The Java Verification Tool KeY: A Tutorial (preprint)

Bernhard Beckert[1][0000−0002−9672−3291], Richard Bubel[2], Daniel Drodt[2][0000−0003−3036−8220], Reiner Hähnle[2][0000−0001−8000−7613], Florian Lanzinger[1][0000−0001−8560−6324], Wolfram Pfeifer[1][0000−0002−9478−9641], Mattias Ulbrich[1][0000−0002−2350−1831], and Alexander Weigl[1][0000−0001−8446−4598]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{beckert, lanzinger, wolfram.pfeifer, ulbrich, weigl}@kit.edu
[2] Technische Universität Darmstadt, Darmstadt, Germany
{richard.bubel, daniel.drodt, reiner.haehnle}@tu-darmstadt.de

Abstract. The KeY tool is a state-of-the-art deductive program verifier for the Java language. Its verification engine is based on a sequent calculus for dynamic logic, realizing forward symbolic execution of the target program, whereby all symbolic paths through a program are explored. Method contracts make verification scalable, because one can prove one method at a time to be correct relative to its contract. KeY combines auto-active and fine-grained proof interaction, which is possible both at the level of the verification target and its specification, as well as at the level of proof rules and program logic. This makes KeY well-suited for teaching program verification, but also permits proof debugging at the source code level. The latter made it possible to verify some of the most complex Java code to date. The article provides a self-contained introduction to the working principles and the practical usage of KeY for anyone with basic knowledge in logic and formal methods.

Keywords: Program verification · Deductive verification · Dynamic Logic · Java Modeling Language.

"... and the aeroplane shot further away and again,
in a fresh space of sky, began writing a K, an E, a Y perhaps?"
— Virginia Woolf, *Mrs. Dalloway*

1 Introduction

What is KeY? The KeY tool [3,4,15] is a state-of-the-art program verification tool for one of the most widely used programming languages: Java. Its capabilities enable the formal specification and verification of unmodified industrial Java code at source-code level. Notable examples of its application include the TimSort effort [27] and, more recently, the verification of a Java implementation of in-place super scalar sample sort [8], one of the fastest general-purpose sorting algorithms [18]. In addition to its role as a program verifier, KeY serves

as a versatile research platform for implementing various formal methods for Java by leveraging KeY's symbolic execution engine. For instance, KeY has been used to facilitate the generation of test cases with high code coverage [6] and to implement a symbolic-state debugger [39]. The maturity of KeY's verification approach and of the tool permit it being taught in BSc- and MSc-level courses. KeY is an academic, noncommercial tool that can be used freely by anyone (it is published under GNU Public License V2). It is completely written in Java, so it can run on any platform for which a Java virtual machine is available.

The roots of the KeY project trace back to 1999, when the continuous development and refinement of KeY and its verification methodology was started. On the occasion of KeY's 25th birthday, this tutorial serves to showcase the mature program verification and analysis tool that KeY is today.

This Tutorial, its Accompanying Material, and Further Reading This tutorial caters to all who want to learn about the KeY tool methodology: Newcomers to the field as well as seasoned researchers in formal methods outside deductive verification. It offers an exploration of KeY's underlying methodology, its capabilities, and practical application. The tutorial covers KeY's basics while giving a glimpse at its advanced features. Participants of the live conference tutorial gain hands-on experience with KeY. By the end of the tutorial you are able to actively use KeY for (simple) verification tasks and to understand which advanced KeY features permit the verification of complex algorithms.

Video recordings of the conference tutorial, the slides, and all presented examples, as well as the KeY tool itself are available for download at

www.key-project.org/tutorial-fm-2024

For further reading, the book on the KeY system, published in 2016 [4], contains two tutorial chapters on using the KeY tool, based on both simple (chapter on "Using the KeY Prover" [7]) and more advanced (chapter on "Formal Verification with KeY: A Tutorial" [13]) examples.

KeY's Verification Methodology in a Nutshell KeY's deductive program verification engine is based on a sequent calculus for Java Dynamic Logic [17]. The calculus rules realize symbolic execution on the target program, whereby all symbolic paths through a program are explored. Method contracts make verification scalable, because one can prove one method at a time to be correct relative to its contract. Contracts in KeY do not need to be expressed in Dynamic Logic, but can be supplied at the level of source code as *Java Modeling Language* (JML) annotations [46].

