Journal of Automated Reasoning (2024) 68:7
https://doi.org/10.1007/s10817-023-09692-0

Schematic Program Proofs with Abstract Execution

Theory and Applications

Dominic Steinhöfel[1] · **Reiner Hähnle**[2]

Received: 11 January 2023 / Accepted: 19 December 2023

Abstract
We propose *Abstract Execution*, a static verification framework based on symbolic execution and dynamic frames for proving properties of *schematic* programs. Since a schematic program may potentially represent infinitely many concrete programs, Abstract Execution can analyze infinitely many programs at once. Trading off expressiveness and automation, the framework allows proving many interesting (universal, behavioral) properties *fully automatically*. Its main application are correctness proofs of *program transformations* represented as pairs of schematic programs. We implemented Abstract Execution in a deductive verification framework and designed a graphical workbench supporting the modeling process. Abstract Execution has been applied to correct code refactoring, analysis of the cost impact of transformation rules, and parallelization of sequential code. Using our framework, we found and reported several bugs in the refactoring engines of the Java IDEs IntelliJ IDEA and Eclipse, which were acknowledged and fixed.

Keywords Schematic Programs · Symbolic Execution · Deductive Verification · Program Transformation · Second-Order Program Properties

1 Introduction

Abstract Execution (AE) generalizes *symbolic execution* to programs with "holes," i.e., *abstract* or *schematic* programs: Abstract programs may contain symbols that stand for arbitrary statements or expressions. Symbolic execution of such abstract elements is achieved by approximating them with abstract specifications of conditions on normal or exceptional behavior, frames and footprints, etc. The symbolic execution "flavor" we consider here is *complete* symbolic execution. This refers to the logic-based variant used in deductive verification [5, 9, 37] that comes with first-order and invariant reasoning. We do not regard

✉ Reiner Hähnle
reiner.haehnle@tu-darmstadt.de

Dominic Steinhöfel
dominic.steinhoefel@cispa.de

1 CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, 66123 Saarbrücken, Germany
2 TU Darmstadt, Software Engineering Group, Hochschulstraße 10, 64289 Darmstadt, Germany

the incomplete (often *dynamic*) symbolic execution variants employed in test case generation [25].

Abstract Execution permits to prove *universal second-order* properties of program behavior by implicitly quantifying over all permissible instances of abstract elements. However, given that schematic elements and their specifications are abstract, it is generally not possible to prove interesting functional properties of a single abstract program. The power of AE derives from being able to *compare* the execution of two related abstract programs with the *same* abstract elements. For loop-free abstract programs this tends to be *fully automatic* and even in the presence of loops it is usually much easier to find coupling invariants than functional ones [2, 10].

Second-order program properties involving the comparison of the behavior of two programs occur in any area of programming, where relative correctness of two program schemata is of concern: Rule-based compilation [48, 81] and optimization [46, 50], code refactoring [24], program synthesis [75], Correctness-by-Construction [45], to name a few.

Mechanized proofs of such properties traditionally are often performed with *interactive* proof assistants [56, 59, 82]. An example is the work on verified compilers [48, 81]. This permits specifying arbitrarily complex properties, but a substantial effort is required to *manually* write proof scripts. Existing *automatic* approaches, on the other hand, target *specific* applications (e.g., regression verification [26], "peephole" optimizations [50], symbolic execution rules [6]) and lack expressiveness. AE is positioned in a "sweet spot" in between these extremes, combining considerable expressiveness and generality with a high degree of automation.

1.1 The Setting of Abstract Execution

Most areas mentioned above involve the transformation of schematic programs. Proving the correctness of program transformation rules can be understood as a *relational verification* [12] problem over programs with placeholders. For example, the pair of schematic programs "$p\ q$" and "$q\ p$" (where p, q represent arbitrary statements) describes a program transformation swapping two statements. If we can prove that, under certain assumptions, all instances of the schematic programs before and after the transformation behave equivalently, the transformation is *safe*.

AE is implemented on top of KeY [5], a highly automatic deductive verification framework for Java programs based on symbolic execution. Our setting of AE extends the Java language by *Abstract Statements (ASs)* "`\abstract_statement P;`," and *Abstract Expressions (AExps)* "`\abstract_expression` T `e;`," where `P` and `e` are the *identifiers* of an abstract statement and expression, respectively, and T is the type of the abstract expression `e`. ASs and AExps are called *Abstract Program Element (APE)*, programs containing APEs are called *abstract* (or *schematic*) programs. AE universally closes over APEs in programs.

Without additional constraints, APE represents all of its well-formed concrete instances. This is insufficient to express meaningful properties. For instance, the above mentioned transformation "$p\ q \rightarrow q\ p$," which is a *refactoring technique* called *Slide Statements* [22], is generally unsound: if we instantiate p with "`x=1;`" and q with "`y=x;`," the final value of `x` will generally be different in executions of the original and the transformed code. Therefore, AE provides a specification language to constrain the *behavior* of concrete instantiations represented by APEs. *An APE is the declaration of a placeholder symbol (e.g.,* `P`*) together with all specification clauses constraining it.* It represents all concrete programs satisfying the specification; if multiple APEs with the same identifier are declared in a program, those represent the same programs (if applicable, modulo renaming of input/output locations).

1.2 Specifying Abstract Programs by Example

In the following, we call the memory locations which APE may *write to* its *frame*, and the locations it may *read from* its *footprint*.

Remark 1 (Wording: Frames and Footprints) In everyday language, the notion of a "footprint," as in "carbon footprint," is used for *effects* on the outside world. Here, we adhere to the meaning coined in the context of "dynamic frames," where *frames* are regarded as "the part of the world which the operation has license to change" [40]. *Footprint*, on the other hand, is a standard term for accessible location sets in the context of *dependency contracts* [88], a research area closely related to AE.

Slide Statements is safe, i.e., retains the external behavior of the affected code, under the following conditions: (1) The frames of p and q must be disjoint, (2) the frame of p and footprint of q must be disjoint, (3) the frame of q and footprint of p must be disjoint, (4) if p completes abruptly (e.g., by throwing an exception), q must complete normally (and vice versa), and (5) if either p or q completes abruptly, the other may not have *relevant* side effects. Conditions (1) to (3) ensure that p and q are "independent," i.e., do not interfere; Condition (4) establishes that the *reason* for (abrupt) completion of the program is the same before and after the transformation. For example, it cannot happen that before, the program completes because of an exception thrown by p, while afterward, it completes due to a **`return`** by q. Condition (5) is required because if one statement completes abruptly, the other one can only change the state either before *or* after the transformation, which is why any changes must be confined to locations we are not interested in.

To impose constraints on frames and footprints of abstract elements, we have to define which locations APEs may read and write. However, no further constraints other than the given ones should be enforced: frames and footprints should apply to *all* programs satisfying Conditions (1) to (3) and (5). We achieve this by using abstract, set-valued specification variables inspired by the theory of *dynamic frames* [40]. Specifically, we introduce constants *frP*, *fpP*, *frQ*, *fpQ*, etc., each representing an abstract set of program variables or heap locations that can be used to refer to the same frame or footprint in multiple specifications.

The abstract program model for *Slide Statements* is shown in Fig. 1 (to simplify the example, we only consider normal completion as well as completion due to a thrown exception or returned value, disregarding, e.g., abrupt completion due to a **`break`** statement). Our specification language extends the Java Modeling Language (JML) [47]. Constraints on ASs are imposed inside specification comments starting with "`@`"; the keyword "**`ae_constraint`**" initiates the declaration of a constraint. In lines 25/26 and 33/34, we assign the newly introduced dynamic frame specification variables to the ASs, where the keyword **`assignable`** specifies a frame, and **`accessible`** a footprint of AS or AExp. Conditions (1) to (3) are encoded in lines 2–4. To realize mutual exclusion of abrupt completion (Condition (4)), we first bind abrupt completion of ASs `P` and `Q` to abstract predicates *throwsExcP*, *returnsP*, *throwsExcQ* and *returnsQ*, resp., with "**`exceptional_behavior requires`** ↳ " and "**`return_behavior requires`**" in lines 27–30 and 35–38. These predicates represent unknown conditions in the same way that dynamic frames represent unknown location sets, with the intention of giving them a name for future reference. The binding via "**`... requires`**" is *both* necessary and sufficient: The respective behavior is *always* demanded when the specified conditions hold and *only* then. The function "**`\value`**`(···)`" maps to the (abstract) value of a location set at the point in the program where it is used. It is needed since the *locations* represented by an abstract location set like *fpP* do not change during program execution, while their *values* can change. Because furthermore, the same

Listing 1: Input Model

```
1  /*@ ae_constraint
2    @       \disjoint(frP, frQ)
3    @    && \disjoint(frP, fpQ)
4    @    && \disjoint(frQ, fpP)
5    @    && \mutex(
6    @       returnsP(\value(fpP)),
7    @       returnsQ(\value(fpQ)))
8    @    && \mutex(
9    @       returnsP(\value(fpP)),
10   @       throwsExcQ(\value(fpQ)))
11   @    && \mutex(
12   @       throwsExcP(\value(fpP)),
13   @       throwsExcQ(\value(fpQ)))
14   @    && \mutex(
15   @       throwsExcP(\value(fpP)),
16   @       returnsQ(\value(fpQ)))
17   @    && (throwsExcP(\value(fpP))
18   @        || returnsP(\value(fpP))
19   @          ==> \disjoint(frQ, rel))
20   @    && (throwsExcQ(\value(fpQ))
21   @        || returnsQ(\value(fpQ))
22   @          ==> \disjoint(frP, rel));
23   @*/
24
25 //@ assignable frP;
26 //@ accessible fpP;
27 /*@ exceptional_behavior requires
28   @    throwsExcP(\value(fpP)); */
29 /*@ return_behavior requires
30   @    returnsP(\value(fpP)); */
31 \abstract_statement P;
32
33 //@ assignable frQ;
34 //@ accessible fpQ;
35 /*@ exceptional_behavior requires
36   @    throwsExcQ(\value(fpQ)); */
37 /*@ return_behavior requires
38   @    returnsQ(\value(fpQ)); */
39 \abstract_statement Q;
```

Listing 2: Output Model

```
/*@ ae_constraint
  @       \disjoint(frP, frQ)
  @    && \disjoint(frP, fpQ)
  @    && \disjoint(frQ, fpP)
  @    && \mutex(
  @       returnsP(\value(fpP)),
  @       returnsQ(\value(fpQ)))
  @    && \mutex(
  @       returnsP(\value(fpP)),
  @       throwsExcQ(\value(fpQ)))
  @    && \mutex(
  @       throwsExcP(\value(fpP)),
  @       throwsExcQ(\value(fpQ)))
  @    && \mutex(
  @       throwsExcP(\value(fpP)),
  @       returnsQ(\value(fpQ)))
  @    && (throwsExcP(\value(fpP))
  @        || returnsP(\value(fpP))
  @          ==> \disjoint(frQ, rel))
  @    && (throwsExcQ(\value(fpQ))
  @        || returnsQ(\value(fpQ))
  @          ==> \disjoint(frP, rel));
  @*/

//@ assignable frQ;
//@ accessible fpQ;
/*@ exceptional_behavior requires
  @    throwsExcQ(\value(fpQ)); */
/*@ return_behavior requires
  @    returnsQ(\value(fpQ)); */
\abstract_statement Q;

//@ assignable frP;
//@ accessible fpP;
/*@ exceptional_behavior requires
  @    throwsExcP(\value(fpP)); */
/*@ return_behavior requires
  @    returnsP(\value(fpP)); */
\abstract_statement P;
```

Fig. 1 Abstract Program Model for *Slide Statements*

program may, or may not, throw an exception (return, etc.) depending on the evaluation of its footprint in the current environment, the abstract predicates are defined parametrically in the values of the footprints. Now, we can stipulate that at most one of the predicate holds (i.e., at most one AS completes abruptly) in lines 5–16 using the "**\mutex**" keyword.

To encode Condition (5) in the model, we employ a further dynamic frame specification variable *rel* representing an underspecified set of relevant locations. We use it to specify the property that the program performs equivalently before and after the transformation. If **\result_1** represents the value of *rel* before and **\result_2** its value after the transformation, this property is specified as **\result_1** $\doteq$ **\result_2**. Without further constraints, the model has to be proven under the assumption that *all* locations are in *rel*, i.e., we prove full equivalence. In lines 17–22 of the model, we relax the proof goal by declaring the frame of Q disjoint from *rel* if AS P completes abruptly, and vice versa. This is more liberal than preventing the normally completing statement from changing *any* part of the state.

Our AE tool proves correctness of the transformation specified in Fig. 1 fully automatically in less than 20 seconds. Such safety conditions on program transformations as shown above are hard to find. Indeed, nearly all conditions presented in this paper were not mentioned in the literature. We discovered them with a feedback loop on interpreting failed proof attempts.

This process is supported by our implementation of AE in a semi-automatic program prover that permits proof inspection.

1.3 Organization of This Paper

Our formalization of AE is based on a dynamic program logic and an abstract, formal definition of Symbolic Execution, which we expound in Sect. 2. Sect. 3 presents the concrete and abstract syntax of our AE framework and defines the semantics of abstract programs. The core of the framework are our rules for executing APEs and simplifying abstract stores, which we present in Sect. 4. In Sect. 5, we explain details about the feedback loop for extracting preconditions for safe transformations and our approach to proving loop transformations. Furthermore, we provide an overview of the applications of AE to correct code refactoring, cost impact of transformation rules, and parallelization of sequential code. The implementation of AE in the program verification framework KeY is described in Sect. 6. Sect. 7 describes related work, and Sect. 8 concludes the paper and outlines ideas for future applications and extensions.

Novelty
This work is a heavily revised and much-extended version of a conference paper [79]. In contrast to the latter, it is fully based on dynamic frames (Sect. 3.1 presents the extended specification language), which enables the specification of general transformations. Furthermore, we introduce abstract *expressions* (AExps), whereas [79] relied on an "abstract expression idiom" using ASs. The brief, informal semantics definition [79] is made precise by translating it into dynamic logic (Sect. 3.3). This reduction makes it possible to mechanically check whether a given concrete program instantiates an abstract one. The symbolic execution rules for APEs and all simplification rules for abstract stores (Sect. 4) have been replaced by wholly revised versions, including rules for better support of heap-related properties. We use a new technique for proving loop transformations based on "abstract strongest loop invariants" (Sect. 5.1.2) that no longer requires manual loop coupling. For the application to correct refactoring (Sect. 5.1.3), we derived more precise safety preconditions for the analyzed refactoring techniques and added a transformation not considered in [79]. Furthermore, we found several unreported bugs in the refactoring engines of major Java IDEs, which we also discuss in Sect. 5.1.3. Finally, we give an overview of other applications of AE conducted since the publication of [79] (included in Sect. 5).

Most of the technical content in this paper is included in a Ph.D. thesis [76]. Here, we give a more condensed account and formalize AE rules in an abstract symbolic execution framework to make the theory independent of KeY's program logic.

2 JavaDL and Symbolic Execution

This section introduces the program logic, including theories of heaps and location sets on which we rely. We also define a general theory of symbolic execution wherein we later express abstract execution rules.

2.1 Program Logic

Our framework is based on Java Dynamic Logic (JavaDL), a first-order dynamic logic for sequential Java programs. In this section, we provide the essential parts of the logic needed to keep the paper self-contained and refer to [5] for a full account. JavaDL extends typed first-order logic by three modal operators: *Modalities* $[p]\varphi$ and $\langle p\rangle\varphi$, as well as *updates* $\{\mathcal{U}\}\varphi$.

The *box modality* $[p]\varphi$ expresses that *if* the program p terminates, *then* it terminates in a state where postcondition φ holds; the *diamond modality* additionally requires p to terminate. Updates denote certain limited state changes. In particular, they always terminate. The *empty update Skip* represents an empty state change, an *elementary update* $\mathtt{x} := t$ the transition where variable $\mathtt{x}$ is assigned the value of term t. Two updates $\mathcal{U}_1$ and $\mathcal{U}_2$ can be combined into a *parallel* update $\mathcal{U}_1 \,||\, \mathcal{U}_2$, where both state changes are executed *simultaneously*. In case of conflicting assignments to the same variable, the syntactically later one "wins." For example, the parallel composition $\mathtt{x} := 1 \,||\, \mathtt{x} := 2$ is equivalent to the elementary update $\mathtt{x} := 2$. Updates are applied to terms and formulas: $\{\mathcal{U}\}t$ 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. In a sequential update $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$, the right-hand sides of $\mathcal{U}_2$ are interpreted in the state after the transition described by $\mathcal{U}_1$, while in parallel compositions, they are interpreted in the *same* pre-state. The formula $\{\mathcal{U}_1\}\{\mathcal{U}_2\}\varphi$ is equivalent to $\{\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2\}\varphi$. We use the notation $\mathcal{U}_1 \circ \mathcal{U}_2$ for $\mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2$ and write Upd for the set of all updates.

Terms and formulas are standard; we denote by Fml and Trm_T the sets of formulas and terms of type T. We write PVSym for the set of *program* variables $\mathtt{x}$ and VSym for the set of *logic* variables v. The semantics of JavaDL is based on first-order Kripke structures $K = (\mathcal{D}, I, \mathcal{S}, \varrho)$ consisting of a *domain* $\mathcal{D}$, an interpretation function I of function and predicate symbols, a set $\mathcal{S}$ of *states* σ mapping program variables to domain values, and a *program transition relation* ϱ associating with legal program fragments p a transition relation $\varrho(p) \in \mathcal{S} \times \mathcal{S}$ such that $(\sigma_1, \sigma_2) \in \varrho(p)$ iff p, when started in σ_1, completes normally (without throwing an exception, breaking, or returning, etc.) in σ_2. This is sufficient, because programs that might terminate abnormally are locally transformed into normally terminating programs by the rules of the Java DL calculus, see Example 2 below. Full details are in [5, Chapter 3].

A *legal program fragment* p for a *context program Prg* is a sequence of Java statements which may appear legally (according to the rules of the Java Language Specification [28]) in the extension of *Prg* by an additional class C with a suitable method m into which p is embedded as a body. Updates, terms, and formulas are evaluated using an overloaded *valuation function* $val\,(K, \sigma, \beta|\cdot)$, where β is a (logic) *variable assignment*. It assigns to updates $\mathcal{U} \in \mathsf{Upd}$ a state transformer $val\,(K, \sigma, \beta|\mathcal{U}) \in \mathcal{S} \times \mathcal{S}$, to terms $t \in \mathsf{Trm}_T$ a domain value $val\,(K, \sigma, \beta|t) \in \mathcal{D}$ of type T, and to formulas $\varphi \in \mathsf{Fml}$ a truth value *tt* or *ff*. For closed formulas (without free logic variables), we omit β. For example, the valuation of the term $\{\mathtt{x} := \mathtt{y}\}\mathtt{x} > 0$, which expresses that in all states where $\mathtt{x}$ was updated to the value of $\mathtt{y}$, $\mathtt{x}$ is strictly positive, is computed as follows:

$$
\begin{aligned}
val\,\big(K, \sigma, \beta|\{\mathtt{x} := \mathtt{y}\}\mathtt{x} > 0\big) &= val\,\big(K, val\,(K, \sigma, \beta|\mathtt{x} := \mathtt{y})\,(\sigma), \beta|\mathtt{x} > 0\big) \\
&= val\,\big(K, \sigma[\mathtt{x} \mapsto val\,(K, \sigma, \beta|\mathtt{y})], \beta|\mathtt{x} > 0\big) \\
&= val\,\big(K, \sigma[\mathtt{x} \mapsto \sigma(\mathtt{y})], \beta|\mathtt{x} > 0\big) \\
&= \sigma[\mathtt{x} \mapsto \sigma(\mathtt{y})](\mathtt{x}) > 0 \;=\; \sigma(\mathtt{y}) > 0
\end{aligned}
$$

We write $K, \sigma \models \varphi$ for $val\,(K, \sigma, \beta|\varphi) = tt$. If $K, \sigma \models \varphi$ for *all* K and σ, we write $\models \varphi$ and say that φ is *valid*. JavaDL has a sound sequent calculus [5] to derive from judgments $\Gamma \vdash \Delta$ the validity of the semantic entailment $\Gamma \models \Delta$.

JavaDL implements a heap theory based on the theory of arrays [53]. A heap is a sequence of mappings from pairs of objects and fields to values. Writing to the heap is accomplished by a function *store*, which takes a heap, object, field, and a value, and returns an updated heap. Reading values is done by the function *select*, taking a heap, an object, and a field, and returning the field's value. For example, the configu-

ration *store*(h, `Person`, `age`, 42) represents a heap identical to h, but where the value of `Person.age` is 42. To evaluate the expression `Person.age` in it we compute *select*(*store*(h, `Person`, `age`, 42), `Person`, `age`) $\doteq$ 42. A *sequence* of mappings is modeled using nested *store* expressions, as in *store*(*store*(h, `Person`, `age`, 42), `Person`, `weight`, 83).

Semantically, the current heap configuration in a state $\sigma \in \mathcal{S}$ is stored in σ(`heap`), for a designated variable `heap` of type *Heap*. The value of location (o, f) in a state σ is σ(`heap`)(o, f). The pair (o, f) is an element of JavaDL's *LocSet* type for *location sets*. Its domain are pairs of objects and fields; in Sect. 3.2, we extend this by *program variable locations*. The *Heap* and *LocSet* theories in JavaDL are closely related: For instance, the function *anon*:$Heap \times LocSet \times Heap \to Heap$ anonymizes the fields of the location set in the first heap argument; when accessing those, the values in the second heap are used instead. See [5] for details on the heap and location set models.

2.2 Symbolic Execution

Symbolic Execution (SE) [8, 91] is a popular program analysis technique introduced in the 1970s [15, 18, 41] for exploring a large number of execution paths of a program. The key idea is to treat inputs to a program as abstract symbols. Whenever the execution depends on the concrete value of a symbolic variable, SE follows the branching execution paths in parallel. Symbolic Execution engines maintain for each explored path (1) a *path condition* describing the conditions satisfied by the branches taken along that path, (2) a *symbolic store* mapping variables to (symbolic) values, and (3) a *program counter* pointing to the next instruction to execute. Branch execution updates the path condition, while assignments update the symbolic store [8]. The triple consisting of these elements is called a *Symbolic Execution State (SES)*.

Semantically, a *symbolic* execution state s represents a (potentially infinite) set of *concrete* execution states $\sigma \in \mathcal{S}$, in the same way as a symbolic parameter represents a potentially infinite set of concrete parameters. We call the set of concrete execution states *concretizations* for s. Based on the notion of concretization, we develop two desirable properties of SE transition relations: *Exhaustiveness*, satisfied by overapproximating SE, and *precision*, satisfied by underapproximating SE. The definitions in this section are a digest of [76, Chapt. 3]. Specifically, the definitions of exhaustiveness and precision and their implications on the correctness of SE do not appear in previous publications. In Sect. 3, we extend this framework to *abstract* SESs, and define SE rules for SESs with abstract program counters in Sect. 4.

We formally define our notion of SESs. We represent path conditions by closed formulas and symbolic stores by JavaDL updates. Updates have the advantage that we can evaluate, for example, a formula, in a symbolic store by simply *applying* the update to the formula. A program counter in our framework is the whole remaining program (instead of a pointer to the next instruction).

Definition 1 (**Symbolic Execution State**) A *Symbolic Execution State (SES)* is a triple $(C, \mathcal{U}, p)$ of (1) a set of *closed* formulas $C \in 2^{\mathsf{Fml}}$, the *path condition*, (2) an update $\mathcal{U} \in \mathsf{Upd}$, the *symbolic store*, and (3) a legal program fragment p, the *program counter*. We omit p for empty program counters and denote the set of all SESs by $\mathbb{S}_{SE}$.

Based on the valuation function of JavaDL, we define the *concretization function concr* which, given an initial concrete state σ, concretizes a symbolic state s to a concrete state relative to a given structure K. The union $\bigcup_\sigma concr(s, \sigma)$ for all initial states represents the set of concretizations. We begin with a "K-indexed" version $concr_K$ and then define *concr* as the union for all structures K. The idea is that all different interpretations of uninterpreted

function and predicate symbols are captured in the concretizations. If, for instance, new Skolem symbols are introduced after a loop invariant application, the represented concrete state space is extended, which must be reflected in the definition.

Definition 2 ((*K*-indexed) Concretization Function) The *K-indexed concretization function* $concr_K : \mathbb{S}_{SE} \times \mathcal{S} \to 2^{\mathcal{S}}$ maps an SES $(C, \mathcal{U}, p)$ and a concrete state $\sigma \in \mathcal{S}$ (1) to the empty set $\emptyset$ if *either* $K, \sigma \not\models \bigwedge C$, *or*, where $\sigma' := val\,(K, \sigma|\mathcal{U})\,(\sigma)$, there is no σ'' such that $(\sigma', \sigma'') \in \varrho(p)$, or otherwise (2) to the singleton set $\{\sigma''\}$ such that $(\sigma', \sigma'') \in \varrho(p)$, where σ' is as before. The *concretization function concr* is defined as $concr(s, \sigma) := \bigcup_K concr_K(s, \sigma)$.

