Ever Change a Running System: Structured Software Reengineering Using Automatically Proven-Correct Transformation Rules

Dominic Steinhöfel

Abstract Legacy systems are business-critical software systems whose failure can have a significant impact on the business. Yet, their maintenance and adaption to changed requirements consume a considerable amount of the total software development costs. Frequently, domain experts and developers involved in the original development are not available anymore, making it difficult to adapt a legacy system without introducing bugs or unwanted behavior. This results in a dilemma: businesses are reluctant to change a working system, while at the same time struggling with its high maintenance costs. We propose the concept of *Structured Software Reengineering* replacing the ad hoc forward engineering part of a reengineering process with the application of behavior-preserving, proven-correct transformations improving nonfunctional program properties. Such transformations preserve valuable business logic while improving properties such as maintainability, performance, or portability to new platforms. Manually encoding and proving such transformations for industrial programming languages, for example, in interactive proof assistants, is a major challenge requiring deep expert knowledge. Existing frameworks for automatically proving transformation rules have limited expressiveness and are restricted to particular target applications such as compilation or peep-hole optimizations. We present *Abstract Execution*, a specification and verification framework for statement-based program transformation rules on Java programs building on symbolic execution. Abstract Execution supports *universal quantification* over statements or expressions and addresses properties about the (big-step) *behavior* of programs. Since this class of properties is useful for a plethora of applications, Abstract Execution bridges the gap between expressiveness and automation. In many cases, fully automatic proofs are in possible. We explain REFINITY, a workbench for modeling and proving statement-level Java transformation rules, and discuss our applications of Abstract Execution to code refactoring,

D. Steinhöfel (✉)
CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Software Engineering Group, TU Darmstadt, Darmstadt, Germany
e-mail: dominic.steinhoefel@cispa.de

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2020*,
https://doi.org/10.1007/978-3-030-83128-8_10

cost analysis of program transformations, and transformations reshaping programs for the application of parallel design patterns.

1 Introduction

"When Companies Become Prisoners of Legacy Systems": this title of a *Wall Street Journal* article [41] well describes the predominant perception of legacy software systems in academia. Indeed, legacy systems are often associated with *high maintenance costs* [12, 13, 40], ranging between 67% and 90% of the total software costs [13]. In a more recent case study [12], an amount of 75% was reported for a financial services company. Interestingly, this perception is not always shared in industry. In an interview with 26 industry practitioners [26], they describe their legacy systems as businesses-critical, reliable, and proven systems. The same survey names the loss of expert knowledge about the software, high maintenance costs, and the desire to increase flexibility as main drivers for *modernization* of legacy systems.

Practitioners are facing a dilemma: they are reluctant to "change a running system" encoding valuable domain knowledge and representing high business value, while struggling with the costs to keep it running. The stakes are high, and the risks to introduce bugs or unwanted behavior during legacy system modernization are considerable [19]. A major challenge is that the original source code has been evolving substantially, documentation might be missing, and domain experts as well as developers involved in past developments might not be available any more [19, 52]. Therefore, even minor modifications present high risk [52].

Software Reengineering is the "examination and alteration of a subject system to reconstitute it in a new form" [11]. The goal is to preserve domain knowledge encoded in existing software, while improving desirable nonfunctional properties like maintainability, performance, and flexibility. In the case of business-critical legacy systems, reengineering efforts must be especially careful to maintain the essential semantics of the subject system, for example, through the introduction of automated tests [16]. While automated testing should be part of any serious software development, it bears two problems: first, writing meaningful, fine-granular tests for legacy systems is nontrivial in presence of missing documentation and domain experts. Second, post hoc verification only informs about a failure *after* its introduction.

Regression Verification [20] allows automatically verifying the equivalence of closely related versions of the same program. If the changes applied during reengineering are small enough, regressions can in principle be excluded using this methodology. However, this requires analyzing whole methods after each change, and side effects or irregular completion (e.g., exceptions) are not considered.

We propose *structured* software reengineering as an alternative. Following Chikofsky and Cross [11], reengineering (1) starts with a *reverse engineering* phase to achieve a more abstract description of the system, followed by (2) a *forward engineering* or *restructuring* phase. Structured reengineering does not impose requirements on phase (1), which aims to carve out reengineering goals and concrete suggestions for code modifications. Instead of then proceeding with the usual procedure interleaving manual changes and post hoc verification (using testing or more heavyweight techniques), structured reengineering consists of applying sequences of *proven-correct program transformation rules* to the legacy system. Such rules consist of a schematic description of code fragments on which they are applicable and of a set of preconditions. Each rule has been *certified* in advance to guarantee that *for all applicable input programs*, an application of the rule results in a semantically equivalent program. Such proofs only have to be conducted *once* per rule; whenever applying it to a concrete program matching the schematic description, one merely has to show that the preconditions are satisfied. Additionally, a rule can be equipped with nonfunctional guarantees, for instance, on the execution time of the transformed result or with an informal description of its intention.

This approach has the following advantages:

1. It guarantees the preservation of the business logic contained in the original legacy system, even in the absence of domain experts and documentation.
2. The preconditions which have to be checked prior to the execution of the transformation are local to the transformed piece of code. This is especially beneficial during applications in huge (method/class/module) contexts.
3. If the applied rules guarantee additional (nonfunctional) properties on top of semantic preservation, such properties of the resulting system are obtained on the fly. This also helps to assure that the transformation process proceeds toward the reengineering goals, and to select rules accordingly.

To be practical, structured reengineering requires *catalogs of transformation rules* as well as *tool support*, such as the integration into an IDE. While this seems to be an obstacle, it turns out that documentation and tool support already exist for a popular class of code transformations, namely *refactoring* rules. Refactoring aims to improve the internal structure of code while maintaining its semantics [18]; the goal is to make the code better to read and maintain by humans. Fowler published a well-organized catalog of refactoring techniques [17, 18]. Many of them, such as *Extract Method* and *Slide Statements*, are implemented in major IDEs (e.g., Eclipse or IntelliJ IDEA). The intention of each refactoring technique is thoroughly documented and motivated along examples. However, the set of preconditions necessary for semantic preservation is almost always incomplete, and naively using refactorings offered by an IDE can easily *break the input program* [14, 44]. We discuss this in Sect. 4.

To illustrate the application of structured reengineering, we consider an example from [18] computing the total salary and average age of the employees of a company. The corresponding code is depicted in Fig. 1, Listing 1. Observe that the loop in Lines 4–7 performs two different tasks: it (1) computes the average age, operating on the variable `avgAge` and reading the `age` field of the elements in the `people` array, and (2) computes the total salary, operating on the variable `totalSalary` and reading the `salary` field of the elements in the `people` array. The *Split Loop*

Listing 1 Original program.

```
1 People[] people = employees();
2 float avgAge = .0;
3 int totalSalary = 0;
4 for (People person: people) {
5   avgAge += person.age;
6   totalSalary += person.salary;
7 }
8 avgAge = avgAge / people.length;
```

Listing 2 After *Split Loop* and *Slide Statements*.

```
1 People[] people = employees();
2 float avgAge = .0;
3 int totalSalary = 0;
4 for (People person: people)
5   avgAge += person.age;
6 avgAge = avgAge / people.length;
7 for (People person: people)
8   totalSalary += person.salary;
```

Fig. 1 Computing average age and total salary for a list of employees

refactoring technique [18] suggests to split a loop doing two different things into two loops (see Fig. 1, Listing 2). If code readability is a chosen reengineering goal, it makes sense to apply this technique. Additionally, this makes it easier to apply further optimizations, such as converting the loops to map-reduce transformations using (parallel) streams.

Program transformations can be expressed as input-output pairs of programs containing placeholders representing arbitrary expressions or statements. After instantiating the placeholders s.t. the input program matches the program fragment to transform, the original fragment is replaced by the output program with the same instantiations. Thus, the *Split Loop* transformation can be expressed as

$$\texttt{for}\ (\mathit{itExpr})\ \{\ P\,Q\ \}\ \rightsquigarrow\ \texttt{for}\ (\mathit{itExpr})\ \{\ P\ \}\ \ \texttt{for}\ (\mathit{itExpr})\ \{\ Q\ \}$$

However, this simple rule comes with strings attached. The most common obstacle is that locations written and accessed by statements P and Q are overlapping; in this case, the loop must not be split. Apart from preconditions on written and accessed locations, abrupt completion has to be regarded: P and Q might complete abruptly because of a (labeled) **`break`** and **`continue`**, a **`return`**, or because of a thrown exception. The latter also applies for *itExpr*. In Sect. 4, we present certified preconditions for refactoring techniques which we derived using a refinement loop based on feedback from failed proof attempts. For our example, Listing 2 shows the result of the application of *Split Loop* on Listing 1. In addition, we also moved the division in Line 8 from Listing 1 to before the second loop using a *Slide Statements* refactoring. This keeps together the code concerned with the computation of the average age, which one could now extract into a new method using an *Extract Method* refactoring. Those techniques are also discussed in Sect. 4.

A critical part in the development of transformation rule catalogs is the effort that needs to be invested to prove the desired properties of the rules. Formalizing and proving the rules in interactive proof assistants like Isabelle [37] or Coq [8] requires substantial effort for manually writing proof scripts, as can be observed in the work on verified compilers [29, 31, 51]. Previously proposed approaches for proving program transformations *automatically* (e.g., [9, 20, 33, 49]) are tailored to a particular target application (e.g., symbolic execution rules or peephole optimizations) and have a limited expressiveness (e.g., only abstract statements *or* expressions, no loops, no specification of written and accessed locations).

We developed *Abstract Execution (AE)* [47, 50], a specification and verification framework based on Symbolic Execution (SE). AE trades off the flexibility of interactive proof assistants with the automation offered by specialized techniques: it restricts the properties of transformations that can be proved to *universal*, *behavioral* properties on statement level (no existential quantifiers, no reasoning about internal structure of placeholders, no transformations on class level), and offers an *expressive specification language* for describing represented concrete programs. The result is a system in which *many meaningful transformations* (such as all refactoring rules in Sect. 4) can be proven *fully automatically*.