KeY features a domain-specific textual language (called *taclets*) for adding axioms of theories, lemmas, and proof rules. This allows to extend and to tailor the deduction engine without having to know implementation internals.

KeY's Interaction Patterns and User Interface In contrast to most other program verification tools, KeY seamlessly combines *auto-active* interaction and

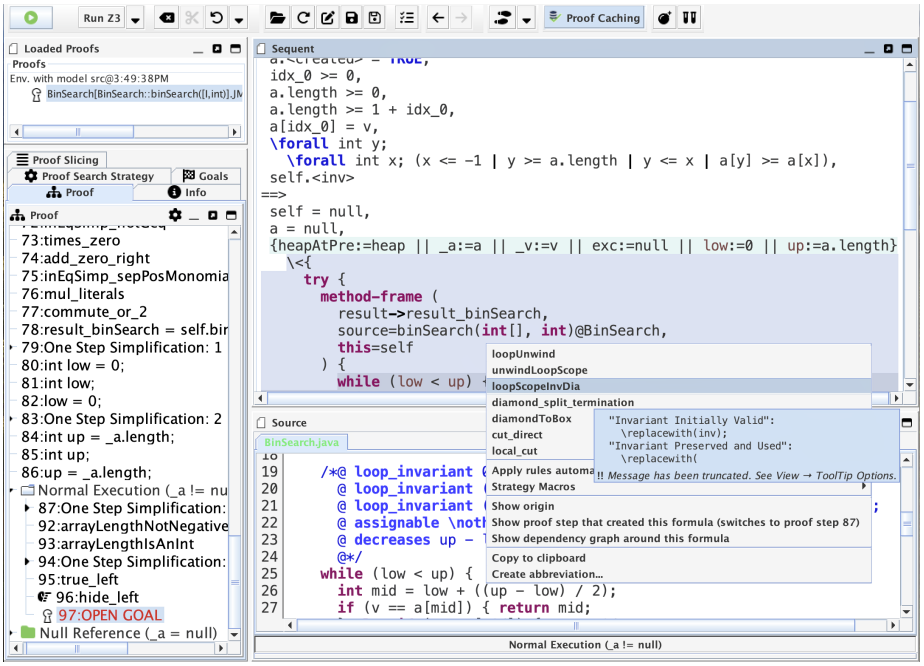

Fig. 1. GUI of the KeY tool: Applicable rules are shown in a context menu after the user has clicked on the loop in a JavaDL formula.

fine-grained interaction: Interaction is possible both at the level of the Java verification target and its JML specification (auto-active interaction pattern), as well as at the level of proof rules and the underlying program logic (fine-grained interaction pattern). In auto-active verification (used in, for example, Why3 [22], Frama-C [45], AutoProof [61], VeriFast [42], VerCors [21], VCC [26], and Dafny [48]), interaction at the input level (adding, removing, or rephrasing specifications, adding hints) is user-friendly as it does not require knowledge and understanding of intermediate proof states. But the lack of insight into the intermediate proof steps makes it hard to identify which additional specification annotations might be needed or which need to be rephrased. On the other hand, fine-grained interaction (most popular with general purpose theorem provers such as Isabelle [51] or Coq [20]), where the proof is constructed either manually or with the help of proof scripts (which may use automatic proof tactics), can give deep insights into possible issues and provides effective control as the user can inspect partial proofs, but it requires a considerable amount of expertise.

KeY's front end for interactive proof construction and exploration is a graphical user interface (GUI), see Fig. 1. Upon loading an annotated Java file, proof obligations are automatically translated into Java Dynamic Logic and presented in the GUI. This GUI is a central component of KeY. Its design is based on a point-and-click interaction style to support the user in proof exploration and

proof construction. For instance, the selection of a calculus rule—out of over 1500(!)—is greatly simplified by allowing the user to highlight any syntactical sub-entity of the proof goal simply by positioning the mouse; a dynamic context menu will offer only the few proof rules which apply to this entity. For instantiating quantified variables, drag-and-drop mechanisms greatly simplify the use of instantiation dialogues. Other supported forms of interaction in the context of proof construction are the inspection of proof trees, the pruning of proof branches, and unlimited undoing of proof steps.