Definition 3 (Semantics of SES) The semantics $[\![s]\!]$ of an SES $s \in \mathbb{S}_{SE}$ is defined as the union of its concretizations: $[\![s]\!] := \bigcup_{\sigma \in \mathcal{S}} concr(s, \sigma)$.

The following example demonstrates the application of Def. 3 along a program containing a loop, which is abstracted using a loop invariant.

Example 1 (Concretization of SES) We consider a program p which decrements a positive variable inside a loop until it reaches 0 and adds 2 afterward:

```
while (i > 0){ i--; } i += 2;
```

Assume we want to show that `i` always has the value 2 after p terminates. Since the initial value of `i`, and therefore the number of loop iterations, is unknown, we abstract the loop with an invariant. SE starts with the initial SES $s = (\{\texttt{i} \geq 0\}, \mathit{Skip}, p)$, where the path condition contains the precondition that `i` is nonnegative. A suitable loop invariant for the **`while`** loop in p is $\texttt{i} \geq 0$. This loop invariant is *inductive:* It is strong enough to imply the postcondition. Together with the negated loop guard $\texttt{i} \leq 0$, which holds *after* termination of the loop, this is sufficiently strong to infer that `i` is 0 after loop termination. After the application of a typical loop invariant rule, one has to show, as a side condition, that $\texttt{i} \geq 0$ is an inductive loop invariant. Hence, we obtain the successor state $s_1 = (\{c \geq 0, c \leq 0\}, \texttt{i} := c, \texttt{i += 2;})$, where $\texttt{i} := c$ is an anonymizing update with a Skolem constant c, $c \geq 0$ the loop invariant, and $c \leq 0$ the branch condition signifying that the loop has been exited. Semantics-preserving simplification of the path condition yields the state $s_1' = \big(\{c \doteq 0\}, \texttt{i} := c, \texttt{i += 2;}\big)$. Its semantics is computed as follows:

$$
\begin{aligned}
[\![s_1']\!] &= \bigcup_{\sigma} concr(s_1', \sigma) \\
&= \bigcup_{\sigma \in \mathcal{S}} \bigcup_{K} \big\{\sigma' : (val\,\big(K, \sigma|\texttt{i} := c\big)\,(\sigma), \sigma') \in \varrho(\texttt{i += 2;}) \text{ and } K, \sigma \models c \doteq 0\big\} \\
&\overset{(*)}{=} \bigcup_{\sigma \in \mathcal{S}} \bigcup_{K} \big\{\sigma' : (val\,\big(K, \sigma|\texttt{i} := 0\big)\,(\sigma), \sigma') \in \varrho(\texttt{i += 2;})\big\} \\
&= \bigcup_{\sigma \in \mathcal{S}} \big\{\sigma' : (\sigma[\texttt{i} \mapsto 0], \sigma') \in \varrho(\texttt{i += 2;})\big\} = \bigcup_{\sigma \in \mathcal{S}} \big\{\sigma[\texttt{i} \mapsto 2]\big\}
\end{aligned}
$$

Consequently, s_1' represents all concrete states where `i` attains the value 2. Step $(*)$ results from the following considerations: if all structures K in the specified set are such that $K, \sigma \models c \doteq 0$, then the transformers created for $val\,(K, \sigma|\texttt{i} := c)$ are equivalent to those created for $val\,(K, \sigma|\texttt{i} := 0)$. After this simplification, there remain no more uninterpreted function symbols and the union over all K can be omitted.

$$\text{assignment}\ \frac{(C, \mathcal{U} \circ \texttt{x} := se, \pi\ \omega)}{(C, \mathcal{U}, \pi\ \texttt{x=}se\texttt{;}\ \omega)} \qquad \text{ifThenElse}\ \frac{(C \cup \{\{\mathcal{U}\}(se \doteq \text{TRUE})\}, \mathcal{U}, \pi\ p_1\ \omega) \quad (C \cup \{\{\mathcal{U}\}(se \doteq \text{FALSE})\}, \mathcal{U}, \pi\ p_2\ \omega)}{(C, \mathcal{U}, \pi\ \texttt{if(}se\texttt{)}\ \ p_1\ \ \texttt{else}\ \ p_2\ \omega)}$$

Fig. 2 Example SE Rules

For any number $k \leq 0$, the formula $\texttt{i} \geq k$ is also a valid (though not inductive) loop invariant. If k is *strictly* negative, the SES resulting after executing the loop has more than one concretization. If we choose $k := -1$, for example, $\texttt{i}$ can attain the values -1 *or* 0 after the loop. Consequently, the concretizations for the final SES after the program comprise some where $\texttt{i}$ is increased by two, and some where it is only increased by one: we are in the realm of *overapproximating* SE.

In this paper, an *SE transition relation* maps an SES to a non-empty set of successor SESs.[1] The big-step extension δ^* of SE transition relation is the reflexive and transitive closure.

Most practical SE transition relations can be defined as a set of *schematic* SE *rules*, where each such rule represents a family of SE transitions (one transition for each consistent instantiation of the contained schematic placeholders). We use sequent calculus notation: let *inp* and $o_1, \ldots, o_n$, for $n \geq 0$, be SESs. The SE rule

$$\text{ruleName}\ \frac{o_1\ o_2\ \cdots\ o_n}{inp}\ \text{(conditions)}$$

represents all instances of the SE transitions $(inp, \{o_1, \ldots, o_n\}))$ resulting from consistent replacement of schematic placeholders in the input and output states. The rule is read bottom-up, has a name (here "ruleName") written on the left, and may have conditions written on the right. In the following, we show two example SE rules.

Example 2 (SE Rules) Fig. 2 shows two SE rules for assignments and conditional statements. Schematic placeholders are C, $\mathcal{U}$, se, π, ω, etc. The schema variable $\mathcal{U}$ can be instantiated to any update, $\pi\omega$ to a Java context, se for a side effect-free expression, and so on. In the Java context $\pi\omega$, π is an *inactive prefix* containing opening braces, labels, and the opening of various scoping frames for methods, exceptions, etc. Scopes are created by SE rules on the fly to ensure that only normally terminating programs ever need to be symbolically executed. For example, **try-catch** statements are scoped inside a "**try** {". Dually, ω consists of closing braces, the remaining program to execute symbolically, and closings of scopes such as **catch**/**finally** clauses. Together, $\pi\omega$ constitute a valid Java program. Let, for instance,

$$s := (\overbrace{\ \emptyset\ }^{C}, \overbrace{\texttt{x := z}}^{\mathcal{U}}, \overbrace{\texttt{try}\ \{\ \texttt{y=x;}}^{\pi}\ \overbrace{\texttt{z=x;}\ \}\ \texttt{finally}\ \{\}}^{\omega})$$
$$s' := (\emptyset, (\texttt{x := z}) \circ (\texttt{y := x}), \texttt{try}\ \{\ \texttt{z=x;}\ \}\ \texttt{finally}\ \{\})$$

be two SESs. Then, the assignment rule covers the SE transition $(s, \{s'\})$.

We define two aspects of the correctness of symbolic transition relations: *Exhaustiveness* and *precision*. These properties are comparable to "recall" and "precision" in binary classification. Exhaustiveness is the property that during a symbolic transition, the set of concrete states represented by an input state is not *decreased*, whereas precision is the property this

[1] For simplicity, we do not introduce more general m-to-n transition relations as in [76].

set is not *increased*. The definitions (called "strong" exhaustiveness and precision in [76]) fix the interpretations of uninterpreted function and predicate symbols across transitions. For exhaustiveness, *fresh* symbols may be created by a transition. This is, for instance, needed in loop invariant rules where assigned locations in loop bodies are anonymized using fresh constants, see Example 1.

Definition 4 (*Exhaustive SE Transition Relation*) An SE transition relation $\delta \subseteq \mathbb{S}_{SE} \times 2^{\mathbb{S}_{SE}}$ is called *exhaustive* iff for each transition (inp, O), structure K and concrete states $\sigma, \sigma' \in \mathcal{S}$, it holds that $\sigma' \in concr_K(inp, \sigma)$ implies that there is (1) a "conservative extension" K' of K interpreting all function and predicate symbols occurring in inp the same way as K (in particular, $concr_K(inp, \sigma) = concr_{K'}(inp, \sigma)$), (2) an SES $o \in O$ and (3) a concrete state $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_{K'}(o, \sigma'')$.

Definition 5 (*Precise SE Transition Relation*) An SE transition relation $\delta \subseteq \mathbb{S}_{SE} \times 2^{\mathbb{S}_{SE}}$ is called *precise* iff for each transition (inp, O), $o \in O$, structure K and concrete states $\sigma, \sigma' \in \mathcal{S}$, it holds that $\sigma' \in concr_K(o, \sigma)$ implies that there is a concrete state $\sigma'' \in \mathcal{S}$ s.t. $\sigma' \in concr_K(inp, \sigma'')$.

Exhaustiveness is a crucial property for SE used in program verification, whereas precision is important for uncovering bugs and creating feasible test cases. Lems. 1 and 2 (proven in [76]) below formalize this intuition. To express them, we first define the concept of *labeled* SES. A labeled SES s^φ denotes the weakest precondition of s relative to postcondition φ. If s^φ is true, the labeled SES is called *valid*.

Definition 6 (*Labeled Symbolic Execution State*) We write s^φ for the *Labeled Symbolic Execution State* $s \in \mathbb{S}_{SE}$ with postcondition $\varphi \in \mathsf{Fml}$. Its semantics $val(K, \sigma | s^\varphi)$ is defined such that for all formulas ψ, $K, \sigma \models \psi$ implies $K, \sigma \models s^\varphi$ if, and only if, it holds that $K, \sigma \models \psi$ and for all $\sigma' \in concr_K(s, \sigma)$, $K, \sigma' \models \varphi$.

For example, the weakest precondition of SES $s = (\{\mathrm{x} \geq 0\}, Skip, \mathrm{x}{-}{-}\texttt{;})$ relative to postcondition $\mathrm{x} \geq 0$ is $\mathrm{x} > 0$. Consequently, $K, \sigma \models s^{\mathrm{x} \geq 0}$ iff $\sigma(\mathrm{x}) > 0$.

Lemma 1 *(Bugs discovered by precise SE are feasible) Let δ be a precise SE transition relation and $inp \rightarrow_{\delta^*} O$. If a postcondition $\varphi \in \mathsf{Fml}$ is not true for a state $o \in O$, i.e., $\not\models o^\varphi$, it follows that $\not\models inp^\varphi$.*

Lemma 2 *(A property proven by exhaustive SE holds for the inputs) Let δ be an exhaustive SE transition relation and $inp \rightarrow_{\delta^*} O$. If a postcondition $\varphi \in \mathsf{Fml}$, which only contains rigid symbols already present in inp, holds for all states $o \in O$, i.e., $\models o^\varphi$, it follows that $\models inp^\varphi$ holds.*

In Sect. 4, we devise SE rules for AE that are both precise and exhaustive with respect to the semantics of AE defined in the following section.

3 Syntax and Semantics of Abstract Execution

In Sect. 1.2, we introduced the essentials of the concrete syntax of our specification framework for abstract programs. Here, we explain features of the language that we omitted so far, define its abstract syntax and give it a semantics. Finally, we introduce *abstract updates*, a syntactic concept for second-order symbolic state changes.

Listing 3 Example Program Demonstrating Additional Specification Language Features

```
1  /*@ ae_constraint
2    @    \disjoint(frameP, footprintP) &&
3    @    \subset(frameP, rel) &&
4    @    \disjoint(x, frameP) &&
5    @    \disjoint(x, footprintP);
6    @*/
7
8  if (
9    /*@ assignable \nothing;
10     @ accessible footprintP;
11     @ normal_behavior ensures \result <==> throwsExcP(\value(footprintP));
12     @ exceptional_behavior requires false;
13     @*/
14   \abstract_expression boolean e
15 ) {
16   /*@ assignable frameP, \hasTo(x);
17     @ accessible footprintP;
18     @ normal_behavior ensures x >= 0;
19     @ exceptional_behavior
20     @   requires throwsExcP(\value(footprintP));
21     @   ensures x == -1;
22     @*/
23   \abstract_statement P;
24
25   //@ assert x >= 0 <==> !throwsExcP(\value(footprintP));
26   //     ^-- true, yet unreachable
27 }
```

3.1 Specification Language

We demonstrate additional specification language features along the abstract example program in Listing 3 containing almost all features not yet mentioned in Sect. 1.2. Our specification syntax admits the notation **\disjoint**(x, *frameP*), where x is interpreted as the singleton containing x. Intuitively, instantiations of this abstract program complete normally without changing the state, or else they complete because an exception thrown by AS P. In the resulting state after the thrown exception, the program variable x attains the final value −1. Additionally, the locations to which the abstract location set *frameP* is instantiated might be changed. One possible valid instance of the abstract program is

if (y == 0) { x = -1; z = w / y; x = 0; },

which instantiates *frameP* to {z} and *footprintP* to {w, y}.

Enforcing Assignments and Specifying "All" or "Nothing"
As a default, locations listed in **assignable** clauses are *upper bounds*: the set of represented concrete programs includes programs not assigning anything. Yet, sometimes one wants to *enforce* the assignment of a specific location. This can be done using the **\hasTo**(·) keyword, which is a JML extension specific to AE. For example, line 16 in Listing 3 imposes that all instantiations of AS P assign the variable x. Additional keywords that can be used in **assignable** and **accessible** clauses are "**\nothing**" (no location can be assigned, used in line 9) and "**\everything**" (any location may be assigned, the default).

Notation We occasionally use a simplified syntax for APEs instead of the concrete syntax with specification comments: the notations P(*assignables*: ≈ *accessibles*) and e(*assignables*: ≈ *accessibles*) represent AS and AExp with identifier symbols P

and e, respectively, both with frame *assignables*, footprint *accessibles* and, unless otherwise stated, expected to complete normally. We generally use capital letters P, Q, … for AS identifiers and lower case letters e, f, … for AExp identifiers. We distinguish **\hasTo** locations by a superscript exclamation mark as in P(x!, y: ≈ *accessibles*).

The **\hasTo** specifier enables simplification steps that would not be possible otherwise. Consider, for example, the following program, where an AS should operate on a (not "relevant") temporary variable tmp instead of the variable x and is therefore surrounded by a *set* and *reset* statement as follows:

```
tmp=x; P(tmp!: ≈ tmp, y); x=tmp;
```

Since P *has to* assign tmp, we can drop the *set* statement after considering the assignment of tmp in the footprint of P, without changing the semantics of the program, resulting in "P(tmp!: ≈ x, y); x=tmp;". Assuming that tmp is not read after this program fragment, and again using the information that it *has to* be assigned by P, we can merge the remaining statements and obtain "P(x!: ≈ x, y);". These simplifications would not have been possible without **\hasTo**. For instance, the *set* statement could not have been dropped in the first simplification step without **\hasTo** because then, P also represented the empty statement, and the *set* statement was still effective.

The *order* of frame and footprint specifications of APEs matters. The program Q(x, y: ≈ *accessibles*) has the same effect on x than Q(w, z: ≈ *accessibles*) has on w (since both AS declarations have the same identifier); however, the effect of Q(y, x: ≈ *accessibles*) (with frame elements swapped) on x will be different.

Additionally to **\disjoint**, which we used in Sect. 1.2 for declaring the disjointness of location sets, the specification language supports the JML set operators for intersection, set difference, set union, and subset (see also [5, Sect. 9.3]). In Listing 3, we declare in line 3 that *frameP* is a subset of the set *rel* of all relevant locations.

In Sect. 1.2, we explained how to couple the abrupt completion behavior of APEs to expressions built from abstract predicates. Listing 3 likewise couples the exceptional behavior of P to the abstract expression *throwsExcP*(*value*(*footprintP*)). In addition to such preconditions, we can also specify functional *postconditions*, i.e., guarantees on the state after execution of the APE. The specification in line 18 imposes that if P completes normally, the variable x has to be nonnegative afterward; similarly, if it completes due to a thrown exception, it has to equal −1 (line 21).

A noteworthy construction is used in the functional postcondition of AExp e in line 11: this boolean expression has to evaluate to true (i.e., its "**\result**" is true) iff the expression *throwsExcP*(*value*(*footprintP*)) also holds. Consequently, P always throws an exception, since otherwise, the body of the **if** statement would not be executed. AExp e always completes normally, since line 12 stipulates that it completes exceptionally iff **false** holds—that is, never. The condition asserted in line 25 of the listing is true (even though the **assert** statement is unreachable as P always completes exceptionally). This is because we defined in lines 2 and 5 that all frame elements of P are disjoint from *footprintP*, which thus retains its original value after execution of P.

AE supports additional specification cases for abrupt completion due to a (labeled) continue or (labeled) break from a block or loop. In the implementation, these cases are only considered if the specified APE occurs inside a loop (or labeled block). To specify them, use the keywords "**continue_behavior**," "**continue_behavior** (*lbl*)," "**break_behavior**" or "**break_behavior** (*lbl*)" (where *lbl* is a label occurring in the context) similarly as we used **exceptional_behavior** before.

3.2 Abstract Syntax

Abstract Execution analyzes analyzes *abstract program fragments*. A program fragment is a sequence of statements that could occur inside a method body. An abstract program fragment contains at least one APE, as well as declarations of abstract location sets, function and predicates symbols, and constraints on these elements.

We first extend the *LocSet* theory to accommodate the subsequent definitions, and then formally define APEs and abstract program fragments.

3.2.1 Program Variable Locations

The *LocSet* theory in JavaDL has been designed to represent heap locations (o, f). We extend this to also represent *program variable locations* of a new type *ProgVar*. Please keep in mind that these refer to program variable *locations* of a given name, not to the *value* of that variable in some state. We extend the vocabulary of *LocSet* by

$$\begin{gathered} pv{:}\mathsf{PVSym} \to \mathit{ProgVar} \qquad \mathit{singletonPV}{:}\mathit{ProgVar} \to \mathit{LocSet} \\ \cdot^{!}{:}\mathit{LocSet} \to \mathit{LocSet} \qquad \mathit{value}{:}\mathit{LocSet} \to \mathit{Any} \\ \mathit{heapLocs}{:}\mathit{LocSet} \to \mathit{LocSet} \qquad \mathit{pvLocs}{:}\mathit{LocSet} \to \mathit{LocSet} \\ \mathit{anonPV}{:}\mathit{ProgVar} \times \mathit{LocSet} \times \mathit{ProgVar} \to \mathit{ProgVar} \end{gathered}$$

Function *pv* is a constructor for program variable locations from a program variable symbol; *singletonPV* is a *LocSet* constructor for program variable locations. If, for example, $\mathtt{x}$ is a program variable, $pv(\mathtt{x})$ maps to the corresponding location. The difference between $pv(\mathtt{x})$ and $\mathtt{x}$ is that the latter is affected by state changes, while the former is not: We have $(\{\mathtt{x} := 17\}\mathtt{x}) \doteq 17$, but $(\{\mathtt{x} := 17\}pv(\mathtt{x})) \doteq pv(\mathtt{x})$. An expression $set^{!}$ represents the same locations as *set*; its purpose is to mark locations that have to be overwritten in assignable specifications. The semantics of a term $value(set)$ are the values attained by the locations represented by the location set *set*. For instance, the meaning of $value(singletonPV(pv(\mathtt{x})))$ is the value of program variable $\mathtt{x}$ in the current state. The functions *heapLocs* and *pvLocs* are filters for heap and program variable location sets, respectively. A term $anonPV(pv(\mathtt{x}), set, pv(\mathtt{x}'))$ (for conditional anonymization of program variables) evaluates to the program variable location $pv(\mathtt{x}')$ if $singletonPV(pv(\mathtt{x}))$ is in *set* and to $pv(\mathtt{x})$ otherwise.

For simplicity, we write $\mathring{\mathtt{x}}$ for $singletonPV(pv(\mathtt{x}))$, and use standard set notation for location sets, e.g., $loc \in set$ expresses that the location *loc* is in *set*. Dynamic frame specification variables are encoded as uninterpreted constant symbols of type *LocSet*.

3.2.2 Abstract Program Elements and Fragments

APEs are tuples of (1) an *identifier*, (2) a *type* (ASs have the designated pseudo-type statement), (3) a *frame* and (4) *footprint* specification, (5) a *termination specifier*, either partial (APE has to terminate) or total (APE may diverge), and (6) a set of *specifications* especially for sufficient and necessary preconditions of abrupt completion behavior. Specifications also comprise postconditions. In relational verification, the postcondition is frequently omitted and thus logically equals true. Normal completion does not have a precondition in AE; APE completes normally iff it does not complete abruptly. We continue writing "APE $\mathtt{P}$" short for "the APE with identifier symbol $\mathtt{P}$." Subsequently, we formally define the abstract syntax of APEs.

Definition 7 (*Abstract Program Element*) An Abstract Program Element is a tuple

$$(id, type, assignables, accessibles, term, specs)$$

of an *identifier id*, a *type type* (type statement for statements), a *frame specification assignables* and a *footprint specification accessibles* (both tuples of terms of type *LocSet*), a *termination specifier term* ∈ {partial, total} and *behavioral specifications specs*. The latter is a tuple of the form

$$(normalPost, \\ returnsSpec, excSpec, continuesSpec, breaksSpec, \\ continuesSpecLbl, breaksSpecLbl)$$

where (1) *normalPost* ∈ Fml, (2) *returnsSpec*, *excSpec*, *continuesSpec*, *breaksSpec* are pairs (*Pre*, *Post*) of formulas defining pre- and postconditions for abrupt completion of the APE due to a **`return`**, exception, **`continue`**, and **`break`**, respectively, (3) *continuesSpecLbl*, *breaksSpecLbl* are partial functions from Java labels to pairs of pre- and postconditions for abrupt completion due to a labeled **`continue`** or labeled **`break`**, (4) all preconditions are mutually exclusive, (5) pre- and postconditions may contain local variables of the context, and the special program variables `heap`, to access heap locations, $\texttt{heap}^{pre}$ to access heap locations in the state before the APE was executed, `exc` to refer to the exception in the case that the APE completes abruptly due to a thrown exception (postcondition of *excSpec* only), and `res` to refer to the result value returned by AS (postcondition of *returnsSpec* only).

Abstract Program Fragments (APFs) contain at least one APE, along with global declarations of AE specification variables and constraints on them (**`ae_constraint`**). We distinguish two types of specification variables: abstract location sets (for dynamic frames and footprints), and abstract function and predicate symbols used in the abstract specification of the behavior of APEs. APF defines the domains of the specification elements *continuesSpecLbl* and *breaksSpecLbl*: APEs must supply pre- and postconditions for exactly the labels in the context of their appearance in the APF. Constraints can also be declared *locally* within APF to, e.g., refer to globally unavailable locations such as the exception variable of a **`catch`** clause. They are w.l.o.g. treated globally in the following definition: local constraints can be converted to global ones by interpreting them in the symbolic state of their occurrence.

Definition 8 (*Abstract Program Fragments*) An *Abstract Program Fragment* is a tuple (*p*, *APEs*, *locSpecVars*, *funcAndPredSymbols*, *constraints*), where (1) *p* is a sequence of statements containing exactly the APEs in the non-empty set *APEs*, (2) *locSpecVars* is a set of dynamic frame specification variables (*LocSet* constants), comprising the symbols used in *APEs*, (3) *funcAndPredSymbols* is a set of abstract function and predicate symbols used in pre- and postconditions of the APEs, and (5) *constraints* is a set of formulas constraining the behavior of the APEs.

3.3 Semantics of Abstract Program Elements and Fragments

We define the semantics of the AE framework indirectly by reduction to JavaDL: A statement or expression is an instance of AS or AExp if it satisfies a JavaDL formula that serves as its logical representation. This approach results in longer definitions than for a direct model-theoretic semantics, but has notable advantages:

(1) The translation to a formal program logic enforces the precise description of legal instances and does not permit omitting important details.
(2) The semantics is *constructive* in the sense that it gives rise to a directly implementable approach to verify that a given program fragment instantiates APF.

(3) The previous two points facilitate *validation*, which, given the complex definition, is an important aspect.