We implemented AE for the JAVA language as an extension of the KeY program prover [2]. As KeY allows manual inspection of generated proof trees, we can draw feedback from failed proof attempts that lets us further refine our transformation models in an iterative manner. To further increase the usability of our approach, we developed REFINITY [48] (available at http://www.key-project.org/REFINITY), presented in Sect. 3. REFINITY offers editor support dedicated to developing transformation rules in AE and greatly simplifies the interaction with the KeY system.

To kick-start the creation of a catalog of transformation rules suitable for structured reengineering, we formalized and proved three types of transformation rules:

1. We prove nine statement-level *refactoring rules* from Fowler's textbooks [17, 18], including two with loops (Sect. 4). In all cases, full semantic equivalence of programs before and after the transformation is proved.
2. Three rules for *restructuring sequential code for subsequent parallelization* [22] are discussed in Sect. 5. This comprises two complex loop transformations, for which we still achieve more than 99.7% proof automation. We prove semantic equivalence and further properties needed for parallel execution of the code.
3. In Sect. 6, we derive results about the *cost impact* of seven loop transformation rules, interacting with an external cost analysis tool. The resulting *Quantitative* AE framework [4] is the first approach to—fully automatically—analyze and certify the cost of schematic programs, and the first to reason about the cost impact of transformation *rules*.

Structure of This Chapter

We explain our Abstract Execution framework in Sect. 2. The REFINITY workbench for modeling and proving JAVA code transformation rules is presented in Sect. 3. Our applications on refactoring rules, restructuring for parallelization, and cost analysis of transformations are presented in Sects. 4–6. We conclude and mention promising future work in Sect. 7.

The contents of this chapter are derived from two main chapters of the PhD thesis of the author [47, Chapters 4 and 6], the original publication on Abstract Execution [50], and from three follow-up publications [4, 22, 48].

2 Abstract Execution

Abstract programs contain schematic placeholders representing many concrete statements or expressions. They naturally occur in many areas of computer science. Program synthesis [42, 46], for example, can be phrased as "show that there *exists* an instantiation of a *scaffold* satisfying my desired property." The scaffold is an abstract program, and the instantiation the result of the synthesis process. Consider, for example, the abstract program

```
int f(int[] a) { P return a[0]; }
```

containing an abstract statement P. To synthesize a version of method `f` returning the smallest element of array `a` (assuming that `a` is non-empty), we can *instantiate* P with "`Arrays.sort(a);`", and thus prove that there is a concrete program matching the scaffold and satisfying the desired specification.

Universal properties quantifying over programs are traditionally proven by *structural induction* [10]. Early work relied on pen-and-paper proofs [32, 36]. Recently, interactive theorem provers are used to mechanize correctness proofs, for example, in the CompCert verified compiler [31] or in the verification of the seL4 microkernel [28]. To prove, for example, that after executing "**int** `x=1;` P" the value of `x` is 1, one should differentiate the cases of P being an assignment, conditional statement, loop, etc. In the case of industrial programming languages like JAVA, this approach quickly gets out of hand. The core idea of Abstract Execution [47, 50] is that it does not matter what *kind of statement* P is instantiated with, as long as it *does not write to variable* `x` and *completes normally* (e.g., does not throw an exception). In other words, we are concerned with the observable *behavior* of the possible instantiations.

We implemented this kind of reasoning as an extension of *Symbolic Execution* [6, 10, 27]. The principle of SE is to explore *all paths in a program* by replacing concrete inputs by symbols. Whenever the execution depends on the concrete value of an input (as for an **if** statement with symbolic guard), SE performs a case distinction and follows each path separately. During the execution, assignments are tracked in a *symbolic store*, while case distinctions update the *path condition*. A *symbolic state* is a pair $(PC, Store)$ consisting of a path condition PC and a symbolic store $Store$. For example, executing an **if** statement with condition "`a>=0`" creates two successor states, one where $\mathtt{a} \geq 0$ is added to the path condition, and one where $\mathtt{a} < 0$ is added. A symbolic store is, in the simplest case, a sequence of assignments such as $(\mathtt{a} := \mathtt{b}, \mathtt{b} := -\mathtt{b})$. This represents all (infinitely many) states where `a` attains the initial value of `b`, and `b` attains its negated initial value. In a state $(PC, Store)$, all right-hand sides in $Store$ are constrained by the conditions in PC: the state $(\{\mathtt{b} < 0\}, (\mathtt{a} := \mathtt{b}, \mathtt{b} := -\mathtt{b}))$ represents all concrete states where `a` attains the initial, *negative* value of `b`, and `b` the *positive* inverted value. For example, the concrete states $\mathtt{a} \mapsto -1, \mathtt{b} \mapsto 1$ and $\mathtt{a} \mapsto -17, \mathtt{b} \mapsto 17$ are elements of this set of *concretizations*. We refer to [47, Chapter 3] for a discussion of SE and its semantics.

But how can we symbolically execute an *abstract* statement or expression? Our solution (1) reflects the abstractness on the level of symbolic stores and path conditions, and (2) performs a case distinction for all completion modes (normal completion, abrupt completion due to a thrown exception, **break**, etc.) in separate SE branches.

The central concept are *abstract (store) updates*. If an abstract statement P depends on a set of locations *footprintP* and may write to a set of locations *frameP*,[1] executing P in a state $(PC, Store)$ leads, for the case of normal completion, to a state

$$(PC, Store \circ \mathcal{U}_P(frameP :\approx \texttt{\textbackslash value}(footprintP)))$$

where $\mathcal{U}_P(frameP :\approx \texttt{\textbackslash value}(footprintP))$ is the abstract update and $\circ$ a *concatenation operator* for symbolic stores. The semantics of the abstract update is a collection of nonabstract updates with left-hand sides in *frameP*, and all right-hand sides depend at most on the values of locations in *footprintP*. Thus, an abstract store containing abstract updates represents up to infinitely many nonabstract stores, each representing up to infinitely many concrete assignments. The concatenation is resolved by interpreting the locations in *footprintP* in the context of *Store*; assignments by the abstract update of locations in *frameP* overwrite already existing ones in *Store*.

Abstract Execution provides a versatile specification framework allowing to precisely and, at the same time, abstractly specify assigned and read locations, and conditions for abrupt completion. Moreover, one can define functional assertions on the resulting symbolic state. Subsequently, we expound our specification language in Sect. 2.1. In Sect. 2.2, we show how to symbolically execute abstract programs.

2.1 Specifying Abstract Programs

AE extends the JAVA language with two keywords: "**\abstract_statement** *P*;" and "**\abstract_expression** *Type e*;" for declaring an *abstract statement* with identifier P and an *abstract expression* of type *Type* with identifier e, respectively. *We use the term Abstract Program Element (APE) to refer to both abstract statements and expressions.* If two APEs with the *same identifier* appear in a program (or a context of two programs), they are assumed to have the same semantics. This is especially useful for expressing transformations: two APEs on the left- and right-hand sides of a transformation rule represent the same concrete programs.

To specify frame, footprint, and abrupt completion behavior of APEs, we use JAVA Modeling Language (JML) [2, 30] specification comments starting with "@". For example, the abstract statement

```
//@ assignable x, y;
//@ accessible \nothing;
\abstract_statement P;
```

[1] We adopt the convention to call locations changed by a program its *frame* and locations that may be read by a program its *footprint* (see, e.g., [2, 25]).

Table 1 Problematic cases for *Slide Statements*

	Statement 1	Statement 2	Counterexample		
			Inputs	Result before	Result after
(1)	`x++;`	`y = 2*z;`	—		
(2)	`x++;`	`y = 2*x;`	$x = 1, y = 1$	$x = 2$, $y = 4$	$x = 2$, $y = 2$
(3)	`x++;`	`x = y;`	$x = 1, y = 1$	$x = 1$, $y = 1$	$x = 2$, $y = 1$
(4)	`x = 20 / y;`	`y = 5;`	$x = 1, y = 0$	$x = 1$, $y = 0$, **Exc.**	$x = 4$, $y = 5$
(5)	`x = 20 / 0;`	`y = 5;`	$x = 1, y = 0$	$x = 1$, $y = 0$, **Exc.**	$x = 1$, $y = 5$, **Exc.**

represents all concrete statements assigning *at most* variables `x` and `y`, while having no access to the state at all. Thus, "`x=17;`" is an instance of P, while "`x=y;`" is not (it reads `y`). There also exists the option to *enforce* some assignment of a location: "**`assignable`** `x,` **`\hasTo`**`(y);`" specifies that `y` *has to* be assigned. Then "`x=17;`" would no longer be an instance of P (but, e.g., "`x=17; y=17;`" is).

When modeling transformation rules, *concrete* frames and footprints (such as `x` and `y`) are usually insufficient: a transformation should be applicable regardless of the names (and numbers) of variables and fields available in a particular context. We leverage the *theory of dynamic frames* [25] to that end. A dynamic frame is a set-valued specification variable representing an *unknown set of concrete locations*. Instead of enumerating variables in a frame as we did above, a placeholder symbol is used, for example, *frameP* or *footprintP*. If needed, the meaning of these symbols can be refined in specifications using usual set operations. For example, one can specify $\mathtt{x} \in \mathit{frameP}$, or $\mathit{frameP} \subseteq \mathit{frameQ}$. These operations have corresponding representatives in JML; for simplicity, we continue using the mathematical notation.

We explain how to express constraints about frames and footprints along a refactoring technique from Sect. 4: *Slide Statements*, which we already mentioned in the introduction. The purpose of this refactoring is to bring statements closer together which participate in a common task, contributing to better understandability and preparing for subsequent refactorings like *Extract Method*.

To motivate the constraints which we are going to define, we present some example cases with counterexamples in Table 1. The two statements in Line 1 of the table can be swapped without problems: Their behavior is fully independent. In Line 2, the frame of the first statement intersects with the footprint of the second statement. Thus, applying *Slide Statements* leads to different results for variable `y` before and after the application: *frames and footprints of both statements must be disjoint*. This is not enough: a later occurring statement can overwrite changes of an earlier one. An example for this is shown in Line 3, where the overall result equals the result of the statement occurring last: *the frames of both statements must be disjoint*.

