Abstract Execution*

Dominic Steinhöfel[0000−0003−4439−7129] and Reiner Hähnle[0000−0001−8000−7613]

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
{steinhoefel,haehnle}@cs.tu-darmstadt.de

Abstract. We propose a new static software analysis principle called *Abstract Execution*, generalizing Symbolic Execution: While the latter analyzes all possible execution paths of a *specific program*, Abstract Execution analyzes a *partially unspecified program* by permitting *abstract symbols* representing unknown contexts. For each abstract symbol, we faithfully represent each possible concrete execution resulting from its substitution with concrete code. There is a wide range of applications of Abstract Execution, especially for verifying *relational properties* of schematic programs. We implemented Abstract Execution in a deductive verification framework and proved correctness of eight well-known statement-level refactoring rules, including two with loops. For each refactoring we characterize the preconditions that make it semantics-preserving. Most preconditions are not mentioned in the literature.

1 Introduction

Reasoning about *abstract programs*, i.e. programs containing an abstract context represented by placeholder symbols, is required whenever one aims to rigorously analyze program transformation techniques. Notably in compiler validation, to argue that a specific compilation or optimization step preserves the meaning of any input program is a standard task. An established approach to this problem formalizes the abstract syntax and the semantics of the target programming language as a set of inductive definitions, then proves properties of abstract programs via structural induction over the program syntax [7]. Early work relied on pen-and-paper proofs [22,24]. Recently, interactive theorem provers are used to mechanize correctness proofs, e.g., in CompCert [21] and CakeML [31]. The main drawback is the very high effort required to mechanize a programming language and to perform interactive proofs. In this paper we take a different approach to reason about abstract programs that is automatic and based on symbolic execution. To make it work, we need to answer two questions: (i) Can one specify abstract program contexts sufficiently without giving full inductive definitions? (ii) If yes, which specification constructs are needed for abstract contexts?

We propose a new static software analysis principle called Abstract Execution (AE) that allows to *automatically* reason about *abstract* sequential programs

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

with *side effects*. An essential component of AE is a specification language for abstract program contexts. This permits, in contrast to prior work [13,14,18,29], to specify irregular termination behavior (exceptions, etc.) and fine-grained assumptions on abstract programs. Here we target sequential JAVA programs, but the principles of AE are equally applicable to other sequential languages.

Abstract Execution generalizes Symbolic Execution (SE) [5,8,17]. Symbolic Execution means to execute programs with symbolic expressions as input values. When SE is embedded into a program logic [1], these symbolic values are represented by first-order Skolem constants. Skolem symbols can also be viewed as *abstract* symbols whose concrete domain value is not specified. In this sense, SE is already abstract and it amounts to execution of *abstract* programs that permit variables and fields to be initialized with unknown values. For example, *symbolic* execution of a program "`i++;`" amounts to execution of the *abstract* program "i_0`++;`", where i_0 is a fresh Skolem symbol of type **`int`**.

AE in addition permits not only abstract *values*, but whole *programs* to appear as undefined expressions. In the program "**`if`** `(i>0)` p_0 **`else`** p_1", for example, the placeholders p_i can be substituted with *arbitrary* concrete programs, as long as the result is well-formed. From a semantic point of view, the difference between abstract values and abstract programs is that (i) an abstract program may change the value of arbitrary variables and fields, and (ii) its execution may terminate abruptly, i.e., by returning, throwing an exception, or continuing or breaking out of a loop. In other words, an abstract program can be seen as an unknown partial function between execution states, i.e. its big-step semantics $[\![p]\!]$. In logic, such a function can be represented by a *second-order* Skolem symbol. Therefore, AE amounts to a limited form of second-order reasoning: let p be an abstract program and φ a first-order formula. Then $\vdash_{ae} [p]\varphi$ holds iff the weakest precondition of p w.r.t. φ is first-order derivable. For example:

$$\vdash_{ae} [\texttt{int i; boolean b=}b_0\texttt{; if (b) } p_0 \texttt{ else i=0;}](b_0 \doteq \text{FALSE} \rightarrow \texttt{i} \doteq 0)$$

To check the relation $\vdash_{ae}$ *automatically*, we must provide a technical solution to the following question: how can one implement weakest precondition reasoning over *abstract* programs such that the result is a *first-order* formula? These are the main building blocks of our solution to realize limited second-order inference over programs in terms of first-order deduction: (1) *second-order* Skolemization to represent the effects of placeholders like p_0 on local variables and the heap, (2) explicit modeling of all possible ways of irregular termination in separate SE branches, (3) over-approximation of returned values and thrown exceptions by first-order Skolemization, and (4) a specification language to describe the possible effect of second-order Skolem symbols as well as to define conditions when irregular termination of concrete instances for abstract programs can happen.

Applications of Abstract Execution AE is applicable to many problems involving reasoning about abstract programs. It can be instantiated to (at least) the following tasks: (1) Execution of abstract method calls [6], a special case of AE; (2) automatic soundness proofs of program transformation and of (3) rule-based compi-

lation [29]; (4) sound, automatic ("lazy") symbolic execution over programs with loops and calls, (5) incremental program development and synthesis [28]; (6) a modular version of *proof carrying code* [26], where the contract of called methods needs not to be known to the certificate provider. It is impossible to discuss all these in this paper. Therefore, we focus on program transformation—task (2). We study refactoring rules as described in Fowler's well-known books [11,12]. We model refactoring techniques as abstract programs, formalize assumptions under which a refactoring is sound, and prove behavioral equivalence of the original and refactored version for all concrete programs satisfying the abstract context.

The paper is structured as follows. In Sect. 2 we describe how to construct *abstract* JAVA programs. Sect. 3 expounds our logic for AE. Sect. 4 contains our case study about proving correctness of refactoring techniques. Sect. 5 discusses related work, Sect. 6 concludes and gives an outlook to the future. An appendix with more material is available at `key-project.org/papers/ae/`.

2 Specifying Abstract Programs

An abstract JAVA program is a program containing at least one *Abstract Placeholder Statement (APS)* symbol. The syntax to declare an APS is:

```
abstract_statement P;
```

The symbol `P` is an identifier for an abstract statement. Semantically, every APS with the same identifier occurring in a program or proof represents the same program. The above APS may be substituted with any *concrete* JAVA program accessing and assigning arbitrary fields and local variables, except that it is not allowed to declare local variables visible outside. Additionally, a concrete program may (1) throw any type of exception, (2) return from the method it executes, (3) break to any surrounding block label, (4) continue to and break from a surrounding loop, (5) continue to the label of a surrounding loop.

The possible behaviors of an APS are constrained by an *abstract specification*. The syntax of the specification language extends *block contracts* [1,19] of the JAVA Modelling Language (JML). JML [20] is a specification language for JAVA used to describe the behavior of JAVA classes and methods. JML specifications are embedded into JAVA code via comment lines starting with an "`@`" sign. *An APS is the declaration of an abstract placeholder symbol together with all specification clauses that constrain it.* We explain the involved concepts by specifying a variant of Fowler's *Consolidate Duplicate Conditional Fragments* refactoring [11] step-by-step. The result, a fully specified program after refactoring is shown in Listing 3 (on p. 5). Table 1 summarizes all specification constructs that may be used in an APS.

Fig. 1 shows an unconstrained formalization of the refactoring. The abstract code uses an idiom to formalize abstract *expressions*: it introduces a variable representing the abstract expression (in this case, the boolean `b`) and precedes it with the APS `Init` that assigns to `b` an unknown value. The idiom works as expected if `Init` is constrained so it assigns a value exactly to `b` and not to any

Table 1: Specification constructs for APSs

Spec. Construct	Explanation
`locals(P)`	Refers to the Skolem (abstract) location set of local variables of an APS with symbol `P` visible from outside.
`declares` *skLocs*;	Specifies that an APS/method declares a list *skLocs* of Skolem location set specifiers `locals(·)`, opt. wrapped in `final(·)` modifiers, which can be used in APSs in the visible scope afterwards.
`assignable` *locs*;	Declares the location set *locs* to be assignable by the APS. *locs* is a list of variables, fields, and Skolem location set specifiers, optionally wrapped in a `hasTo(·)` modifier.
`accessible` *locs*;	Declares *locs* to be accessible by the APS.
`return_behavior requires` φ;	Specifies that the APS returns iff φ holds.
`exceptional_behavior requires` φ;	Spec. that the APS throws an exc. iff φ holds.
`break_behavior requires` φ; `continue_behavior requires` φ;	Specifies that the APS breaks/continues during loop execution iff φ holds.
`break_behavior` (*lbl*) `requires` φ; `continue_behavior` (*lbl*) `requires` φ;	Specifies that the APS breaks/continues to the (loop) label *lbl* iff φ holds.