The logical representation of APF needs to cover the following aspects: (1) the frame specification, including (2) the specific semantics of **\hasTo** (motivated in Sect. 3.1), (3) the footprint specification, [2] (4) the termination condition, (5) the contract for normal completion, including **return**, and (6) the contract for the various cases of abrupt completion. The following conjunction *represents*(*ape*, *p*) of formulas evaluates to *tt* iff the program *p* is a legal instance of (is represented by) the APE *ape*. The first four conjuncts correspond to cases (1)–(4) above. The next two conjuncts relate to case (5), and the remaining conjuncts to case (6), where the two formulas *breaksForLbl*(*breaksSpecLbl*, *lb*, *p*) and *continuesForLbl*(*continuesSpecLbl*, *lb*, *p*) for labeled **break**s and **continue**s, respectively, receive an additional parameter *lb* representing the specific label to be considered.

$$\begin{aligned} represents(ape, p) := {} & frameFor(assignables, p) \\ & \wedge hasToFor(assignables, p) \\ & \wedge footprintFor(accessibles, assignables, p) \\ & \wedge terminationFor(term, p) \\ & \wedge normalCompletionFor(specs, p) \\ & \wedge returnsFor(returnsSpec, p) \\ & \wedge excFor(excSpec, p) \\ & \wedge breaksFor(breaksSpec, p) \\ & \wedge continuesFor(continuesSpec, p) \\ & \wedge \bigwedge_{dom(breaksSpecLbl)} breaksForLbl(breaksSpecLbl, lb, p) \\ & \wedge \bigwedge_{dom(continuesSpecLbl)} continuesForLbl(continuesSpecLbl, lb, p) \end{aligned}$$

For the sake of readability, we moved the formal definitions to Appendix A. Based on *represents*, we define the semantics of a *single* APE as follows:

Definition 9 (*Semantics of APE*) Let *abstrStmt* be AS. Its semantics $[\![abstrStmt]\!]$ is the set of all concrete statements represented by it, formally:

$$[\![abstrStmt]\!] := \{stmt \mid \models represents(abstrStmt, stmt)\}$$

The definition works accordingly for AExps.

Legal instantiations of Abstract Program *Fragments* first have to provide instantiations of the APE specification variables (i.e., of abstract location sets, and function and predicate symbols) satisfying the global constraints; second, they have to provide legal *and consistent* instantiations of the APEs s.t. the resulting program is a legal concrete program fragment. An instantiation of a set of APEs is consistent if two APEs with the same identifier are instantiated by statements or expressions that are equal up to the renaming of used locations, if the frame and footprint definitions of the APE occurrences they are instantiating differ.

[2] The formula *footprintFor* asserts that executing *p* in two generic environments that agree only on the value of *p*'s *footprint* locations, in each case has the same effect on all *frame* locations. This is the reason for the presence of the *assignables* argument in *footprintFor*: Demanding that the two final states be equivalent for *all* locations would be too strong.

In our subsequent definition of the semantics of APFs, the notation $S[subst]$ denominates the result of applying the substitution *subst* on all elements of the set S; similarly for program (elements) p instead of sets.

Definition 10 (*Semantics of Abstract Program Fragment*) Let

$$\mathcal{F} = (p, APEs, locSpecVars, funcAndPredSymbols, constraints)$$

be APF. A program fragment p^0 is a *legal instantiation* of $\mathcal{F}$ if it arises from a substitution $subst_{locSpecVars}$ of concrete locations for specification variables, a substitution $subst_{funcAndPredSymbols}$ for abstract function and predicate symbols, as well as an instantiation $subst_{APEs}$ of concrete statements and expressions for APEs such that:

(1) $subst_{locSpecVars}$ substitutes concrete heap locations or program variables for elements of *locSpecVars*.
(2) $subst_{APEs}$ substitutes statements or expressions for elements of *APEs*.
(3) $subst_{funcAndPredSymbols}$ substitutes, for elements of *funcAndPredSymbols*, JavaDL terms or formulas containing at most locations corresponding to the arguments passed to the substituted symbol after applying $subst_{locSpecVars}$.
(4) The formulas $constraints[subst_{funcAndPredSymbols}][subst_{locSpecVars}]$ are valid (i.e., the global constraints on AE specification variables are satisfied).
(5) $p^0 = p[subst_{APEs}]$.
(6) Each instantiation in $subst_{APEs}$ is *represented* by the APE it instantiates, respecting the instantiations of specification variables: For all $ape \in APEs$, it holds that

$$ape[subst_{APEs}] \in [\![ape[subst_{funcAndPredSymbols}][subst_{locSpecVars}]]\!] \ .$$

(7) Each instantiation in $subst_{APEs}$ is *consistent*: For all APEs ape_1, $ape_2 \in APEs$ with the same identifier symbol, it holds that $ape_1[subst_{APEs}]$ and $ape_2[subst_{APEs}]$ are equal modulo renaming of elements in the frame and footprint specifications in ape_1 and ape_2 (after applying $subst_{locSpecVars}$).

The semantics $[\![\mathcal{F}]\!]$ of $\mathcal{F}$ is then defined as the set of its legal instantiations.

Example 3 (Instantiating APFs) The abstract program model in Listing 4, a simplified version of Listing 1 from the introduction, represents a transformation rule swapping two statements that are *independent*, i.e., cannot overwrite state changes nor interfere with the footprint of the other statement. In addition, at most one statement may complete abruptly (we only consider exceptions and **`return`**s in this example). We show that the concrete program fragment

$$p^0 := \texttt{x = y / z; a++; b = 2*a;}$$

is an instance of the abstract model by constructing substitutions as required by Def. 10 that yield p^0. Intuitively, the first assignment instantiates AS `P`, and the latter two AS `Q`. First, we instantiate *frameP* with {`x`}, *fpP* with {`y`, `z`}, *frameQ* with {`a`, `b`} and *fpQ* with {`a`}. This instantiation satisfies the constraints specified in lines 2–4 in the listing (required by Condition (4) in Def. 10).

Our instantiation of AS `P` throws an `ArithmeticException` if `z` is zero; consequently, we instantiate *throwsExcP* to $\texttt{z} \doteq 0$. Condition (3) requires that this instantiation uses at most locations corresponding to the "footprint" of the occurrences of the symbol *throwsExcP* in the abstract program, i.e., *fpP*, after substituting abstract location sets. Since we instantiate *fpP* to {`y`, `z`}, the term $\texttt{z} \doteq 0$, which accesses only `z`, satisfies this condition. All other

Listing 4 Abstract Program Model for Example 3

```
1  /*@ ae_constraint
2    @    \disjoint(frameP, frameQ) &&
3    @    \disjoint(frameP, fpQ) &&
4    @    \disjoint(frameQ, fpP) &&
5    @
6    @    \mutex(returnsP(\value(fpP)), returnsQ(\value(fpQ))) &&
7    @    \mutex(returnsP(\value(fpP)), throwsExcQ(\value(fpQ))) &&
8    @    \mutex(throwsExcP(\value(fpP)), throwsExcQ(\value(fpQ))) &&
9    @    \mutex(throwsExcP(\value(fpP)), returnsQ(\value(fpQ)));
10   @*/
11
12 //@ assignable frameP;
13 //@ accessible fpP;
14 //@ exceptional_behavior requires throwsExcP(\value(fpP));
15 //@ return_behavior requires returnsP(\value(fpP));
16 \abstract_statement P;
17
18 //@ assignable frameQ;
19 //@ accessible fpQ;
20 //@ exceptional_behavior requires throwsExcQ(\value(fpQ));
21 //@ return_behavior requires returnsQ(\value(fpQ));
22 \abstract_statement Q;
```

abstract predicates occurring in the program are instantiated to false, which trivially satisfies the requirement on accessed locations.

With this instantiation of the abstract predicate symbols, Condition (4) is satisfied: since only one abstract predicate is not instantiated to false, it is easy to see that mutual exclusion of the abrupt completion conditions (lines 6–9 in the listing) is ensured.

By substituting AS `P` with "`x = y / z;`" and `Q` with "`a++; b = 2*a;`," we obtain p^0 and the validity of Condition (5). Condition (6) refers to Def. 9. For example, we have to show that "`a++; b = 2*a;`" is represented by the following result of instantiating abstract location set and predicate symbols in AS `Q`:

```
//@ assignable a, b;
//@ accessible a;
//@ exceptional_behavior requires false;
//@ return_behavior requires false;
\abstract_statement Q;
```

We abstain to discuss the details of this condition for brevity. Intuitively, this partially instantiated AS represents all normally completing statements assigning at most `a` and `b` while accessing at most `a`, which comprises "`a++; b = 2*a;`."

The instantiation trivially satisfies Condition (7) since there are not two APE occurrences with the same identifier symbol.

3.4 Syntax and Semantics of Abstract Updates

Abstract updates are the main building block of the AE calculus. They represent syntactically unbounded many concrete state changes. Abstract Execution turns APEs $\mathtt{P}(assignables : \approx accessibles)$ into abstract updates $\mathcal{U}_{\mathtt{P}}(assignables: \approx accessibles)$. While a concrete update $\mathtt{x} := t$ assigns the value of a term t to a concrete variable `x`, an abstract update $\mathcal{U}_{\mathtt{P}}(lhs_1, \ldots, lhs_n: \approx rhs_1, \ldots, rhs_m)$ has multiple left-hand and right-hand slots. It repre-

sents all state changes writing to any subset of the left-hand side locations, where the assigned values may only be built from combinations of constants and the memory locations specified on the right. Both, left- and right-hand may be empty, also the lhs_i / rhs_j may be abstract location sets instead of concrete program variables. Like APEs, abstract updates have an identifier symbol such as $\mathcal{U}_{\mathtt{P}}$ above, with the same semantic implication: abstract updates with the same identifier represent the same state changes, parametric in the arguments they are passed. We connect syntactically abstract updates and APEs by their names, as in $\mathcal{U}_{\mathtt{P}}$ for AS `P`, but do not enforce a *semantic* connection.

We define the syntactic category of *abstract update symbols*. An abstract update symbol is an operator with a name (such as $\mathcal{U}_{\mathtt{P}}$), a list of parameters (the assignable locations), and an arity. *Abstract updates* are created from the application of an abstract update symbol to a list of terms (the right-hand sides, or "accessibles"). The length of the list has to match the arity of the symbol. Abstract updates can be used in the construction of sequential and parallel updates and update applications.

Definition 11 (Abstract Update Symbol) We define an *abstract update symbol* $\mathcal{U}_{\mathtt{P}}(assignables)$ as an identifier $\mathcal{U}_{\mathtt{P}}$, a n-tuple of *LocSet* parameter terms *assignables*, and an arity m (with $n, m \geq 0$). Each abstract update symbol with the same identifier has (1) the same number n of assignable locations, and (2) the same arity m. The set of all abstract update symbols is denoted by $\mathrm{Upd}^{\mathcal{A}}$. To the set Upd of updates we add, for $\mathcal{U}_{\mathtt{P}}(assignables) \in \mathrm{Upd}^{\mathcal{A}}$, *abstract updates* $\mathcal{U}_{\mathtt{P}}(assignables\colon \approx accessibles)$, which may occur in compound update constructions. The right-hand side *accessibles* is an m-tuple of argument terms, where m is the arity of the abstract update symbol.

To define the semantics of abstract updates, we extend the interpretation function I of Java DL Kripke structures such that $I(\mathcal{U}_{\mathtt{P}}(assignables))$ returns a function $(\mathcal{D})^m \to \mathcal{S} \to \mathcal{S}$ that, depending on the values of the right-hand side of an abstract update, returns a state transformer. We then extend the valuation function of dynamic logic accordingly. The interpretation of an abstract update symbol has to respect its "frame" (i.e., *assignables*). Furthermore, we have to ensure that the interpretation of abstract updates *with the same identifier* is equivalent "modulo frame changes." For instance, the abstract update $\mathcal{U}_{\mathtt{P}}(\mathring{\mathtt{x}}\colon \approx accessibles)$ should have the same effect on `x` that $\mathcal{U}_{\mathtt{P}}(\mathring{\mathtt{y}}\colon \approx accessibles)$ has on `y`: it has to hold that $\mathtt{x} \doteq \mathtt{y} \to \{\mathcal{U}_{\mathtt{P}}(\mathring{\mathtt{x}}\colon \approx accessibles)\}\mathtt{x} \doteq \{\mathcal{U}_{\mathtt{P}}(\mathring{\mathtt{y}}\colon \approx accessibles)\}\mathtt{y}$. We need the premise $\mathtt{x} \doteq \mathtt{y}$ because the left-hand sides in the abstract updates are not declared as "has-to:" they do not have to be written, in which case the variables have to be equal in the pre-state for the equality to hold. Omitting the premise yields the constraint on the semantics of has-to left-hand sides.

Definition 12 (*Semantics of Abstract Update*) An interpretation function I of a Java DL Kripke structure $(\mathcal{D}, I, \mathcal{S}, \varrho)$ assigns to a symbol $\mathcal{U}_{\mathtt{P}}(assignables) \in \mathrm{Upd}^{\mathcal{A}}$ with arity m a function $(\mathcal{D})^m \to \mathcal{S} \to \mathcal{S}$, such that:

(1) *Frame Condition:* Let $accessibles \in (\mathcal{D})^m$ and $\sigma \in \mathcal{S}$. For all locations $loc \in \mathcal{D}^{LocSet}$, it holds that either $loc \in val\,(K, \sigma\,|assignables)$, or

$$\sigma(loc) = I(\mathcal{U}_{\mathtt{P}}(assignables))(accessibles)(\sigma)(loc).$$

(2) *State Transformers for Same Identifier Are Equivalent:* Let, for any $i = 1, \ldots, n$, be $\mathcal{U} = \mathcal{U}_{\mathtt{P}}(s_1, \ldots, s_i, \ldots, s_n) \in \mathrm{Upd}^{\mathcal{A}}$, $accessibles \in (\mathcal{D})^m$ and $\sigma \in \mathcal{S}$. For

all location set terms s_i' representing the same number of concrete locations as s_i (i.e., $|val\,(K, \sigma|s_i)| = |val\,(K, \sigma|s_i{}')|$), there has to be a bijective mapping ι between $val\,(K, \sigma|s_i)$ and $val\left(K, \sigma|s_i'\right)$, such that for all $loc \in val\,(K, \sigma|s_i)$, it holds that

$$I(\mathcal{U})(accessibles)(\sigma)(loc) = I(\mathcal{U}')(accessibles)(\sigma')(\iota(loc)),$$

where $\mathcal{U}' := \mathcal{U}_{\mathbb{P}}(s_1, \ldots, s_i', \ldots, s_n)$ and $\sigma' := \sigma[\iota(loc) \mapsto \sigma(loc)]$.

(3) *Has-To Condition:* For $\mathcal{U} = \mathcal{U}_{\mathbb{P}}(s_1, \ldots, s_i^{!}, \ldots, s_n)$, the requirement of Condition (2) has to hold for has-to locations $s_i' = (s_i'')^{!}$ and $\sigma' := \sigma$.

The mapping ι in Condition (2) of Def. 12 is required because a single element of the *assignables* of an abstract update symbol can be an *abstract* location set and therefore represent many concrete locations. The definition requires that state transformers created for the two abstract updates with the same identifier and equal *accessibles* arguments transform a pre-state σ, where a location *loc* has the same value as its corresponding location $\iota(loc)$, to a state where they *still* have the same value (though potentially different from the value in σ). If *loc* is a "has-to" location, the value in the resulting state will be equal independent of the value in the pre-state.

Extending the valuation function $val\,(K, \sigma, \beta|\cdot)$ is straightforward.

Definition 13 (Valuation of Abstract Update) We extend the JavaDL valuation function $val\,(K, \sigma, \beta|\cdot)$ as follows, for $\mathcal{U}_{\mathbb{P}}(assignables) \in \mathrm{Upd}^{\mathcal{A}}$ with arity m:

$$val\,(K, \sigma, \beta|\mathcal{U}_{\mathbb{P}}(assignables: \approx t_1, \ldots, t_m)) =$$
$$I(\mathcal{U}_{\mathbb{P}}(assignables))(val\,(K, \sigma, \beta|t_1), \ldots, val\,(K, \sigma, \beta|t_m))$$

Subsequently, we conclude the section on the syntax and semantics of Abstract Execution by considering *Abstract* Symbolic Execution States.

3.5 Syntax and Semantics of Abstract Symbolic Execution States

We generalize the notion of SES from Sect. 2.2. The only changes are that due to Def. 11, symbolic stores can also include abstract updates, and we use APFs instead of concrete program fragments as program counters.

Definition 14 (*Abstract Symbolic Execution State*) An *Abstract* SES is a triple $(C, \mathcal{U}, \mathcal{F})$ of (1) a set of closed formulas $C \in 2^{\mathsf{Fml}}$, the *path condition*, (2) a (potentially abstract) update $\mathcal{U} \in \mathsf{Upd}$, the *symbolic store*, and (3) an *Abstract* Program Fragment $\mathcal{F}$, the *program counter*. We write $\mathbb{S}_{SE}^{\mathcal{A}}$ for the set of all abstract SESs.

As in the concrete case, the semantics of abstract SESs is based on the concept of *concretization functions*. The concretization function for abstract SESs takes a *concrete* program fragment as additional argument: if a given concrete program is represented by the abstract program counter, the concretization for this program is part of the semantics of the abstract SES; otherwise, the result is the empty set. The semantics of the abstract SES is obtained by constructing the union over the set *Stmt* of all concrete program fragments.

Definition 15 (*Semantics of Abstract SES*) The *K-indexed abstract concretization function* $concr_K^{\mathcal{A}}: \mathbb{S}_{SE}^{\mathcal{A}} \times \mathcal{S} \times Stmt \to 2^{\mathcal{S}}$ maps an abstract SES $(C, \mathcal{U}, \mathcal{F})$, a concrete state $\sigma \in \mathcal{S}$ and concrete program element p either (1) to the empty set $\emptyset$ if $p \notin [\![\mathcal{F}]\!]$, or (2) to the set $concr_K\,(C, \mathcal{U}, p)$ otherwise. The *abstract concretization function* $concr^{\mathcal{A}}$ is defined as $concr^{\mathcal{A}}(s, \sigma, p) := \bigcup_K (s, \sigma, p)$. The semantics $[\![s]\!]$ of an abstract SES $s \in \mathbb{S}_{SE}^{\mathcal{A}}$ is defined as $[\![s]\!] := \bigcup_{\sigma \in \mathcal{S}} \bigcup_{p \in Stmt} concr^{\mathcal{A}}(s, \sigma, p)$.

4 Rules for Abstract Execution and Abstract Store Simplification

The fundamental idea of Abstract Execution is to perform second-order reasoning about universal properties of program behavior by *Symbolic Execution*: Abstract Statements and Abstract Expressions are translated into *abstract updates*; abrupt completion is taken into account by explicit branches in the symbolic execution tree. Thus, the core constituents of our reasoning system, presented in this section, are SE rules for APEs and simplification rules for abstract stores containing abstract updates.

4.1 Symbolic Execution Rules for Abstract Program Elements

AE is necessarily less expressive and complete than full structural induction over program syntax in higher-order logic, because it approximates induction with the fixed set of descriptive elements contained in abstract updates obtained from APEs. The advantage is that the resulting symbolic execution rules can be instantiated by matching, without having to guess an induction hypothesis. This has a dramatic impact on the efficiency and automation of AE. Nevertheless, because of the complications due to abrupt termination, the full AE rule for ASs without any abbreviations is rather lengthy. Therefore, we begin with a *concise* AE rule for the case of ASs (with frame *frame* and footprint *footprint*) that *complete normally*:

abstractStatementSimple

$$\frac{(C \cup \{\{\mathcal{U} \circ \mathcal{U}_{\mathtt{P}}(\mathit{frame}\colon \approx \mathit{value}(\mathit{footprint}))\}(\mathit{normalPost})\},\ \mathcal{U} \circ \mathcal{U}_{\mathtt{P}}(\mathit{frame}\colon \approx \mathit{value}(\mathit{footprint})),\ \pi\ \ \omega)}{(C, \mathcal{U}, \pi\ \ \texttt{\textbackslash abstract\_statement}\ \ \mathtt{P;}\ \ \omega)}$$

The rule removes the AS `P` and appends to the symbolic store an abstract update. The abstract update symbol $\mathcal{U}_{\mathtt{P}}$ is created fresh when first executing AS with identifier `P`, but is reused for further executions of ASs with the same identifier to ensure that ASs with the same identifier behave equivalently when executed with the same footprint values. The path condition is extended by the postcondition of normal completion, *normalPost*, in the state after execution of the AS. A precondition is not added, since it is not allowed for normal completion. This rule is exhaustive *and* precise, since the semantics of the abstract update symbol $\mathcal{U}_{\mathtt{P}}(\mathit{frame})$ is aligned with the semantics of a normally completing AS with identifier `P`. Only for ASs with non-trivial postconditions, the path condition has to be updated to achieve precision.

Complexity is added by considering abrupt completion. We first discuss the complete AE rule for AExps, depicted in Fig 3. The statement in the program counter of the rule's conclusion is the assignment of AExp `e` to a variable `v`. The execution of `e` can either complete normally, in which case an abstract value is assigned to `v`, or else because of a thrown exception. In the latter case, the assignment to `v` must not happen. Instead, the exception is thrown. Consequently, the active statement of the conclusion is not simply removed from the program counter (as in abstractStatementSimple), but *replaced* with a conditional throw of an abstract exception object `exc` if symbolic flag `throwsExc` is true, and an assignment of a symbolic value `res` to `v` otherwise.

The generated symbolic store is more complex as in abstractStatementSimple, because the values of `throwsExc`, `exc` and `res` have to be suitably initialized. Variable `throwsExc` refers to the pre-state before executing `e`. The corresponding state update, `throwsExc` $:= \mathit{throwsExc}^{\mathtt{e}}(\mathit{value}(\mathit{footprint}))$, is therefore added to the symbolic store before the abstract update $\mathcal{U}_{\mathtt{e}}(\mathit{frame}\colon \approx \mathit{value}(\mathit{footprint}))$. The values of `exc` and `res` are

$$\frac{\begin{array}{l}\text{abstractExpression}\\ \big(C \cup \big\{\{\mathcal{U} \circ (\texttt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})))\}\\ \qquad (\texttt{throwsExc} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec})),\\ \quad \{\mathcal{U} \circ (\texttt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))) \circ\\ \quad\ \mathcal{U}_{\mathsf{e}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint})) \circ\\ \quad\ \ (\texttt{exc} := \mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})) \,||\, \texttt{res} := \mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})))\}\\ \quad\ \ ((\texttt{throwsExc} \doteq \text{TRUE} \rightarrow \texttt{exc} \neq \texttt{null} \wedge \mathit{post}(\mathit{excSpec})) \wedge\\ \quad\ \ (\texttt{throwsExc} \neq \text{TRUE} \rightarrow \mathit{normalPost}))\big\},\\ \mathcal{U} \circ (\texttt{throwsExc} := \mathit{throwsExc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))) \circ\\ \ \mathcal{U}_{\mathsf{e}}(\mathit{frame} :\approx \mathit{value}(\mathit{footprint})) \circ\\ \quad (\texttt{exc} := \mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint})) \,||\, \texttt{res} := \mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))),\\ \pi\ \texttt{if (throwsExc) throw exc;}\\ \ \texttt{v = res;}\ \omega\big)\end{array}}{\big(C, \mathcal{U}, \pi\ \texttt{v=\abstract\_expression}\ T\ \texttt{e;}\ \omega\big)}$$

Fig. 3 Symbolic Execution Rule for AE of Abstract Expressions

interpreted in the post-state, and are set to $\mathit{exc}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))$ and $\mathit{res}^{\mathsf{e}}(\mathit{value}(\mathit{footprint}))$, respectively, The abstract functions $\mathit{throwsExc}^{\mathsf{e}}$, $\mathit{exc}^{\mathsf{e}}$, and $\mathit{res}^{\mathsf{e}}$ of types `boolean`, `Throwable` and T are, similar to the abstract update symbol $\mathcal{U}_{\mathsf{e}}$, created fresh when first executing AExp with identifier `e`, but are reused for every further execution of AExp with the same identifier.

The path condition is extended by two formulas. First, the value of the $\mathit{throwsExc}^{\mathsf{e}}$ expression is bound to the evaluation of the precondition for exceptional completion, $\mathit{pre}(\mathit{excSpec})$, in the pre-state. Second, the assumptions concerning the postconditions for both completion modes are evaluated in the whole new symbolic store, i.e., the post-state. If an exception is thrown, i.e., `throwsExc` evaluates to TRUE, the exception object is assumed to be non-null, and the postcondition $\mathit{post}(\mathit{excSpec})$ is assumed to hold. In the converse case for normal completion, *normalPost* is assumed. These postconditions may contain the variables `exc` (for exceptional completion) and `res` (for normal completion), see Def. 7.

If $\mathit{pre}(\mathit{excSpec})$ is satisfiable in the pre-state, subsequent symbolic execution after an application of abstractExpression will result in two SE branches, one for normal completion, and one for completion due to a thrown exception.