Figure 2 depicts an abstract program model for the *Slide Statements* refactoring including the two constraints on frames and footprints. The abstract statements `A` and `B` from the input model (Listing 3) occur in reverse order in the output

Listing 3 Input Model

```
1 /*@ ae_constraint
2   @    frameA ∩ footprintB = ∅  &&
3   @    frameB ∩ footprintA = ∅  &&
4   @    frameA ∩ frameB = ∅;  */
5
6 //@ assignable frameA;
7 //@ accessible footprintA;
8 \abstract_statement A;
9
0 //@ assignable frameB;
1 //@ accessible footprintB;
2 \abstract_statement B;
```

Listing 4 Output Model

```
/*@ ae_constraint
  @    frameA ∩ footprintB = ∅  &&
  @    frameB ∩ footprintA = ∅  &&
  @    frameA ∩ frameB = ∅;  */

//@ assignable frameB;
//@ accessible footprintB;
\abstract_statement B;

//@ assignable frameA;
//@ accessible footprintA;
\abstract_statement A;
```

Fig. 2 Abstract program model for *Slide Statements* ignoring abrupt completion

model (Listing 4). Both abstract statements are annotated with dynamic frame symbols *frameA*/*footprintA* and *frameB*/*footprintB*, respectively, for their frames and footprints. Our constraints on them are implemented using an **`ae_constraint`** declaration in Lines 1–4: the constraints in Lines 2 and 3 enforce disjointness of frames and footprints, while the constraint in Line 4 enforces disjointness of the frames. Assuming normal completion, AE can prove these constraints sufficient.

In the JAVA language, we however have to take abrupt completion (here, exceptions and **`return`**s) into account. Line 4 in Table 1 shows an example where the order of the statements determines not only the result values of the considered variables but also the resulting *completion mode* (exception vs. normal completion) for the shown test case. These statements would, though, anyway not be amenable for *Slide Statements* since the requirement on the disjointness of frames and footprints is not met. In the scenario shown in Line 5, frames and footprints *are* disjoint; the outcome is still different before and after transformation, since early exceptional termination of the first statements impedes the assignment of 5 to `y` before, but not after the swap. This leads us to the additional requirement for semantic preservation in the presence of abrupt completion: *if one statement completes abruptly, the assignments by the other statement must not be "relevant."* If we are not concerned about the final value of variable `y`, we may reorder these statements. Furthermore, *abrupt completion must be mutually exclusive*, because otherwise (1) the assignments by *both* statements would have to be irrelevant, and (2) there could be different *reasons* for abrupt completion (**`return`** vs. thrown exception, different thrown exception types/objects).

Figure 3 shows the extension of the *Slide Statements* transformation model considering exceptions. For space reasons, we ignore **`return`**s here; the full model is available in [47, Appendix E]. If no conditions on abrupt completion behavior of APEs is provided, as in Fig. 2, AE automatically explores all paths corresponding to each possible completion mode. In Fig. 3, we bind exceptional behavior, using the "**`exceptional_behavior requires`** `...;`" keyword, to uninterpreted predicates *throwsExcA*/*throwsExcB* (Lines 15–16 and 21–22). The

Listing 5 Input Model

```
 1 /*@ ae_constraint
 2   @   frameA ∩ footprintB = ∅ &&
 3   @   frameB ∩ footprintA = ∅ &&
 4   @   frameA ∩ frameB = ∅ &&
 5   @   \mutex(
 6   @      throwsExcA(\value(footprintA)),
 7   @      throwsExcB(\value(footprintB))) &&
 8   @   (throwsExcA(\value(footprintA))
 9   @      ⟹ frameB ∩ relevant = ∅) &&
10   @   (throwsExcB(\value(footprintB))
11   @      ⟹ frameA ∩ relevant = ∅); */
12
13 //@ assignable frameA;
14 //@ accessible footprintA;
15 /*@ exceptional_behavior requires
16   @   throwsExcA(\value(footprintA)); */
17 \abstract_statement A;
18
19 //@ assignable frameB;
20 //@ accessible footprintB;
21 /*@ exceptional_behavior requires
22   @   throwsExcB(\value(footprintB)); */
23 \abstract_statement B;
```

Listing 6 Output Model

```
/*@ ae_constraint
  @   frameA ∩ footprintB = ∅ &&
  @   frameB ∩ footprintA = ∅ &&
  @   frameA ∩ frameB = ∅ &&
  @   \mutex(
  @      throwsExcA(\value(footprintA)),
  @      throwsExcB(\value(footprintB))) &&
  @   (throwsExcA(\value(footprintA))
  @      ⟹ frameB ∩ relevant = ∅) &&
  @   (throwsExcB(\value(footprintB))
  @      ⟹ frameA ∩ relevant = ∅); */

//@ assignable frameB;
//@ accessible footprintB;
/*@ exceptional_behavior requires
  @   throwsExcB(\value(footprintB)); */
\abstract_statement B;

//@ assignable frameA;
//@ accessible footprintA;
/*@ exceptional_behavior requires
  @   throwsExcA(\value(footprintA)); */
\abstract_statement A;
```

Fig. 3 Model for *Slide Statements* with exceptions, extensions highlighted in gray

intended meaning is "the abstract statement `A`/`B` throws an exception *if, and only if*, the predicate *throwsExcA*/*throwsExcB* holds." Since whether or not a statement throws an exception is usually determined by its footprint (e.g., "`x/y`" throws an arithmetic exception iff `y` is zero), we define these predicates parametric in the values of the footprints: the dynamic frame *footprintA* represents a set of *locations*, while the expression **`\value`**(*footprintA*) represents the *values* of this set. Analogously to our usage of dynamic frame specification variables, we can now use these predicates in constraints (Lines 5–11 in Fig. 3). In Lines 5–7, we stipulate that *either* statement `A` *or* statement `B` may throw an exception (but not both) using the "**`\mutex`**" keyword. Assuming an abstract location set *relevant* representing relevant locations, we subsequently impose that if one statement completes abruptly due to a thrown exception, the frame of the other statement only contains *irrelevant* locations (Lines 8–11).

Our implementation of AE automatically *proves* semantic equivalence of the full model (including the specifications for **`return`**s, which work similarly to the ones for exception) in less than 20 s—*once and for all*, for all input programs satisfying the specified constraints. Given such a proof, we can be sure that our specified constraints are *sufficient for a safe application of the refactoring rule*.

In addition to "**`exceptional_behavior`**", the keywords "**`return_behavior`**" for abrupt completion due to a **`return`** and, for loop bodies, "**`break_behavior`**" for **`break`**s and "**`continue_behavior`**" for **`continue`**s are supported. For labeled **`break`**s, one writes "**`break_behavior`** (*lbl*)" (similarly for labeled **`continue`**s). Moreover, "**requires**" can be replaced by "**ensures**" to specify a *post*condition on the resulting state. While this is rarely needed for transformation rules, it can be useful, for example, in incremental, "correct-by-construction" program development.[2]

The specification framework we have just presented is designed with real applications *and* automatic proofs in mind. In Sects. 4–6, we demonstrate that it is strong enough to model interesting transformations from different areas of code optimization. Next, we introduce our Symbolic Execution rules for AE.

2.2 Symbolic Execution of Abstract Program Elements

Symbolic execution of an APE results in separate execution branches for each possible abrupt completion mode. Figure 4a visualizes the relevant part of the execution tree for an abstract statement (we omitted labeled **break**s and **continue**s). Assuming that we start in a symbolic state $(PC, Store)$, we produce symbolic output states s_{normal} for normal completion, s_{exc} for abrupt completion due to a thrown exception, and so on. In the figure, symbolic states are input states for the execution of the annotated statement. The variables `res` and `exc` of types `Object` and `Throwable`, respectively, are created fresh, that is, they are not declared anywhere else in the context program. The execution tree for an abstract expression (Fig. 4b) looks similarly, but as expressions can only complete normally or due to a thrown exception, an abstract expression node only has two successors. We point out that also expressions can have a non-empty frame, as in "`x > y++`".

In the remainder of this section, we show how the states s_{normal}, s_{exc} etc. are formally constructed. This is an important part of our framework, but not required for understanding the sections that follow. Readers not interested in the formalism can therefore safely skip to Sect. 3, where we describe how to use AE in practice.

Our central concept is the *abstract update*: for an APE `P` with frame *frameP* and footprint *footprintP*, an abstract update $\mathcal{U}_{\mathtt{P}}(\mathit{frameP} :\approx \mathbf{\backslash value}(\mathit{footprintP}))$ has the same effect on the state as `P`. However, while `P` may complete abruptly (e.g., throw an exception), the abstract update *always completes normally*.

To express the symbolic input states for successors of APEs in the execution tree, we need to *apply* symbolic stores to formulas: we write $[\![\mathit{Store}]\!]\mathit{formula}$ for the transformation of *formula* according to *Store*. For example, $[\![\mathtt{x} := 17]\!]\mathtt{x} \geq 0$ is equivalent to $17 \geq 0$ and thus to true.

Subsequently, we define the symbolic states resulting from the execution of an abstract statement "**\abstract_statement** `P;`" in the symbolic state $(PC, Store)$. We write *normalPost*, *excPre*, *excPost*, etc., for the pre- and postconditions specified using JML "**requires**" and "**ensures**" clauses in the scope of the respective behavior ("**normal_behavior**", "**exceptional_behavior**", etc.).

[2] The use of AE for Correctness-by-Construction has been explored in a Master's thesis [53].