KeY's Verification Process The user provides Java source code with annotations written in JML. These consist of requirement specifications as well as auxiliary specifications such as loop invariants and contracts of called methods. KeY translates these into proof obligations in Java Dynamic Logic. Now the user is left with the choice of trying to let the prover verify fully automatically or of starting interactively by applying calculus rules to the proof obligation. If the user chooses to start the automated proof search strategy offered by the prover, the result can be either that (i) the prover succeeds in finding a proof or (ii) the prover stops after a number of steps with a partial proof. This is the point in the proof process, where the user gets involved. The user inspects the proof state and decides whether to continue with fine-grained interaction or to continue in auto-active style and revise the JML annotation or the source code.

Recently, a lightweight proof scripting language was provided that complements the GUI's point-and-click style interaction. It fosters proof reuse and mitigates the need to redo the initial part of failed proof attempts by hand [18].

2 Verification Approach

Arguably, the most important structuring concept in programming languages are *procedures*[3] (aka functions, methods, routines, etc.). They serve several purposes: (i) To *abbreviate* code by a mere *signature* (unique name plus argument and return types) that would else have to be repeated many times over; (ii) hence, a program can be *structured* into a set of methods (which, in turn, might be grouped into modules, objects, traits, etc.), each containing closely related code; (iii) since the desired functionality of a method might not be exactly the same at each call site, the usage of methods fosters *abstraction* of different behavior into a common implementation. Finally, methods can be used to (iv) *encapsulate* a computation, i.e. to contain its possible effect on the memory state.

With methods being such a central structuring concept of programs, it is natural that verification proofs should reflect and be able to benefit from the structure inherent to the program under verification. In fact, without *structural similarity* between the verification target and the proof object, it is exceedingly difficult to verify complex software systems. This is why most modern deductive verification approaches, including KeY, are *contract-based* [36].

[3] Since the target language of KeY is Java, we mostly adopt the terminology "method" in this paper from now on.

2.1 The Principle of Contract-Based Verification

A contract, in the context of software verification, is a specification artifact that makes it possible to mimic the method structure of a program in a correctness proof. The central idea is to describe the effect of a possible execution of a given method in terms of logical formulas. This has a decisive consequence: Whenever in a proof over a program, it is necessary to reason about a called method, then, instead of going into the implementation of that method, it is sufficient to consider the formulas in the method's contract. This simple and natural approach has dramatic consequences for program verification: (i) Since contracts consist not of code, but of formulas, we replace program execution by *substitution* and *deduction*, and (ii) in contrast to code which, in general, admits an unbounded number of different behaviors, contracts consist of a *finite* number of formulas.

Together, these two observations enable *procedure-modular* verification: the structure of verification proofs follows the structure of methods and an unbounded number of possible method executions is described with a finite number of logical formulas. But how, exactly, are method contracts defined? And how does one ensure that a given method implementation conforms to its contract?

2.2 Method Contracts

A method contract, similar to a legal contract, has two main aspects: (i) It specifies the conditions under which it goes into effect and, if so, (ii) it gives guarantees. It may also (iii) specify collateral effects and give (iv) a temporal statute on delivery. Translated to the domain of programs and methods:

Definition 1 (Method Contract). *A* method contract *for a method* `m` *is a tuple* $(pre, post[, mod]\,[, trm])$*, where pre and post are formulas, called* pre- *and* postcondition*, respectively, the optional* modifier *mod is a set expression over memory locations, and the optional* termination witness *trm is a first-order term over a type equipped with a well-order* $\prec$*.*

The semantics *of a method contract is as follows: if* `m` *is started in any execution state where pre holds, then, in any state where* `m` *terminates, post must hold as well. If mod is present, then* `m` *may change at most the value of memory locations in mod (otherwise, it may change anything). If trm is present, then its value must decrease with respect to* $\prec$ *before the subsequent call to* `m`*, thus enforcing termination of recursive calls. Otherwise,* `m` *may diverge.*

Just like legal contracts, method contracts interface to two parties: the client and the provider. In programs, the client is the code containing a call to `m`, while the provider is the implementor of `m`. It is the latter's responsibility to ensure *post* and, if present, *mod* and *trm*, *provided that pre* holds at the time when `m` is called which is the caller's responsibility.