Figure. 4 shows the AE rule abstractStatment for ASs. It is in essence an extension of abstractExpression with the additional cases for abrupt completion of a statement compared to an expression (**return**s, (labeled) **break**s, (labeled) **continue**s). In each case, a conditional in the program counter yields a separate SE branch. Since statements generally do not evaluate to a value (with the exception of "expression statements"), the program counter does not contain an assignment.

The labels $lb_{b_1}, \ldots, lb_{b_n}$ are (distinct) loop or block labels declared in the prefix π; $lb_{c_1}, \ldots, lb_{c_m}$ are all *loop* labels only. The rule is only applicable in the context of a non-void method (since we **return** a value in the program counter) and within a loop (since we **break** and **continue**). For different contexts, we provide dedicated variants of rule abstractExecution, which we do not detail here for brevity.

abstractStatement

$$\frac{\begin{array}{l}(C \cup \big\{\{\mathcal{U} \circ \mathcal{U}_{init}\} mutualExclusionFor,\\ \quad \{\mathcal{U} \circ \mathcal{U}_{init}\}(\texttt{normal} \doteq \text{TRUE} \leftrightarrow notAbruptly),\\ \quad \{\mathcal{U} \circ \mathcal{U}_{init}\} behavioralPreconds,\\ \quad \{\mathcal{U} \circ \mathcal{U}_{init} \circ \mathcal{U}_{\mathrm{P}}(frame :\approx value(footprint)) \circ\\ \quad\quad (\texttt{exc} := exc^{\mathrm{e}}(value(footprint)) \,||\, \texttt{res} := res^{\mathrm{e}}(value(footprint)))\}\\ \quad\quad behavioralPostconds\big\},\\ \mathcal{U} \circ \mathcal{U}_{init} \circ \mathcal{U}_{\mathrm{P}}(frame :\approx value(footprint)) \circ\\ \quad (\texttt{exc} := exc^{\mathrm{e}}(value(footprint)) \,||\, \texttt{res} := res^{\mathrm{e}}(value(footprint))),\\ \pi\ \texttt{if (throwsExc) throw exc;}\\ \quad \texttt{if (returns) return res;}\\ \quad \texttt{if (breaks) break;}\\ \quad \texttt{if (continues) continue;}\\ \quad \texttt{if (breaks\_}lb_{b_1}\texttt{) break } lb_{b_1}\texttt{;}\ \cdots\ \texttt{if (breaks\_}lb_{b_n}\texttt{) break } lb_{b_n}\texttt{;}\\ \quad \texttt{if (continues\_}lb_{c_1}\texttt{) continue } lb_{c_1}\texttt{;}\ \cdots\\ \quad\quad \texttt{if (continues\_}lb_{c_m}\texttt{) continue } lb_{c_m}\texttt{;}\ \omega)\end{array}}{(C, \mathcal{U}, \pi\ \backslash\texttt{abstract\_statement P;}\ \omega)}$$

Fig. 4 Symbolic Execution Rule for AE of Abstract Statements (abbreviations and label symbols are explained in the text)

For readability, we use abbreviations in Fig. 4. The update $\mathcal{U}_{init}$ initializes all boolean flags such as `throwsExc` and `returns`, as in the case of abstractExpression:

$$\begin{array}{rl}\mathcal{U}_{init} := & \texttt{throwsExc} := throwsExc^{\mathrm{P}}(value(footprint)) \,||\\ & \texttt{returns} := returns^{\mathrm{P}}(value(footprint)) \,||\\ & \texttt{breaks} := breaks^{\mathrm{P}}(value(footprint)) \,||\\ & \texttt{continues} := continues^{\mathrm{P}}(value(footprint)) \,||\\ & \texttt{breaks\_}lb_{b_1} := breaksLb^{\mathrm{P}}(lb_{b_1}, value(footprint)) \,||\, \cdots \,||\\ & \texttt{breaks\_}lb_{b_n} := breaksLb^{\mathrm{P}}(lb_{b_n}, value(footprint)) \,||\\ & \texttt{continues\_}lb_{c_1} := continuesLb^{\mathrm{P}}(lb_{c_1}, value(footprint)) \,||\, \cdots \,||\\ & \texttt{continues\_}lb_{c_m} := continuesLb^{\mathrm{P}}(lb_{c_m}, value(footprint))\end{array}$$

The formula *mutualExclusionFor* declares mutual exclusion of all flags that appear in $\mathcal{U}_{init}$ as left-hand sides, such that at most one of them can evaluate to TRUE. AS completes normally iff it does not complete abruptly, i.e., if all abrupt completion flags evaluate to FALSE. This is captured in the formula *notAbruptly* defined as $\texttt{throwsExc} \doteq \text{FALSE} \wedge \texttt{returns} \doteq \text{FALSE} \wedge \cdots$. The formula *behavioralPreconds* binds the values of the flags to the corresponding preconditions defined by the AS. Since the specifications for labeled **`break`**s and **`continue`**s are parametric in the label, the corresponding formulas are also passed the label

as a parameter:

$$
\begin{aligned}
&\mathit{behavioralPreconds} := \\
&\quad (\texttt{throwsExc} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{excSpec})) \wedge \\
&\quad (\texttt{returns} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{returnsSpec})) \wedge \\
&\quad (\texttt{breaks} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{breaksSpec})) \wedge \\
&\quad (\texttt{continues} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{continuesSpec})) \wedge \\
&\quad (\texttt{breaks\_}\mathit{lb}_{b_1} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_1}))) \wedge \cdots \wedge \\
&\quad (\texttt{breaks\_}\mathit{lb}_{b_n} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_n}))) \wedge \\
&\quad (\texttt{continues\_}\mathit{lb}_{c_1} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_1})) \wedge \cdots \wedge \\
&\quad (\texttt{continues\_}\mathit{lb}_{c_m} \doteq \text{TRUE} \leftrightarrow \mathit{pre}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_m})))
\end{aligned}
$$

Finally, *behavioralPostconds* adds the assumptions about all postconditions:

$$
\begin{aligned}
&\mathit{behavioralPostconds} := \\
&\quad (\texttt{normal} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{normalSpec})) \wedge \\
&\quad (\texttt{throwsExc} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{excSpec}) \wedge \texttt{exc} \not\doteq \texttt{null}) \wedge \\
&\quad (\texttt{returns} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{returnsSpec})) \wedge \\
&\quad (\texttt{breaks} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpec})) \wedge \\
&\quad (\texttt{continues} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpec})) \wedge \\
&\quad (\texttt{breaks\_}\mathit{lb}_{b_1} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_1}))) \wedge \cdots \wedge \\
&\quad (\texttt{breaks\_}\mathit{lb}_{b_n} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{breaksSpecLbl}(\mathit{lb}_{b_n}))) \wedge \\
&\quad (\texttt{continues\_}\mathit{lb}_{c_1} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_1})), \wedge \cdots \wedge \\
&\quad (\texttt{continues\_}\mathit{lb}_{c_m} \doteq \text{TRUE} \rightarrow \mathit{post}(\mathit{continuesSpecLbl}(\mathit{lb}_{c_m})))
\end{aligned}
$$

Observe that according to Def. 7, preconditions are mutually exclusive, so we can connect them to the Boolean flags with equivalence "$\leftrightarrow$" in *behavioralPreconds*. This requirement does not, and usually will not, have to hold for postconditions, which is why we use implication "$\rightarrow$" in *behavioralPostconds*.

The most important feature of our AE rules is that they are *exhaustive* because this implies by Lem. 2 that we can soundly prove abstract program properties by using them. In addition, they should be *precise*, such that they allow proving (modulo inherent incompleteness of used theories like arithmetic) everything that is logically valid. Indeed, our rules satisfy both properties. Subsequently, we state the corresponding theorems and provide proof sketches. For full proofs, we refer to [76]. [3]

Theorem 1 *The rule abstractStatement (Fig. 4) is exhaustive.*

Proof Sketch We have to prove that for all instantiations of the conclusion SES in abstractStatement, each concretization (concrete state represented by the SES) is also a concretization of the premise SES. The core insights used in the proof are:

[3] In [76], soundness and completeness for a validity calculus are proven instead of exhaustiveness and precision of SE; the arguments are similar, though, since these notions are related.

(1) We perform a case distinction over the reasons for (normal or abrupt) completion of the AS instantiation. This is in the spirit of AE, which reasons about programs based on their *effect* rather than their syntactic structure.
(2) We defined the semantics of APEs as a conjunction of JavaDL formulas. For a fixed, but arbitrary instantiation of the AS in the conclusion, we can assume the validity of this conjunction, and exploit the fact that it shares common elements with the premise SES to perform strong, semantics-preserving simplifications.
(3) Re-using abstract updates and first-order symbols such as $exc^{\mathbb{P}}$ for APEs with the same identifier is soundness-critical; however, it is admissible since this only happens in the AE rules, and for equivalent APEs (modulo frame changes). Using truly fresh *first-order symbols* every time would be sound, but either incomplete or require non-trivial postconditions in the presence of multiple APEs with the same identifier symbol. The usage of fresh *abstract updates* would even require specifying the whole framed post-state for completeness. Note that all terms with such re-used symbols depend on the current value of the relevant context (the footprint of the APE). The contrary would be unsound. □

Introducing abstract updates and first-order symbols freshly upon the first encounter with APE with a given identifier symbol, but re-using them later for APEs with this identifier is soundness-critical, but greatly simplifies both symbolic reasoning and the required specification effort in the presence of multiple APEs with the same identifier. While this calls for a discussion as in Item (3) for exhaustiveness, it *simplifies* the argument for *precision*.

Theorem 2 *The rule abstractStatement (Fig. 4) is precise.*

Proof Sketch We have to prove that for all instantiations of the premise SES in abstractStatement, each concretization (concrete state represented by the SES) is also a concretization of the conclusion SES. For a given structure and initial concrete state, we may assume the validity of the path condition of the premise SES, since otherwise, no concretization is produced. It follows that the formula imposing mutual exclusion of abrupt completion is valid, which is why we can, as for exhaustiveness, proceed by case distinction over behavior. Because due to the semantics of APFs, APEs with the same identifiers are behaviorally isomorphic, we have to use the fact that logic symbols introduced for ASs with the same identifier symbols are re-used (since fresh symbols would yield more concretizations). This argument is non-standard: there is no formal connection between interpretations of abstract update symbols and the ASs they have been introduced for, but since there is only one rule for executing ASs and this rule always uses the *same* symbols for ASs with the same identifier symbol, we narrow down the interpretations of introduced logic symbols to the feasible ones. □

The proofs for abstractExpression work analogously.

Theorem 3 *The rule abstractExpression (Fig. 3) is exhaustive.*

Theorem 4 *The rule abstractExpression (Fig. 3) is precise.*

The AE rules abstractStatement and abstractExpression transform APEs into second-order abstract updates in the symbolic store. To facilitate further reasoning with the resulting SESs, we have to provide sufficiently strong simplification rules for abstract updates, which we present in the following section.

4.2 Update Simplification Rules

We first provide an intuition of the mechanics of update simplification by discussing how *concrete* updates are simplified. Afterward, we introduce our simplification rules for *abstract* updates.

4.2.1 Concrete Update Simplification Rules

Symbolic execution transforms assignments to changes in the symbolic store. To evaluate a postcondition in a symbolic store after the execution terminated, the store has to be *applied* to the postcondition. Consider the program "`x = 17; y = x; x = 42;`." The resulting symbolic store for this program is $\{\mathtt{x} := 17\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := 42\}$. To evaluate the postcondition $\mathtt{y} > 0$, we first have to simplify the store to a *parallel normal form* $\mathtt{v}_1 := t_1 \,||\, \cdots \,||\, \mathtt{v}_n := t_n$ with distinct left-hand sides $\mathtt{v}_i$ and terms t_i without updates. This is achieved by several update *simplification rules*. Java DL provides suitable rules for concrete updates. For the example above, we first apply the rule seqToPar twice to transform the sequential update into a parallel update $\mathtt{x} := 17 \,||\, \{\mathtt{x} := 17\}(\mathtt{y} := \mathtt{x} \,||\, \{\mathtt{y} := \mathtt{x}\}(\mathtt{x} := 42))$. Next, we turn $\{\mathtt{y} := \mathtt{x}\}(\mathtt{x} := 42)$ into $\mathtt{x} := \{\mathtt{y} := \mathtt{x}\}42$ using one of the applyOnRigid rules. Inside the right-hand side of the resulting update, we can drop the update application $\{\mathtt{y} := \mathtt{x}\}$ using the $\mathsf{dropUpdate}_2$ rule since the variable $\mathtt{y}$ does not occur in the term 42. Formally, this is captured by the condition $\mathtt{y} \notin \mathit{fpv}(42)$, where the function $\mathit{fpv}(t)$ returns the "free" program variables in the term t. We continue by applying the update $\{\mathtt{x} := 17\}$, which leads to the update $\mathtt{x} := 17 \,||\, \mathtt{y} := \{\mathtt{x} := 17\}\mathtt{x} \,||\, \{\mathtt{x} := 17\}(\mathtt{x} := 42)$. The second application of $\mathtt{x} := 17$ can be dropped as before; the term $\{\mathtt{x} := 17\}\mathtt{x}$ is simplified to 17 by applying the update to the variable $\mathtt{x}$ using rule applyOnTarget. This results in the simplified update $\mathtt{x} := 17 \,||\, \mathtt{y} := 17 \,||\, \mathtt{x} := 42$. Using the rule $\mathsf{dropUpdate}_1$, we drop the update $\mathtt{x} := 17$ since it is overwritten by a later update in the parallel construction, leading to the update $\mathtt{y} := 17 \,||\, \mathtt{x} := 42$, which is in parallel normal form. Applying that update to our postcondition $\mathtt{y} > 0$ yields the true formula $17 > 0$.

4.2.2 Abstract Update Simplification Rules

For abstract updates, we can reuse most of the existing machinery. One must strengthen the rule $\mathsf{dropUpdate}_2$, because the condition $\mathtt{x} \notin \mathit{fpv}(t)$ is not sufficient if t contains **`\value`** terms depending on dynamic frame specification variables. Consider, for instance, the formula $\mathit{value}(\mathit{locs}) \doteq \{\mathtt{x} := 17\}\mathit{value}(\mathit{locs})$, where *locs* is a dynamic frame variable of type *LocSet*. This formula is only true, i.e., $\{\mathtt{x} := 17\}$ can only be dropped, if we know from the current execution context that $\mathtt{x}$ is not in *locs*. Consequently, in addition to $\mathtt{x} \notin \mathit{fpv}(t)$, we have to ensure by checking the path condition that $\mathtt{x} \notin \mathit{locset}$ for each dynamic frame variable *locset* such that $\mathit{value}(\mathit{locset})$ occurs in t to be able to apply $\mathsf{dropUpdate}_2$.

The replacement $\mathsf{dropUpdate}_2'$ for $\mathsf{dropUpdate}_2$, as well the other simplification rules for abstract updates, therefore, not only depend on the term in focus, but also on the execution context captured by the path condition C of a symbolic state. In analogy to the condition $\mathtt{x} \notin \mathit{fpv}(t)$, expressing that $\mathtt{x}$ is *irrelevant* for the term t, we formalize a predicate $\mathit{irrelevant}(C, \mathit{locset}, t)$ expressing that the location set *locset* is not relevant for the target term t. It holds if the path condition implies $\mathit{locset} \cap \{s\} \doteq \emptyset$ for each *LocSet* constant s such that $\mathit{value}(s)$ is a subterm of t. Additionally, there are some special cases: The simplification rule contraction for abstract updates discussed below introduces placeholders "_" to frames that are by definition "irrelevant," and treated accordingly. For program variables, we assert that the variable does not occur freely in the target, for heap locations that there is no free occurrence of the `heap` variable. The latter is a safe overapproximation; fine-grained heap-

related simplifications require dedicated rules (we explain an example further below). We define *irrelevant* on *tuples* of locations.

Definition 16 (*Location Set Irrelevance Checking*) Let $C \subseteq \mathsf{Fml}$ be a path condition, *locs* a tuple of locations (program variables, heap locations, and dynamic frame variables) and $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$. We define $\mathit{irrelevant}(C, \mathit{locs}, t)$ as

$$\begin{aligned}\mathit{irrelevant}(C, (s_1, s_2, \ldots, s_n), t) :=& \forall i = 1, \ldots, n; (s_i = \_ \vee \\ & (\forall\, \mathit{value}(s) \text{ in } t; C \text{ implies } s_i \cap s \doteq \emptyset) \wedge \\ & (s_i = \mathtt{x} \rightarrow \mathtt{x} \notin \mathit{fpv}(t)) \wedge (s_i = (o, f) \rightarrow \mathtt{heap} \notin \mathit{fpv}(t))) \ .\end{aligned}$$

In addition to *irrelevant*, which tells us that assigning a location set has *no* effect on the valuation of a target, we need a predicate $\mathit{overwrites}(C, \mathit{locs}_1, \mathit{locs}_2)$ expressing that assignment *of all* **\hasTo** *locations* in the location tuple locs_1 will at least assign all locations in the location tuple locs_2. Depending on the type of a location in locs_2, there are several ways to conclude that the location is overwritten. In the simplest case, a location literally occurs in locs_1, as in $\mathit{overwrites}(C, \mathtt{x}^!, \mathtt{x})$ or $\mathit{overwrites}(C, \mathit{abstrLocSet}^!, \mathit{abstrLocSet})$; the judgment is independent from C then. If the location tuple is a singleton (either a heap or program variable location), we check whether a suitable expression $(o, f) \in s$ occurs in the context. Otherwise, we have to find a combination of **\hasTo** locations in C such that the union of these locations covers locs_2.

Definition 17 (*Location Set Overwrite Checking*) Let $C \subseteq \mathsf{Fml}$ be a path condition, and $\mathit{locs}_1 \in (\mathsf{Trm}_{\mathit{LocSet}})^n$, $\mathit{locs}_2 \in (\mathsf{Trm}_{\mathit{LocSet}})^m$ tuples of location set terms, where $n, m \in \mathbb{N}$. The predicate $\mathit{overwrites}(C, \mathit{locs}_1, \mathit{locs}_2)$ is defined as [4]

$$\begin{aligned}\mathit{overwrites}(C, (s_1^1, s_2^1, \ldots, s_n^1), (s_1^2, s_2^2, \ldots, s_m^2)) :=& \forall i = 1, \ldots, m; \Big(s_i^2 = \_ \text{ or} \\ \exists s_k^1 = s^!; \big((s_i^2 = s) \vee (s_i^2 = (o, f) \wedge ((o, f) \in s) \in C) \vee (s_i^2 = \mathtt{x} \wedge (\mathtt{x} \in s) \in C)\big) \vee & \\ \exists s_{k_1}^1 = (s_1')^!, \ldots, s_{k_l}^1 = (s_l')^!; \big((s_i^2 \subseteq (s_1' \cup \cdots \cup s_l')) \in C\big)\Big) \ . &\end{aligned}$$

We subsequently discuss the "core" abstract update simplification rules in Fig. 5. In addition to those, we provide a second rule set dedicated to simplifying *heap-related* terms that frequently arise in the context of AE proofs. Understanding these rules requires no new insights or techniques. To make the paper self-contained, we include the heap-related abstract update simplification rules in Appendix B.

In the rules we use the following tuple notation: If *frame* is an n-tuple, we write *value*(*frame*) instead of $(\mathit{value}(0), \ldots, \mathit{value}(n))$ and similarly for application of updates, etc.

Rule $\mathsf{dropUpdate}_2'$ was already discussed. Three further rules, $\mathsf{dropUpdate}_3$ to $\mathsf{dropUpdate}_5$, for dropping updates. Rules $\mathsf{dropUpdate}_3$ and $\mathsf{dropUpdate}_4$ correspond to the existing $\mathsf{dropUpdate}_1$ in Java DL, dropping an earlier update within a parallel composition if a later one dominates it. The first of these rules replaces an earlier concrete update $\mathtt{a} := t'$ by *Skip* if $\mathtt{a}$ is overwritten by the frame of a later abstract update. The second rule treats the case of an earlier *abstract* update that is dropped. This case is more complex due to the nature of abstract updates. We can drop the abstract update $\mathcal{U}_{\mathtt{P}}(\mathit{frame}\colon \approx \mathit{footprint})$

[4] The notation "$\exists s_k^1 = s^!; \varphi(s)$" is short for "$\exists k \in 1, \ldots, n; (s_k^1 \text{ is hasTo} \wedge \text{let } s := s_k^1 \text{ in } \varphi(s))$."

$\{\ldots \,||\, \mathsf{a} := t' \,||\, \ldots\}t \rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \ldots\}t$ dropUpdate$'_2$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$, $\mathit{irrelevant}(C, \mathsf{a}, t)$

$\{\ldots \,||\, \mathsf{a} := t' \,||\, \ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t$ dropUpdate$_3$
$\rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$, $\mathit{overwrites}(C, \mathit{frame}, \mathsf{a})$

$\{\ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}_1 \,||\, \ldots \,||\, \mathcal{U}_n\}t$ dropUpdate$_4$
$\rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \mathcal{U}_1 \,||\, \ldots \,||\, \mathcal{U}_n\}t$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$,
the left-hand side of update $\mathcal{U}_i$ has elements $\mathit{fr}^i_1, \ldots, \mathit{fr}^i_{k_i}$,
$\mathit{overwrites}(C, (\mathit{fr}^1_1, \mathit{fr}^1_2, \ldots, \mathit{fr}^1_{k_1}, \ldots, \mathit{fr}^n_{k_n}), \mathit{frame})$

$\{\ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}\}t \rightsquigarrow \{\ldots \,||\, \mathit{Skip} \,||\, \mathcal{U}\}t$ dropUpdate$_5$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$, $\mathit{irrelevant}(C, \mathit{frame}, t)$
and $\mathit{irrelevant}(C, \mathit{frame}, \mathcal{U})$

$\{\ldots \,||\, \mathcal{U}_\mathrm{P}(\ldots, \mathit{frame}_i, \ldots :\approx \mathit{footprint}) \,||\, \mathcal{U}\}t$ contraction
$\rightsquigarrow \{\ldots \,||\, \mathcal{U}_\mathrm{P}(\ldots, \_, \ldots :\approx \mathit{footprint}) \,||\, \mathcal{U}\}t$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$, $\mathit{irrelevant}(C, \mathit{frame}_i, t)$
and $\mathit{irrelevant}(C, \mathit{frame}_i, \mathcal{U})$

$\{\ldots \,||\, \mathcal{U}_\mathrm{Q}(\mathit{frame}' :\approx \mathit{footprint}') \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t$ reorderUpdate$_1$
$\rightsquigarrow \{\ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}_\mathrm{Q}(\mathit{frame}' :\approx \mathit{footprint}') \,||\, \ldots\}t$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$,
identifier "$\mathcal{U}_\mathrm{P}$" is lexicographically smaller than "$\mathcal{U}_\mathrm{Q}$",
$\mathit{irrelevant}(C, \mathit{frame}', \mathit{value}(\mathit{frame}))$

$\{\ldots \,||\, \mathsf{a} := t' \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \ldots\}t$ reorderUpdate$_2$
$\rightsquigarrow \{\ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathsf{a} := t' \,||\, \ldots\}t$
where $t \in \mathsf{Trm}_A \cup \mathsf{Fml} \cup \mathsf{Upd}$, $\mathit{irrelevant}(C, \mathsf{a}, \mathit{value}(\mathit{frame}))$

$\{\mathcal{U}\}\mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \rightsquigarrow \mathcal{U}_\mathrm{P}(\mathit{frame} :\approx (\{\mathcal{U}\}\mathit{footprint}))$ applyOnRigid$_9$

$\mathcal{U}_\mathrm{P}(\ldots, \mathtt{x}^!, \ldots :\approx \mathit{footprint})$ extractHasTo
$\rightsquigarrow \mathcal{U}_\mathrm{P}(\ldots, \_, \ldots :\approx \mathit{footprint}) \,||\, \mathtt{x} := f^\mathrm{P}_k(\mathit{footprint})$
where $\mathtt{x}^!$occurs at position k within the abstract update,
f^P_k is created *dependently fresh* for position k and the identifier $\mathcal{U}_\mathrm{P}$

Fig. 5 Abstract Update Simplification Rules

from a parallel update if there is a series of updates occurring later in the parallel scope that, together, overwrite *frame*. A simple case would be to replace $\mathcal{U}_\mathrm{P}(\mathtt{x}: \approx \mathit{footprint})$ by *Skip* in $\mathcal{U}_\mathrm{P}(\mathtt{x}: \approx \mathit{footprint}) \,||\, \mathtt{x} := t'$, but for more complicated *frame* expressions, it is not required that a *single* update overwrites all contained locations at once. The rule dropUpdate$_5$ corresponds to dropUpdate$_2$, handling the case of an (abstract) update that is dropped since the locations it assigns are irrelevant for the target term.