Listing 1: Before

```
abstract_statement Init;
if (b) {
  abstract_statement P;
  abstract_statement Q1;
} else {
  abstract_statement P;
  abstract_statement Q2;
}
```

Listing 2: After

```
abstract_statement P;

abstract_statement Init;
if (b) {
  abstract_statement Q1;
} else {
  abstract_statement Q2;
}
```

Fig. 1: Unconstrained formalization of the "Consolidate Duplicate Conditional Fragments" Refactoring, "Pullout Prefix" variant [11]

other variable. JML uses the **assignable** clause to specify which locations can be assigned a value, but does not *enforce* the assignment. Hence, we extend JML with the **hasTo(·)** keyword. The specification "`//@ assignable hasTo(b);`" enforces that the specified abstract code assigns a value *exactly* to `b`.

Observe that the refactoring is unsound, whenever the APS `P` influences the value of `b`. If, for instance, `P` sets `b` to **true**, the **else** branch of the **if** statement in the refactored program is never reached. A drastic solution is to specify "`//@ ↳ assignable \nothing;`" for `P` which excludes any assignment. This, however, restricts the refactoring rule too severely to be useful. Assume the depicted program fragments occur in the scope of a method with a variable `result` that is returned at the end. Then we might constrain `P` with "`//@ assignable result;`" which forbids assignments to `b` (because it allows assignments *exactly* to `result`), but renders `P` still useful. But this is not restrictive enough: The abstract program `Init` that initializes `b` may still *access* arbitrary locations and assign them

Listing 3: Fully specified abstract program for the refactored version

```
//@ axiom mutex{throwsExc(P),
//@             throwsExc(Init), returns(P)};

//@ declares   locals(P);
//@ assignable locals(P), result;
//@ accessible locals(P), result, args;
//@ return_behavior requires returns(P);
//@ exceptional_behavior requires throwsExc(P);
abstract_statement P;

//@ assignable hasTo(b);
//@ accessible args;
//@ return_behavior requires false;
//@ exceptional_behavior requires throwsExc(Init);
abstract_statement Init;
```

```
if (b) {
  //@ declares   locals(Q1);
  //@ assignable \everything;
  //@ accessible \everything;
  abstract_statement Q1;
} else {
  //@ declares   locals(Q2);
  //@ assignable \everything;
  //@ accessible \everything;
  abstract_statement Q2;
}
```

to b. Thus, P can indirectly influence the control flow by assigning a value to the variable result which could then affect Init's choice for b. To address this issue we proceed as follows:

We add to all APSs in Fig. 1 (except Init) a "**declares**" annotation. It declares abstract "Skolem" location sets that can be instantiated with arbitrary concrete local variable declarations visible from outside. For example, to P we add the annotation "`//@ declares locals(P);`". To specify that a method containing APSs receives an unknown set "args" of parameters, we annotate it with "`//@ declares args;`". Abstract location sets in declarations can be declared final by surrounding them with "**final**$(\cdot)$". This prevents them from occurring in **assignable** clauses. Continuing the example, we add to P the annotation "`//@ assignable locals(P), result;`", to Init "`//@ accessible args;`".

Proving correctness of the refactoring still fails, however, for two reasons. The first is: we have not excluded that Init contains a **return** statement. Since Init's only task is to initalize b, this should never happen. We add the annotation "`//@ return_behavior requires false;`" to Init, specifying that a **return** requires the specified condition—here falsity—which excludes returning.

The second reason why the refactoring is not yet correct is that Init might raise an exception. This is entirely possible and a real problem: If we permit P to return or throw an exception, but Init may also throw an exception, then the refactored and the original program have different behavior. We have two options: (i) We deny Init to throw an exception by adding the annotation "`//@ exceptional_behavior requires false;`" or, more generally, (ii) we enforce that *if* Init throws an exception, then P can neither throw an exception nor can it return. The latter is achieved with the help of the abstract functions **throwsExc**(Init), **throwsExc**(P), and **returns**(P) of *Boolean* type. They qualify the **requires** clauses that restrict exceptional and returning behavior of Init and P. A global **axiom** declares them to be mutually exclusive.

Listing 3 shows the specified refactored program. Similar annotations apply to the original version. For Q1 and Q2, we permit assigning/accessing all loca-

$$\textsf{assignment}\ \frac{\Gamma \vdash \{\mathcal{U}\}\{\mathrm{x} := e\}[\pi\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathrm{x}{=}e;\ \omega]\varphi, \Delta}\ e \text{ is side effect-free}$$

Fig. 2: SE rule for variable assignment

tions ("`\everything`"). This is default, so these declarations can be left out. The annotation "`//@ declares final(args);`" would have to be added to the surrounding method declaration. With the specification in Listing 3 we can prove *equivalence* of the original and refactored program for *any concrete instance* matching the abstract program structurally and satisfying its I/O and control flow constraints. Having proved equivalence, the refactoring can be applied in either direction. This is relevant: many of Fowler's refactorings are bi-directional. We continue with formalizing AE over constrained APSs in a program logic.

3 Abstract Execution Logic

Our implementation of AE is realized on top of the symbolic execution framework of the deductive verification system KeY [1]. It is based on JAVA Dynamic Logic (JavaDL), a program logic for the JAVA language.

3.1 Principles of JavaDL

JavaDL extends sorted First-Order Logic (FOL). JAVA programs appear inside logical formulas as *modalities*, of which there are two types: The box modality $[p]\varphi$ expresses that *if* program p terminates, then the postcondition φ holds in any final state (partial correctness). The diamond modality $\langle p\rangle\varphi$ additionally requires p to terminate (total correctness). To prove the validity of formulas, i.e. $[p]\varphi$ or $\langle p\rangle\varphi$ holds in any initial state, JavaDL has a sequent calculus comprising FOL and theory-specific rules, as well as rules realizing SE of JAVA programs. A Hoare triple $\{\psi\}\, p\, \{\varphi\}$ is equivalent to the JavaDL formula $\psi \to [p]\varphi$.

The SE rules of the JavaDL calculus reduce a JAVA statement to first-order assumptions and a separate syntactic category called symbolic *updates* representing symbolic state transitions. The atomic building blocks of updates are the empty update *Skip*, representing the identity state transition, and the *elementary update* $\mathrm{x} := t$ for the state transition where variable x is assigned the value of term t. Two updates $\mathcal{U}_1$, $\mathcal{U}_2$ can be combined into a *parallel* update $\mathcal{U}_1 \,||\, \mathcal{U}_2$: the state changes of $\mathcal{U}_1$ and $\mathcal{U}_2$ are executed *simultaneously*; in case both assign to the same variable, the assignment in $\mathcal{U}_2$ "wins". Only in the absence of such "conflicts", parallel composition is commutative. Updates are applied to terms t and formulas φ: $\{\mathcal{U}\}t$ and $\{\mathcal{U}\}\varphi$ represent the value of term t and truth value of formula φ after the state change effected by $\mathcal{U}$, respectively. Parallel update composition $\{\mathcal{U}_1 \,||\, \mathcal{U}_2\}\varphi$ is different from sequential composition $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$. In the sequential case, right-hand sides of $\mathcal{U}_2$ are interpreted in the state resulting from $\mathcal{U}_1$. In the parallel case, they are interpreted in the *same* pre-state. The formula $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$ is equivalent to $\{\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2\}\varphi$.

Fig. 2 depicts an SE rule using an update to represent the effect of an assignment of an expression without side effects to variable x. As usual, sequent calculus rules are read "bottom-up", i.e. the rule symbolically executes the assignment by turning it into a symbolic update. SE rules operate on the first active (i.e. executable) statement of a program, here the assignment. The remaining program is contained in $\pi\,\omega$, where the prefix π consists of opening braces, labels, try-blocks, etc., and the postfix ω of closing braces, blocks, and remaining statements. The program $\pi\,\omega$ is a well-formed JAVA program. Sequent calculus rules have zero or more premises (sequents on top of the rule); zero premises characterize a rule that closes a proof case, more than *one* premise causes a proof to split. An example of the latter is the rule for an **if**-statement (see web appendix). The conclusion (bottom part) of a rule consists of exactly one sequent. A rule is *sound* if the correctness of the conclusion follows from the correctness of all premises. A sequent $\Gamma \vdash \Delta$ is correct if the conjunction of the formulas in the set Γ implies the disjunction of those in Δ. Details are in [1].

We need a recently introduced concept of JavaDL for reasoning about loops: *loop scopes* [30]. A loop scope $\circlearrowleft_{\mathtt{x}} p \,_{\mathtt{x}}\circlearrowleft$ is a scope for a loop body p. It results from SE of a loop **while**(b){p}. The boolean flag x encodes *completion information* about the loop: it is set to TRUE if the loop is *exited* (either normally or by irregular termination) and to FALSE if it *continues* with another iteration. Using the value of x, a postcondition can distinguish both cases.