2.3 Java Modeling Language

So far, the components of contracts are left abstract. In deductive verification, one typically uses (typed) first-order formulas or expressions to formalize them

and a program logic to express the semantics of contracts. However, languages such as Java are very much richer than first-order logic which makes it tedious to write contracts. For this reason, it is common to specify contracts based on *behavioral modeling notations* having a rich syntax and thus being closer to the target language. Contracts written in such a modeling language are then automatically desugared into first-order and program logic [35]. In KeY we use the *Java Modeling Language* (JML) [46].

We introduce JML contracts by example. Listing 1 shows a contract for `binSearch()`, a recursive Java implementation of binary search between indices `low` (inclusive) and `up` (exclusive) on an integer array `a`:

```
/*@ private normal_behavior
  @   requires    0 <= low <= up <= a.length;
  @   requires    (\forall int x, y; 0 <= x < y < a.length; a[x] <= a[y]);
  @   ensures     \result == -1 || low <= \result < up;
  @   ensures     (\exists int idx; low <= idx < up; a[idx] == v) ?
  @                \result >= low && a[\result] == v : \result == -1;
  @   assignable  \nothing;
  @   measured_by up - low;
  @*/
private int binSearch(int[] a, int v, int low, int up) {
  if (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])     { return mid; }
    else if (v < a[mid]) { return binSearch(a, v, low, mid); }
    else                 { return binSearch(a, v, mid + 1, up); }
  }
  return -1;
}
```

Listing 1. Recursive implementation of binary search with JML specification

We observe that JML is placed in Java comments augmented by a trailing/leading `@` sign (the `@`'s between the first and last are purely cosmetic). JML permits visibility modifiers ("`private`") with the same semantics as Java. Any side effect-free Java expression may occur in JML, any `boolean` expression can serve as a formula. Non-Java keywords in JML expressions are indicated by a leading backslash. Beyond Java expressions, first-order universal and existential quantifiers are allowed in JML. These (i) are evaluated over the domain specified in their variable declaration and (ii) have an optional *range* expression, as shown. This idiom, where a quantifier ranges over integers and is further restricted by upper and lower bounds, is characteristic of specifications over array types.

The JML keyword `requires` indicates the *pre* slot of a contract. In the example, two `requires` clauses state the various index bounds and that array `a` is sorted. If a keyword occurs multiple times, as here, then the conjunction of all expressions must hold. The JML keyword `ensures` indicates the *post* slot of a contract. The first `ensures` clause expresses that the returned value is either a valid index of `a` or `-1`. The keyword `\result` always denotes the returned value of a method. The second `ensures` clause is a conditional expression (observe that

quantifiers may appear nested), saying that (i) either the searched element `v` is present in `a` between the given bounds, in which case a valid index, where `v` can be found, is returned or else (ii) the constant `-1` is returned.

JML keyword `assignable` indicates the *mod* slot of a contract. A search method is expected not to change the computation state, so we specify the empty set of locations, for which keyword `\nothing` stands. To prove termination of the recursive implementation we add a `measured_by` clause. The top-level call will be of the form `binSearch(a,v,0,a.length)`, so initially the measure is equal to the value of `a.length`. It decreases at each call, because `low < low + ((up - low) / 2) + 1` holds; it is never negative, because the relation `low <= up` is maintained.

2.4 Dynamic Logic

How does one *prove*, for example, that the implementation of `binSearch` in Listing 1 conforms to its contract? In KeY we use dynamic logic, a program logic due to Pratt [53]. In a nutshell, dynamic logic is obtained from Hoare logic [41] by closing it syntactically with respect to first-order formulas. In consequence, correctness assertions may be nested which adds useful expressiveness.