If only *some* of an abstract update's left-hand sides are ineffective and the rules dropUpdate$_4$ and dropUpdate$_5$ are not available, we have to perform a more fine-grained

simplification step than dropping the whole update. The formula

$$\{\mathcal{U}_{\mathbb{P}}(\mathtt{x},\ \mathtt{z}\colon \approx \mathit{footprint})\}\mathtt{z} \doteq \{\mathcal{U}_{\mathbb{P}}(\mathtt{y},\ \mathtt{z}\colon \approx \mathit{footprint})\}\mathtt{z}$$

is valid, but not provable with the rules discussed so far. In this situation, the rule contraction is applicable. It replaces ineffective parts of an abstract update's left-hand side with the "irrelevant" location "_." For the example above, this results in

$$\{\mathcal{U}_{\mathbb{P}}(\_,\ \mathtt{z}\colon \approx \mathit{footprint})\}\mathtt{z} \doteq \{\mathcal{U}_{\mathbb{P}}(\_,\ \mathtt{z}\colon \approx \mathit{footprint})\}\mathtt{z},$$

which is trivially provable. The symbol "_" receives special treatment in the definitions of the relations *irrelevant* and *overwrites* as it is always considered to be irrelevant or overwritten, respectively. An abstract update with *only* "_" left-hand sides can be dropped by rules $\mathsf{dropUpdate}_4$ and $\mathsf{dropUpdate}_5$ independently of their context.

In contrast to concrete updates, abstract updates cannot be applied to a target term by performing a simple substitution. Generally, some abstract update applications cannot be simplified away and remain in the final states resulting from symbolic execution. Especially thinking of correctness proofs of *transformations*, where we compare the execution result for two abstract programs, we need to establish a *normal form* to be able to compare the results. Consider, for example, the *Slide Statements* refactoring from the introduction. In the resulting symbolic store, the abstract updates occur in a different order and have to be normalized to show equivalence. This normal form is established by the rules $\mathsf{reorderUpdate}_1$ and $\mathsf{reorderUpdate}_2$. Abstract updates are moved to the front of a parallel update as long as this does not change the semantics. However, an abstract update may only be pushed past another abstract update if it has a *lexicographically smaller identifier* symbol. If there are no conflicts between the elementary abstract and concrete updates within a parallel update, it is normalized to a block of abstract updates ordered according to the lexicographic order of their identifiers, followed by a block of concrete elementary updates.

Even though it is generally impossible to apply abstract updates by performing a substitution in the target we can (1) apply updates *on* an abstract update, and (2) perform an effective simplification for a special case, namely for program variables marked as **\hasTo** in the left-hand side of an abstract update. The corresponding rules are $\mathsf{applyOnRigid}_9$ and extractHasTo. The rule $\mathsf{applyOnRigid}_9$ belongs to a class of simplification rules pushing update applications down into the term structure. It specifies that the application of an update $\mathcal{U}$ to an abstract update is equal to the abstract update with the same left-hand side, but $\mathcal{U}$ applied to the footprint. For situation (2), consider the formula $\{\mathcal{U}_{\mathbb{P}}(\mathtt{x}^{!}\colon \approx \mathit{footprint})\}\varphi$. Since the update *has to* change the value of x (based on value of the term *footprint*), the formula is equivalent to $\{\mathtt{x} := f(\mathit{footprint})\}\varphi$ for a suitably chosen function symbol f. "Suitably chosen," in this case, means that the function has to be chosen *dependently fresh* for the identifier symbol $\mathcal{U}_{\mathbb{P}}$ of the abstract update to conform to the semantics of abstract updates (Def. 12). This is generalized to abstract updates with multiple left-hand sides by using function symbols $f_k^{\mathbb{P}}$ indexed not only with the identifier, but also with the index k of the respective left-hand side. It is not always feasible to completely convert an abstract update into a concrete one, as in the following example:

$$\{\mathcal{U}_{\mathbb{P}}(\mathtt{z},\ \mathtt{x}^{!},\ \mathit{locset}\colon \approx \mathit{footprint})\}\varphi$$

The assignable program variable location "z" is not marked as **\hasTo**, and the abstract location set *locset* cannot be converted to a concrete update. Therefore, we extract **\hasTo** program variable locations individually and replace their positions in the left-hand side of the abstract update by the irrelevant location "_." Our simplification rule

extractHasTo incorporates these considerations. Applying extractHasTo to the example above yields $\{\mathcal{U}_{\mathbb{P}}(\mathtt{z}, \_, \mathit{locset}: \approx \mathit{footprint}) \,||\, \mathtt{x} := f_2^{\mathbb{P}}(\mathit{footprint})\}\varphi$.

5 Applications of Abstract Execution

Abstract Execution has been applied in a variety of different scenarios: (1) Deriving preconditions for the safe application of refactoring rules [76, 80], (2) analyzing the cost impact of program transformation rules [7], (3) the parallelization of sequential code [33], (4) "Correct-by-Construction" program development [89], (5) modular verification of software product lines with delta-oriented programming [66], and (6) the correctness of rule-based compilation [78]. Subsequently, we briefly overview all of these applications, discussing motivation, approach and results for each.

5.1 Safe Refactoring

Refactoring is the process of changing code in such a way that it does *not alter its external behavior*, yet improves its *internal structure* [22]. Careful refactoring can contribute to the maintainability and reusability of code. Consequently, many actions performed during software development are refactorings (ca. 30% as reported in [72]). While programmers still frequently manually refactor their code [57], most mainstream IDEs implement (semi-)automated refactoring techniques. [5]

Code refactoring is generally a complex activity, and it is easy to break the refactored code accidentally [20]. The reason is that refactoring techniques come with preconditions and constraints that have to be satisfied to ensure the preservation of program semantics. If those are violated, the resulting program may not compile, or—which is worse—compile, but expose a different behavior. IDEs automatically check some of these preconditions, but not all [74]: Even refactoring with tool support does not exclude the possibility of unexpected changes to a program's behavior [17, 73]. With the help of AE we could show in our work on correct refactoring [76, 79] that documentation of crucial preconditions in existing standard literature (e.g., [21, 22]) is vastly incomplete for many refactoring techniques.

We used AE to model nine statement-level refactorings. [6] We extracted sufficient preconditions ensuring their safety in a feedback loop driven by the interpretation of failed proof goals, ultimately leading to a proof certificate for these preconditions. We chose six refactorings from Fowler's original book [21] and three from the second edition [22], which includes two techniques with loops. For each technique, we created a model consisting of two abstract programs, one representing the starting point, and one the result of the refactoring. Our proof goal is behavioral *equivalence*; thus, we obtain preconditions for, e.g., *Extract Method* at the same time as for its inverse *Inline Method*. For *all* refactoring techniques, we discovered new preconditions that had not been mentioned in the literature. Subsequently, we explain how we created proof obligations for proving behavioral equivalence (Sect. 5.1.1), discuss how to prove transformations with loops (Sect. 5.1.2) and provide an overview of the discovered preconditions (Sect. 5.1.3), including a description of four bugs we discovered in the implementations of *Extract Method* in IntelliJ IDEA and Eclipse. For an extensive discussion including full models for all refactoring techniques, we refer to [76, Chapter 6].

[5] For the JVM ecosystem, for instance, IntelliJ IDEA, Eclipse and NetBeans together cover 92% of the IDE market [87]. All of these implement automatic refactoring techniques.

[6] Most practically applied refactorings are confined to method bodies [72]. However, existing work on the correctness of code refactoring almost exclusively regards high-level techniques such as "move field" or "pull up method." We thus focus on a significant blind spot by addressing statement-level transformations.

5.1.1 Proof Obligation for Behavioral Equivalence

A refactoring model includes (1) two *abstract programs* "*left*" and "*right*," (2) a *relational precondition* "*Pre*," (3) a set of *relevant locations* "*relevant*," and (4) a *postcondition* "$Post(s_1, s_2)$," where s_1 is a sequence consisting of a possibly returned value, a possibly thrown exception, and the values of the relevant locations for the *left* program, and similarly s_2 for the *right* program. From these constituents, we create a proof obligation collecting the outcomes of the left and right program in two uninterpreted predicates P and Q. As a default, we use a single abstract location set *relevant* for the relevant locations. Since this location set may represent *any* set of locations (unless the model imposes more specific constraints), both abstract programs have to coincide in their effects on the full program state. The resulting proof goal follows the syntactic pattern

$$\varphi_{left} \wedge \varphi_{right} \wedge Pre \wedge \cdots \vdash \exists\, Seq\ s_1, s_2;\ (P(s_1) \wedge Q(s_2) \wedge Post(s_1, s_2)) \quad , \tag{1}$$

where formulas φ_{left} and φ_{right} collect the results from executing the left and right programs in the predicates P and Q, respectively. Formulas φ_{left} and φ_{right} follow the pattern "$\{\mathcal{U}_{init}\}\neg\langle\cdots$`left()`$\cdots\rangle\neg P(\ldots)$" and "$\{\mathcal{U}_{init}\}\neg\langle\cdots$`right()`$\cdots\rangle\neg Q(\ldots)$", respectively. The double negations are a technical necessity to make use of the diamond modality (enforcing termination of the left and right programs) in proof assumptions. The formulas φ_{left} and φ_{right} are defined as

$$\begin{aligned}
\varphi_{left} := \{&\texttt{_result} := null \,||\, \texttt{_exc} := null\} \\
&\neg\langle \texttt{try \{ _result=obj.left()@Problem; \}} \\
&\quad \texttt{catch (Throwable t) \{ exc=t; \}}\rangle \\
&\quad\quad \neg P(\texttt{_result}, \texttt{_exc}, value(relevant))
\end{aligned}$$

$$\begin{aligned}
\varphi_{right} := \{&\texttt{_result} := null \,||\, \texttt{_exc} := null\} \\
&\neg\langle \texttt{try \{ _result=obj.right()@Problem; \}} \\
&\quad \texttt{catch (Throwable t) \{ exc=t; \}}\rangle \\
&\quad\quad \neg Q(\texttt{_result}, \texttt{_exc}, value(relevant))
\end{aligned}$$

where `Problem` is a Java class consisting of two methods `left` and `right` containing the *left* and *right* abstract program of our model, respectively. After the successful execution of both abstract programs, there will be exactly one instance of each of the uninterpreted predicates P and Q, which makes instantiating the existential quantifier in the succedent of this proof obligation trivial. This encoding is more efficient than the alternative of expressing the problem using an equivalence "$\leftrightarrow$," since the proof does not have to split, and, consequently, about 50% of the proof steps are saved.

Example 4 (Proof Obligations) We instantiate proof obligation schema (1) to the model from Example 3. Listing 4 shows the contents of the `left()` method except the "**`return`** ↳ **`null;`**" statement at the end. The `right()` method contains the ASs in reverse order. The precondition *Pre* is instantiated to the **`ae_constraint`** formula from Listing 4. We instantiate the postcondition $Post(s_1, s_2)$ to $s_1[2] \doteq s_2[2]$, where $s_i[2]$, the third component of sequence s_i, contains the final value of the abstract location set *relevant* after symbolic execution of `left()` and `right()`. We obtain:

$$\varphi_{left} \wedge \varphi_{right} \wedge (frameP \cap frameQ \doteq \emptyset) \wedge (frameP \cap footprintQ \doteq \emptyset) \wedge \cdots \vdash$$

$$\exists\, Seq\ s_1, s_2;\ \big(P(s_1) \wedge Q(s_2) \wedge s_1[2] \doteq s_2[2]\big)$$

The formulas φ_{left} and φ_{right} have exactly the shape given above, if `Problem` is the class that declares methods `left()` and `right()`. A proof of this obligation splits into multiple subgoals. For example, there is one case, where P throws an exception and Q returns. Subgoals of this kind are immediately discarded, because the precondition (part of the **`ae_constraint`**s in Listing 4) requires those cases to be mutually exclusive. We focus now on the case where both ASs complete normally. The resulting proof obligation has the following shape:

$$\begin{aligned}
&P(\mathbf{null}, \mathbf{null},\\
&\quad \{\mathcal{U}_{\mathrm{P}}(frameP\colon \approx footprintP)\}\\
&\qquad \{\mathcal{U}_{\mathrm{Q}}(frameQ\colon \approx footprintQ)\}value(relevant))\wedge\\
&Q(\mathbf{null}, \mathbf{null},\\
&\quad \{\mathcal{U}_{\mathrm{Q}}(frameQ\colon \approx footprintQ)\}\\
&\qquad \{\mathcal{U}_{\mathrm{P}}(frameP\colon \approx footprintP)\}value(relevant))\wedge\\
&(frameP \cap frameQ \doteq \emptyset) \wedge (frameP \cap footprintQ \doteq \emptyset) \wedge \cdots\\
\vdash\ &\exists\, Seq\ s_1, s_2;\ \big(P(s_1) \wedge Q(s_2) \wedge s_1[2] \doteq s_2[2]\big)
\end{aligned}$$

The predicates P and Q are uninterpreted, so the only promising way to instantiate existentially $P(s_1)$ and $Q(s_2)$ in the conclusion, is to use the arguments of P and Q's occurrence in the premise. After eliminating the quantifier in this way, the P and Q terms in the conclusion are identical to the instances in the premise and can be discharged. It remains to prove the instantiated postcondition $s_1[2] \doteq s_2[2]$:

$$\begin{aligned}
&\{\mathcal{U}_{\mathrm{P}}(frameP\colon \approx footprintP)\}\{\mathcal{U}_{\mathrm{Q}}(frameQ\colon \approx footprintQ)\}value(relevant) \doteq\\
&\quad \{\mathcal{U}_{\mathrm{Q}}(frameQ\colon \approx footprintQ)\}\{\mathcal{U}_{\mathrm{P}}(frameP\colon \approx footprintP)\}value(relevant)
\end{aligned}$$

The disjointness assumptions in the precondition (for example, $frameP \cap frameQ \doteq \emptyset$) permit the (abstract) update simplification rules to close this proof goal.

Our graphical workbench REFINITY [77] automates the construction of such proof obligations. We discuss REFINITY in Sect. 6.

5.1.2 Proving Transformations with Loops

Symbolic execution of loops requires advanced techniques: When loop guards are symbolic, we cannot know the number of iterations after which the loop will terminate. Frequently, *loop invariants* (see also Sect. 3), which are specifications respected by every loop iteration, are employed to abstract loop behavior regardless of the number of iterations. Finding good loop invariants is generally hard; indeed, coming up with sufficiently precise *specifications* was identified as the "new bottleneck" of formal verification (for example, [5]). In program equivalence proofs based on functional verification techniques, one even needs the *strongest* possible invariant for each occurring loop ([12], [76, Sec. 5.4.2]). This notwithstanding, we discovered a possibility to *generically* specify *abstract* strongest loop invariants for abstract programs.

Consider a loop with guard $g(\mathtt{x})$ operating on a single variable `x`. The formula $Inv(\mathtt{x})$ is a strongest loop invariant when it is (1) preserved by every run and (2) there is *exactly one* value v such that $Inv(v)$ holds and $g(v)$ does *not* hold. Condition (2) means that there remains

no degree of freedom in the choice of the value of x after loop termination: *Inv* describes the *exact*, final value. We can formalize the condition as $\exists v; \forall x; ((Inv(x) \wedge \neg g(x)) \leftrightarrow x = v)$.

Generalizing this to a loop with an abstract expression as guard and dynamic frame specification variables as frame and footprint yields a condition constraining instantiations of abstract invariant formulas to abstract *strongest* ones:

$$\exists \_fr, \_fp; \forall fr, fp; \\ \big((Inv(fr, fp) \wedge \neg guardIsTrue(fr, fp)) \leftrightarrow \big(\_fr \doteq fr \wedge \_fp \doteq fp\big)\big)$$

This assumes that *fr* and *fp* are the loop frame and footprint, and *guardIsTrue* is a predicate that holds if the loop guard evaluates to true. We add this as a global precondition and use *guardIsTrue* and *Inv* for the specifications inside our program.

As such, this only allows reasoning about normally completing loop bodies. Loop invariants are generally only required to hold before each further loop iteration; in particular, they need not hold after abrupt completion due to a **`break`** or **`return`**. By generalizing abstract strongest loop invariants to what we call *strongest* abstract strongest loop invariants—strongest loop invariants that also need to be respected after abrupt completion—we can also reason about abruptly completing loops. Assuming that *breaksBody* is a predicate that holds if, and only if, the abstract statement in the loop body will complete abruptly because of a **`break`**, our condition for abstract invariants *Inv* gets

$$\exists \_fr, \_fp; \quad \forall fr, fp; \\ \big(\big(Inv(fr, fp) \wedge \big(\neg guardIsTrue(fr, fp) \vee breaksBody(fp)\big)\big) \\ \leftrightarrow \big(\_fr \doteq fr \wedge \_fp \doteq fp\big) \big)$$

Listing 5 shows an example of a fully abstract loop with specifications for abrupt completion. In lines 1–7, the strongest abstract strongest loop invariant condition is stated. In lines 9–10 we assume the invariant initially. The postconditions for the abstract statement `Body` ensure that the invariant is preserved and bind the predicates for abrupt completion. Finally, *Inv* is used as a loop invariant in line 11.

Proving Validity of Instantiation for Models with Loops

Abstract loop invariants have a significant advantage over concrete ones: They are easy to come up with. In Listing 5, annotating a loop with a strongest abstract strongest invariant is a matter of introducing a fresh symbol (e.g., *Inv*) and using it at suitable positions. This approach is a *generic* recipe that can be applied to various models.

But, of course, there is no free lunch: The complexity avoided by using *generic* abstract functional invariants, instead of concrete coupling invariants [12], returns when one wants to *prove* that a given concrete program with loops is a valid *instance* of a given abstract program. In this case, not only one needs to discover a meaningful invariant for the loop of the concrete program, which is difficult enough, but one has to discover the *strongest* loop invariant to serve as an instance of its abstract counterpart. Even though strongest invariants always exist for deterministic programs, they might be impossible to specify in a given specification language (for example, JML).

On the other hand, for many applications of AE it is sufficient to stay at the abstract level and avoid concrete strongest loop invariants altogether: For example, we show in the subsequent Sect. 5.1.3 that the mechanics for the *Remove Control Flag* refactoring as described in Fowler's book [21] likely *yields incorrect results*. Furthermore, we show how this can be mitigated. Such insights for *schematic* transformations can be obtained without ever considering concrete instances.

Listing 5 Abstract Strongest Loop Invariant with Abrupt Completion

```
/*@ ae_constraint
  @   (\exists any _fr,_fp; (\forall any fr,fp; ((
  @         Inv(fr, fp) &&
  @         (   !guardIsTrue(fr, fp) || throwsExcBody(fp)
  @          || returnsBody(fp) || breaksBody(fp))
  @       ) <==> (fr == _fr && fp == _fp))
  @   )); */

/*@ ae_constraint
  @   Inv(\value(loopFrame), \value(loopFootprint)); */
/*@ loop_invariant Inv(\value(loopFrame), \value(loopFootprint));
  @ assignable loopFrame;
  @*/
while (
    /*@ assignable \nothing;
      @ accessible loopFrame, loopFootprint;
      @ normal_behavior ensures \result <==>
      @   guardIsTrue(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior requires false; */
    \abstract_expression boolean e
) {
    /*@ assignable loopFrame;
      @ accessible loopFootprint;
      @ normal_behavior ensures
      @   Inv(\value(loopFrame), \value(loopFootprint));
      @ exceptional_behavior ensures
      @   throwsExcBody(\value(loopFootprint)) &&
      @   Inv(\value(loopFrame), \value(loopFootprint));
      @ return_behavior ensures
      @   returnsBody(\value(loopFootprint)) &&
      @   Inv(\value(loopFrame), \value(loopFootprint));
      @ break_behavior ensures
      @   breaksBody(\value(loopFootprint)) &&
      @   Inv(\value(loopFrame), \value(loopFootprint));
      @ continue_behavior requires ensures
      @   Inv(\value(loopFrame), \value(loopFootprint)); */
    \abstract_statement Body;
}
```

5.1.3 Preconditions for Safe Refactoring

Slide Statements

The idea behind the *Slide Statements* refactoring technique (see technique (see 1.2) is to reorder statements to keep those together that have a common purpose [22]. Under the assumption that both statements complete normally, the preconditions documented in [22] are complete: Neither statement must write to the locations read by the other once, and they must also not write to the same locations. However, no preconditions are mentioned for *abrupt* completion. We inferred in addition that (1) at most one of `A` and `B` may complete abruptly in any state and (2) if one statement completes abruptly, the other one must not write to the *relevant* state.

The *Consolidate Duplicate Conditional Fragments* [21] technique is a special case of *Slide Statements* for moving common statements in all branches of an **`if`** or **`try`** statement to before or after that statement. For extracting a prefix of an **`if`**, the same preconditions as for *Slide Statements* apply. Extracting a postfix from an **`if`** comes without preconditions.

For **`try`** statements, the moved postfix must not throw an exception or access the caught exception object. If the postfix is moved to a **`finally`** block, the remaining statement in the **`try`** block must not return.

Slide Statements is implemented as a lightweight refactoring in IntelliJ IDEA 2021.1 (Eclipse 4.19 does not support it). The IntelliJ implementation permits to move single statements (not separated by a "`;`") one position up or down. No preconditions are checked, which makes it easy to, for example, move a variable occurrence to a position before its definition. We did not file a bug report for IntelliJ since we formed the impression that the lightweight realization does not aim for correct results under all circumstances.

Consolidate Conditional Expression

For the case of sequential or nested conditionals with "the same result" [21], this refactoring proposes to merge these conditionals into a single check to improve clarity. There are two variants of this technique: (1) Transforming a sequence of **`if`** statements to a single one with a disjunction as the guard, and (2) transforming a nested **`if`** statement to a single one with a conjunction as the guard. Schematically:

$$\begin{array}{lllllllllllllllll}
(1) & \texttt{if} & (e_1) & \{ & P & \} & \texttt{if} & (e_2) & \{ & P & \} & \rightsquigarrow & \texttt{if} & (e_1 & \texttt{||} & e_2) & \{ & P & \} \\
(2) & \texttt{if} & (e_1) & \{ & & & \texttt{if} & (e_2) & \{ & P & \} & \rightsquigarrow & \texttt{if} & (e_1 & \texttt{\&\&} & e_2) & \{ & P & \}
\end{array}$$

The crucial part of modeling this refactoring is the interpretation of having "the same result." In our opinion, supported by the examples supplied in [21], P should *always* either return or throw an exception: it is never executed twice. Our analysis results in the conclusion that under this assumption, both variants of the refactoring can be applied *without additional preconditions*. This is notable, since Fowler mentions that conditionals *must not have any side effects*, which is, however, only necessary if one uses Boolean connectives without short-circuit evaluation ("`|`" and "`&`"). In the case of variant (2), P can furthermore complete arbitrarily (i.e., also normally).