3.2 Formalization of Abstract Execution

We first give a definition of the domain of *locations* used in **declares**, **accessible** and **assignable** specifications of APSs. The symbol "*allLocs*" is introduced for the "**\everything**" specifier in **accessible** and **assignable** specifications.

Definition 1. *The set* $Locs^{Concr}$ *of* concrete locations *consists of program variables* x *and, for an object* o *and field identifier* f*, field locations* (o, f)*. The set* $Locs^{Sk}$ *of* Skolem location sets *consists of uninterpreted functions* loc^{Sk} *representing arbitrary sets of concrete locations* $Locs^{Concr}$*. We define* $Locs = Locs^{Concr} \cup Locs^{Sk} \cup \{allLocs\}$*, where the symbol allLocs represents all concrete locations* $Locs^{Concr}$*. The set of* assignable *locations is defined as* $Locs^{Assgn} = Locs \cup \{loc^{!} \mid loc \in Locs \setminus \{allLocs\}\}$ *which also includes "have-to" locations* $loc^{!}$*.*

In Sect. 2, we introduced the specification elements of APSs. These are formalized in the subsequent definition of the logic representation of an APS.

Definition 2. *Let id be an identifier symbol,* $decls \subseteq Locs^{Sk}$*,* $assignables \subseteq Locs^{Assgn}$ *and* $accessibles \subseteq Locs$*. An* Abstract Placeholder Statement *is a tuple*

$$(id, decls, assignables, accessibles, specs)$$

where "specs" represents behavioral specifications and is a tuple of the form:

$$(returnsSpec, excSpec, continuesSpec,$$
$$breaksSpec, continuesSpecLbl, breakSpecLbl)$$

The elements returnsSpec, excSpec, continuesSpec, breaksSpec are optional: they are the empty set or a singleton of a formula specifying when an APS returns, throws an exception, continues, and breaks, respectively. Elements breakSpecLbl, continuesSpecLbl are partial functions from labels to formulas, specifying when an APS continues a labeled loop or breaks from a labeled block or loop.

We write "APS P" short for "the APS with identifier symbol P". Abstract Execution reasons about the behavior of *all possible concrete programs* with which APSs "legally" may be instantiated, formally:

Definition 3. *Let p_a be an abstract program with occurrences of APS symbols $\mathtt{P}_1, \ldots \mathtt{P}_n$. We call the substitution with concrete programs $\mathtt{P}_i^0$ for each $\mathtt{P}_i$ a* legal instantiation *iff (1) the result from substituting all occurrences of $\mathtt{P}_i$ by $\mathtt{P}_i^0$ in p_a is a compilable* Java *program, and (2) all $\mathtt{P}_i^0$ satisfy all constraints of the APSs declaring the $\mathtt{P}_i$.*

Example 1. We substitute P in Listing 3 with "`int z=result++;`". This is legal if z is undeclared in the visible scope. The substitution instantiates the location set "`locals(P)`" with {z}, affecting further instantiations referring to it. The substitution of "`y=z;`" for P is illegal: First, it *assigns* a variable which is not contained in its assignable set; second, if z is not contained in the instantiation of args, it *accesses* an undeclared variable z; third, the program is not *compilable* if y and z are undeclared. Let param be in the instantiation of args. Substituting P with "`int z=result/param;`" and Init with "`b=result/param;`" is illegal since *both* could throw an exception, contradicting the axiom.

SE of an APS must over-approximate the behavior of *all* legal instantiations. To model this in the logic, we use second-order *Skolemization*. Given an APS (P, *decls*, *assignables*, *accessibles*, *specs*), we create what we call a *Skolem update* "$\mathcal{U}_{\mathtt{P}}(\mathit{assignables} :\approx \mathit{accessibles})$" with *Skolem path condition* "$C_{\mathtt{P}}(\mathit{accessibles})$" fresh for P. The term "fresh for" means that the symbols $\mathcal{U}_{\mathtt{P}}$, $C_{\mathtt{P}}$ are created freshly (as usual for Skolemization) when P first occurs in a proof context, but are re-used each time when P re-occurs. This ensures that each occurrence of an APS symbol represents the *same* program. We define the meaning of Skolem update and Skolem path condition by extending the notion of legal instantiation. In the definition we assume all Skolem location sets in Locs^{Sk} to be instantiated with concrete locations.

Definition 4. *Let an APS P with assignables $\subseteq \mathit{Locs}^{Assgn} \setminus \mathit{Locs}^{Sk}$, accessibles $\subseteq \mathit{Locs} \setminus \mathit{Locs}^{Sk}$ be given. An abstract update $\mathcal{U}_{\mathtt{P}}(\mathit{assignables} :\approx \mathit{accessibles})$ may be instantiated with any concrete update $\mathtt{x}_1 := t_1 \,||\, \ldots \,||\, \mathtt{x}_n := t_n$ for which the following conditions hold: (1) either allLocs $\in$ accessibles or the t_i depend at most on locations in accessibles; (2) either allLocs $\in$ assignables or for each $\mathtt{x}_i$ one of $\mathtt{x}_i \in$ assignables or $\mathtt{x}_i^! \in$ assignables; (3) for all $\mathtt{x}^! \in$ assignables, there is an i such that $\mathtt{x} = \mathtt{x}_i$. An abstract path condition $C_{\mathtt{P}}(\mathit{accessibles})$ may be instantiated with any closed formula φ depending at most on locations in accessibles.*

$$\mathsf{simpleAERule}\ \frac{\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathtt{P}}(\mathit{allLocs} :\approx \mathit{allLocs})\}(C_{\mathtt{P}}(\mathit{allLocs}) \to [\pi\ \omega]\varphi), \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{abstract\_statement}\ \mathtt{P};\ \omega]\varphi, \Delta}$$

Fig. 3: Simple AE rule without abrupt termination

Definition 5. Abstract *JavaDL extends JavaDL syntax as follows: (1) updates can be Skolem updates; (2) Skolem path conditions are also formulas; (3) programs can be abstract.* Abstract *sequents and sequent rules are defined as before, but range over abstract JavaDL formulas.*

A JavaDL sequent calculus rule is *sound* if the validity of the conclusion follows from the validity of all premises [1]. We can leave this definition *unchanged* provided that we define validity of abstract sequents suitably:

Definition 6. *A sequent S^0 is a* legal instantiation *of an abstract sequent S if S^0 results from substituting all Skolem updates, Skolem path conditions and APS symbols in S with legal instantiations. An abstract sequent is* valid *iff all its legal instantiations are valid in JavaDL.*

One of the simplest possible AE rules (first mentioned in [29]) is shown in Fig. 3. It is only applicable for an APS whose specification and legal instantiations exclude irregular termination.

Theorem 1. *The Abstract Execution rule* simpleAERule *(Fig. 3) is sound.*

Proof. Let $\mathtt{P}^0$ be any legal instantiation of $\mathtt{P}$. Since $\mathtt{P}^0$ cannot terminate irregularly, symbolic execution transforms the sequent $\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{P}^0\ \omega]\varphi, \Delta$ to one of the shape $\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}^0\}(C^0 \to [\pi\ \omega]\varphi), \Delta$.[1] Assume the premise of simpleAERule is valid (otherwise, the rule is trivially sound). The instantiations $\mathcal{U}^0$ of $\mathcal{U}_{\mathtt{P}}$ and C^0 of $C_{\mathtt{P}}$ are legal (the *allLocs* location allows reading and writing arbitrary locations). So, by assumption, $\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}^0\}(C^0 \to [\pi\ \omega]\varphi), \Delta$ is valid and, by soundness of SE, also the conclusion. Since $\mathtt{P}^0$ was chosen arbitrarily, the abstract sequent in the conclusion of simpleAERule is valid. □

The simpleAERule rule is unsatisfactory: it is too restrictive on irregular termination. It is also too abstract, because the abstract update and path condition in the premise may write and read any location. The abstract update can erase all variables and the whole heap, which prevents proving interesting properties. More useful rules can be obtained for *specific contexts* in which an APS occurs in the conclusion. Depending on the context, legal instantiations can lead to different ways of irregular termination.