Definition 2 (Dynamic Logic, DL). *Dynamic logic extends first-order logic with a binary operator $[p]\phi$ and is inductively defined as follows: (1) Every first-order formula is a DL formula. (2) If p is a program and ϕ a DL formula, then $[p]\phi$ is a DL formula.*

Formula $[p]\phi$ is valid in a first-order model $\mathcal{M}$ if the following holds: For any execution state s, if p is started in s and it terminates in a state s', then ϕ is valid in $\mathcal{M}$ and s'.

Obviously, $[p]\phi$ expresses partial correctness of p with respect to postcondition ϕ, whenever ϕ is a first-order formula. A first-order contract $(pre, post)$ for p can be expressed as $pre \to [p]post$, which corresponds to the *Hoare triple* $\{pre\}p\{post\}$ [41]. The remaining contract elements are discussed later.

DL being syntactically closed, we can define the dual operator $\langle\cdot\rangle$ by $\langle p\rangle\phi \equiv \neg[p]\neg\phi$, which expresses *total correctness* of p, keeping in mind $[p]\phi$ trivially holds for non-terminating program and assuming programs are deterministic.

It is important to observe that DL is a *modal logic*: in general, executing p changes the execution state. However, it is convenient to assume the value of *first-order* variables stays invariant. For example, we might want to write $\forall x.\,(x \geq 0 \to [p](\backslash\texttt{result} \geq x))$ and be sure that x is *not* changed by p. To achieve this, it is necessary, unlike in Hoare logic, to sharply differentiate between memory locations in programs (variables, arrays, fields, ...) and first-order variables. The former are modeled as *non-rigid* constants and functions whose interpretation may change from state to state; the latter are, as usual, evaluated under a *rigid* first-order model and variable assignment. In consequence, we do not permit quantification over program variables—this would be a second-order formula. On the other hand, it makes perfect sense to write a DL formula such as $\texttt{i} \doteq 0 \to [p]\texttt{i} \not\doteq 0$ and expect it to be valid, for example, when p is `++i`.

For the KeY tool, we use a Java-specific extension of dynamic logic called JavaDL. The main difference to vanilla DL [37] is that JavaDL contains many predefined rigid and non-rigid first-order functions and predicates, including suitable first-order theories that model Java features in first-order logic. The intended first-order models $\mathcal{M}$ (Definition 2) must be defined accordingly. For example, there is a non-rigid function that returns the length of an array `a` in the current execution state as $\mathit{length}(\mathtt{a})$. In JavaDL we also permit Java-style syntax `a.length`. JavaDL is a *typed* first-order logic whose type system includes (i) all Java primitive and reference types that (ii) are ordered according to Java's typing rules. Obviously, all JavaDL terms are assumed to be well-formed according to Java rules which is enforced by KeY's parser.

2.5 State Updates

A common approach to perform logical inference in program logic is to compute the *weakest precondition* [31] of $[p]\mathit{post}$, i.e. the logically weakest formula $\mathit{wp}(p, \mathit{post})$ such that $\mathit{wp}(p, \mathit{post}) \rightarrow [p]\mathit{post}$ holds. It is constructed from *post* by unraveling p backwards. For example, in Hoare calculus $\mathit{wp}(x := e, \phi) = \phi[x/e]$ (assuming x is a scalar variable and e a simple expression). The weakest precondition computation is iterated until the beginning of a program p is reached. Branching statements split into several weakest preconditions, such that the overall result of the process is a finite set of first-order formulas $\mathit{vc}_1, \ldots, \mathit{vc}_n$ for which $\bigwedge_i \mathit{vc}_i \rightarrow [p]\mathit{post}$ holds. The vc_i are called *verification conditions* and can be discharged, for example, with automated theorem provers or SMT solvers. Iterative or recursive constructs require strongest invariants to compute *wp*, otherwise, (stronger) necessary conditions are obtained.

This *verification condition generation* (VCG) is simple and amenable to automation, but is problematic whenever full automation is not achievable: (i) Verification conditions tend to become large and complex, and then they are difficult to understand in case they are not provable; (ii) executing a program backwards is unnatural for humans and makes it hard to follow a failed verification attempt.