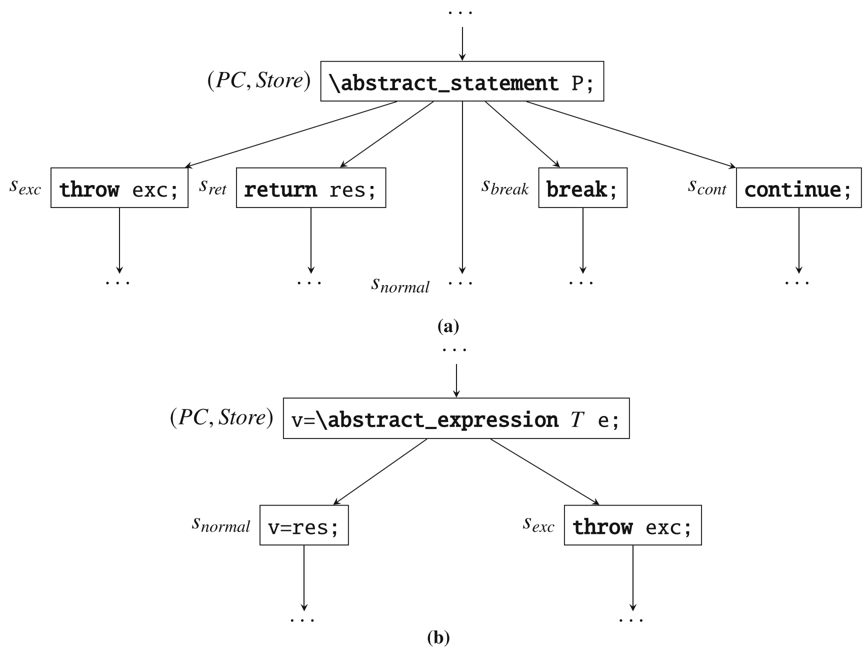


Fig. 4 Execution tree fragments for abstract statements and abstract expressions. Symbolic input states for tree nodes ($(PC, Store)$, s_{exc}, etc.) are annotated on the left-hand side of the nodes. (**a**) Execution tree fragment for an abstract statement. (**b**) Execution tree fragment for an abstract expression

Normal Completion

We use the abbreviation *completesNormally* to denote the negated disjunction of the preconditions for all other completion modes: $\mathit{completesNormally} \iff \neg(\mathit{excPre} \vee \mathit{returnPre} \vee \mathit{breakPre} \vee \mathit{continuePre})$. This precondition is evaluated in the initial store *Store*, while the postcondition *normalPost* is evaluated in the resulting store after the application of the abstract update (and can thus represent constraints on the resulting state). In case the abstract statement is specified to *always complete abruptly* in *Store*, the precondition $[\![\mathit{Store}]\!](\mathit{completesNormally})$ evaluates to false. Then, this symbolic execution branch becomes infeasible and is not followed any further. This holds similarly for all other branches discussed below.

Formally, we define the symbolic state s_{normal} as

$$\begin{aligned}
s_{normal} := (PC \cup \Big\{ & [\![\mathit{Store}]\!](\mathit{completesNormally}), \\
& [\![\mathit{Store} \circ \mathcal{U}_{\mathtt{P}}(\mathit{frameP} :\approx \texttt{\textbackslash value}(\mathit{footprintP}))]\!](\mathit{normalPost})\Big\}, \\
& \qquad \mathit{Store} \circ \mathcal{U}_{\mathtt{P}}(\mathit{frameP} :\approx \texttt{\textbackslash value}(\mathit{footprintP})))
\end{aligned}$$

Abrupt Completion Due to a Thrown Exception

The state s_{exc} for completion due to a thrown exception is constructed similarly to s_{normal}, just with the appropriate pre- and postconditions. There is one addition: we initialize the exception variable `exc` (see Fig. 4a) to a value $exc^{\mathtt{P}}(\mathtt{\backslash value}(footprint))$, where "$exc^{\mathtt{P}}$" is a unary function symbol exclusively introduced in symbolic execution of the abstract statement `P`: it is generated *fresh* when first encountering `P`, but is *re-used* whenever another occurrence of `P` is executed. We follow the same procedure for abstract update symbols $\mathcal{U}_{\mathtt{P}}$. This is especially useful for equivalence proofs, since abstract statements with the same identifiers exhibit the same behavior—*when executed in the same state*. It is thus essential to pass the current value of `P`'s footprint to the $exc^{\mathtt{P}}$ function, since `P` might throw different exception objects for different initial execution states. The variable `exc` can be referred to in the postcondition *excPost* to constrain the values of the thrown exception object.

$$
\begin{aligned}
s_{exc} := \Big(PC \cup \Big\{&\llbracket Store \rrbracket(\ excPre\ ),\\
&\llbracket Store \circ \mathcal{U}_{\mathtt{P}}(frameP :\approx \mathtt{\backslash value}(footprintP)) \circ\\
&\quad (\mathtt{exc} := exc^{\mathtt{P}}(\mathtt{\backslash value}(footprint)))\ \rrbracket(\ excPost\ )\Big\},\\
&\qquad Store \circ \mathcal{U}_{\mathtt{P}}(frameP :\approx \mathtt{\backslash value}(footprintP)) \circ\\
&\qquad\qquad (\mathtt{exc} := exc^{\mathtt{P}}(\mathtt{\backslash value}(footprint)))\ \Big)
\end{aligned}
$$

Abrupt Completion Due to a Returned Result

The symbolic state s_{ret} for a **`return`** statement of a result is constructed exactly the same way as s_{exc} for a thrown exception. If the abstract statement is executed in the context of a `void` method, the initialization of the `res` variable is omitted (and the execution tree contains a "**`return`**`;`" instead of a "**`return`** `res;`" node). As in the exceptional case, the postcondition *returnPost* can refer to the `res` variable to characterize the returned result (in the context of a non-`void` method).

$$
\begin{aligned}
s_{ret} := \Big(PC \cup \Big\{&\llbracket Store \rrbracket(returnPre),\\
&\llbracket Store \circ \mathcal{U}_{\mathtt{P}}(frameP :\approx \mathtt{\backslash value}(footprintP)) \circ\\
&\quad (\mathtt{res} := result^{\mathtt{P}}(\mathtt{\backslash value}(footprint)))\rrbracket(returnPost)\Big\},\\
&\qquad Store \circ \mathcal{U}_{\mathtt{P}}(frameP :\approx \mathtt{\backslash value}(footprintP)) \circ\\
&\qquad\qquad (\mathtt{res} := result^{\mathtt{P}}(\mathtt{\backslash value}(footprint)))\Big)
\end{aligned}
$$

Abrupt Completion Due to a Break or Continue Statement
The state s_{break} for abrupt completion due to a **break** looks exactly like s_{normal} for normal completion with different pre- and postconditions, as initialization of a returned or thrown object is not required. If the abstract statement occurs outside any loop, we omit this state and the corresponding node in the execution tree.

$$\begin{aligned} s_{break} := \Big(PC \cup \big\{&[\![Store]\!](breakPre), \\ &[\![Store \circ \mathcal{U}_{\mathrm{P}}(frameP :\approx \texttt{\textbackslash value}(footprintP))]\!](breakPost)\big\}, \\ &\qquad Store \circ \mathcal{U}_{\mathrm{P}}(frameP :\approx \texttt{\textbackslash value}(footprintP))\Big)\end{aligned}$$

The **continue** case is analogous. For *abstract expressions*, s_{normal} has a simpler *completesNormally* condition (i.e., $\neg excPre$) and contains an initialization of the `res` variable similar to s_{ret} for abstract statements; s_{exc} is identical.

To reason about the symbolic states arising from the execution of APEs, we provide a number of simplification and normalization rules for symbolic stores with abstract updates. We refer to [47, Sec. 4.3.2] for their full definitions. We demonstrate some of these rules in the following example.

Example 1 (Execution of the Slide Statements Model) We investigate the “normal completion” case of the *output* model for the *Slide Statements* refactoring from Listing 6, Fig. 3 on Page 206. The final state after executing the abstract statements is

$$\begin{aligned}\Big(\{&frameA \cap footprintB = \emptyset,\ frameB \cap footprintA = \emptyset,\ frameA \cap frameB = \emptyset, \\ &mutexFormula, \\ &\neg throwsExcB(\texttt{\textbackslash value}(footprintB)), \\ &\neg[\![\mathcal{U}_{\mathrm{B}}(frameB :\approx \texttt{\textbackslash value}(footprintB))]\!]throwsExcA(\texttt{\textbackslash value}(footprintA))\}, \\ &(\mathcal{U}_{\mathrm{B}}(frameB :\approx \texttt{\textbackslash value}(footprintB)) \circ \mathcal{U}_{\mathrm{A}}(frameA :\approx \texttt{\textbackslash value}(footprintA)))\Big)\end{aligned}$$

For simplicity, we omit conditions on abrupt completion due to other reasons than a thrown exception. The first four elements in the path condition stem from the **ae_constraint** specification in Lines 1–11 in Listing 6, where *mutexFormula* abbreviates the condition on mutual exclusion of abrupt completion (Lines 5–11).

First, we apply the abstract update $\mathcal{U}_{\mathrm{B}}(\dots)$ to the *throwsExcA* expression in the path condition and resolve the update concatenation in the store by the rule transforming $(Store_1 \circ Store_2)$ to $(Store_1,\ [\![Store_1]\!]Store_2)$, resulting in

$$\begin{aligned}\Big(\{&frameA \cap footprintB = \emptyset,\ frameB \cap footprintA = \emptyset,\ frameA \cap frameB = \emptyset, \\ &mutexFormula, \\ &\neg throwsExcB(\texttt{\textbackslash value}(footprintB)), \\ &\neg throwsExcA([\![\mathcal{U}_{\mathrm{B}}(frameB :\approx \texttt{\textbackslash value}(footprintB))]\!]\texttt{\textbackslash value}(footprintA))\},\end{aligned}$$

$$(\mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB})),$$
$$[\![\mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB}))]\!](\mathcal{U}_{\mathtt{A}}(\mathit{frameA} :\approx \texttt{\textbackslash value}(\mathit{footprintA}))))\big)$$

Next, we simplify the application $[\![\mathcal{U}_{\mathtt{B}}(\dots)]\!]\mathcal{U}_{\mathtt{A}}(\dots)$ in the store to

$$\mathcal{U}_{A}(\mathit{frameA} :\approx [\![\mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB}))]\!](\texttt{\textbackslash value}(\mathit{footprintA}))).$$