Fig. 4 shows a rule for AE within a loop scope and a non-void method, but outside the scope of *loop* labels. In contrast to simpleAERule, nonVoidLoopAERule

[1] If the statement causes a split, like an `if` statement, we still can combine the arising sequents to a single one by state merging [27].

nonVoidLoopAERule

$$
\frac{\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{\mathtt{P}}(\mathit{assignables} :\approx \mathit{accessibles})\}\\
\quad\{\mathtt{returns} := \mathit{returns}_0 \,||\, \mathtt{result} := \mathit{result}_0 \,||\, \mathtt{exc} := \mathit{exc}_0 \,||\\
\quad\ \mathtt{breaks} := \mathit{breaks}_0 \,||\, \mathtt{continues} := \mathit{continues}_0 \,||\\
\quad\ \mathtt{breaksToLbl\_1} := \mathit{breaksToLabel1}_0 \,||\, \cdots \,||\\
\quad\ \mathtt{breaksToLbl\_n} := \mathit{breaksToLabeln}_0\}\\
\Big(\quad C_{\mathtt{P}}(\mathit{accessibles})\\
\ \wedge\ \mathit{mutex}\,(\mathtt{returns}, \mathtt{exc} \neq \mathtt{null}, \mathtt{breaksToLbl\_1}, \cdots, \mathtt{breaksToLbl\_n})\\
\ \wedge\ (\mathtt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^{?}\ \ \wedge\ (\mathtt{exc} \neq \mathtt{null} \leftrightarrow \mathit{excSpec})^{?}\\
\ \wedge\ (\mathtt{breaks} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksSpec})^{?}\\
\ \wedge\ (\mathtt{continues} \doteq \mathrm{TRUE} \leftrightarrow \mathit{continuesSpec})^{?}\\
\ \wedge\ (\mathtt{breaksToLbl\_1} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLbl1Spec})^{?} \wedge \cdots\\
\ \wedge\ (\mathtt{breaksToLbl\_n} \doteq \mathrm{TRUE} \leftrightarrow \mathit{breaksLblnSpec})^{?}\\
\rightarrow [\pi\ l_1 : \{ \cdots \{\ l_n : \{\\
\quad \circlearrowleft_{\mathtt{x}}\ \mathtt{if}\ (\mathtt{returns})\ \mathtt{return}\ \mathtt{result};\ \mathtt{if}\ (\mathtt{exc}\ \mathtt{!=}\ \mathtt{null})\ \mathtt{throw}\ \mathtt{exc};\\
\quad\ \ \mathtt{if}\ (\mathtt{breaks})\ \mathtt{break};\ \ \ \mathtt{if}\ (\mathtt{continues})\ \mathtt{continue};\\
\quad\ \ \mathtt{if}\ (\mathtt{breaksToLbl\_1})\ \mathtt{break}\ l_1;\ \cdots\ \mathtt{if}\ (\mathtt{breaksToLbl\_n})\ \mathtt{break}\ l_n;\\
\quad\ \ \mathit{Rest}_1\ {}_{\mathtt{x}}\circlearrowleft\\
\quad \mathit{Rest}_2\ \}\}\cdots\}\ \omega]\varphi\Big), \Delta
\end{array}}{\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}[\pi\ l_1 : \{ \cdots \{ l_n : \{\\
\quad \circlearrowleft_{\mathtt{x}}\ \mathtt{abstract\_statement}\ \mathtt{P};\ \mathit{Rest}_1\ {}_{\mathtt{x}}\circlearrowleft\ \mathit{Rest}_2\\
\}\}\cdots\}\ \omega]\varphi, \Delta
\end{array}}
$$

Fig. 4: AE rule for an APS within a loop scope.

uses the *assignables*/*accessibles* specifications of the APS syntax. Irregular termination is modeled by **if** statements inside the loop scope in the premise. The conditionals depend on variables initialized with fresh constants in the update after the abstract update. E.g., `returns` is initialized with a constant $\mathit{returns}_0$.

Without a specific context, SE will split at each **if** statement and follow both branches, e.g., one where `P` returns and one where it does not. Using the behavior specification, this can be fine-tuned: For example, in the path condition in the premise, there is an optional (marked with $^{?}$) conjunct "$(\mathtt{returns} \doteq \mathrm{TRUE} \leftrightarrow \mathit{returnsSpec})^{?}$". This lets one control the value of the guard `returns` with the formula *returnsSpec*. The behavior specifications stem from the specifications of the abstract symbol `P` in the conclusion as detailed in Sect. 2. The function *mutex* is interpreted such that at most one of its arguments is true at any time: here this specifies that there is not more than one reason for a program to terminate irregularly.[2] A proof of the following Thm. 2 is provided in the web appendix.

Theorem 2. *The Abstract Execution rule* nonVoidLoopAERule *(Fig. 4) is sound.*

Design principles for AE rules. The principles underlying the above rules apply to other sequential languages than JAVA as well. To create a new AE rule, we proceed as follows. Given a context in which an APS declaration is the active statement (a loop, method, labeled block, etc.), we model possible side effects of

[2] It is possible that, for instance, during *returning* an exception is thrown: this simply means that *exception* is the reason for termination.

that APS with separate, conditioned SE branches in the premise. For soundness it is crucial not to miss any irregular termination cases. We point out that instead of performing exhaustive structural induction, we merely distinguish *different paths of program completion*. For paths depending on values (as for a `return` or exception), Skolem constants are introduced. The conditioned premises depend on flags that establish a link to the APS's specification; abstract updates and path conditions are added as in nonVoidLoopAERule. AE rules are not specific to the target application of this paper (correctness of refactoring rules), but can be used in any of the areas mentioned in the introduction.

3.3 Abstract Update Simplification

The JavaDL calculus comes with many simplification rules for concrete updates: $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$ simplifies to $\{\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2\}\varphi$, updates applied to formulas without program variables are removed, etc. In addition, spurious updates, such as those assigning variables not occurring in their scope, are removed. To reason about abstract programs, we need corresponding rules for abstract updates.

We designed a set of simplification rules for abstract updates (see Table 2): (1) Remove spurious updates: From the *assignables* part of an abstract update, delete those not occurring in the scope or that are overwritten before being read; (2) two rules handle the interplay between concrete and abstract updates; (3) two rules handle *concatenation* of abstract updates: When we cannot further simplify a formula $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$, we connect the abstract updates by a concatenation operator, resulting in $\{\mathcal{U}_1 \circ \mathcal{U}_2\}\varphi$. This is not needed for concrete updates which are directly simplified as shown above. Within a concatenation, abstract updates can be commuted if their assignable/accessible specifications do not interfere.

4 Proving the Correctness of Refactoring Techniques

We studied five refactoring techniques from Fowler's classic book [11] and three from the second edition [12]. We choose refactorings operating at the statement level, because they are directly expressible in JavaDL and—for the time being—exclude techniques that reorganize class hierarchies, rename constructs, or move methods. For each of the eight techniques we formalized the starting point and the result of the refactoring as a suitably specified abstract program (see Sect. 2), and then proved their *equivalence* with the AE calculus discussed in Sect. 3. Thus, we obtain soundness of, for example, *Extract Method* at the same time as of its inverse, *Inline Method*. All proofs are fully mechanized in KeY [1].

Methodology. For each refactoring, we create a JAVA class `Refactor` with two public methods: `before` contains an abstract program representing the input to the refactoring, `after` the refactored result. We start with minimal annotations in the occurring APSs including `declares` directives and standard return and assignable specifications for the abstract expressions idiom. The following JavaDL formula performs AE of an abstract program `p` on a `Refactor` object `o`

Table 2: Simplification rules for abstract updates

Simplification Scheme	Description
$\{\mathcal{U}_\mathtt{P}(\mathtt{x}, \mathtt{y} :\approx \ldots)\}\varphi(\mathtt{x}) \rightsquigarrow$ $\{\mathcal{U}_\mathtt{P}(\mathtt{x} :\approx \ldots)\}\varphi(\mathtt{x})$	Ineffective assignables are removed, abstract updates with no assignables dropped.
$\{\mathtt{x}_1 := t_1 \,\|\|\, \mathtt{x}_2 := t_2 \,\|\| \ldots\}$ $\{\mathcal{U}_\mathtt{P}(\ldots, \mathtt{x}_1, \ldots :\approx$ $\ldots, \mathtt{x}_2, \ldots)\}\varphi \rightsquigarrow$ $\{\mathtt{x}_1 := t_1 \,\|\| \ldots\}$ $\{\mathcal{U}_\mathtt{P}(\ldots, \mathtt{x}_1, \ldots :\approx$ $\ldots, t_2, \ldots)\}\{\mathtt{x}_2 := t_2\}\varphi$	Applies variable assignments to the accessibles of the abstract update and pushes down elementary updates not assigned by the abstract update. Elementary updates that *have to* be assigned (overwritten) are dropped later.
$\{\mathcal{U}_\mathtt{P}(\ldots, \mathtt{y}', \ldots :\approx \ldots)\}$ $\{\ldots \,\|\| \mathtt{x} := \mathtt{y} \,\|\| \ldots\}\varphi(\mathtt{x}) \rightsquigarrow$ $\{\mathcal{U}_\mathtt{P}(\ldots, \mathtt{x}', \ldots :\approx \ldots)\}$ $\{\ldots \,\|\| \ldots\}\varphi(\mathtt{x})$	Eliminates a renaming substitution "y for x": Since the concrete update assigns to x a value chosen by $\mathcal{U}_\mathtt{P}$ (which *has to* assign y), the abstract update can as well directly choose that value. Sound because y is not contained in φ.
$\{\mathcal{U}_\mathtt{P}\}\{\mathcal{U}_\mathtt{Q}\}\varphi \rightsquigarrow \{\mathcal{U}_\mathtt{P} \circ \mathcal{U}_\mathtt{Q}\}\varphi$	Sequential to "concatenated" update application.
$\mathcal{U}_\mathtt{P}(\mathit{assgn}_1 :\approx \mathit{access}_1)\circ$ $\mathcal{U}_\mathtt{Q}(\mathit{assgn}_2 :\approx \mathit{access}_2) \rightsquigarrow$ $\mathcal{U}_\mathtt{Q}(\mathit{assgn}_2 :\approx \mathit{access}_2)\circ$ $\mathcal{U}_\mathtt{P}(\mathit{assgn}_1 :\approx \mathit{access}_1)$	Abstract updates within concatenations can be commuted if their assignable and accessible specifiers are *independent*; i.e., assgn_1 has to be disjoint from assgn_2 and access_2, and similarly vice versa.