Extract Method, Decompose Conditional, and Move Statements to Callers

Method extraction is a well-known refactoring technique implemented in many IDEs. Fowler [21] names as preconditions that the extracted code may not assign more than one local variable referenced in the outside context. [7] We discovered two *additional* constraints: (1) The extracted fragment must not return since this changes the control flow. (2) If the extracted method assigns a local variable from the outside context, then it must not throw an exception after that variable has been assigned a value. For the latter additional precondition, consider the example in Fig. 6. For an empty `intList`, the division in line 4 completes abruptly because of a 0 divisor. Then, the final value of `avg` is 0 before, but `ERROR` after the transformation. While this can even be considered an improvement for the example scenario, it changes the program's semantics; besides, in the reverse direction corresponding to the *Inline Method* refactoring, one would introduce a more obvious bug. *Decompose Conditional* and *Move Statements to Callers* are variants of *Extract Method.* We modeled and verified these, too. There were no additional insights compared to *Extract Method.*

Discovered Bugs for Extract Method in IntelliJ and Eclipse

Considering the implementation in IDEs, Eclipse 4.19 issues a warning when extracting a fragment containing a **`return`** statement, while IntelliJ IDEA 2021.1 tries to work around

[7] Assignment to more than one local variable requires to return (and decompose at the caller side) a complex object, such as an array that holds the changed locals. This is possible, but it is not covered by the refactoring technique considered here.

```
1  int avg = ERROR;
2  try {
3    avg = sumOfElems(intList);
4    avg /= intList.size();
5  } catch (ArithmeticException ae) {}
6  averages.add(avg);
```

⇝

```
int avg = ERROR;
try {
  avg = average(intList);
} catch (ArithmeticException ae) {}
averages.add(avg);
```

Fig. 6 Wrong Extraction of a Query Method

Listing 6: Input Program

```
int m(int input) {
    if (input == 0) {
        return 1;
    } else if (input == 1) {
        return 0;
    }
    return input;
}
```

Listing 7: Output Program

```
int m(int input) {
    return extracted(input);
    return input; // <- unreachable
}

private int extracted(int input) {
    if (input == 0) {
        return 1;
    } else if (input == 1) {
        return 0;
    } // <- missing return
}
```

Fig. 7 Wrong Application of "*Extract Method*" by IntelliJ IDEA

this problem: The extracted method returns null if no return occurred, which is checked at the call site. If a non-null value is returned, the caller returns that value; otherwise, the caller proceeds normally. We suspected that this approach by IntelliJ will not produce correct results for all input programs, and indeed quickly discovered a counterexample where IntelliJ produced *uncompilable code without showing a previous warning*. Consider the method `m` in Listing 6 of Fig. 7. Applying *Extract Method* in IntelliJ IDEA 2021.1 to the highlighted lines yields the code in Listing 7. This code does not compile since the second **`return`** statement in the refactored version of `m` becomes unreachable, and, furthermore, the method `extracted` misses a **`return`** statement. Getting around this issue is not trivial. One option is to return a null value at the end of `extracted` and return from `m` if a non-null value was returned or resuming execution in `m` otherwise (this is implemented in IntelliJ for different situations, for example, **`if`** statements without **`else`** branches). It is, however, not *the* ultimate solution, for instance, when the extracted fragment returns non-trivial types. We reported a bug to the IntelliJ developers, who fixed the problem in later IDEA versions. [8]

Precondition (2) in the previous paragraph is unchecked by either IDE for the *Extract Method* direction. However, both IntelliJ and Eclipse produce a correct result for the inverse *Inline Method* direction by replacing `avg` with a temporary variable `avg1` that is only assigned to `avg` at the end of the inlined method body. For the *Extract Method* direction, we figured out a workaround for the problem related to exceptions and suggested it in a bug reported to the IntelliJ developers. [9]

Eclipse allows factoring out **`break`** and **`continue`** statements from within loops, which immediately yields uncompilable code. We reported this bug to the Eclipse community. [10]

[8] "**IDEA-271736** 'Extract Method' of 'if-else if' fragment with multiple returns yields uncompilable code in IDEA 2021.01," https://youtrack.jetbrains.com/issue/IDEA-271736, "major" priority. This bug report was eventually classified as a duplicate of our report **IDEA-271801** (see below), which was fixed in build 213.1344.

[9] "**IDEA-271752** 'Extract Method' yields semantically incorrect result in presence of runtime exceptions (can be fixed!)," https://youtrack.jetbrains.com/issue/IDEA-271752, "normal" priority, fixed (with an unspecified build number).

[10] "**Bug 574254** 'Extract Method' allows extraction of break and continue statements, yielding uncompilable results," https://bugs.eclipse.org/bugs/show_bug.cgi?id=574254, "major" priority, fixed in 4.21 M3.

IntelliJ is more considerate in handling **`break`** and **`continue`** statements. In some cases, it produces correct results. However, it is still easy to come up with examples where IntelliJ produces uncompilable code or semantically incorrect results when factoring out conditionals containing **`break`**s or **`continue`**s from within loops. Furthermore, the implementation is inconsistent and produces correct or wrong results for the same input depending on the surrounding code in the class. We filed another bug report for this issue. [11]

Decompose Conditional [21] is a variant where condition and both branches of an **`if`** statement are extracted to individual methods. For the branches, this is identical to *Extract Method*; there is no precondition for the extracted *condition*.

Move Statements to Callers [22] is a variant of *Inline Method* where a prefix (and not the whole body) of a method is moved to the callers. Conversely, *Move Statements into Method* moves statements before an invocation to inside the called method. The same restrictions as for *Extract Method* / *Inline Method* apply.

Replace Exception with Test

In our example in Fig. 6, we anticipated that a division by zero would raise an `ArithmeticException` and used a **`try`** statement to react accordingly. Fowler motivates the *Replace Exception with Test* refactoring [21] by declaring this procedure a code smell that should be avoided. Rather, we should *check* the problematic condition in case of the example `intList.size() == 0` *beforehand* in an **`if`**/**`else`** statement replacing the **`try`**/**`catch`**. Note that then, method extraction in Fig. 6 would have been safe. Yet, this also demonstrates that *Replace Exception with Test* is generally unsafe, even though no preconditions are mentioned in [21]. Schematically, this refactoring can be written as

$$\texttt{try}\ \{\ P\ \}\ \texttt{catch}\ (\ldots)\ \{\ Q\ \} \rightsquigarrow \texttt{if}\ (!\mathit{cond})\ \{\ P\ \}\ \texttt{else}\ \{\ Q\ \}$$

where *cond* is the condition under which P will throw an exception. The general problem with this refactoring is that whenever P throws an exception, it might have changed the relevant state *before* completing abruptly. After the refactoring, P is not executed at all. The refactoring *can* be safely applied if P neither assigns any "relevant" location nor the locations assigned by Q, or if both P and Q do not assign any relevant location, and, furthermore, Q always completes normally.

Since these situations are unlikely in practice, we came up with a workaround that always ensures the safety of the refactoring technique: If Q contains a prefix *resetting* all locations assigned by P to default values that are independent of P's assignments, then intermediate changes by `P` are neutralized, resulting in the same effect before and after the refactoring. This situation can always be achieved by adding suitable reset statements to the **`catch`** clause directly before Q.

To the best of our knowledge, *Replace Exception with Test* is not implemented in any major Java IDE.

Split Loop

Splitting a loop, where this is possible, contributes to readability by dividing loops with separate concerns. It can also make sense to split a loop to prepare for code parallelization (see Sect. 5.3). As we showed in our work on the cost impact of transformations (Sect. 5.2), the performance impact of this transformation is minor. This might be counterintuitive, but

[11] "**IDEA-271801** 'Extract Method' of conditional with break inside loop yields semantically incorrect OR UNCOMPILABLE result in some cases," https://youtrack.jetbrains.com/issue/IDEA-271801, "major" priority, fixed in build 213.1344.

the only overhead introduced by dividing a loop is the double evaluation of the loop guard, which is usually insignificant. The schematic representation of *Split Loop* is

```
Init while (g) { Upd P Q } ⇝
           Init while (g) { Upd P }
           Init while (g) { Upd Q }
```

We extracted the following preconditions: (1) The guard g must not have any side effects and P, Q must not write to the footprint of g, (2) the initialization statement *Init* and the loop update statement *Upd* must write (initialize/update) g's footprint and must complete normally, *Init*'s footprint must be empty, and *Upd*'s footprint equals g's footprint, (3) the frames and footprints of P and Q must be independent in the sense of *Slide Statements*, i.e., not overwrite each other and not influence each other's evaluations, (4) P must not complete abruptly, (5) Q must not complete abruptly before P committed its final result (i.e., established its invariant). All these preconditions are undocumented in [21, 22].

Observe that loops over an iterator (with a guard like "`it.hasNext()`") do not satisfy these preconditions: a call to "`it.next()`" in P or Q changes the state on which the evaluation of g depends, which is not allowed due to condition 5.1.3. Therefore, it is unsafe to apply *Split Loop* to such loops.

Remove Control Flag

A "control flag" in a loop determines when the loop should terminate. The *Remove Control Flag* refactoring [21] suggests to resort to **`break`** or **`continue`** statements instead to better communicate the control flow. Schematically:

```
    while (!done && g) { if (cond) { P done=true; } Q }
⇝   while (          g) { if (cond) { P break;     } Q }
```

We found that the shortcut introduced by the abrupt completion, however, generally breaks semantic equivalence. Any code that would have been executed after setting the control flag (Q in the schema) is now skipped by the shortcut, and must thus not have effects visible outside of the loop. Otherwise, we have to duplicate Q:

```
    while (!done && g) { if (cond) { P     done=true; } Q }
⇝   while (          g) { if (cond) { P Q   break;     } Q }
```

Following the mechanics described in [21] likely yields incorrect results.

For the proofs of *Split Loop* and *Remove Control Flag*, we used abstract strongest abstract loop invariants, as described in Sect. 5.1.2.

Discussion: Inadequate Refactoring Support in IDEs

Most statement-level refactoring techniques discussed by Fowler [21, 22] are not supported in mainstream IDEs. Indeed, only variable or method extraction or inlining are implemented in IntelliJ IDEA and Eclipse, apart from renaming or class-level refactorings. We consider this problematic, as automated code refactoring can prevent many coding errors. In addition, even an *imperfect* implementation is a step forward, enabling users to submit bug reports and gradually improve the implementation.

5.2 Performance Impact of Transformations

Refactoring, as described above, changes a program without affecting its *functionality* to optimize "soft" properties, such as readability or maintainability. When refactoring, programmers

Listing 8: Input Model

```
int i = 0;

//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
    i · (acP (t, w) + acQ (t, z) + 2);
//@ decreases t − i;
while (i < t) {
    //@ assignable x;
    //@ accessible t,w;
    //@ cost_footprint t,w;
    \abstract_statement P;
    //@ assignable y;
    //@ accessible i,t,y,z;
    //@ cost_footprint t,z;
    \abstract_statement Q;
    i++;
}
//@ assert \cost == 2 +
    t · (acP (t, w) + acQ (t, z) + 2);
```

Listing 9: Output Model

```
int i = 0;
//@ assignable x;
//@ accessible t,w;
//@ cost_footprint t,w;
\abstract_statement P;
//@ loop_invariant i ≥ 0 && i ≤ t;
//@ cost_invariant
    i · (acQ (t, z) + 2);
//@ decreases t − i;
while (i < t) {
    //@ assignable y;
    //@ accessible i,t,y,z;
    //@ cost_footprint t,z;
    \abstract_statement Q;
    i++;
}
//@ assert \cost == 2 +
    acP (t, w) + t · (acQ (t, z) + 2);
```

Fig. 8 specification lines are model for *Code Motion*. Lighter gray specification lines are manually annotated, darker gray gray lines are inferred automatically

are encouraged to disregard the manipulated program's performance (e.g., runtime), this being an orthogonal aspect. Still, it is certainly worthwhile to know that, for example, *Split Loop* does not affect performance up to a constant factor.

In [7], we apply *Quantitative* Abstract Execution (QAE) to statically derive the *cost effect of program transformation schemas*. In contrast to existing relational cost analysis approaches (e.g., [63, 64]) that work by first *applying* a transformation—say, *Split Loop*—and analyzing performance *after the fact*, QAE obtains *abstract* cost bounds of transformations before they are even applied. Using these bounds, one can reason about the cost effect of a transformation in general, or obtain a concrete cost effect for a given target program by instantiating an abstract bound for that program.

Compared to standard AE models, their quantitative counterparts contain specifications of accessible locations of APE that are relevant for its execution cost, the so-called *cost footprint*. The total execution cost of an abstract program is tracked in a special variable **`\cost`** (cf. the **`\result`** variable for a method's returned result). Loop invariants are split into functional invariants, which, for the purpose of cost analysis, only need to be strong enough to prove termination (the **`decreases`** clause), and *cost invariants* capturing the value of the **`\cost`** variable throughout loop execution.

Fig. 8 shows the QAE models for *Code Motion*, a standard optimization implemented in compilers [4], where a loop-invariant statement is moved to before a loop. QAE allows to prove that the execution cost is not increased by the transformation (and *decreased* if the cost of AS `P` is nonzero and the loop is executed at least once).

Frame, footprint, and cost footprint of the involved ASs are manually specified; the loop invariants and decreases terms are automatically inferred. The QAE toolchain consists of a *cost analyzer*, which infers these additional annotations, and a *verifier*, which proves them correct. The approach is parametric in a *cost model*: In particular, one can analyze (abstract) execution time and memory consumption in the heap. Thanks to the automatic inference of loop invariants and optimized proof strategies, the whole inference and certification process

works *fully automatically* and does not require auxiliary specifications. The analysis results in abstract cost bounds parametric in the execution cost of the involved APEs, and a proof certificate for their correctness.

QAE has been evaluated with seven optimization techniques, comprising, e.g., *Split Loop* and *Loop Tiling*. All models contain loops, for which the cost analyzer was able to automatically infer sufficiently strong invariants and postconditions.

5.3 Safe Code Parallelization

Parallelization of sequential code is one the most important approaches, sometimes the only available one, to improve time performance. For this reason, code parallelization is one of the central research topics in the area of high-performance computing (HPC). In this community, design patterns are an established and powerful method to parallelize sequential programs [35, 51]. It makes sense to start with *sequential* programs, because these already serve their intended purpose, also one avoids loss of domain knowledge, documentation, and previous investments. In addition, *patterns* embody best practices, as well as correct and efficient usage of parallelization interfaces—knowledge that many programmers lack. Therefore, a pattern-based approach to parallelization constitutes a safe, efficient and even semi-automatic [60] migration path from sequential to parallel code.

Unfortunately, pattern-based parallelization suffers from a severe practical limitation: sequential legacy code typically does not exactly have the form that allows immediate application of a pattern. Hence a certain amount of *code restructuring* is unavoidable in most cases before pattern-based parallelization becomes applicable. The DiscoPoP parallelization framework [60] developed a small number of code transformation schemata [52] that in many cases are sufficient to bring sequential code into the form required for pattern-based parallelization to succeed, i.e. these restructuring schemata *prepare* code for parallelization, but they still work on *sequential* code.

Consider, for example, the `for`-loop in Listing 10, where $stmt_2$ depends on the result of $stmt_1$ and $stmt_1$ depends on the result of $stmt_3$ (across iterations). At first sight, the code might seem not parallelizable because of a forward-dependency among loop iterations. However, an astute programmer might find a case where it is possible to successfully parallelize the code by just reordering the statements, placing $stmt_3$ before $stmt_1$, as depicted in Listing 11. Such a transformation preserves the semantics of the original code and makes it parallelizable using the pipeline pattern that achieves functional parallelism, similar to an assembly line. The pipeline consists of various stages that process data concurrently as it passes through the pipeline. It can be used to parallelize the body of a loop if the loop cannot be parallelized in the conventional way by simply dividing the iteration space (the *do-all* pattern). The pipeline pattern assigns each computational unit to a processor and provides a mechanism for passing on data elements to the next unit. Then it runs the computational units in parallel. In the example, the execution of different loop iterations can overlap as long as $stmt_3$ is completed in iteration i before $stmt_1$ and $stmt_2$ start in iteration $i + 1$. This was not possible before because $stmt_3$ came last.

The code transformation required to make the pipeline pattern applicable is called *Computational Unit Repositioning* in [52]. It is an instance of the *Slide Statement* refactoring technique mentioned in Sect. 5.1.3. Consequently, the preconditions for a safe application of the transformation are the same as for *Slide Statements*: Neither statement depends on the output of the other one, and cannot overwrite state changes of the other. The fully automatic proof in REFINITY for this simple transformation technique consists of ca. 1,000 nodes and takes less than 7 seconds.

Listing 10: Original code

```
for (i=0; i<n-1; i++) {
  a[i]   = b[i]; // → stmt1
  c[i]   = a[i]; // → stmt2
  b[i+1] = b[i]; // → stmt3
}
```

Listing 11: After restructuring

```
for (i=0; i<n-1; i++) {
  b[i+1] = b[i]; // → stmt3
  a[i]   = b[i]; // → stmt1
  c[i]   = a[i]; // → stmt2
}
```

Fig. 9 A rudimentary sample code parallelizable using the pipeline pattern

In [33] we formalized and verified in addition two more complex transformation schemata from [52] called *Loop Splitting* and *Geometric Decomposition*. These proofs were not fully automatic and required a number of user interactions. The formalization also required the concept of strongest loop invariant introduced in Sect. 5.1.2.

In addition, it was necessary to extend the AE framework with the possibility to specify and reason about *families* of abstract location sets. This extension allows for versatile specifications, but is a challenge for the prover. Still, we reached a degree of automation of 99.7% even for the most complex loop transformation.

Our formal models cover not merely necessary criteria for proving the correctness of (sequential) program transformations, but also stronger constraints for the subsequent addition of parallelization directives. Crucial preconditions on memory access should be automatically checkable by parallelization tools, or can at least be closely approximated. By precisely stating these requirements explicitly, we hope we cleared the way to safer parallelization.

One obvious future work direction is the generalization of AE to parallel programs. This would allow us to go one step further: To mechanically prove that the constraints in our models are sufficiently strong to ensure the preservation of the sequential program semantics *after* parallelization.

5.4 Correctness-by-Construction

A posteriori verification (also called *post hoc verification*) designates the approach to deductive verification, whereby a program is verified *after* it has been constructed and, possibly, even deployed. This is by far the most common approach today, even though it is acknowledged that developing specifications post hoc for a program that was not designed with verification in mind makes the task of specification and verification considerably harder than necessary [11, 30].

Interestingly, early work in formal software development often argued for a different, a *constructive* approach, often termed *correctness-by-construction* [19, 29, 90]. Its starting point is not the program code, but a mathematical formalization of a program's intended behavior (a specification), from which, in a series of refinement steps, a correctness proof together with executable code is gradually developed. The success of the B method [1, 2] notwithstanding, correctness-by-construction was never realized for an industrial programming language. One of the problems was the lack of tool support, but in the past years there has been a renewed interest [44, 45].

In traditional correctness-by-construction the refinement rules are directly derived from the axiomatic program semantics. For example, Hoare's rule for sequential composition [34]

$$\frac{\{P\}S_1\{I\} \quad \{I\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

ifElseTransl

$$\frac{(C \cup \{\mathtt{b} \doteq \mathrm{TRUE}\}, \mathcal{U}, \pi\ P_1\ \omega \dashv\vdash (q \lhd_n P_2)^{(n)})@(obs_1) \quad (C \cup \{\mathtt{b} \doteq \mathrm{FALSE}\}, \mathcal{U}, \pi\ P_1'\ \omega \dashv\vdash (q \lhd_n P_2')^{(n)})@(obs_2)}{(C, \mathcal{U}, \pi \begin{array}{c}\texttt{if (b)}\\ P_1\\ \texttt{else}\\ P_1'\end{array} \omega \dashv\vdash \left(q \lhd_n \left(\left(\begin{array}{l}\texttt{\%1 = load i1, i1* \%b}\\ \texttt{br i1 \%1, label \%2,}\\ \texttt{\qquad label \%3}\\ \texttt{br label \%4;<label>:\%2}\\ \texttt{br label \%4;<label>:\%3}\\ \texttt{\qquad ;<label>:\%4}\end{array}\right) \lhd_3 P_2' \lhd_2 P_2\right)\right)^{(n)})@(obs_1 \cup obs_2)}$$

Fig. 10 Dual SE rule for translating an `if` statement to LLVM IR

gives rise to a refinement rule, whereby a specification $\{P\}S\{Q\}$ of an as yet unknown program S is refined into a program in the shape of a sequential composition. Its specification is extended with an intermediate assertion I that must hold in between S_1 and S_2. There are at two main issues with this refinement approach: (1) the number of refinement rules is small and fixed, which results in a large number of fine-grained refinement steps; (2) refinement rules for complex language constructs (for example, aliasing, exception handling, dynamic dispatch, etc.) are complicated and need to be proven correct.

Based on the observation that S, S_1, and S_2 are APEs, it is possible to re-formulate a refinement rule as a program transformation rule (Sect. 5.1) in the AE framework. Therefore, AE has the potential to overcome the mentioned limitations: since AE was developed in the context of a deductive verification tool for Java, complex language constructs are supported, specifically, exceptions and framing. A tool such as REFINITY (Sect. 6) lets one prove problem-specific refinement rules on the fly and could even provide immediate feedback on incorrect refinement attempts. A first proof-of-concept has been given in [89], but the full potential of AE for correctness-by-construction remains to be explored.

5.5 Sound Rule-Based Compilation

The principles underlying Abstract Execution, that is, the transformation of APEs to abstract updates and distinguishing different completion modes in different symbolic execution branches, do not only apply to Java. If we permit abstract programs in the source *and target* languages of a compiler, we can reason about the compiler's correctness. We investigated this idea based on the example of a rule-based compiler from Java to LLVM IR [78]. The compiler consists of translation rules for each syntactic element of the Java source language. We express these rules as schematic SE rules, only that we use *dual* SE states over program *pairs*. A dual SE state has the shape $(C, \mathcal{U}, p_1 \dashv\vdash p_2)@(obs)$, where C and $\mathcal{U}$ are a path condition and an update, respectively, p_1, p_2 are programs in the source and target language of the rule-based compiler, and *obs* is a set of *observable locations*. Intuitively, a dual SE state expresses the judgment that executing the two programs p_i in the state determined by C and $\mathcal{U}$ has the same effect on the locations in *obs*. We restrict the set of locations to the "observable" ones to enable the introduction of intermediate assignments to registers by the compiler.

Based on this formalism, we can express the compilation of a Java `if` statement to LLVM IR as the dual SE rule depicted in Fig. 10. In the rule, P_1 and P_1' are abstract Java statements, and P_2 and P_2' are abstract LLVM IR statements. An LLVM IR statement $p \lhd_n q$ is the statement arising from inserting the statement q into the statement p at position n such that in the resulting statement, the temporary registers `%1`, `%2`, etc., are assigned in sequential order as required by LLVM IR.

The attractive feature of this AE-based compiler is that we can *automatically* reason about the correctness of the translation rules. To that end, we define the *validity* of a dual SE state $(C, \mathcal{U}, p_1 \dashv\vdash p_2)@(obs)$ as the validity of the following *justifying formula*:

$$\{\mathcal{U}\}\Big(\bigwedge C\Big) \rightarrow \big(\{\mathcal{U}\}[p](obs') \leftrightarrow \{\mathcal{U}\}[q]^{(n)}(obs')\big),$$

As usual in sequent calculi, a dual SE rule is sound if we can conclude the conclusion's validity from the premises' validity. In other words, we can prove the correctness of the compiler by discharging proof obligations for each translation rule in our AE framework. As a trust anchor, we assume that the implementations of AE for the source and target languages are sound. This requirement corresponds to formalizing the semantics for those languages in proven-correct compilers written (and proved) in interactive proof assistants (e.g., the CompCert [48], Jinja [42], or CakeML [81] verified compilers). The difference is that our proofs are highly automated as we can rely on the AE framework. In contrast, interactive proofs require substantial manual work: The CompCert code, for example, consists of 44% proof scripts [48].

5.6 Verification of Software Product Lines

A *software product line* (SPL, also called product family) [62] is a set of related programs, so-called *product variants* that exhibit *commonality* as well as *variability*. Commonality typically manifests itself in terms of common core functionality and a code base shared by all variants. Variability derives from the need to support a possibly large number of different feature combinations (or *products*). The main argument for family-based software development is the possibility to factor out the commonalities and thus avoid having to develop and maintain a large number of variants in isolation. It is also much faster to realize a new product as a variant than developing it from scratch. Family-based development is most productive in variant-rich application areas, such as consumer products, embedded (IoT) devices, but also operating systems.

Regarding analysis, specification, and verification of software, the quest to lift single product-based approaches to feature- and family-oriented [83] approaches resulted in various proposals [31, 32, 43, 84–86]. The fundamental design space of SPL verification is demarcated by two extremes: In the ideal scenario for *feature-oriented* verification, one verifies the core code and family-specific code separately for each feature. A suitable composition mechanism then guarantees the correctness of each valid variant. The main drawback is that compositionality requires serious constraints on the admissibility of contracts and feature implementation. For example, Hähnle & Schaefer [31] proposed an adaptation of *Liskov's Substitution Principle* (LSP) [49] in the context of delta-oriented programming [67]. Further contract composition principles are discussed by Thüm et al. [84]. The problem with constraints on contract admissibility is that it often imposes too severe restrictions on software design to be of practical use. On the other end of the design space lies *product-based* verification, where each valid product is specified and verified in isolation. This is usually prohibitive in cost, particularly, with respect to specification [11]. Besides, it excludes systematic reuse, the purported main advantage of software product lines.

It turns out that AE is the basis of an interesting trade-off [66] between the two extremes in terms of a fully compositional verification approach without too restrictive constraints that would render it impractical. The underlying variability principle is delta-oriented programming (DOP) [67], where each feature is implemented by one or more delta modules (deltas, for short) that are applied successively to a core variant. In DOP one specifies incremental code transformations at the granularity of a method declaration with code *deltas*. This aligns

with contract-based specification. Hence, each modification of a method in a delta is assumed to be specified with a contract. Moreover, a delta for a method can be declared relative to a previous version of that method that is called using the keyword `original` in the delta's code.

