\begin{array}{l}\texttt{if } (b)\ \{p\}\ \texttt{else}\ \{q\};\omega \sim \texttt{if } (b)\ \{\overline{\texttt{p;}\omega}\}\ \texttt{else}\ \{\overline{\texttt{q;}\omega}\}\\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi)\end{array}$$

with $val_{D,s_a,\beta}(U) = s$ holds.

We have to check that the four requirements stated in Def. 10 are satisfied. First, we need to check requirement (i), namely:

$$val_{D,s,\beta}(\{\mathcal{U}\}[\texttt{if } (b)\ \{p\}\ \texttt{else}\ \{q\};\omega]) = tt$$

This requirement is equivalent to the soundness proof of the conditional rule for the standard calculus version and skipped.

The most interesting requirement to be checked is (ii). We need to show that the specialized program *s-obs-bisimulates* the original program:

$$s \models \texttt{if } (b)\ \{\texttt{p}\}\ \texttt{else}\ \{\texttt{q}\};\omega \sim_{obs} \texttt{if } (b)\ \{\overline{\texttt{p;}\omega}\}\ \texttt{else}\ \{\overline{\texttt{q;}\omega}\}$$

Case $val_{D,s,\beta}(b) = true$: Validity of the first premise ensures that

$$s \models \texttt{p;}\omega \sim_{obs} \overline{\texttt{p;}\omega}$$

which means according to its definition

$$val_{D,s,\beta}(\texttt{p;}\omega) \sim_{obs} val_{D,s,\beta}(\overline{\texttt{p;}\omega})$$

With that we get

$$\begin{aligned} val_{D,s,\beta}(\texttt{if } (b)\ \{\texttt{p}\}\ \texttt{else}\ \{\texttt{q}\};\omega) &= val_{D,s,\beta}(\texttt{p};\omega)\\ &\sim_{obs} val_{D,s,\beta}(\overline{\texttt{p;}\omega})\\ &= val_{D,s,\beta}(\texttt{if } (b)\ \{\overline{\texttt{p;}\omega}\}\ \texttt{else}\ \{\overline{\texttt{q;}\omega}\}) \end{aligned}$$

The second case $val_{D,s,\beta}(b) = false$ is analogous. Taking both cases we can conclude the proof of requirement (ii).

Requirement (iii) is satisfied if

$$use_{p;\omega} \cup use_{q;\omega} \cup \{b\}$$

is a superset of all observable locations of $\texttt{if } (b)\ \{p\}\ \texttt{else}\ \{q\};\omega$ and ϕ. From the premises we get directly that $use_{p;\omega}$ and $use_{q;\omega}$ are supersets of all observable locations of the branches, the remaining program and formula ϕ. The only additional location which is read by the conditional statement except those of its branches and which may not yet be included is variable `b`. The union of all these sets is the set used in the rule's conclusion and satisfies obviously requirement (iii). Finally, we need to check requirement (iv) which can be done analogous to the check for requirement (iii). □