and records the result ("Flag" in the postcondition is explained later):

$$\begin{aligned}\mathrm{AE}(\mathtt{p}, \mathrm{Flag}) = \{&\mathtt{result} := \mathit{result}_0 \,\|\| \ldots\}\\ &\langle \mathtt{try}\ \{\mathtt{result=o.p(result,\ldots)@Refactor;}\}\\ &\ \mathtt{catch}\ (\mathtt{Throwable\ t})\ \{\mathtt{result=t;}\}\rangle \mathit{Post}(\mathtt{result}, \mathrm{Flag})\end{aligned} \tag{1}$$

Equivalence of the original and the refactored program is established by proving the formula "AE(before, TRUE) $\leftrightarrow$ AE(after, TRUE)". This is loaded into KeY and an automatic proof is started. In all but one refactoring technique (first in Table 3), the proof cannot be finished and open goals remain. The reason is—quite simply—that Fowler's refactoring techniques are not sound in general [10]. As he points out, they rely on robust test suites and a "try-compile-and-test" loop trusted to unveil potential faults introduced by a refactoring.

In our setting we have the opportunity to restrict the programs that can be soundly refactored via suitable annotations of the APSs used to describe refactoring source and target. Fowler in most cases does not mention these restrictions. Fortunately, inspection of uncloseable proof goals provides clear hints on the nature of the required annotations.

A typical example is when all open proof goals expect an APS P to throw an exception (assumption exc $\neq$ null occurs in each unprovable goal). This is addressed by adding a constraint on P that forbids to throw exceptions. Another common situation concerns too liberal **accessible**/**assignable** specifications. These lead to open goals that contain a sequent of the form $\{\mathcal{U}\}\varphi \vdash \{\mathcal{U}'\}\varphi$ that

Listing 4: Before

```
done = false; i = 0;
while (!done && i < threshold) {
  if (condition) {
    abstract_statement Body;
    done = true;
  }
  i++;
}
return result;
```

Listing 5: After

```
i = 0;
while (i < threshold) {
  if (condition) {
    abstract_statement Body;
    break;
  }
  i++;
}
return result;
```

Fig. 5: The *Remove Control Flag* Refactoring Technique

becomes valid when the abstract updates $\mathcal{U}$ and $\mathcal{U}'$ are identical. Any differences give hints on possible annotations that permit to close the proof.

Once a proof is complete, the formalization of a refactoring should be checked for validity, i.e. whether the intention of the refactoring technique has been faithfully captured. Specifications should not be more restrictive than necessary and permit substituting non-trivial programs for APSs. For example, it is easy, but useless, to find a proof with "`assignable \nothing;`" specifications. For each behavioral restriction, there should be a convincing justification. We discovered non-trivial and justifiable restrictions for almost all the investigated refactoring techniques (summarized in Table 3). Source code samples are in the appendix.

Complexity of checking legal substitutions. A closed equivalence proof about abstract programs asserts that those programs behave equivalently for all legal substitutions of concrete programs for APS symbols (Def. 3). Consequently, for each concrete program one must check that all the constraints specified in APSs are satisfied. These include syntactic restrictions (e.g., when inlining a method, there are no recursive calls in the body) as well as behavioral ones. The latter are not necessarily automatically checkable or even decidable. For example, to decide whether a program throws an exception, is equivalent to reachability. Even so, a formalized and proven refactoring technique makes its requirements *explicit* that before were mentioned only informally (if at all). Not in general, but in practice quite often, constraints can be proven in KeY.

Multiple specifications. It can make sense to create *multiple* formalizations of the same refactoring technique. The restrictions that ensure soundness can differ depending on (1) the program *context* (inside/outside a loop or labeled block), and (2) the termination mode (normal, exceptional, break, etc.). The "consolidate duplicate conditional fragments" technique in Table 3 exemplifies this.

Programs with loops. We studied two refactoring techniques involving loops: *Remove Control Flag* [11] and *Split Loop* [12]. To handle loops we make use of the Flag in formula (1) to separate runs leaving the loop from those leading to further executions of the loop body. The value of `Flag` is controlled by the loop scope parameter (the invariant rule in Fig. 7 in the appendix contains details).

Table 3: Studied refactoring techniques and discovered behavioral constraints.

Refactoring Technique	Discovered Restrictions	Justification & Remarks
Consolidate duplicate conditional fragments (Extract Postfix) [11]	none	—
Consolidate duplicate conditional fragments (Extract Prefix) [11]	• `if` guard may only throw exc. if prefix terminates normally; • pulled out statement may assign heap and parameters iff guard does not access it.	Irregular termination in both guard and prefix affect final result. Influence on accessibles of guard can change control flow (whether `if` or `else` is taken).
Consolidate duplicate conditional fragments (Postfix of `try-catch` to `finally`) [11]	• Program in `try` block may not return; • program in `catch` block may not return or throw exception.	A `finally` block is always executed, even after a return. This changes the returned result before and after the refactoring.
Consolidate duplicate conditional fragments (Postfix of `try-catch`, no `finally`) [11]	none	Fowler talks about moving the postfix "to the final block", leaving unspecified whether this refers to the `finally` block or to the statements after `try`. Only in the latter case no restrictions apply.
Decompose Conditional [11]	Special case of "Extract Method", see below.	
Extract Method [11]	Extracted fragment • must not return; • may only assign heap and a local variable.	A return after extraction does not affect the top-level method.
Replace Exception with Test [11]	There has to be a "rollback" program in the `catch`/`else` block, program in `try`/`if` may only assign variables reset by that program.	The program in `try`/`if` may change part of the state before throwing an exception, therefore the result after the exception/test can differ.
Move Statements to Callers [12]	Neither the moved statement nor the remaining program may return.	If the remaining program in the called method returns early, the moved one is not executed after, but before refactoring.
Slide Statements [12]	All programs participating in the sliding, i.e. the swapped parts and the one in the middle • must not return or throw an exception; • must be *independent*.	*Independent* means the participating programs write to locations that the others do not access. Abrupt termination would change the result.
Split Loop [12]	Explained in text.	Example with loop.
Remove Control Flag [11]	Explained in text.	Example with loop.

The *Remove Control Flag* refactoring in Fig. 5 is interesting, because the number of runs of the loop before and after the refactoring differs by one (the guard needs to be executed one extra time before). This complicates the proof since we obtain *Post*(_result, FALSE) for one case and *Post*(_result, TRUE) for the other. We solve this by harmonizing the iteration structure via an *unrolling technique* [16] and an intermediate refactoring. Alternatively, one can code the unrolling inside a modified loop invariant rule (Fig. 9 in the appendix).

The *Split Loop* refactoring splits a loop with two independent parts into two successive loops. We had to supply several annotations to the APSs. For instance, the first part of the loop body must not break or continue (since otherwise, the second part is skipped, which is not the case after the refactoring), while the second part must not return, throw an exception, or break, since then we would have to relate runs continuing loop iteration with others exiting the loop.

Performance All proofs are performed automatically in KeY without user interaction. For the refactorings with loops, currently small proof scripts ($\approx$ 40 lines) are needed for loop coupling. The proofs have 2,900–40,000 nodes (median: 7,100) and take 6–300 s (median: 29 s) to complete. All problem files with detailed statistics, together with a KeY version implementing AE, are available at `key-project.org/papers/ae/`.