In KeY we assume that contracts and loop invariants (Sect. 3) for complex programs must be at least partially created manually. Getting them right requires understanding of intermediate proof situations. For this reason the JavaDL inference system is not based on a VCG architecture, but on *forward symbolic execution*. Unfortunately, computing the dual of $\mathit{wp}(p, \mathit{post})$, i.e. the *strongest postcondition* of a program started in a state satisfying *pre*, is expensive and unnatural. Therefore, we use a technical trick that avoids computing explicit strongest postconditions:

Definition 3 (Elementary Update). *Let* `v` *be a program variable of primitive type and* `e` *a simple (not nested) and side effect-free expression such that the assignment* `v = e` *is well-formed. Let* e *be the first-order representation of* `e`.[4] *Then* $\mathtt{v} := e$ *is called an* elementary update.

[4] From now on we adopt the convention that the first-order translation of a Java expression uses the same letter, but is typeset in Roman font.

The semantics *of an elementary update* $\mathtt{v} := e$ *are all state transitions, where the value of* $\mathtt{v}$ *in the final state is set to the value that* e *had in the first state.*

Updates capture the effect of symbolic state changes and are streamlined to represent (simple) assignments. By prefixing JavaDL formulas with updates we can express that a formula is evaluated in the state represented by these updates:

Definition 4 (JavaDL with Updates). *If* u *is an update,* ϕ *a DL formula, and* e *a DL expression, then* $\{u\}\,\phi$ *is a DL formula and* $\{u\}\,e$ *a DL expression.*

2.6 A JavaDL Calculus

For a simple assignment `v = e`, the DL formulas $[\mathtt{v = e;}\ p]\phi$ and $\{\mathtt{v} := e\}\,[p]\phi$ are logically equivalent. This observation is the basis for a *forward symbolic execution calculus* to prove the validity of JavaDL formulas: For each type of Java statement st in a program formula $[st;\ p]\phi$, we compute a finite set of formulas that implies $[st;\ p]\phi$ and, therefore, can *replace* it. These formulas have the form $\{\phi_i \to \mathcal{U}_i[st_i;\ p]\phi\}_i$, where $\mathcal{U}_i$ are updates, the st_i typically are (possibly empty) sub-statements of st, and the ϕ_i (optional) preconditions (the above replacement of an assignment is a special case of this general schema). This characterization permits to further reduce the $[st_i;\ p]\phi$ and so on. All that remains to do is to turn this schema into a calculus and to design the actual rules.

We assume the reader is familiar with the basics of sequent calculi (see, for example, [32]). As usual, we use naming conventions for schema variables: $\phi, \psi, \ldots$ stand for JavaDL formulas, Γ for sets of JavaDL formulas, and $\mathcal{U}$ denotes an arbitrary sequence of updates. More schema variables are introduced as needed. A typical (unary) rule may have the general form of the left rule schema below where DL formulas ϕ, ψ are rewritten while the update $\mathcal{U}$ and formulas in Γ remain unchanged (Γ might contain assumptions or theories).

$$\frac{\Gamma, \mathcal{U}\phi' \implies \mathcal{U}\psi'}{\Gamma, \mathcal{U}\phi \implies \mathcal{U}\psi} \qquad \frac{\phi' \implies \psi'}{\phi \implies \psi}$$

To make rule notation more succinct, we drop context formulas and leading updates in the following like in the rule schema on the right, where contexts are *implicit*, but we *actually mean the rule on the left.* With this convention in place, we formalize the observation at the beginning of this subsection as the sequent rule given below on the left. On the right is the rule that stops symbolic execution once there is no further statement left to evaluate:

$$\frac{\implies \{\mathtt{v} := e\}\,[p]\phi}{\implies [\mathtt{v = e;}\ p]\phi}\ \text{assignment} \qquad \frac{\implies \phi}{\implies [\,]\phi}\ \text{emptyBox}$$

Example 1. Let us prove correctness of in-place value swap (ignoring possible arithmetic overflow), as formalized in the sequent:

$$\mathtt{i} \doteq i_0,\ \mathtt{j} \doteq j_0 \implies [\mathtt{int\ i,j;\ i = i+j;\ j = i-j;\ i = i-j;}](\mathtt{i} \doteq j_0 \wedge \mathtt{j} \doteq i_0)$$