The expression $[\![\mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB}))]\!](\texttt{\textbackslash value}(\mathit{footprintA}))$ occurs two times in the resulting state: once in the path condition and once in the symbolic store. Since the path condition contains the assumption $\mathit{frameB} \cap \mathit{footprintA} = \emptyset$, we know that the assignments due to $\mathcal{U}_{\mathtt{B}}(\dots)$ are irrelevant for $\texttt{\textbackslash value}(\mathit{footprintA})$; we deduce that we can drop $[\![\mathcal{U}_{\mathtt{B}}(\dots)]\!]$ and obtain the much simpler state

$$\big(\{\mathit{frameA} \cap \mathit{footprintB} = \emptyset,\ \mathit{frameB} \cap \mathit{footprintA} = \emptyset,\ \mathit{frameA} \cap \mathit{frameB} = \emptyset,$$
$$\mathit{mutexFormula},$$
$$\neg\mathit{throwsExcB}(\texttt{\textbackslash value}(\mathit{footprintB})),\ \neg\mathit{throwsExcA}(\texttt{\textbackslash value}(\mathit{footprintA}))\},$$
$$(\mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB})), \mathcal{U}_{\mathtt{A}}(\mathit{frameA} :\approx \texttt{\textbackslash value}(\mathit{footprintA})))\big)$$

There is one more rule we are going to apply; this time, it is not a simplification but a *normalization* rule. The state we obtain by executing the *input* model for *Slide Statements* in Listing 5 equals the one from above for the output model, with one exception: the order of the abstract updates in the store. Indeed, we cannot simply swap elements of a store, since a later element might overwrite assignments by an earlier one. However, we know from the path condition that $\mathit{frameA} \cap \mathit{frameB} = \emptyset$. Therefore, we can apply a normalization rule reordering abstract updates according to the lexicographic order of their identifiers. Our final result is

$$\big(\{\mathit{frameA} \cap \mathit{footprintB} = \emptyset,\ \mathit{frameB} \cap \mathit{footprintA} = \emptyset,\ \mathit{frameA} \cap \mathit{frameB} = \emptyset,$$
$$\mathit{mutexFormula},$$
$$\neg\mathit{throwsExcB}(\texttt{\textbackslash value}(\mathit{footprintB})),\ \neg\mathit{throwsExcA}(\texttt{\textbackslash value}(\mathit{footprintA}))\},$$
$$(\mathcal{U}_{\mathtt{A}}(\mathit{frameA} :\approx \texttt{\textbackslash value}(\mathit{footprintA})), \mathcal{U}_{\mathtt{B}}(\mathit{frameB} :\approx \texttt{\textbackslash value}(\mathit{footprintB})))\big)$$

This state is equivalent to the final state for the input model (as the path condition is a set, the order of path constraints is irrelevant). Assuming that this also holds for the abrupt completion cases, we conclude the equivalence of the input and output models for *Slide Statements*, and follow that applying this transformation is semantics-preserving for all concrete instantiations satisfying the constraints we formalized.

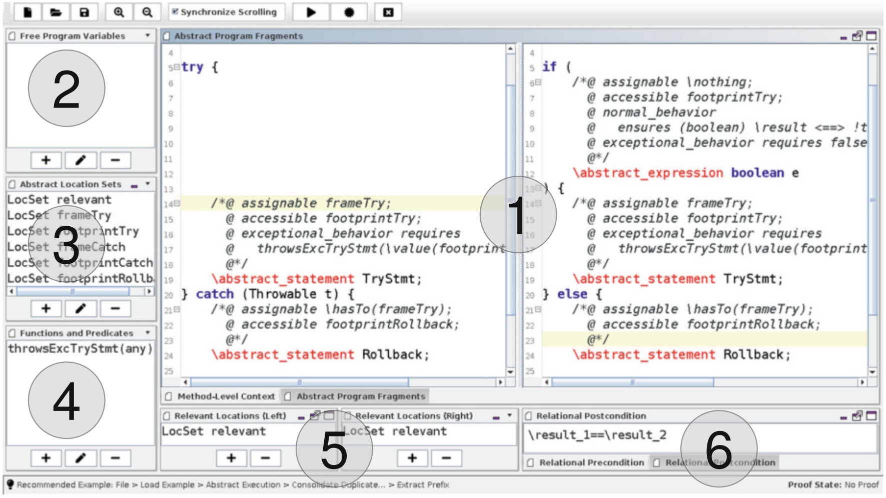
3 The REFINITY Workbench

We implemented AE by extending the KeY [2] program prover. The extension consists of ~5.5k lines of JAVA and ~400 lines of *Taclet* code. Taclets are KeY's language for SE rules, which we had to significantly extend for our AE taclets. More implementation details are available in [47, Sec. 4.4].

Using this implementation, one can reason about programs containing APEs. However, symbols such as dynamic frames and abstract predicates have to be declared in KeY input files separately from the JAVA code. Proof obligations for showing the equivalence of two abstract program fragments (which is the prime application of AE and needed for our case studies described in Sects. 4 and 5) have to be manually defined. Those proof obligations are all structurally similar, but tedious to write from scratch. Additionally, there is no editing support for abstract programs, which require a significant amount of JML specification lines.

Addressing these issues, we developed REFINITY [48], a workbench for specifying and proving properties of statement-level JAVA code transformation rules. A major driver for the development of REFINITY was an invited talk in a tutorial session at iFM'19,[3] where participants without any background in using KeY successfully applied REFINITY to prove the correctness of two refactoring techniques.

Figure 5 shows the REFINITY GUI. Input and output models for a transformation rule are written to the two text fields at marker ① . When editing, one can



```
LocSet relevant
LocSet frameTry
LocSet footprintTry
LocSet frameCatch
LocSet footprintCatch
LocSet footprintRollb
```

```
throwsExcTryStmt(any)
```

```
try {







   /*@ assignable frameTry;
     @ accessible footprintTry;
     @ exceptional_behavior requires
     @   throwsExcTryStmt(\value(footprint
     @*/
   \abstract_statement TryStmt;
} catch (Throwable t) {
   /*@ assignable \hasTo(frameTry);
     @ accessible footprintRollback;
     @*/
   \abstract_statement Rollback;
```

```
if (
   /*@ assignable \nothing;
     @ accessible footprintTry;
     @ normal_behavior
     @   ensures (boolean) \result <==> !t
     @ exceptional_behavior requires false
     @*/
   \abstract_expression boolean e
) {
   /*@ assignable frameTry;
     @ accessible footprintTry;
     @ exceptional_behavior requires
     @   throwsExcTryStmt(\value(footprint
     @*/
   \abstract_statement TryStmt;
} else {
   /*@ assignable \hasTo(frameTry);
     @ accessible footprintRollback;
     @*/
   \abstract_statement Rollback;
```

```
LocSet relevant
```

```
LocSet relevant
```

```
\result_1==\result_2
```

Fig. 5 The REFINITY window

[3] https://ifm2019.hvl.no/refa/#pcrr.

use a number of keyboard shortcuts for creating stubs for APEs and transformation constraints. Field ② contains global program variables which can be referred to in the input and output model. In the compartment labeled ③, we define dynamic frame variables used in the model ("LocSet" is the sort for abstract location sets, i.e., dynamic frames). REFINITY models include by default an additional location set "*relevant*" representing all relevant locations. If we do not impose further constraints, for example, exclude some locations from *relevant*, correctness has to be proven under the assumption that *all* locations are in this set. Abstract predicates (like *throwsExcA*(. . .) in the previous section) and functions are declared in input field ④.

Fields ⑤ and ⑥ specify global assumptions and proof objectives. The effects of the abstract programs specified in field ① are recorded in two *sequences* `\result_1` and `\result_2` for the input and output model. Their elements can be accessed using array syntax, for example, `\result_1[0]`. Positions 0 and 1 are reserved for returned results and thrown exceptions. Starting from position 2, the final values of "relevant locations" declared in fields ⑤ (with the dynamic frame *relevant* being the default) are stored. The standard postcondition, which we see in field ⑥, is `\result_1==\result_2`. Without constraints about *relevant*, this specifies that returned values, thrown exceptions, and the whole memory after termination have to be identical. More fine-grained postconditions can also be specified: for example, if the first relevant location is an integer variable, "`\result_1[2]>2*\result_2[2]`" is admissible. The global "Relational Precondition" (⑥) has access to the *initial* values of free program variables (field ②) and abstract location sets (field ③).

Pressing ▶ transforms the model into a KeY proof obligation and starts the automatic proof search. If KeY reports success, the specified model is correct. Saved proof certificates can be validated against the loaded model using the ✹ button.

During development of a new model, KeY will usually finish unsuccessfully, leaving one or more proof goals open. In rare cases and for highly complicated models, the reason could be that KeY needs more time or is not able to close the proof although the model is valid—we hit a *prover incapacity*. In the latter case, one can try to close the proof by interacting with the prover. More likely, though, are problems in the model. Inspecting the open goals provides feedback on how to refine the model to make it sound. Possible *refinements* include (1) declaring the disjointness of dynamic frames, (2) imposing mutual exclusion on abrupt completion behavior, (3) declaring a functional postcondition for APEs, and (4) refining the relational *post*condition or (4) adding a relational *pre*condition.

The proof obligation REFINITY generates for KeY consists of a JAVA class with two methods `left(...)` and `right(...)` containing the abstract program fragments, and a problem description file containing proof strategy settings, declarations of variable, function, predicate, and abstract location set symbols and the proof goal (expressed in KeY's program logic "JAVA Dynamic Logic" [2]). Such proof goals easily span around 40 lines in concrete syntax.

REFINITY spares the user from having to deal with such technicalities, simplifying the modeling process. It automatically creates the mentioned files, starts a KeY proof with reasonable presets, and displays proof status information in its status bar. Additionally, it supports syntactic extensions unsupported by KeY.

Execution of Abstract 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* are employed to abstract loop behavior. A loop invariant is a specification respected *by every loop iteration*, which can be used to abstract away from the concrete loop regardless of the number of iterations. It is generally hard to come up with suitable invariants; indeed, *specifications* have been identified as the "new bottleneck" of formal verification (see, e.g., [2]). In program equivalence proofs using functional verification techniques, one even needs the *strongest* possible invariant for each occurring loop ([7], [47, Sec. 5.4.2]).