5 Related Work

The idea of Abstract Execution was first mentioned in our earlier work [29], where it is used to formalize the correctness of compilation rules of a JAVA-to-LLVM IR compiler. There, APSs could not be annotated and irregular termination was excluded; also, every APS can assign and access any location. In the present paper we lift these restrictions, provide an implementation and a case study. Abstract execution of APSs can be seen as a generalization of *abstract operation contracts* [6,15] to abstract *block* contracts. In the former, contracts are abstract, but programs concrete; we generalize this to abstract *programs*. This amounts to encoding limited second-order inference (no induction, no higher-order quantification) over programs into first-order (dynamic) logic.

The principal use cases for AE reside in the area of *relational verification* [4], which includes, but is not limited to: general-purpose relational proofs about programs [2,16], correctness proofs for refactorings [13], regression verification [14], proven-correct compilation [21,31] and compiler optimizations [23,18], program synthesis [28], information flow properties (e.g., by self-composition [3,9]).

There are several approaches to prove relational properties of *concrete* programs (e.g., LLRÊVE [16]). Barthe et al. [2] propose the construction of *product programs* from two variable-disjoint programs as a general-purpose technique for verifying relational program properties. After execution of the product program, the result is checked for correctness, i.e. equality. This works even for structurally different programs. Instead, we execute both programs in isolation in an equivalence proof. This has the drawback of requiring a certain structural

similarity of the programs and explicit loop coupling, but it is more resilient in the presence of irregular or non-termination. Product programs and AE are not mutually exclusive: One can create a *product* of *abstract* programs.

Garrido & Meseguer [13] prove correctness of three JAVA refactoring techniques based on an executable Maude semantics of JAVA. They focus on refactorings not targeted by us (e.g., *Pull Up Field*). It is unclear whether this approach works for statement-based refactorings including loops. Alive [23] permits proving automatically the correctness of "peephole optimizations" for LLVM. While this approach reasons about classes of programs, it is parametric only in register names and imposes other serious restrictions (e.g., no loops). Eilertsen et al. [10] generate semantic correctness assertions that ensure preservation of program semantics after refactoring. They work on concrete programs and perform run-time, not static checking.

Godlin & Strichman [14] perform "Regression Verification" by transforming loops into recursive functions and replacing recursive calls with uninterpreted function symbols. The latter are similar to APSs, however, side effects or irregular termination cannot be modeled, because functions are pure. Mechtaev et al. [25] propose a mechanism for proving existential second-order properties based on symbolic functions. Their goal is to find existential witnesses for those functions by synthesis from a user-specified grammar. In contrast, we aim at *universal* properties, and APSs represent statements (with side effects), not functions. The PEC system [18] for proving the correctness of compiler optimizations uses *meta variables* ranging over expressions, variables and statements. The latter are "single-entry-single-exit", whereas APSs can have multiple exit points, including irregular termination. In addition, we permit annotations that constrain possible behavior. The property to be proven in [18] is a certain bi-simulation relation which is somewhat inflexible and requires lockstep execution.

6 Conclusion and Future Work

We proposed *Abstract Execution* of programs that contain APS symbols, a new software analysis principle extending symbolic execution. AE permits to *automatically* reason about partially unspecified programs. APSs allow irregular termination and include specification of assignable and accessible locations as well as of termination behavior. This generalizes other approaches going into similar directions. We implemented our method and applied it to eight JAVA refactoring techniques, of which two require reasoning about loops. Our formalization of the refactoring techniques makes implicit requirements explicit. It helps to better understand and safely apply refactorings. We plan to investigate how to support structurally different programs (e.g., comparing iterative and recursive versions of the same algorithm), concurrent programs, and we intend to look at other application areas. To prove the correctness of compiler optimizations automatically using AE is a natural follow-up to our work on refactoring techniques.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book: From Theory to Practice, LNCS, vol. 10001. Springer (2016)
2. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs. In: Butler, M.J., Schulte, W. (eds.) FM: Formal Methods, 17th Intl. Symp. on Formal Methods, Limerick, Ireland. LNCS, vol. 6664, pp. 200–214. Springer (2011)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure Information Flow by Self-Composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17,Pacific Grove, CA, USA. pp. 100–114. IEEE Computer Society (2004)
4. Beckert, B., Ulbrich, M.: Trends in Relational Program Verification. In: Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday. pp. 41–58 (2018)
5. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution. ACM SIGPLAN Notices **10**(6), 234–245 (Jun 1975)
6. Bubel, R., Hähnle, R., Pelevina, M.: Fully Abstract Operation Contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece. LNCS, vol. 8803, pp. 120–134. Springer (Oct 2014)
7. Burstall, R.M.: Proving Properties of Programs by Structural Induction. The Computer Journal **12**(1), 41–48 (1969)
8. Burstall, R.M.: Program Proving as Hand Simulation with a Little Induction. In: Information Processing '74, pp. 308–312. Elsevier/North-Holland (1974)
9. Darvas, A., Hähnle, R., Sands, D.: A Theorem Proving Approach to Analysis of Secure Information Flow. In: Hutter, D., Ullmann, M. (eds.) Proc. 2nd International Conference on Security in Pervasive Computing. LNCS, vol. 3450, pp. 193–209. Springer (2005)
10. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer Refactorings. In: Proc. 7th Intern. Symp. on Leveraging Applications of Formal Methods, ISoLA. pp. 517–531 (2016)
11. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series, Addison-Wesley (Jun 1999)
12. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Signature Series, Addison-Wesley Professional (Nov 2018), 2nd edition
13. Garrido, A., Meseguer, J.: Formal Specification and Verification of Java Refactorings. In: Proc. 6th IEEE Intern. Workshop on Source Code Analysis and Manipulation. pp. 165–174. SCAM '06, IEEE Computer Society (2006)
14. Godlin, B., Strichman, O.: Regression Verification: Proving the Equivalence of Similar Programs. Softw. Test., Verif. Reliab. **23**(3), 241–258 (2013)
15. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in Software Verification by Abstract Method Calls. In: Bonacina, M.P. (ed.) Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA. LNCS, vol. 7898, pp. 300–314. Springer (2013)
16. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational Program Reasoning Using Compiler IR - Combining Static Verification and Dynamic Analysis. J. Autom. Reasoning **60**(3), 337–363 (2018)
17. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7), 385–394 (Jul 1976)
18. Kundu, S., Tatlock, Z., Lerner, S.: Proving Optimizations Correct Using Parameterized Program Equivalence. In: Proc. PLDI 2009. pp. 327–337 (2009)

19. Lanzinger, F.: A Divide-and-Conquer Strategy with Block and Loop Contracts for Deductive Program Verification. Bachelor Thesis, Institute of Theoretical Informatics, Karlsruhe Institute of Technology (April 2018)
20. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 2013), http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf, draft revision 2344
21. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM **52**(7), 107–115 (2009)
22. London, R.L.: Correctness of a Compiler for a Lisp Subset. In: Prof. of ACM Conf. on Proving Assertions About Programs. pp. 121–127. ACM (1972)
23. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Practical Verification of Peephole Optimizations with Alive. Commun. ACM **61**(2), 84–91 (2018)
24. McCarthy, J., Painter, J.: Correctness of a Compiler for Arithmetic Expressions. Mathematical Aspects of Computer Science **1** (1967)
25. Mechtaev, S., Griggio, A., Cimatti, A., Roychoudhury, A.: Symbolic Execution with Existential Second-Order Constraints. In: Proc. 2018 Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. pp. 389–399 (2018)
26. Necula, G.C.: Proof-carrying code. In: Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France. pp. 106–119. ACM Press (Jan 1997)
27. Scheurer, D., Hähnle, R., Bubel, R.: A General Lattice Model for Merging Symbolic Execution Branches. In: Ogata, K., Lawford, M., Liu, S. (eds.) Proc. 18th Intern. Conf. on Formal Engineering Methods (ICFEM). LNCS, vol. 10009, pp. 57–73. Springer (Nov 2016)
28. Srivastava, S., Gulwani, S., Foster, J.S.: From Program Verification to Program Synthesis. In: Proc. 37th POPL. pp. 313–326 (2010)
29. Steinhöfel, D., Hähnle, R.: Modular, Correct Compilation with Automatic Soundness Proofs. In: Proc. 8th Intern. Symp. on Leveraging Applications of Formal Methods (ISoLA). LNCS, vol. 11244, pp. 424–447. Springer (2018)
30. Steinhöfel, D., Wasser, N.: A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows. In: Polikarpova, N., Schneider, S. (eds.) Proc. of 13th Intern. Conf. on Integrated Formal Methods. pp. 279–294. Springer (2017)
31. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A New Verified Compiler Backend for CakeML. In: Proc. 21st Intern. Conf. on Functional Programming. pp. 60–73. ACM (2016)
32. Wasser, N., Steinhöfel, D.: Technical Report: Using Loop Scopes with for-Loops. arXiv e-prints **1901.06839** (Jan 2019), https://arxiv.org/abs/1901.06839

Appendix

We provide additional information and explanations that had to be left out of the paper for space reasons. Numbered (sub-)sections relate to those of the paper.