The new idea of [66], compared to the cited earlier work, is to impose relatively liberal constraints on deltas and contracts that *permit overriding* of behavior and are not compositional in the general case. Compositionality is regained by imposing a *normal form* on the code declared in a delta. The obvious drawback is that legacy code generally does not follow this normal form, but [66] showed that a small number of *behavior-preserving* program transformations are sufficient to achieve normal form in practice for several case studies involving legacy code (only in one instance limited remodeling was needed). It was possible to use the *Safe Refactoring* approach described in Sect. 5.1 with minor modifications.

Because the LSP is broken by overriding, unlike in [31, 84], correctness of calls to `original` is no longer guaranteed by a general composition principle (plus the correctness of `original`-free methods). In addition, the correctness of contracts with calls to `original` must be established with the help of the constraints implied by normal form. However, since calls to `original` can be seen as an AS, this can be directly modeled and proven with AE. For the case studies in [66] all necessary transformation schemata and contracts were proven fully automatic.

6 Implementation

We implemented Abstract Execution on top of the SE engine of the deductive program verifier KeY [5]. Deviating from the uniform representation in Sect 4, there exist dedicated AS execution rules tailored to different *contexts*. For instance, if AS is executed outside of a loop, **`continue`**s and unlabeled **`break`**s are omitted. This saves one having to exclude behavior that cannot occur explicitly.

AE Rules as Taclets

Most SE rules in KeY are implemented as so-called *taclets* [65]. Taclets provide a textbook-like notation for schematic sequent calculus rules with side conditions and, in particular, for KeY's Java DL calculus. The taclet syntax is easy to learn and even easier to read. The taclet semantics is formally defined relative to possible external application-specific conditions and transformers. *Derivable* taclets, not part of the axiomatic basis of Java DL, can be *automatically* proven correct within KeY itself [14].

Except few complex SE rules requiring complex program transformations implemented directly in Java, all of KeY's Java DL rules are expressed in taclet notation. To extend rule coverage as much as possible, the taclet language offers two extension points: *variable conditions* to answer complex queries about the proof environment and to initialize custom data structures, as well as *transformers* to create terms and program elements or perform complex transformations on existing ones. An implementation of a rule set based solely on extended taclets imposes certain overhead compared to a pure Java implementation, because of a necessarily more fine-grained decomposition into conditions and transformers, as well as a higher amount of parsing. Extension points also hide part of taclet semantics and hinder fully automatic proofs of derived taclets. Nevertheless, we decided to implement the abstract execution rules as extended taclets, because the advantages far outweigh the problems: (1) A taclet specification, even with extensions backed by custom Java code, is still less opaque and better maintainable than a pure Java implementation. (2) The AE rules are considered to be *axioms*, which is why they are anyhow not derivable.

We defined four rules for ASs (for different contexts) and one for AExps (only one context needed) as extended taclets. We make use of 11 new variable conditions and transformers. Each of these is realized as a simple, stand-alone Java class with clear responsibility, exposing as many details as possible in the textual representation of the taclet itself. As a representative example, one of extensions concerns a "for-each" construct for iterating over schema variables of list type. The following taclet code handles the case that AS completes abruptly due to a labeled **break**. It occurs in the SE rules for ASs, specifically, in the part describing the shape of the code resulting from the execution:

```
#foreach (#v1, #label in #vars, #labels) {
    if (#v1) {
        break #label;
    }
}
```

Implementing the for-each construct is more difficult than simply implementing a Java class for a transformer like "`#handle-labeled-breaks(#vars, #labels)`" that could replace the code above, but the result is more transparent and reusable, and comes close to the description of the rule in textbook-style.

Due to the complexity inherent to abstract statements, including completion modes and framing, the AE taclets implementing AS are, to the best of our knowledge, the most complex ones ever implemented. The longest taclet for ASs has 19 variable conditions, a "**\replacewith**" clause (the premises of the rule) of 68 lines and it extends to 79 lines of code in total.

Appendix C shows the AE taclet for AExps. The AS taclets have a similar shape but are even more lengthy since more abrupt completion possibilities must be considered.

Built-In Rules for Abstract Update Simplification

We had to realize a number of abstract update simplification rules as "built-in" rules directly in Java instead of as taclets. The main reason is that these rules depend on a *variable number of premises* in a context that is *initially unknown*. For instance, to implement rule $\mathsf{dropUpdate}_6$ in Fig. 11, we found no straightforward way to extend the taclet mechanism to allow for a more flexible specification of premises without the risk of breaking the existing implementation.

To realize the AS implementation as a pure *extension* of the KeY system without modifying the latter was an important design constraint that greatly improves maintainability of the AS functionality in future versions of KeY.

7 Related Work

Schematic programs are a natural way to describe program transformations *declaratively* in a *modular way*: One describes how to transform, for example, an **if** statement, while delegating the transformation of the *then* and *else* clauses to separate rules. The contents of these clauses are represented as *placeholders*.

One of the first applications of AE, and its original motivation, was the design of a modular, rule-based compiler with automatically proven-correct transformation rules [78] (see Sect. 5.5). Compilers are program transformers; in the area of proven-correct program transformations, the application to *mechanically verified compilers* gained significant interest. Compilers such as CompCert [48], CakeML [81] and Jinja [42] provide strong correctness guarantees. They all have in common that the source and target languages, correctness properties, and proofs are mechanized in *interactive proof assistants* such as Coq [82], Isabelle [59], or Lean [56]. These systems rely on expressive logical frameworks. In contrast, the scope of

AE is restricted to universal, behavioral program properties: AE abstracts away from the inner structure of the programs being proven. Conversely, this restriction is also a selling point of AE. We demonstrated that our framework is expressive enough to be applied to refactoring, cost analysis of transformations, code parallelization, correctness-by-construction, software product line engineering, and rule-based compilation (see Sect. 5). In almost all of these cases, the AE-KeY system found correctness proofs *fully automatically*. In contrast, interactive proof assistants require the user to write proof scripts manually: The CompCert code consists of 44% proof scripts [48].

In the context of the KeY system, Ahrendt et al. [6] and Bubel et al. [14] addressed automatic correctness proofs of Java DL program transformation rules. The former work projects rules to an executable semantics in the rewriting logic of the Maude system [55]; the latter one validates the correctness of *derived* SE rules in the KeY system itself. Both approaches are less expressive compared to AE. For example, Ahrendt et al. only support schematic expressions and Bubel et al. only schematic statements; we have both. The rewriting logic of Ahrendt et al. does not model abrupt completion. Neither of them admits constraints on frames and footprints or the specification of pre- and postconditions for APEs.

Abstract Execution completely decouples reasoning over abstract programs from the problem of checking whether a given concrete instance of an AS is valid. We discuss two approaches that, vice versa, check "eagerly" for instance validity. The first is the calculus for the differential dynamic logic dL of the *KeYmaera X* system [61]. It is based on *uniform substitution* of function, predicate, and *program* symbols. The calculus' axiomatic core is a set of *concrete* formulas with *uninterpreted* function/predicate/program symbols that may be instantiated to further concrete functions/predicates/programs via uniform substitution. Substitutions are sound provided that they do not *clash*, for example, substituting a term with a *free* variable for another term at a position where that variable is *bound* is forbidden. Uninterpreted symbols can be viewed as *abstract*, hence it is possible to express and to derive formulas over AS in the dL calculus. Consider the dL formula

$$[?q;(a;b) \cup ?\neg q;(a;c)]\varphi \leftrightarrow [a;(?q;b \cup ?\neg q;c)]\varphi$$

that represents the *Consolidate Duplicate Conditional Fragments* refactoring (see Sect. 5.1.3), where a, b, c, q, and φ are *arbitrary* programs, conditions, and formulas, respectively. Its validity in dL is provable. The drawback is that the conditions under which substitutions are sound, i.e., when no clash occurs, become rather complex. For example, one must implement the uniform substitution operator such that it avoids the case that an instance of q uses a variable assigned by an instance of a, etc. While manageable for the modeling language used in dL, it becomes extremely complex as soon as programming language features such as reference types, scopes, visibility rules, exceptions, etc., are introduced.

The most ambitious attempt at a substitution-centric approach so far is *partial evaluation* [23]. Given a program p with parameters, one considers a subset $\overline{\in}$ of its parameters to have *static* values $\overline{e}$ at compile time. Viewed from an AE perspective, partial evaluation instantiates every occurrence of $\overline{\in}$ in the *abstract* program $p(\overline{\in})$ with concrete $\overline{e}$, followed by simplification of the resulting *concrete* program using $\overline{e}$. The idea is to obtain a more efficient program than the original p under the assumption that part of the input is static. Like in uniform substitution, the main technical challenge is to find syntactic conditions to avoid clashes that would admit invalid substitutions [70]. For realistic functional or imperative programming language this is so complex that it became a main research direction in partial evaluation, known as *binding-time analysis* [38]. As far as we know, it was never fully axiomatized in a calculus.

In a *program repair* scenario, Mechtaev et al. [54] use abstract programs with parametric schematic *expressions*. Their goal is to synthesize witnesses for these expressions satisfying a postcondition. Compared to our work, Mechtaev et al. (1) address *existential* second-order program proofs, (2) do not consider abrupt completion, and (3) have no concept of abstract *statement.*

Godlin and Strichman [26] perform *regression verification* of closely related program versions. To automate the proofs in the presence of recursive functions, they replace recursive calls with *uninterpreted function symbols*; loops are transformed into recursive functions. Although AExps are related to uninterpreted functions, the latter are *pure*. The approach does not need, and consequently does not support, ASs.

The PEC system [46] uses *meta variables* for expressions, variables, and statements to prove correct compiler optimizations. Similarly to the other approaches discussed, PEC does not support additional specifications to constrain the behavior of placeholders, and its "meta statements" can only complete normally. Alive [50] has a more restricted scope: It automatically proves the correctness of *peephole optimizations* for LLVM. Those optimizations are expressed in a restricted DSL less general than the AE framework. For example, only register names can be abstract, and programs cannot contain loops.

Several works address the correctness of refactoring. We distinguish methods *statically* verifying refactoring techniques, including the extraction of preconditions using formal methods and static enforcement of safe refactoring, as well as *dynamic* techniques using testing and runtime assertions.

Garrido and Meseguer [24] formalized the Java refactoring techniques *Pull Up / Push Down Field*, *Pull Up / Push Down Method*, and *Rename Temporary* in Maude's rewrite logic. They prove the correctness of two refactoring techniques using a mixture of Maude evaluation and pen-and-paper proofs. Our AE-based proofs are fully mechanized and were derived by automatic proof search.

Using dynamic frames, AE abstracts away from concrete variable or field names. Schäfer et al. [68] address the problem of preventing naming and accessibility problems during code refactoring. For example, their framework ensures that a moved reference is still bound to the same declaration. For a safe application of refactoring techniques in practice, the behavioral guarantees from AE should be combined with a framework aware of names and bindings. Silva et al. [71] use Alloy [36] models to verify the type correctness of Java code transformations. They claim that they cover everything except behavioral issues—precisely what is handled by AE.

The design of the REFINITY system (see Sect. 5.1) favors the formalization of statement-level refactoring techniques. Recent work by Abusdal et al. [3] demonstrates that as well *class-level* refactoring techniques can be modeled and proven in our framework. The authors verified *Hide Delegate*, a technique involving multiple classes.

Regarding dynamic techniques, Soares et al. [74] automatically generate test suites for detecting behavioral changes caused by code refactoring using static analysis. Eilertsen et al. [20] add correctness assertions in the course of refactoring applications. The related work by Namjoshi and Zuck [58] generates *witnesses* during program transformation that guarantee the equivalence of the source and target programs.

8 Conclusion and Future Work

We presented the theory, implementation, and known applications of Abstract Execution (AE), a specification and semi-automated verification framework for behavioral, universal second-order program properties of *schematic* programs. In AE, programs may contain

schematic placeholders for both statements and expressions. The behavior of instantiations of these placeholders can be constrained by fine-grained, yet abstract, specifications. Compared to our previous conference publication [79], we (1) extended our framework with *abstract expressions* and *set-valued specification variables* for assigned and used locations, (2) provided a precise, *formal semantics* of abstract programs, (3) detailed our extended set of abstract update *simplification rules*, (4) described our *implementation*, and (5) discussed *recent applications* of the AE framework. In the context of our flagship application to verified code refactoring, we report on our *discovered bugs* in the refactoring engines of popular Java IDEs.

There are many ways to connect to this work. Our present work demonstrates that AE can be useful in different application scenarios. For example, one might analyze compiler optimizations beyond the "peephole" level targeted by frameworks such as Alive [50], or equivalence-preserving transformations for metamorphic testing [16] of compilers.

Going in a different direction, one could generalize concrete programs causing a specific behavior in program transformers (e.g., a crash or non-compiling output) to abstract versions describing the *class* of behavior-triggering inputs. Such an abstraction represents a *hypothesis* about the origins of the failure for *debugging*, in the spirit of Delta Debugging [92], DDSet [27], and the Alhazen tool [39]. This gives rise to another idea: Given an abstract program that forms a hypothesis about a class of behavior-inducing programs, one might help developers by *creating concrete instances* from this schematic one for further testing and validation. In our bug report to the IntelliJ developers, for instance, we could then not only have submitted a single, failing example but an abstract program from which the developers could generate myriads of further tests fully automatically by themselves.

Similar in spirit is *instance checking*: Is this concrete program fragment likely to trigger a known bug described by an abstract one? Or might it be an admissible input to a *Extract Method* refactoring? Such an instance checker, for which we have developed an early prototype, is the first ingredient for *deriving a transformer* from an input-output pair of schematic programs that detects possible input fragments in a larger context and transforms them into an optimized version.

Static instance checking is, in general, *expensive*: We might have to come up with strong loop invariants where we used abstract invariants in models; instantiating concrete, precise frames is another, non-trivial task. Instead, we could follow the approach of Eilertsen et al. [20] and derive *safety assertions* from transformation models checking runtime-enforceable properties of the models. Combined with test generation and the value of derived safety preconditions as a means of documentation, we can get the most out of an existing model even without static instance proofs.

While one can define separate precise pre- and postconditions for all completion modes of APE specification (for example, normal completion versus completion caused by **`break`**), the AE framework currently allows merely a single frame and footprint specification. The possibility to devise *mode-dependent* frames and footprints for each completion mode would enable more precise specifications and thus a broader set of represented concrete instances. Related to this, we extended the specification language to support *parametric location sets* in our application to code parallelization (Sect. 5.3). As a result, however, this application is the only one where the proof search required human interaction. Improving automation for such language extensions is essential for the attractiveness and acceptance of AE.

Automation can also be improved for our relational transformation proofs in general. RE-FINITY reduces a relational problem to the functional verification of individual programs, which works reasonably well with strong, but abstract, loop invariants. Using techniques from relational verification such as relational invariants [12], it might be possible to work

around strongest invariants, simplifying static instance checking and making AE ready for larger transformation proofs. Complicated, realistic transformation proofs might also require stronger support of heap-related properties. Compared to our earlier work [79], we significantly improved our abstract update simplification rules for heaps (Sect. 4.2.2); yet, more complex case studies will likely bring up situations demanding additional strategies for completeness and automation.

Abstract Execution is a promising and practically useful technique for proving properties of infinitely many programs in a single proof, with most applications residing in the area of program transformations. In this work, we have provided a complete account of the current state of the theory behind AE and its applications so far. We believe that this overview will provide a fruitful basis for a plethora of interesting follow-up work on the rigorous verification of program transformations.

Acknowledgments We thank the reviewers for their careful reading of our manuscript and their constructive feedback. The quality of the published paper considerably benefited from it.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Appendix A: Representation Formulas for Semantics of Abstract Statements

We define necessary and sufficient conditions for all aspects of the property of being an instance of APE or APF. For brevity, we only consider statements. The definitions apply to expressions e of type T by wrapping them inside statements $T\ \ \mathtt{x}{=}e\mathtt{;}$ for a fresh variable $\mathtt{x}$ that is added to the frame specification. AExps can only be instantiated by expressions of a subtype of their own type.

(1) *Frame Condition* A statement p satisfies the frame specification of AS iff it assigns *at most* locations in the *assignables* set of the AS. This holds if the following formula is valid, where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are all program variables assigned, but not declared, in p, and $\mathtt{x}_1^{pre}, \ldots, \mathtt{x}_n^{pre}$ as well as $\mathtt{heap}^{pre}$ are fresh program variables of suitable types:

$$
\begin{aligned}
&\mathit{frameFor}(\mathit{assignables}, p) :=\\
&\quad \{\mathtt{heap}^{pre} := \mathtt{heap} \,||\, \mathtt{x}_1^{pre} := \mathtt{x}_1 \,||\, \ldots \,||\, \mathtt{x}_n^{pre} := \mathtt{x}_n\}\\
&\qquad [p']\big((\forall f{:}\mathit{Field}; \forall o{:}\mathit{Object};\\
&\qquad\qquad o.\mathit{created}@\mathtt{heap}^{pre} \doteq \mathrm{FALSE} \vee o.f \doteq o.f@\mathtt{heap}^{pre}\\
&\qquad\qquad\quad \vee (o, f) \in \mathit{assignables}) \wedge\\
&\qquad\quad (\mathtt{x}_1 \doteq \mathtt{x}_1^{pre} \vee \mathring{\mathtt{x}}_1 \in \mathit{assignables}) \wedge \cdots \wedge (\mathtt{x}_n \doteq \mathtt{x}_n^{pre} \vee \mathring{\mathtt{x}}_n \in \mathit{assignables})\big)
\end{aligned}
$$

Intuitively, this formula permits p to create new objects or assign locally declared variables, but otherwise only change locations that are declared assignable. The program p' encapsulates p inside an **exec** statement to catch abrupt completion: inside a box modality, an abruptly completing program yields a trivially valid formula, while inside a diamond modality, the formula is unsatisfiable regardless of the postcondition:

$$
\begin{aligned}
p' := \ & \texttt{exec}\ \{\, p \,\} \\
& \texttt{ccatch}\ (\texttt{\textbackslash Return})\ \{\} \\
& \texttt{ccatch}\ (\texttt{\textbackslash Return Object v})\ \{\} \\
& \vdots \\
& \texttt{ccatch}\ (\texttt{Throwable t})\ \{\}
\end{aligned}
$$

(2) *Has-To Condition* Standard JavaDL is not capable of recording memory access: a program variable that is assigned its original value as in `x = x;` is indistinguishable from the empty statement. A recent extension of JavaDL tracks read and written memory locations in a global history [13]. We refrain from introducing this theory here, and conservatively overapproximate the has-to condition by a static check: let the predicate $hasToFor(assignables, p)$ evaluate to *tt* if each location in *assignables* designated as a has-to location occurs *syntactically* on the left-hand side of at least one assignment statement (including pre-increments, etc.) along any path of the program's CFG.

(3) *Footprint Specification* A program satisfies the footprint (*accessibles*) specification of APE if the evaluation in two *arbitrary* environments—which, however, agree on the values of the accessible locations—yields the same effects on the assignable locations. The condition we define is inspired by *dependency contracts* [69], with the difference that we are interested in the effect on a *set* of locations, not merely the **\result** variable of a method:

$$
\begin{aligned}
& footprintFor(accessibles, assignables, p) := \\
& \quad ([p']Post(value(assignables)) \leftrightarrow \\
& \qquad \{\mathtt{heap} := anon(\mathtt{heap}, allLocs \setminus heapLocs(accessibles), h) \,||\, \\
& \qquad\ \mathtt{x}_1 := value(anonPV(\mathring{\mathtt{x}}_1, allLocs \setminus pvLocs(accessibles), \mathring{\mathtt{x}}_1^a)) \,||\, \ldots \,||\, \\
& \qquad\ \mathtt{x}_n := value(anonPV(\mathring{\mathtt{x}}_n, allLocs \setminus pvLocs(accessibles), \mathring{\mathtt{x}}_n^a))\} \\
& \qquad\ [p]Post(value(assignables)))
\end{aligned}
$$

where

- *Post* is a fresh predicate of suitable type and arity,
- the $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are all program variables occurring in p,
- h is a fresh *Heap* symbol and $\mathtt{x}_i^a$ are fresh program variables of the types of $\mathtt{x}_i$,
- and p' is defined as above.

(4) Termination AnAPE specifies whether instances must terminate or may diverge depending on the value of the *term* field (where total stands for termination). We can express this with the diamond modality of JavaDL: Even the trivial postcondition "true" only holds in the scope of a diamond modality if the enclosed program terminates *normally*. As above we catch termination by abrupt completion by using p' instead of p.

$$terminationFor(term, p) := term \doteq \text{total} \rightarrow \langle p' \rangle \text{true}$$

(5) Normal Completion An instance of APE has to complete normally when none of the preconditions for abrupt completion is met. In this case, it also has to satisfy the postcondition

$\{\mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint})\}h \rightsquigarrow \mathit{anon}(h, \mathit{frame}, \mathit{anonHeap}_\mathrm{P}(\mathit{footprint}))$ abstractHeapUpdate
where h is of sort *Heap* and
$\mathit{wellFormed}(\mathit{anonHeap}_\mathrm{P}(\mathit{footprint}))$ is added to the path condition

$\mathit{select}_A(\{\ldots \,||\, \mathcal{U}_\mathrm{P}(\mathit{fr}_1, \ldots, \mathit{fr}_n :\approx \mathit{footprint}) \,||\, \ldots\}\mathtt{heap}, o, f)$ $\mathrm{dropUpdate}_6$
$\rightsquigarrow \mathit{select}_A(\{\ldots \,||\, \mathit{Skip} \,||\, \ldots\}\mathtt{heap}, o, f)$
where $\forall i = 1, \ldots, n;\ ((o, f) \notin \mathit{fr}_i) \in C$

$\{\ldots \,||\, \mathtt{heap} := \mathit{anon}(h, \mathit{frame}, \mathit{anonHeap}) \,||\, \mathcal{U}\}t$ dropAnonInUpdate
$\rightsquigarrow \{\ldots \,||\, \mathtt{heap} := h \,||\, \mathcal{U}\}t$
where $\mathit{irrelevant}(\mathit{Ctx}, \mathit{frame}, t)$ and $\mathit{irrelevant}(C, \mathit{frame}, \mathcal{U})$

$\mathit{select}_A(\mathit{anon}(h, \mathit{frame}, \mathit{anonHeap}), o, f) \rightsquigarrow \mathit{select}_A(h, o, f)$ dropAbstractAnonInSelect
where $((o, f) \notin \mathit{frame}) \in C$

$\mathit{anon}(\mathit{anon}(h, \mathit{frame}_2, \mathit{anonHeap}_2), \mathit{frame}_1, \mathit{anonHeap}_1)$ dropAbstractAnon
$\rightsquigarrow \mathit{anon}(h, \mathit{frame}_1, \mathit{anonHeap}_1)$
where $\mathit{overwrites}(C, \mathit{frame}_1, \mathit{frame}_2)$ and
$\mathit{irrelevant}(C, \mathit{frame}_2, \mathit{anonHeap}_1)$

$\{\ldots \,||\, \mathtt{heap} := \mathit{store}(h, o, f, t)\}\mathit{value}(\mathit{frame})$ dropStoreToValue
$\rightsquigarrow \{\ldots \,||\, \mathtt{heap} := h\}\mathit{value}(\mathit{frame})$
where $((o, f) \notin \mathit{frame}) \in C$

$\{\mathtt{heap} := \{\mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint})\}\mathtt{heap} \,||\, \mathcal{U}\}\mathtt{heap}$ flattenAbstractHeapUpdate
$\rightsquigarrow \{\mathcal{U}_\mathrm{P}(\mathit{frame} :\approx \mathit{footprint}) \,||\, \mathcal{U}\}\mathtt{heap}$

$\{\ldots \,||\, \mathtt{heap} := t_1 \,||\, \mathtt{v} := t_2\}t \rightsquigarrow \{\ldots \,||\, \mathtt{v} := t_2 \,||\, \mathtt{heap} := t_1\}t$ pushHeapUpdateToEnd
where $\mathtt{v} \neq \mathtt{heap}$

$\mathit{anon}(\mathit{anon}(h, \mathit{frame}_1, t_1), \mathit{frame}_2, t_2)$ reorderAbstractAnon
$\rightsquigarrow \mathit{anon}(\mathit{anon}(h, \mathit{frame}_2, t_2), \mathit{frame}_1, t_1)$
where $\mathit{frame}_1 \cap \mathit{frame}_2 \doteq \emptyset \in C$ and
frame_1 is lexicographically smaller than frame_2

Fig. 11 Heap-Related Abstract Update Simplification Rules

for normal completion. We first define the precondition *normal* for normal completion as the conjunction of the negations of all abrupt completion preconditions (*pre* extracts the precondition of a pre- and postcondition pair):

$$\begin{aligned}\mathit{normal} :=& \neg \mathit{pre}(\mathit{returnsSpec}) \wedge \neg \mathit{pre}(\mathit{returnsValSpec}) \wedge \neg \mathit{pre}(\mathit{excSpec}) \\ & \wedge \neg \mathit{pre}(\mathit{continuesSpec}) \wedge \neg \mathit{pre}(\mathit{breaksSpec}) \\ & \wedge \bigwedge_{l \in \mathit{dom}(\mathit{continuesSpecLbl})} \neg \mathit{pre}(\mathit{continuesSpecLbl}(l)) \wedge \bigwedge_{l \in \mathit{dom}(\mathit{breaksSpecLbl})} \neg \mathit{pre}(\mathit{breaksSpecLbl}(l))\end{aligned}$$

In the following formalization of the normal completion requirement, we use a labeled **`break`** with a fresh label l which is only reached if p completed normally. A fresh boolean variable `_normal`, initialized to **`false`**, is set to **`true`** inside an **`exec`** statement if the labeled **`break`** was reached, which is required by the postcondition.