Luckily, there is a way to *generically* specify *abstract* strongest loop invariants which we can use in AE. Assume a loop with guard $g(\mathtt{x})$ operating on a single variable $\mathtt{x}$. The formula $\mathit{Inv}(\mathtt{x})$ is a strongest loop invariant for that loop if it is (1) preserved by every run, and (2) there is *exactly one* value v s.t. $\mathit{Inv}(v)$ holds and $g(v)$ does *not* hold. Condition (2) means that there remains no degree of freedom in the choice of the value of $\mathtt{x}$ after loop termination: *Inv* describes the *exact*, final value. We can rewrite the condition to $\exists v;\ \forall \mathtt{x};\ ((\mathit{Inv}(\mathtt{x}) \land \neg g(\mathtt{x})) \rightarrow \mathtt{x} = v)$.

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. We can add this condition as a precondition in REFINITY and use the abstract invariant formula in our program. We refer to [47, Sec. 6.2] for a full account.

REFINITY can be downloaded at `key-project.org/REFINITY/`. It comes with a number of examples to get started with modeling transformation rules. The next three sections are devoted to particular applications of REFINITY and AE.

4 Correctness of Refactoring Rules

Refactoring aims to change code in a way that does *not alter its external behavior*, yet improves its *internal structure* [18]. In the context of software reengineering, refactoring can contribute to better understandability, and thus to the maintainability of a legacy system. Generally, refactoring is a risky process, especially for poorly understood legacy systems. It has been shown that common refactorings can easily, and accidentally, change a program's behavior [15]. Most refactorings come with preconditions. If those are not met, the transformed program might not compile, or—which is worse—compile, but behave in a different way. Testing contributes to safe refactoring, but can be misleading for insufficiently robust test suites [5].

In contrast to other code transformations, code refactoring is well supported in modern IDEs. Unfortunately, relying on refactoring tools does not automatically guarantee the preservation of program behavior [14, 44], as these tools typically do not implement *all* preconditions [45]. Indeed, documentation of preconditions for safe refactoring in literature is vastly incomplete, as we discovered in our case study.

Using REFINITY [48], we modeled nine statement-level refactoring techniques from [17, 18], including two with loops. In an iterative process, we refined the models by adding additional constraints, until we could prove them semantically equivalent.

Most *practically applied* refactorings are confined to method bodies [43]. However, existing work on correctness of code refactoring almost exclusively regards high-level techniques such as “move field” or “pull up method.” AE and REFINITY thus focus on a significant blind spot by addressing *statement-level* transformations.

By documenting precise preconditions for safe, statement-level refactoring, we contribute insights useful for building more robust refactoring tools. Those can eventually be adopted for structured reengineering of legacy systems.

In the remainder of this section, we outline our derived preconditions for safe applications of the nine considered refactoring techniques. Note that since we prove *equivalence* of input and output models, we simultaneously consider *dual refactoring techniques* (such as *Inline Method* for *Extract Method*). A more detailed account and full abstract program models are provided in [47, Sec. 6.3 and Appendix E].

Slide Statements

We already discussed *Slide Statements* in Sect. 2.1. The idea of this technique is to reorder statements to keep those together which fulfill a common purpose [18]. Let *frameA*/*frameB* and *footprintA*/*footprintB* be the frames and footprints of the participating statements `A` and `B`, respectively. We derived the following preconditions (only the first three of which are documented in [18]):

1. *frameA* and *frameB* have to be disjoint.
2. *frameA* and *footprintB* have to be disjoint.
3. *frameB* and *footprintA* have to be disjoint.
4. `A` may only complete abruptly if `B` completes normally, and vice Versa.
5. if `A` completes abruptly, `B` performs no assignments relevant to the outside, and vice versa.

Consolidate Duplicate Conditional Fragments

This variation of *Slide Statements* consists in moving code that is executed in all branches of a conditional statement to outside that conditional. We modeled four variants: extracting (1) a common *prefix* from an **`if`** statement, (2) a common *postfix* from an `if` statement, (3) a common postfix from a `try-catch` statement to after the `try-catch`, (4) a common postfix from a `try-catch` statement to the `finally` branch of the `try- catch`. The two variants (3) and (4) are distinguished due to an ambiguity in the refactoring's description ("(moving) to the final block" [17]).

Variant (1) comes with the very same preconditions as *Slide Statements*. Variant (2) can be applied without preconditions. The extraction of a postfix from a `try` statement (variant (3)) to after that statement can be applied under the assumption that P does not throw an exception, which also can be deduced from the description in [17]. Furthermore, P must not access the caught exception object. Those restrictions also apply for variant (4). For this variant of the refactoring, schematically

$$\begin{array}{ll} & \texttt{try}\ \{\ Q_1\ P\ \}\ \ \texttt{catch}\ \ (T\ \ \texttt{e})\ \{\ Q_2\ P\ \} \\ \rightsquigarrow & \texttt{try}\ \{\ Q_1\ \}\ \ \texttt{catch}\ \ (T\ \ \texttt{e})\ \{\ Q_2\ \}\ \ \texttt{finally}\ \{\ P\ \} \end{array}$$

we derived the additional, unmentioned precondition that Q_1 must not complete due to a `return`. Otherwise, P would be executed after, but not before the refactoring.

Consolidate Conditional Expression

For the case of sequential or nested conditionals with "the same result" [17], this refactoring proposes to merge these into a single check to improve clarity. The two variants of this technique are schematically represented as

$$\begin{array}{ll|ll} & \texttt{if}\ \ (\mathit{expr}_1)\ \{\ P\ \}\ \texttt{if}\ \ (\mathit{expr}_2)\ \{\ P\ \} & & \texttt{if}\ \ (\mathit{expr}_1)\ \{\ \texttt{if}\ (\mathit{expr}_2)\ \{\ P\ \}\ \} \\ \rightsquigarrow & \texttt{if}\ (\mathit{expr}_1 \parallel \mathit{expr}_2)\ \{\ P\ \} & \rightsquigarrow & \texttt{if}\ \ (\mathit{expr}_1\ \&\&\ \mathit{expr}_2)\ \{\ P\ \} \end{array}$$

Our interpretation of "have the same result" is that P *always* returns or throws an exception (as in all examples in [17]). Thus, P is never executed twice in the sequential case. In the case of nested `if` statements, P can complete *arbitrarily*.

Both variants can be applied *without additional preconditions*. Fowler mentions that conditionals *must not have any side effects*, which is, however, *only* necessary for logical connectors *without short-circuit evaluation* (i.e., "|" and not "||", etc.).

Extract Method, Decompose Conditional, and Move Statements to Callers

Method extraction is a well-known refactoring technique implemented in many IDEs. According to [17], it may be applied if the extracted code does not assign more than one local variable referenced in the outside context. We discovered two additional, unmentioned constraints: (1) The extracted fragment must not return, since this changes control flow. (2) If the newly created method is a *query* and the extracted fragment throws an exception, it must not change the value of the returned query result variable before. For the second precondition, consider the following example:

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

When presented an empty list, the computation of the average in Line 4 will complete abruptly because of a division by 0. Thus, the value of `avg` will be 0 (the sum of no elements) at Line 6 before the refactoring, while its value *after* the transformation is that of the constant `ERROR`: The method `average` completes abruptly and `avg` is not assigned in Line 3. One might argue that this is an improvement; still, it is not semantics-preserving. What is more, as we always prove semantic *equivalence*, the reverse direction *Inline Method* is also covered. In the example, one would introduce a bug when following the reverse direction and inlining method `average`.

Decompose Conditional [17] 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 [18] 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 for *Extract Method* above, we used a **`try`** statement to react to the *expected* behavior that a list can be empty. Instead, we could have *tested* the list for emptiness, and reserved exceptions for *unexpected* behavior. Observe that then, method extraction of the two statements computing the list's average would even have been safe. This observation, on the other hand, suggests that the *Replace Exception with Test* is not generally semantics-preserving, though no restrictions are mentioned in literature. Assume a statement P throws an exception if the condition *cond* holds. *Replace Exception with Test* transforms “**`try`** { P } **`catch`** (...) { Q }” to “**`if`** (!(*cond*)) { P } **`else`** { Q }”. In our first proof attempt, we found the problem that if P throws an exception, it might change the relevant state before completing. After the refactoring, this is no longer the case (cf. the change to the variable `avg` above). We derived four scenarios under which this refactoring *is* safe:

One can safely apply *Replace Exception with Test* if it either holds that (1) the frame of P is disjoint from the set of relevant locations *and* from the frame of Q (it may still influence Q's behavior by writing to its *footprint*), or (2) the frames of P and of Q are disjoint from the set of relevant locations, and Q always completes normally, or (3) the frame of P is disjoint from the footprint of Q, and Q *has to* assign *all* locations assigned by P, or (4) statement Q starts with a “rollback” resetting all locations in the frame of P to independent values. None of these conditions have been mentioned before.

Split Loop

Loop splitting is a common optimization technique which we also address in Sect. 5 and 6. Apart from being useful for, for example, code parallelization, it contributes to readability by dividing loops with separate concerns, and by clearing the way for subsequent optimizations such as the replacement by stream operations. Schematically, "**`while`** `(`g`) {` P Q `}`" gets "**`while`** `(`g`) {` P `}` **`while`** `(`g`) {` Q `}`". We isolated the following sufficient preconditions: (1) The frames of P and Q have to be disjoint from the footprint of g, and the frame of g is empty, (2) the frames of P and Q have to be disjoint, (3) the frame of P must be disjoint from the footprint of Q, and vice versa, (4) the guard g and statement P must not complete abruptly, and (5) Q must not complete abruptly before g and P committed their final results (or, established their invariants). None of these have been documented in [17, 18].

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 (1). Therefore, it is not safe to apply *Split Loop* to such loops.

Remove Control Flag

Instead of using a "control flag" for deciding when to terminate a loop, this refactoring suggests to resort to **`break`** or **`continue`** statements to better communicate the intended control flow. The shortcut associated by the introduction of abrupt completion, however, generally breaks semantic equivalence. Any code that would have been executed after setting the control flag (which is skipped by the shortcut) must not have effects visible outside the loop. Otherwise, it has to be duplicated:

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

Relying on the mechanics described in [17] likely produces incorrect results.