2 Specifying Abstract Programs

Remark. The specification constructs listed in Table 1 include some syntactic sugar that is removed before loading a problem into our tool; details about that are provided in the implementation.

3.1 Principles of JavaDL

To help understanding the JavaDL sequent calculus a little better, we briefly discuss three exemplary calculus rules. Figure 6 shows the rules allRight, orLeft and ifElseSplit. The first two are logical first-order rules, whereas the last one is a (splitting) Symbolic Execution rule. Rule allRight processes a universal quantifier in the succedent of a JavaDL sequent by replacing the quantified variable by a fresh Skolem constant. Because a sequent $\Gamma \vdash \Delta$ is equivalent to the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$, the logical rule orLeft splits the proof into two branches for a disjunction in the antecedent of a sequent. A disjunction in the succedent (or a *con*junction in the antecedent) would be handled by simply adding the constituents as individual formulas to the succedent (antecedent). Finally, ifElseSplit also splits the proof into two branches. It is a "classical" Symbolic Execution rule: because the expression *simpleExpr* (which stands for "simple expression", an expression without side effects) might contain symbolic values, we have to evaluate the *then* and *else* branches separately. For each branch, we add as a new precondition that *simpleExpr* evaluates to TRUE resp. FALSE.

$$\textsf{allRight}\ \frac{\Gamma \vdash [x/c](\varphi), \Delta}{\Gamma \vdash \forall x; \varphi, \Delta}\ c \text{ is a fresh constant of suitable type}$$

$$\textsf{orLeft}\ \frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}$$

ifElseSplit

$$\frac{\begin{array}{c}\Gamma, \mathit{simpleExpr} \doteq \text{TRUE} \vdash \{\mathcal{U}\}[\pi\ p_1\ \omega]\varphi, \Delta \\ \Gamma, \mathit{simpleExpr} \doteq \text{FALSE} \vdash \{\mathcal{U}\}[\pi\ p_2\ \omega]\varphi, \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{if}\ (\mathit{simpleExpr})\ p_1\ \texttt{else}\ p_2\ \omega]\varphi, \Delta}$$

Fig. 6: Some example JavaDL calculus rules

3.2 Formalization of Abstract Execution

Proof (Thm. 2). Let $\mathtt{P}^0$ be any legal instantiation of P. We create a program $\mathtt{P}^0_\dagger$, which is equivalent to $\mathtt{P}^0$, as follows: First, we transform $\mathtt{P}^0$ to a program $\mathtt{P}^{0\prime}$ without irregular termination. We define the program transformation operator transf which replaces in the input program each occurrence of

- "**return** *expr*;" with "returns=true; result=*expr*; **break** outer;",
- "**break** l_i;" with "breaksToLbl_i=**true**; **break** outer;"

and, for **break** and **continue** statements on the top level (not in the scope of a loop within the input program), each occurrence of

- "**break**;" with "breaks=true; **break** outer;",
- "**continue**;" with "continues=true; **break** outer;".

The program $\mathtt{P}^{0\prime}$ then is defined as

$$\mathtt{P}^{0\prime} = \begin{pmatrix} \texttt{outer: \{} \\ \quad \textbf{try} \texttt{ \{ transf(P}^0\texttt{) \}} \\ \quad \textbf{catch} \texttt{ (Throwable t) \{ exc=t; \}} \\ \texttt{\}} \end{pmatrix}$$

The label outer is freshly introduced. The other flags introduced by transf coincide with those in the premise of nonVoidLoopAERule. All boolean variables are initialized with **false**, and exc with null. Since the considered JAVA fragment is sequential, does not support reflection, and we additionally do not consider errors (only exceptions), $\mathtt{P}^{0\prime}$ can only terminate regularly (if it does *not* terminate, the conclusion of rule nonVoidLoopAERule is trivially valid). Note that the rule excludes labeled **continue** statements, which is why we also do not consider them here. Then, we define $\mathtt{P}^0_\dagger$ as:

$$\mathtt{P}^0_\dagger = \begin{pmatrix} \mathtt{P}^{0\prime} \\ \textbf{if} \texttt{ (returns) } \textbf{return} \texttt{ result;} \\ \textbf{if} \texttt{ (exc != } \textbf{null}\texttt{) } \textbf{throw} \texttt{ exc;} \\ \textbf{if} \texttt{ (breaks) } \textbf{break}\texttt{;} \\ \textbf{if} \texttt{ (continues) } \textbf{continue}\texttt{;} \\ \textbf{if} \texttt{ (breaksToLbl\_1) } \textbf{break}\ l_1\texttt{;} \\ \ldots \\ \textbf{if} \texttt{ (breaksToLbl\_n) } \textbf{break}\ l_n\texttt{;} \end{pmatrix}$$

The program $\mathtt{P}^0_\dagger$ is equivalent to $\mathtt{P}^0$ since, if $\mathtt{P}^0$ terminates regularly, the behavior of $\mathtt{P}^{0\prime}$ equals that of $\mathtt{P}^0$ (the **try** statement has no effect) and the added **if** statements are not entered. If irregular termination occurs, it is captured and deferred equivalently to the outside. In the following, we consider, without loss of generality, the instantiation of P in the conclusion with $\mathtt{P}^0_\dagger$ instead of $\mathtt{P}^0$.

Assume that $\texttt{P}^0$ terminates normally. Since we can assume that it respects the contract of APS `P`, all the behavior specification formulas like *returnsSpec* are equivalent to false, and the **`if`** statements in the premise are not entered. The premise is therefore, in that case, logically equivalent to the one of rule simpleAERule and the soundness argument similar to Thm. 1, except that the abstract update and path condition do not range over all locations, but exactly over the assignable and accessible locations of $\texttt{P}^0$. Again, since $\texttt{P}^0$ respects the contract of `P`, we can find a suitable legal instantiation of the premise which implies the conclusion.

If $\texttt{P}^0$ terminates irregularly, we easily find a suitable legal instantiation of the premise implying the conclusion for the breaking and continuing cases. For the returning and exceptional cases, the validity of the premise implies that the conclusion is valid for every possible returned result and thrown exception, since the variables `result` and `exc` are set to fresh Skolem constants $returns_0$ and exc_0. In particular, it is therefore valid for the concretely returned result or thrown exception. □

4 Proving the Correctness of Refactoring Techniques

We created a variant of the existing loop invariant rule based on loop scopes in JavaDL [30,32]. The new rule, depicted in Fig. 7, only applies to formulas with a special type of post condition, namely those containing the uninterpreted *Post* predicate used in AE equivalence proofs. In the part of the post condition of the "preserved & use case" where in the case that the loop continues (i.e., the loop scope index `x` is FALSE), normally only the invariant has to be shown, we additionally include the post condition (highlighted in gray), but with the second component of the *Post* predicate set to FALSE. This provides us with the possibility to only assume and show a simple invariant *Inv* containing, e.g., information necessary for showing termination, and to otherwise continue with abstract relational reasoning, thereby relating runs continuing loop execution separately from those leaving the loop.

Two other variants (Figures 8 and 9) are useful for situations where one loop has a bigger amount of iterations than another. The first one implements a general unrolling pattern as described, e.g., in [16], for harmonizing the iteration structure of two loops. The second variant realizes the unrolling pattern as displayed in Listings 6 and 7 along the *Remove Control Flag* example. It is specialized to loops where the guards consist of two conjuncts and only the first should trigger a direct break out of the loop. Both rules are parametric in a number i determining how often the body should be unrolled. Thus, they spare the harmonization of the loop iteration structure by manual code transformation.

Performance

Figures 10 and 11 visualize proof sizes and needed time for proof completion for the studied refactorings. Figure 10 also shows the numbers of APSs for all models. Proof size roughly corresponds to auto mode time; higher numbers of

loopScopeInvariantAE

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if(}expr\texttt{)}\ \{ & \\ \quad\quad body & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]\ ))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while(}expr\texttt{)}\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}$$

Fig. 7: Loop Scope Invariant Rule for Abstract Execution

loopScopeInvariantAEUnrolling

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if(}expr\texttt{)}\ \{ & \\ \quad\quad \{\ (\ \texttt{if(}expr\texttt{)}\ body\ \texttt{else break;}\ )^{i}\ \} & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while(}expr\texttt{)}\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}\ i \geq 1$$

Fig. 8: "Unrolling" Loop Scope Invariant Rule for Abstract Execution

APSs usually lead to bigger proofs, although due to the influence of other factors (unrelated proof tree branching, loops etc.), there is no direct correspondence.