$\mathit{normalCompletionFor}(\mathit{specs}, p) := \mathit{normal} \rightarrow$
`[_normal=false;`

$$\texttt{exec}\ \{p\ \texttt{break}\ l;\}\ \texttt{ccatch}\ (\texttt{\textbackslash Break}\ l)\ \{\texttt{\_normal=true;}\}]$$
$$(\texttt{\_normal} \doteq \text{TRUE} \wedge \mathit{normalPost})$$

(6) Abrupt Completion The formalization of the requirements imposed on abrupt completion work analogously to normal completion, but are simpler, since no labeled **break** is needed. We consider in detail the (most complex) case of completion due to a **return** of a value (similar to *pre*, *post* extracts the postcondition of a pre- and postcondition pair):

$$\mathit{returnsFor}(\mathit{returnsSpec}, p) := \mathit{pre}(\mathit{returnsSpec}) \rightarrow$$
$$[\texttt{\_returned=false;}$$
$$\texttt{exec}\ \{p\}$$
$$\texttt{ccatch}\ (\texttt{\textbackslash Return})\ \{\texttt{\_returned=true;}\}]$$
$$\texttt{ccatch}\ (\texttt{\textbackslash Return Object val})\ \{\texttt{res=val; \_returned=true;}\}]$$
$$(\texttt{\_returned} \doteq \text{TRUE} \wedge \mathit{post}(\mathit{returnsSpec}))$$

Recall that `res` is a special program variable for accessing the returned value in the postcondition of the specification of abrupt completion due to a **return**.

The remaining definitions of $\mathit{excFor}(\mathit{excSpec}, p)$, $\mathit{breaksFor}(\mathit{breaksSpec}, p)$ due to a **break**, and $\mathit{continuesFor}(\mathit{continuesSpec}, p)$ for abrupt completion due to a **continue**, are defined analogously. For the latter two, we do not need an assignment to a variable like `res`, while for exceptional completion, the variable `exc` exposing the thrown exception object is assigned. Furthermore, only one **ccatch** clause is required. The formulas $\mathit{breaksForLbl}(\mathit{breaksSpecLbl}, \mathit{lb}, p)$ and $\mathit{continuesForLbl}(\mathit{continuesSpecLbl}, \mathit{lb}, p)$ for labeled **break**s and **continue**s receive an additional parameter *lb* representing the specific label which should be considered, and are defined analogously otherwise. We omit these cases for brevity.

Appendix B: Heap-Related Abstract Update Rules

The heap-related abstract update simplification rules are given in Fig. 11.

Rule abstractHeapUpdate transforms an abstract update $\mathcal{U}_{\mathtt{P}}(\mathit{frame}: \approx \mathit{footprint})$ to a variable *h* of sort *Heap* into an *anon* term $\mathit{anon}(h, \mathit{frame}, \mathit{anonHeap}_{\mathtt{P}}(\mathit{footprint}))$. Both expressions are equivalent: Changing heap locations in *frame* depending on locations in *footprint* is the same as anonymizing those locations with a heap term $\mathit{anonHeap}_{\mathtt{P}}(\mathit{footprint})$, where $\mathit{anonHeap}_{\mathtt{P}}$ is a function symbol created *fresh for* the APE `P`. The transformation allows us to reuse the existing Java DL machinery for heaps and location sets in proofs (by using the *anon* function) instead of producing further AE-specific simplification rules.

The next five rules allow to drop abstract updates, *anon* applications, or *store* applications. Rule dropUpdate$_6$ matches an abstract update $\mathit{select}_A(\{\ldots \mid\mid \mathcal{U}_{\mathtt{P}}(\mathit{frame}: \approx \mathit{footprint}) \mid\mid \ldots\}\mathtt{heap}, o, f)$ inside a *select* expression. If the selected location is not in the frame of the abstract update, i.e., $(o, f) \notin \mathit{fr}_i$ is in the context for all parts fr_i of *frame*, we can remove the abstract update. Rule dropAnonInUpdate, not containing an abstract update, simplifies heap anonymizations that have no effect since the masked part of the heap is not accessed in subsequent expressions. This situation frequently occurs in abstract contexts. Since the *irrelevant* predicate does not apply if the `heap` variable occurs freely in the target expression, e.g., if there is a modality in the target, "conventional" non-abstract contexts are unaffected. The rules dropAbstractAnonInSelect and dropAbstractAnon also allow dropping *anon* applications: When selecting a field from an *anon* term, the *anon* can be removed if the field is not contained in the anonymized heap portion. Furthermore, we can drop an inner of two nested *anon*s if the outer one overwrites a larger memory portion

and the outer anonymizing heap term is not affected by the inner *anon*. The final "drop" rule, dropStoreToValue, removes a *store* application in a heap update to a value term *value*(*frame*) if the target field of the store is irrelevant to *frame*.

The remaining three rules are normalization rules in the spirit of the reorderUpdate rules from Fig. 5. They bring heap updates to a normal form, which facilitates relational transformation proofs where abstract heap updates cannot be applied to target terms and thus be removed. Rule flattenAbstractHeapUpdate simplifies a nested heap update containing an abstract update. The rule pushHeapUpdateToEnd pushes an update to the `heap` variable to the end of a parallel update. Finally, reorderAbstractAnon reorders nested *anon* applications by ordering them lexicographically if they are independent.

Appendix C: Taclet Code for Abstract Expressions

```
abstractExpression {
  \schemaVar \update U;

  \find (\modality{#allmodal}{..#v = #aexp;...}\endmodality(post))

  \varcond(\new(#normal, boolean))
  \varcond(\new(#throwsExc, boolean))
  \varcond(\new(#exc, java.lang.Throwable))
  \varcond(\new(#returns, boolean))
  \varcond(\new(#result, \typeof(#v)))
  \varcond(\new(#h, Heap))
  \varcond(\initializeParametricSkolemUpdate(U, #aexp))

  \replacewith (
    {   #normal:=#abstrPrecond(#aexp, "normal")
     || #throwsExc:=#abstrPrecond(#aexp, "throwsExc")
     || #h:=heap} (
       (  (#normal = TRUE <-> !#throwsExc = TRUE)
        & (#excPrecondition(#aexp, #throwsExc))
        & (#throwsExc = TRUE -> !#exc = null)) ->
       {U}{
         #exc:=#abstrPrecond(#aexp, "exceptionObject") ||
         #result:=#addCast(#abstrPrecond(#aexp, "resultObject"), #v)
       }(
           (( #throwsExc = TRUE -> !#exc = null &
                #postCondAE(
                  #aexp, "throwsExc", #returns, #result, #exc)) &
            (!#throwsExc = TRUE ->
                #postCondAE(
                  #aexp, "normal", #returns, #result, #exc)) &
            (\forall f; \forall o;
              (  elementOf(o,f,#getFrame(#aexp))
               | !o=null &
                 !boolean::select(
                   #h,o,java.lang.Object::<created>)=TRUE
               | any::select(heap,o,f) = any::select(#h,o,f)))) ->
             \modality{#allmodal}{
               ..
               if (#throwsExc) { throw #exc; }
               #v = #result;
               ...
             }\endmodality(post)
         )))

  \heuristics(abstractExecution, simplify_prog)
};
```

References

1. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press (2010)
3. Abusdal, O.J., Kamburjan, E., Pun, V.K.I., Stolz, V.: A Notion of Equivalence for Refactorings with Abstract Execution. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering - 11th International Symposium, ISoLA 2022, Proceedings, Part II, *LNCS*, vol. 13702, pp. 259–280. Springer (2022). https://doi.org/10.1007/978-3-031-19756-7_15
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley series in computer science / World student series edition. Addison-Wesley (1986). https://www.worldcat.org/oclc/12285707
5. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, *LNCS*, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
6. Ahrendt, W., Roth, A., Sasse, R.: Automatic Validation of Transformation Rules for Java Verification Against a Rewriting Semantics. In: G. Sutcliffe, A. Voronkov (eds.) Proc. 12th LPAR, *LNCS*, vol. 3835, pp. 412–426. Springer (2005). https://doi.org/10.1007/11591191_29
7. Albert, E., Hähnle, R., Merayo, A., Steinhöfel, D.: Certified Abstract Cost Analysis. In: E. Guerra, M. Stoelinga (eds.) Proc. 24th Intern. Conf. on Fundamental Approaches to Software Engineering (FASE), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS), *LNCS*, vol. 12649, pp. 24–45. Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_2
8. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. **51**(3), 50:1-50:39 (2018). https://doi.org/10.1145/3182657
9. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal System Development with KIV. In: T. Maibaum (ed.) Fundamental Approaches to Software Engineering, LNCS, pp. 363–366. Springer (2000)
10. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs. In: M.J. Butler, W. Schulte (eds.) 17th Intl. Symp. on Formal Methods (FM), *LNCS*, vol. 6664, pp. 200–214. Springer (2011)
11. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: F. Cassez, R. Huuck, G. Klein, B. Schlich (eds.) Proc. 7th Conf. on Systems Software Verification (SSV), *EPTCS*, vol. 102, pp. 18–32 (2012). https://doi.org/10.4204/EPTCS.102.4
12. 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). https://doi.org/10.1007/978-3-319-98047-8_3
13. Bubel, R., Hähnle, R., Heydari Tabar, A.: A Program Logic For Dependence Analysis. In: W. Ahrendt, S.L.T. Tarifa (eds.) Proc. 15th Intern. Conf on Integrated Formal Methods (IFM), LNCS. Springer (2019). To appear
14. Bubel, R., Roth, A., Rümmer, P.: Ensuring the Correctness of Lightweight Tactics for JavaCard Dynamic Logic. Electr. Notes Theor. Comput. Sci. **199**, 107–128 (2008). https://doi.org/10.1016/j.entcs.2007.11.015
15. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing '74, pp. 308–312. Elsevier/North-Holland (1974)
16. Chen, T.Y., Cheung, S.C., Yiu, S.: Metamorphic Testing: A New Approach for Generating Next Test Cases. CoRR **abs/2002.12543** (2020). arxiv:2002.12543
17. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated Testing of Refactoring Engines. In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 185–194 (2007)
18. Dannenberg, R., Ernst, G.: Formal Program Verification Using Symbolic Execution. IEEE Transactions on Software Engineering **SE–8**(1), 43–52 (1982)
19. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
20. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer Refactorings. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, *LNCS*, vol. 9952, pp. 517–531 (2016). https://doi.org/10.1007/978-3-319-47166-2_36
21. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Object Technology Series (1999)

22. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Signature Series. Addison-Wesley Professional (2018). 2nd edition
23. Futamura, Y.: Partial computation of programs. In: E. Goto, K. Furukawa, R. Nakajima, I. Nakata, A. Yonezawa (eds.) RIMS Symposium on Software Science and Engineering, Kyoto, Japan, *LNCS*, vol. 147, pp. 1–35. Springer (1982). https://doi.org/10.1007/3-540-11980-9_13
24. Garrido, A., Meseguer, J.: Formal Specification and Verification of Java Refactorings. In: Proc. 6th SCAM, pp. 165–174. IEEE Computer Society, Washington, DC, USA (2006). https://doi.org/10.1109/SCAM.2006.16
25. Godefroid, P.: Test Generation using Symbolic Execution. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 18, pp. 24–33. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012). http://drops.dagstuhl.de/opus/volltexte/2012/3845/
26. Godlin, B., Strichman, O.: Regression Verification: Proving the Equivalence of Similar Programs. Softw. Test., Verif. Reliab. **23**(3), 241–258 (2013). https://doi.org/10.1002/stvr.1472
27. Gopinath, R., Kampmann, A., Havrikov, N., Soremekun, E.O., Zeller, A.: Abstracting failure-inducing inputs. In: S. Khurshid, C.S. Pasareanu (eds.) ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, pp. 237–248. ACM (2020). https://doi.org/10.1145/3395363.3397349
28. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java (TM) Language Specification, 3rd edn. Addison-Wesley Professional (2005)
29. Gries, D.: The Science of Programming. Texts and Monographs in Computer Science. Springer (1981)
30. Hähnle, R., Huisman, M.: Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In: B. Steffen, G.J. Woeginger (eds.) Computing and Software Science - State of the Art and Perspectives, *LNCS*, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
31. Hähnle, R., Schaefer, I.: A liskov principle for delta-oriented programming. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, *LNCS*, vol. 7609, pp. 32–46. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_4
32. Hähnle, R., Schaefer, I., Bubel, R.: Reuse in software verification by abstract method calls. In: M.P. Bonacina (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings, *LNCS*, vol. 7898, pp. 300–314. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_21
33. Hähnle, R., Tabar, A.H., Mazaheri, A., Norouzi, M., Steinhöfel, D., Wolf, F.: Safer Parallelization. In: T. Margaria, B. Steffen (eds.) Proc. 9th Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles (Part II), *LNCS*, vol. 12477, pp. 117–137. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_8
34. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**(10), 576–580 (1969)
35. Huda, Z.U., Jannesari, A., Wolf, F.: Using Template Matching to Infer Parallel Design Patterns. TACO **11**(4), 64:1-64:21 (2015). https://doi.org/10.1145/2688905
36. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006). http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928
37. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: M.G. Bobaru, K. Havelund, G.J. Holzmann, R. Joshi (eds.) Proc. 3rd Intern. Symp. NASA Formal Methods (NFM), *LNCS*, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
38. Jones, N.D.: Static semantics, types, and binding time analysis. Theor. Comput. Sci. **90**(1), 95–118 (1991). https://doi.org/10.1016/0304-3975(91)90301-H
39. Kampmann, A., Havrikov, N., Soremekun, E.O., Zeller, A.: When Does My Program Do This? Learning Circumstances of Software Behavior. In: P. Devanbu, M.B. Cohen, T. Zimmermann (eds.) ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020, pp. 1228–1239. ACM (2020). doi:10.1145/3368089.3409687
40. Kassios, I.T.: The Dynamic Frames Theory. Formal Asp. Comput. **23**(3), 267–288 (2011). https://doi.org/10.1007/s00165-010-0152-5
41. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
42. Klein, G., Nipkow, T.: A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. ACM Trans. PLS **28**(4), 619–695 (2006)

43. Knüppel, A., Krüger, S., Thüm, T., Bubel, R., Krieter, S., Bodden, E., Schaefer, I.: Using Abstract Contracts for Verifying Evolving Features and Their Interactions, *LNCS*, vol. 12345, pp. 122–148. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_5
44. Knüppel, A., Runge, T., Schaefer, I.: Scaling correctness-by-construction. In: T. Margaria, B. Steffen (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles: 9th Intl. Symp. on Leveraging Applications of Formal Methods, ISoLA, Rhodes, Greece, Part I, *LNCS*, vol. 12476, pp. 187–207. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_10
45. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-27919-5
46. Kundu, S., Tatlock, Z., Lerner, S.: Proving Optimizations Correct Using Parameterized Program Equivalence. In: Proc. PLDI 2009, pp. 327–337 (2009)
47. 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 (2013). http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf. Draft revision 2344
48. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM **52**(7), 107–115 (2009)
49. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). https://doi.org/10.1145/197320.197383
50. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Practical Verification of Peephole Optimizations with Alive. Commun. ACM **61**(2), 84–91 (2018)
51. Massingill, B.L., Mattson, T.G., Sanders, B.A.: Parallel Programming with a Pattern Language. Int. J. Softw. Tools Technol. Transf. **3**(2), 217–234 (2001). https://doi.org/10.1007/s100090100045
52. Mazaheri, A., Norouzi, M., Olokin, K., Wolf, F.: Enhancing parallel pattern identification through code-restructuring. Tech. rep., Technical University of Darmstadt, Department of Computer Science, Laboratory for Parallel Programming (2020)
53. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In: P. Braffort, D. Hirschberg (eds.) Computer Programming and Formal Systems, pp. 33–69. North Holland (1963)
54. 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)
55. Meseguer, J., Winkler, T.C.: Parallel Programmming in Maude. In: J. Banâtre, D.L. Métayer (eds.) Research Directions in High-Level Parallel Programming Languages, *LNCS*, vol. 574, pp. 253–293. Springer (1991). https://doi.org/10.1007/3-540-55160-3_49
56. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: A. Platzer, G. Sutcliffe (eds.) CADE, 28th Intl. Conf. on Automated Deduction, *LNCS*, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
57. Murphy, G.C., Kersten, M., Findlater, L.: How Are Java Software Developers Using the Eclipse IDE? IEEE Software **23**(4), 76–83 (2006). https://doi.org/10.1109/MS.2006.105
58. Namjoshi, K.S., Zuck, L.D.: Witnessing Program Transformations. In: F. Logozzo, M. Fähndrich (eds.) 20th International Symposium on Static Analysis (SAS), *LNCS*, vol. 7935, pp. 304–323. Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_17
59. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002). doi:https://doi.org/10.1007/3-540-45949-9
60. Norouzi, M., Wolf, F., Jannesari, A.: Automatic construct selection and variable classification in OpenMP. In: Proc. of the International Conference on Supercomputing (ICS), Phoenix, AZ, USA, pp. 330–341. ACM (2019). https://doi.org/10.1145/3330345.3330375
61. Platzer, A.: A Complete Uniform Substitution Calculus for Differential Dynamic Logic. J. Autom. Reason. **59**(2), 219–265 (2017). https://doi.org/10.1007/s10817-016-9385-1
62. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005). https://doi.org/10.1007/3-540-28901-1
63. Qu, W., Gaboardi, M., Garg, D.: Relational Cost Analysis for Functional-Imperative Programs. Proc. ACM Program. Lang. **3**(ICFP), 92:1-92:29 (2019). https://doi.org/10.1145/3341696
64. Radicek, I., Barthe, G., Gaboardi, M., Garg, D., Zuleger, F.: Monadic Refinements for Relational Cost Analysis. Proc. ACM Program. Lang. **2**(POPL), 36:1-36:32 (2018). https://doi.org/10.1145/3158124
65. Rümmer, P., Ulbrich, M.: Proof Search with Taclets. In: W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P.H. Schmitt, M. Ulbrich (eds.) Deductive Software Verification - The KeY Book - From Theory to Practice, *LNCS*, vol. 10001, pp. 107–147. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6_4
66. Scaletta, M., Hähnle, R., Steinhöfel, D., Bubel, R.: Delta-based verification of software product families. In: C. De Roever (ed.) Proc. 20th Intl. Conf. on Generative Programming (GPCE). ACM Press, New York, NY, USA (2021)

67. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: J. Bosch, J. Lee (eds.) Software Product Lines: Going Beyond, *LNCS*, vol. 6287, pp. 77–91. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15579-6_6
68. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. IEEE Trans. Software Eng. **38**(6), 1233–1257 (2012). https://doi.org/10.1109/TSE.2012.13
69. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic Frames in Java Dynamic Logic. In: B. Beckert, C. Marché (eds.) Intl. Conf. on Formal Verification of Object-Oriented Software (FoVeOOS), *LNCS*, vol. 6528, pp. 138–152. Springer (2011)
70. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. **25**(4), 452–499 (2003). https://doi.org/10.1145/778559.778561
71. da Silva, T.D., Sampaio, A., Mota, A.: Verifying Transformations of Java Programs Using Alloy. In: M. Cornélio, B. Roscoe (eds.) 18th Brazilian Symposium on Formal Methods: Foundations and Applications (SBMF), *LNCS*, vol. 9526, pp. 110–126. Springer (2015). https://doi.org/10.1007/978-3-319-29473-5_7
72. Soares, G., Catao, B., Varjao, C., Aguiar, S., Gheyi, R., Massoni, T.: Analyzing Refactorings on Software Repositories. In: 25th Brazilian Symposium on Software Engineering (SBES), pp. 164–173. IEEE Computer Society (2011). https://doi.org/10.1109/SBES.2011.21
73. Soares, G., Gheyi, R., Massoni, T.: Automated Behavioral Testing of Refactoring Engines. IEEE Trans. Software Eng. **39**(2), 147–162 (2013). https://doi.org/10.1109/TSE.2012.19
74. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making Program Refactoring Safer. IEEE Software **27**(4), 52–57 (2010). https://doi.org/10.1109/MS.2010.63
75. Srivastava, S., Gulwani, S., Foster, J.S.: From Program Verification to Program Synthesis. In: Proc. 37th POPL, pp. 313–326 (2010). https://doi.org/10.1145/1706299.1706337
76. Steinhöfel, D.: Abstract Execution: Automatically Proving Infinitely Many Programs. Ph.D. thesis, Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany (2020). https://doi.org/10.25534/tuprints-00008540.http://tuprints.ulb.tu-darmstadt.de/8540/
77. Steinhöfel, D.: REFINITY to Model and Prove Program Transformation Rules. In: B.C. d. S. Oliveira (ed.) Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS), *LNCS*, vol. 12470, pp. 311–319. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_16
78. Steinhöfel, D., Hähnle, R.: Modular, Correct Compilation with Automatic Soundness Proofs. In: T. Margaria, B. Steffen (eds.) Proc. 8th ISOLA, LNCS (2018)
79. Steinhöfel, D., Hähnle, R.: Abstract Execution. In: Proc. Third World Congress on Formal Methods - The Next 30 Years, (FM), pp. 319–336 (2019). https://doi.org/10.1007/978-3-030-30942-8_20
80. Steinhöfel, D., Hähnle, R.: The Trace Modality. In: L.S. Barbosa, A. Baltag (eds.) Dynamic Logic. New Trends and Applications - Second International Workshop (DaLi), *LNCS*, vol. 12005, pp. 124–140. Springer (2019). https://doi.org/10.1007/978-3-030-38808-9_8
81. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A New Verified Compiler Backend for CakeML. In: Proc. 21st ICFP, pp. 60–73. ACM (2016). https://doi.org/10.1145/2951913.2951924
82. The Coq Development Team: The Coq Proof Assistant, version 8.10.0 (2019). https://doi.org/10.5281/zenodo.3476303
83. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 6:1-6:45 (2014). https://doi.org/10.1145/2580950
84. Thüm, T., Knüppel, A., Krüger, S., Bolle, S., Schaefer, I.: Feature-oriented contract composition. In: Proc. 23rd Intl. Systems and Software Product Line Conference - Volume A, p. 25. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3336294.3342374
85. Thüm, T., Schaefer, I., Apel, S., Hentschel, M.: Family-based deductive verification of software product lines. SIGPLAN Not. **48**(3), 11–20 (2012). https://doi.org/10.1145/2480361.2371404
86. Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S.: Proof composition for deductive verification of software product lines. In: Fourth IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST, Berlin, Germany, pp. 270–277. IEEE Computer Society, Los Alamitos, CA (2011). https://doi.org/10.1109/ICSTW.2011.48
87. Vermeer, B.: IntelliJ IDEA dominates the IDE market with 62% adoption among JVM developers (2020). https://snyk.io/blog/intellij-idea-dominates-the-ide-market. Last accessed: 2021/06/16
88. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. Ph.D. thesis, Karlsruhe Institute of Technology, Karlsruhe (2011)
89. Winterland, D.: Abstract execution for correctness-by-construction. Master's thesis, Technische Universität Braunschweig, Institute of Software Engineering and Automotive Informatics (2020). https://www.tu-braunschweig.de/isf/team/runge
90. Wirth, N.: Program development by stepwise refinement. CACM **14**(4), 221–227 (1971)

91. Yang, G., Filieri, A., Borges, M., Clun, D., Wen, J.: Advances in Symbolic Execution. In: A.M. Memon (ed.) Advances in Computers, *Advances in Computers*, vol. 113, pp. 225 – 287. Elsevier (2019). https://doi.org/10.1016/bs.adcom.2018.10.002. http://www.sciencedirect.com/science/article/pii/S0065245818300627
92. Zeller, A.: Yesterday, My Program Worked. Today, It Does Not. Why? In: O. Nierstrasz, M. Lemoine (eds.) Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings, *Lecture Notes in Computer Science*, vol. 1687, pp. 253–267. Springer (1999). https://doi.org/10.1007/3-540-48166-4_16