We proved both *Slide Statements* and *Remove Control Flag* using abstract strongest invariants. For *Remove Control Flag*, we apply an even stronger type of loop invariant also considering abrupt completion (standard invariants only have to hold at loop *entry*, and not after (abruptly) leaving the loop).

In the subsequent two sections, we regard code transformations from an optimization point of view: how can we transform code such that it can be *better parallelized* (Sect. 5), and what is the effect of a transformation on *execution cost* (Sect. 6)?

5 Restructuring for Parallelization

Legacy systems were typically developed before the widespread use of modern software engineering techniques [40] like parallel programming interfaces such as OpenMP. *Adapting* existing sequential legacy software to *parallel environments* can save time and money, while avoiding the loss of domain knowledge. One powerful method to parallelize programs are *parallel design patterns* [24, 34] embodying best practices and correct as well as efficient usage of parallelization interfaces. This even yields a *semi-automatic* approach [38] for migrating sequential to parallel code.

As legacy code was not written with parallel patterns in mind, it often does not allow immediate application of a pattern: a certain amount of prior *code restructuring* is unavoidable in most cases. The DiscoPoP [38] framework implements a small number of *sequential* code transformation schemata that frequently suffice to bring sequential code into the form required for the application of a parallel pattern. To ensure that the parallelized program retains the functionality of the original legacy code, it is essential to ensure the correctness of the sequential restructuring rules.

In [22], we focused on three representative restructuring techniques, isolated conditions under which they are safe to apply, and proved—using Abstract Execution and REFINITY—that these conditions are sufficiently strong. Those schemata, previously developed within the DiscoPoP framework, are (1) *Computational Unit (CU) Repositioning*, (2) *Loop Splitting*, and (3) *Geometric Decomposition*. The latter two are loop transformations using an advanced memory layout specification mechanism. The corresponding proofs are, for a change, not fully automatic but require a small number of manual rule applications (<0.3% of all applications). We brief the most relevant aspects of the formalizations and proofs for these three schemata.

CU Repositioning

A CU is a piece of code with little to no internal parallelism, and the basic unit of dependence graphs generated by DiscoPoP. *CU Repositioning* prepares code for an application of the *pipeline pattern*. In a pipeline, each unit can depend on units in prior stages, but *not* in later stages [35]. To enable this pattern, it is sometimes required to *reposition* a later occurring CU to the first CU, merging those into a single CU. In practice, this usually means to move statements occurring after a loop or call to before that loop/call. *CU Repositioning* is an instance of *Slide Statements* (cf. Sect. 4) with *the same preconditions*. This is a notable connection between refactoring and parallelization: the same transformation can serve different purposes.

Loop Splitting

Loop Splitting is an optimization splitting one into several loops. Here, it is used to enable the *Do-All* parallel design pattern, which requires that there are no data dependences between different loop iterations. When an initial segment of iterations *does* have external dependences, we can factor out this segment and apply *Do-All* to

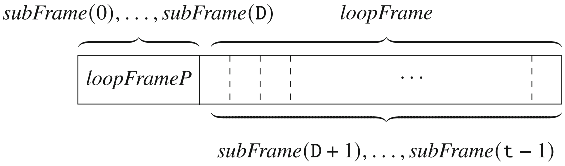

subFrame(0), . . . , *subFrame*(D)
loopFrame
loopFrameP
· · ·
subFrame(D + 1), . . . , *subFrame*(t − 1)

Fig. 6 Abstract memory layout for *Loop Splitting* of a loop with t iterations at index D

the remaining iterations. *Loop Splitting* is a special case of the *Split Loop* refactoring additionally requiring that the parts that are to be parallelized are independent.

Our REFINITY model thus sets up a more advanced memory layout, as depicted in Fig. 6. Each of the t loop iterations operate on their own set of memory locations *subFrame*(i). Assuming that we factor out the first $\mathtt{D} + 1$ iterations, the precondition on the correctness of the restructured *sequential* code is that *subFrame*(0) to *subFrame*(D) (the part pulled out) are disjoint from *subFrame*($\mathtt{D} + 1$) to *subFrame*($\mathtt{t} - 1$) (the subjects to *Do-All*). In addition, the location sets *subFrame*($\mathtt{D} + 1$) to *subFrame*($\mathtt{t} - 1$) have to be disjoint from each other, which is a prerequisite for *Do-All*.

This modeling technique using a *family* of location sets required extensions of AE and REFINITY. The existing proof strategies are yet lacking dedicated support for the arising constraints, which is why we had to perform 7 simple and 16 non-trivial proof steps (out of a total of 15,600 steps) manually, which amounts to 0.1%.

Geometric Decomposition

If a program can be understood as a *sequence of operations* on a *main data structure*, often the best way of parallelization is to decompose this structure. Lists, for example, can be *decomposed* into substructures in a similar manner as dividing a *geometric* region into subregions—hence the name. We focused on the decomposition of a loop with t iterations into N loops of size t/N. These N loops can then be run in parallel.

Geometric Decomposition is a *generalization* of *Loop Splitting*: instead of dividing a loop into *two* parts, it is split into $\mathtt{N} > 1$ parts. Our model uses a similar abstract memory setup as for *Loop Splitting*. The essential correctness precondition is that each bundle of iterations of size t/N has to operate on a separate memory region.

This transformation is the most complex one proven with AE up to now. The proof consists of ~84k rule applications, of which 215 are manual (0.26%).

6 Cost Analysis of Transformation Rules

Apart from running previously sequential code in parallel, one can aim at reducing the *execution cost of sequential code* by applying optimizing transformations, as exercised by many compilers [1]. What is more, when transforming code for *different* reasons (e.g., during code refactoring), one may be concerned about the impact on execution cost. A good example is *Split Loop*: naively, one could use performance as an argument against dividing one loop into two, disregarding understandability of the resulting code. However, the overhead attached to loop splitting merely consists in double evaluations of the loop *guard*, which often is negligible.

Cost analysis for *individual* programs has been addressed by a plethora of tools (e.g., [3, 21, 23]). The relative cost of *different concrete* programs has been subject to formal analyses (e.g., [39]). However, the cost impact of program *transformations*, which can be coined as the relative cost of two *schematic* programs, has not been studied before. We proposed *Quantitative Abstract Execution (QAE)* [4], the first approach to analyze the cost of schematic programs and thus of transformations.

Our technique combines a new frontend to the COSTA cost analyzer [3] and an extension of the AE framework implemented in KeY to a *fully automatic* toolchain. Its workflow is visualized in Fig. 7. One starts with an abstract program and a cost model (number of instructions, allocated memory, etc.) which are input to the cost analyzer ①. Each cost model comes in three flavors differing in their *strength*. Those are, in descending order, *exact*, *upper bound*, and *asymptotic cost*. Generally, we try to derive exact bounds first. The output of the analyzer is a *cost bound* w.r.t. the chosen cost model and a set of *cost invariants* and *ranking functions* for each loop in the program. We enrich the initial transformation model by translating this *quantitative information* to the QAE specification framework. The result of this is given to the QAE implementation on top of KeY ②. We use a proof strategy specifically tailored to the kinds of constraints arising in QAE to obtain a *certificate* for the correctness of the cost bounds in the chosen strength. If the certification process succeeds, we save the certificate and output the bounds ③. Otherwise, we *weaken* the strength of the cost model: from exact cost we descend to upper bound and later to asymptotic cost. Then, we continue at ①. Alternatively, one can inspect the failed proof attempt to obtain feedback for improving components ① or ②.

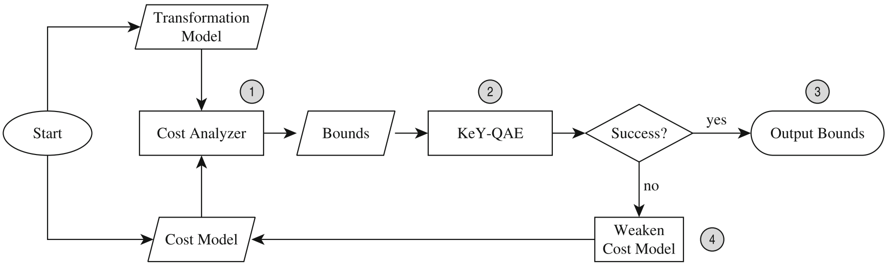


Fig. 7 Workflow of our approach to cost analysis of transformation rules

We implemented our toolchain as a command-line application. Excluding the libraries, and not counting blank lines and comments, it consists of 1,803 lines of PYTHON code (the extension of the cost analyzer), 703 lines of JAVA code (the conversion tool transforming the output of the cost analyzer to input files for KeY), and a 389-line bash script implementing the overall workflow.

We evaluated our approach for seven typical code optimization rules. In six cases, we used the number of executed instructions as a cost model, and in one case the heap consumption. The result consists of *abstract* cost bounds parametric in the concrete cost of the schematic elements from the transformation model. For the case of *Split Loop*, for example, we obtain a bound like $(\#it + 1) \cdot \big(2 \cdot cost_g(fp_g) + cost_P(fp_P) + cost_Q(fp_Q)\big)$ *after* the transformation, where $\#it$ is the number of iterations, and $cost_x/fp_x$ the abstract cost placeholder symbol and cost footprint for APE x, respectively.

Cost analysis took about 50 ms for each problem, while performing the proofs took between 13 s and 30 s. All proofs worked fully automatically and *did not require manual auxiliary specifications*, which was possible for three reasons: (1) We focused exclusively on *quantitative aspects*, leaving aside semantic equivalence (which can be verified separately). Dynamic frames are replaced by representative sets of *program variables* that can be handled by the cost analyzer. (2) The cost analyzer automatically produced *cost invariants* for loops and *ranking functions* needed to show termination. (3) Our new *proof strategy*, integrating external SMT solvers and different strategies for handling arithmetic problems, proved to be *effective for all problems at hand.*

7 Conclusion and Future Work

Legacy software systems are challenging both researchers and practitioners with an intricate problem: Transform *substantial code bases* of *high value*, but *poorly performing*, *insufficiently documented*, and *abandoned* by most of their original developers, into a system implementing the *same functionality*, but making use of *modern software engineering techniques* and *best practices.*

Software reengineering addresses this problem. It generally consists of two phases: (1) A *requirements extraction* and *reverse engineering* phase aiming for a better understanding of the legacy system and the reengineering goals; and (2) a *forward engineering* phase carrying out the actual transformation.