Source Code Samples

The source code of most refactoring models is shown in Figs. 12 to 17. For space reasons, we usually omit showing the JML specifications of the APSs.

Additional Implementation Remarks

In the KeY system, rules can either be programmed in JAVA as "built-in rules", or added as so-called *taclet*s, schematic, theory-specific rules written in a domain-specific language for SE rules [1, Chapter 4]. Our AE rules are written as taclets, because this helps to modularize soundness arguments. Taclets are also easier to read and to maintain than JAVA code. A typical AE taclet needs ca. 50–60 lines of text (excluding comments and empty lines). For the implementation, we had to extend the taclet language by various features like new variable conditions

loopScopeInvariantAEUnrollingSplitCond

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\} Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if}(e_1 \texttt{\&\&} e_2)\ \{ & \\ \quad\quad \{\ (\ \texttt{if}(e_1)\ \{\ \texttt{if}(e_2)\ body\ \}\ \texttt{else break;}\ )^i\ \} & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while}(e_1 \texttt{\&\&} e_2)\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}\ i \geq 1$$

Fig. 9: Special "Unrolling" Loop Scope Invariant Rule for Abstract Execution, Harmonizing Iteration Structure for a Compound Loop Condition

and term transformers (both allow to inline small portions of JAVA code) and a new loop construct for realizing the "$\cdots$" in rules like nonVoidLoopAERule.

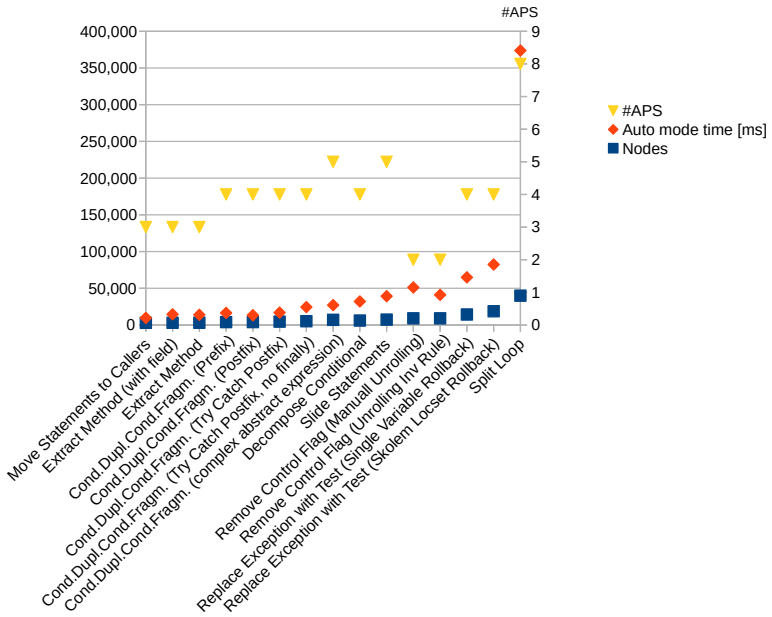

Fig. 10: Performance: Numbers of nodes and APSs and auto mode time

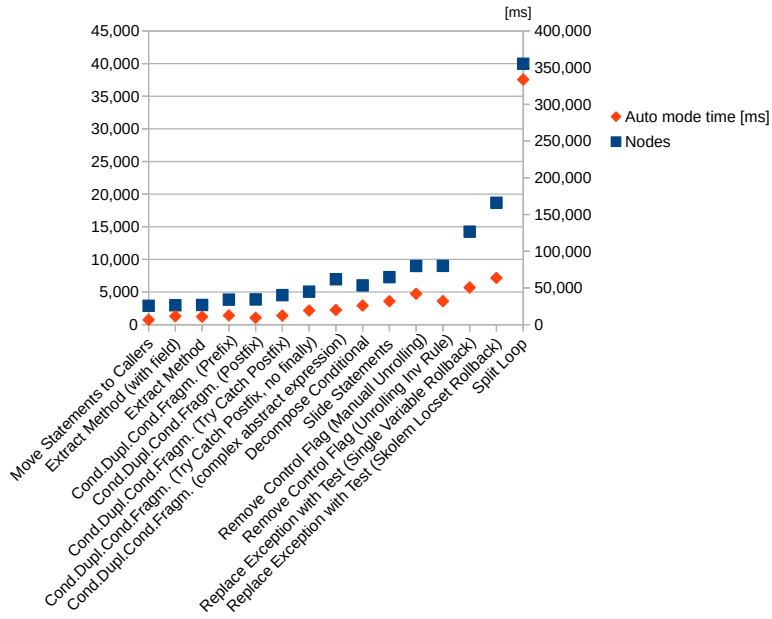

Fig. 11: Performance: Numbers of nodes and auto mode time

Listing 6: Remove Control Flag, Original

```
while (!done && i < threshold) {


  if (condition) {
    abstract_statement Body;
    done = true;
  }

  i++;









}
```

Listing 7: Remove Control Flag, After Unrolling

```
while (!done && i < threshold) {
  if (!done) {
    if (i < threshold) {
      if (condition) {
        abstract_statement Body;
        done = true;
      }
    }
    i++;
  } else break;
  if (!done) {
    if (i < threshold) {
      if (condition) {
        abstract_statement Body;
        done = true;
      }
    }
    i++;
  } else break;
}
```

Listing 8: Before

```
abstract_statement Guard;
if (guard) {
  abstract_statement Then;
} else {
  abstract_statement Else;
}

abstract_statement Post;
return result;
```

Listing 9: After

```
guard = mGuard(result);
if (guard) {
  tmp = mThen(result, guard, tmp);
} else {
  tmp = mElse(result, guard, tmp);
}

abstract_statement Post;
return result;

//...

//@ declares final(args);
private Object mThen(Object result,
    boolean guard, Object tmp) {
  abstract_statement Then;
  return tmp;
}
```

Fig. 12: The *Decompose Conditional* Refactoring Technique

Listing 10: Before

```
abstract_statement P;
abstract_statement Q;
abstract_statement R;
return result;
```

Listing 11: After

```
abstract_statement P;
tmp = extracted(res, tmp);
abstract_statement R;
return result;

// ...

//@ declares final(locals(P)), final(args);
public Object extracted(Object res,
    Object tmp) {
  abstract_statement Q;
  return tmp;
}
```

Fig. 13: The *Extract Method* Refactoring Technique

Listing 12: Before

```
result = called();

abstract_statement B;

return result;

// ...

//@ declares final(args);
public Object called() {
  abstract_statement C;
  abstract_statement A;
  return result;
}
```

Listing 13: After

```
result = called();

abstract_statement A;
abstract_statement B;

return result;

// ...

//@ declares final(args);
public Object called() {
  abstract_statement C;
  return result;
}
```

Fig. 14: The *Move Statements to Callers* Refactoring Technique

Listing 14: Before

```
abstract_statement Init;

try {
  //@ exceptional_behavior
  //@     requires throwsExc;
  abstract_statement Normal;
} catch (Throwable t) {
  abstract_statement Rollback;
  abstract_statement Exceptional;
}

abstract_statement After;
return result;
```

Listing 15: After

```
abstract_statement Init;

if (!throwsExc) {
  //@ exceptional_behavior
  //@     requires throwsExc;
  abstract_statement Normal;
} else {
  abstract_statement Rollback;
  abstract_statement Exceptional;
}

abstract_statement After;
return result;
```

Fig. 15: The *Replace Exception With Test* Refactoring Technique

Listing 16: Before

```
abstract_statement A;
abstract_statement B;
abstract_statement C;
abstract_statement D;
abstract_statement E;
```

Listing 17: After

```
abstract_statement A;
abstract_statement D;
abstract_statement C;
abstract_statement B;
abstract_statement E;
```

Fig. 16: The *Slide Statements* Refactoring Technique

Listing 18: Before

```
abstract_statement PreProc;

abstract_statement InitA;
abstract_statement InitB;

for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outA;
  abstract_statement LoopBodyA;
  //@ assignable outB;
  abstract_statement LoopBodyB;
}

//@ assignable result;
//@ accessible outA;
abstract_statement PostProcA;
//@ assignable result;
//@ accessible outB;
abstract_statement PostProcB;




abstract_statement PostProc;
return result;
```

Listing 19: After

```
abstract_statement PreProc;

abstract_statement InitA;
for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outB;
  abstract_statement LoopBodyB;
}
//@ assignable result;
//@ accessible outA;
abstract_statement PostProcA;

abstract_statement InitB;
for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outB;
  abstract_statement LoopBodyB;
}
//@ assignable result;
//@ accessible outB;
abstract_statement PostProcB;

abstract_statement PostProc;
return result;
```

Fig. 17: The *Split Loop* Refactoring