Phase (2) constitutes the critical step: here, behavior can be altered, and domain knowledge and ultimately *money* can be lost. Testing alone is generally insufficient in the presence of sparse existing test suits, as writing meaningful *new* test cases requires insight into the legacy system and the domain knowledge it implements.

We addressed this by proposing *structured* software reengineering: instead of changing code *arbitrarily* and relying on tests (or good luck), we suggest to use *proven-correct* code transformations from a predefined catalog for incrementally changing a legacy system. Correctly applied, this approach guarantees the preservation of *all* functional behavior of the input system. Furthermore, transformations can have clear nonfunctional *objectives*, such as improving readability, runtime performance, or amenability to parallelization.

To kick-start a catalog of proven transformation techniques, we developed *Abstract Execution* (Sect. 2), the first *general-purpose framework* for *automatic* reasoning about statement-level JAVA code transformations, and REFINITY (Sect. 3), a workbench for encoding and proving transformations. We used these tools to (1) derive preconditions for safe code refactoring and prove them sufficient (Sect. 4), (2) prove the safety of transformations used by code parallelization tools to prepare sequential code for the application of parallel design patterns (Sect. 5), and (3) develop an automatic approach to assess the cost impact of program transformations (Sect. 6).

There are many directions for future work on structured software reengineering: (i) Extending the catalog of proven-correct transformation techniques. For transformations *above* statement level, other techniques could be leveraged. (ii) Efficient checks for whether a transformation specified in AE is applicable for a program remain to be implemented (a prototypical demonstrator is contained in REFINITY). (iii) The AE approach could be adapted to different programming languages, or even to differing languages for source and target of a transformation, to address, for example, Cobol programs which the financial industry still relies on. The feasibility of this is demonstrated by an earlier work addressing the translation of JAVA to LLVM IR using a simple version of abstract statements and updates [49]. (iv) To find its way into industrial practice, structured software reengineering needs robust, usable tool support. We envision an IDE with drag'n'drop support of transformations from a catalog (including documentation) onto the code, with automatic sanity checks, or, where this is not possible, automatically generated test cases or at least appropriate warnings. One could even think of automating the process by automatically matching and applying transformations contributing to a selected optimization goal.

Software has come to stay. Structured software reengineering contributes to sustainable software life cycles whenever it stays longer than expected.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification—The KeY Book. LNCS, vol. 10001. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. **413**(1), 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009

4. 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). LNCS, vol. 12649, pp. 24–45. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-71500-7_2
5. Alves, E.L.G., Massoni, T., de Lima Machado, P.D.: Test coverage of impacted code elements for detecting refactoring faults: an exploratory study. J. Syst. Softw. **123**, 223–238 (2017). https://doi.org/10.1016/j.jss.2016.02.001
6. 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
7. 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
8. Bertot, Y., Castéran, P.: Interactive theorem proving and program development—Coq'Art: the calculus of inductive constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2004). https://doi.org/10.1007/978-3-662-07964-5
9. 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
10. Burstall, R.M.: Program proving as hand simulation with a little induction. In: Information Processing, pp. 308–312. Elsevier, Amsterdam (1974)
11. Chikofsky, E.J., Cross, J.H.II..: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**(1), 13–17 (1990). https://doi.org/10.1109/52.43044
12. Crotty, J., Horrocks, I.: Managing legacy system costs: a case study of a meta-assessment model to identify solutions in a large financial services company. Appl. Comput. Inform. **13**(2), 175–183 (2017). https://doi.org/10.1016/j.aci.2016.12.001
13. Cuadrado, F., García, B., Dueñas, J.C., G., H.A.P.: A case study on software evolution towards service-oriented architecture. In: Proc. 22nd Inter. Conf. on Advanced Information Networking and Applications (AINA), pp. 1399–1404. IEEE Computer Society, Silver Spring (2008). https://doi.org/10.1109/WAINA.2008.296
14. 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)
15. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Margaria, T., Steffen, B. (eds.) Proc. 7th ISoLA. LNCS, vol. 9952 (2016). https://doi.org/10.1007/978-3-319-47166-2_36
16. Feathers, M.C.: Working effectively with legacy code. In: Zannier, C., Erdogmus, H., Lindstrom, L. (eds.) Proc. 4th Conf. on Extreme Programming and Agile Methods. LNCS, vol. 3134, p. 217. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-27777-4_42
17. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley, Reading (1999)
18. Fowler, M.: Refactoring: Improving the Design of Existing Code, 2nd edn. Addison-Wesley Signature Series. Addison-Wesley, Reading (2018)
19. Fürnweger, A., Auer, M., Biffl, S.: Software evolution of legacy systems—a case study of soft-migration. In: Hammoudi, S., Maciaszek, L.A., Missikoff, M., Camp, O., Cordeiro, J. (eds.) Proc. 18th Intern. Conf. on Enterprise Information Systems (ICEIS), pp. 413–424. SciTePress, Setúbal (2016). https://doi.org/10.5220/0005771104130424
20. 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
21. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Shao, Z., Pierce, B.C. (eds.) Proc. 36th POPL. ACM (2009). https://doi.org/10.1145/1480881.1480898

22. Hähnle, R., Heydari Tabar, A., Mazaheri, A., Norouzi, M., Steinhöfel, D., Wolf, F.: Safer parallelization. In: Margaria, T., Steffen, B. (eds.) Proc. 9th Intern. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA): Engineering Principles, Part II. LNCS, vol. 12477. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-61470-6_8
23. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Gordon, A.D. (ed.) 19th European Symposium Programming Languages and Systems (ESOP). LNCS, vol. 6012. Springer, Berlin (2010). https://doi.org/10.1007/978-3-642-11957-6_16
24. Huda, Z.U., Jannesari, A., Wolf, F.: Using template matching to infer parallel design patterns. TACO **11**(4), 64:1–64:21 (2014). https://doi.org/10.1145/2688905
25. Kassios, I.T.: The dynamic frames theory. Formal Asp. Comput. **23**(3) (2011). https://doi.org/10.1007/s00165-010-0152-5
26. Khadka, R., Batlajery, B.V., Saeidi, A., Jansen, S., Hage, J.: How do professionals perceive legacy systems and software modernization? In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th Intern. Conf. on Software Engineering (ICSE), pp. 36–47. ACM, New York (2014). https://doi.org/10.1145/2568225.2568318
27. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
28. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an operating-system kernel. Commun. ACM **53**(6), 107–115 (2010). https://doi.org/10.1145/1743546.1743574
29. 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)
30. 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
31. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
32. London, R.L.: Correctness of a compiler for a lisp subset. In: Proc. ACM Conf. on Proving Assertions About Programs, pp. 121–127. ACM, New York (1972). https://doi.org/10.1145/800235.807080
33. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Practical verification of peephole optimizations with alive. Commun. ACM **61**(2), 84–91 (2018). https://doi.org/10.1145/3166064
34. 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
35. Mattson, T.G., Sanders, B., Massingill, B.: Patterns for parallel programming. Pearson Education, London (2004)
36. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. Mathematical Aspects of Computer Science, vol. 1 (1967)
37. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45949-9
38. Norouzi, M.A., Wolf, F., Jannesari, A.: Automatic construct selection and variable classification in OpenMP. In: Eigenmann, R., Ding, C., McKee, S.A. (eds.) Proc. ACM Intern. Conf. on Supercomputing (ICS). ACM, New York (2019). https://doi.org/10.1145/3330345.3330375
39. 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
40. Ransom, J., Sommerville, I., Warren, I.: A method for assessing legacy systems for evolution. In: Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR), pp. 128–134. IEEE Computer Society, Silver Spring (1998). https://doi.org/10.1109/CSMR.1998.665778
41. Schneider, A.: When companies become prisoners of legacy systems. Wall Street J. (2013). https://deloitte.wsj.com/cio/2013/10/01/when-companies-become-prisoners/

42. Smith, D.R.: KIDS: a semiautomatic program development system. IEEE Trans. Softw. Eng. **16**(9), 1024–1043 (1990). https://doi.org/10.1109/32.58788
43. Soares, G., Catao, B., Varjao, C., Aguiar, S., Gheyi, R., Massoni, T.: Analyzing refactorings on software repositories. In: Proc. 25th Brazilian Symposium on Software Engineering (SBES), pp. 164–173. IEEE Computer Society, Silver Spring (2011). https://doi.org/10.1109/SBES.2011.21
44. 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
45. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Softw. **27**(4), 52–57 (2010). https://doi.org/10.1109/MS.2010.63
46. 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
47. Steinhöfel, D.: Abstract execution: automatically proving infinitely many programs. Ph.D. Thesis, TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany (2020). https://doi.org/10.25534/tuprints-00008540. http://tuprints.ulb.tu-darmstadt.de/8540/
48. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Oliveira, B.C.D.S. (ed.) Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS). LNCS, vol. 12470. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-64437-6_16
49. Steinhöfel, D., Hähnle, R.: Modular, correct compilation with automatic soundness proofs. In: Margaria, T., Steffen, B. (eds.) Proc. 8th ISoLA. LNCS, vol. 11244, pp. 424–447. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-03418-4_25
50. Steinhöfel, D., Hähnle, R.: Abstract execution. In: Proc. Third World Congress on Formal Methods—The Next 30 Years (FM) (2019). https://doi.org/10.1007/978-3-030-30942-8_20
51. 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. ACM, New York (2016). https://doi.org/10.1145/2951913.2951924
52. Vijaya, A., Venkataraman, N.: Modernizing legacy systems: a re-engineering approach. Int. J. Web Portals **10**(2), 50–60 (2018). https://doi.org/10.4018/IJWP.2018070104
53. Winterland, D.: Abstract Execution for Correctness-by-Construction. Master's thesis, Technische Universität Braunschweig (2020)