Symbolic execution of the variable declaration (rule not shown) extends the signature and has no visible effect. After applying the assignment rule three times and then rule (emptyBox), we obtain:

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \{\mathtt{i} := \mathtt{i+j}\}\,\{\mathtt{j} := \mathtt{i\text{-}j}\}\,\{\mathtt{i} := \mathtt{i\text{-}j}\}\,(\mathtt{i} \doteq j_0 \wedge \mathtt{j} \doteq i_0) \qquad (1)$$

The example shows that we need rules for applying an update to a first-order formula or term. Updates can be viewed as *explicit substitutions* [2], thus update application is obvious: A straightforward homomorphism on formulas and terms, except the base case: $\{\mathtt{v} := e\}\,\mathtt{w}$ yields e in case $\mathtt{v} = \mathtt{w}$ and $\mathtt{w}$, otherwise.

We apply the updates (1), starting with the last one, which yields (2) and, after two more update applications, the provable first-order sequent (3):

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \{\mathtt{i} := \mathtt{i+j}\}\,\{\mathtt{j} := \mathtt{i\text{-}j}\}\,(\mathtt{i\text{-}j} \doteq j_0 \wedge \mathtt{j} \doteq i_0) \qquad (2)$$

$$\mathtt{i} \doteq i_0,\, \mathtt{j} \doteq j_0 \implies \mathtt{i+j\text{-}(i+j\text{-}j)} \doteq j_0 \wedge \mathtt{(i+j)\text{-}j} \doteq i_0 \qquad (3)$$

At this point, two important observations can be made: (i) The first-order formula on the right of sequent (3) is the *weakest precondition* of the program and postcondition in Example 1. Updates allow us to compute it in a *forward* fashion. Moreover, it is unnecessary to define substitution on programs (which is highly complex for languages such as Java), because updates are applied only on formulas and terms. Difficulties, such as aliasing or side effects, are dealt with at the level of symbolic execution rules, as we shall see below. (ii) There is a potential inefficiency in the so far *lazily* applied updates. For example, when some code is unreachable, that is discovered late. In addition, iterative substitutions can blow up term size drastically. This is mitigated by performing *eager* update simplification. To this end, we define *parallel* updates $\mathtt{v}_1 := e_1 \,||\, \cdots \,||\, \mathtt{v}_n := e_n$, where each slot is an elementary update and all $\mathtt{v}_i$ are different.

The semantics of parallel updates are those state transitions, where all the elementary updates are performed in parallel, i.e. the *old* values of the right hand side are used in each elementary update. For example, the parallel update $\mathtt{j} := \mathtt{i} \,||\, \mathtt{i} := \mathtt{j}$ simultaneously sets $\mathtt{i}$ to the previous value of $\mathtt{j}$ and vice versa. Since all left-hand sides in a parallel update are different, this is well-defined (for the moment, we ignore aliasing which is discussed in Section 3). To turn a sequence of elementary updates into a parallel update, the following rewrite rule is applied, where u is any, possibly parallel, update:

$$\{u\}\,\{\mathtt{v} := e\} \rightsquigarrow \{u\backslash\mathtt{v} \,||\, \mathtt{v} := \{u\}\,e\} \qquad \text{seqToPar}$$

where update $u\backslash\mathtt{v}$ is identical to u, except elementary updates with left-hand side $\mathtt{v}$ are dropped. This is to keep left-hand sides unique and is justified by the fact that any $\mathtt{v}$ occurring in u on the left is overwritten by the later update of $\mathtt{v}$.

If we apply rule (seqToPar) to the formula in sequent (1), we obtain

$$\{\mathtt{i} := \mathtt{i+j} \,||\, \mathtt{j} := \mathtt{(i+j)\text{-}j}\}\,\{\mathtt{i} := \mathtt{i\text{-}j}\}\,(\mathtt{i} \doteq j_0 \wedge \mathtt{j} \doteq i_0)\ .$$

Then, before applying rule (seqToPar) again, it is possible to perform arithmetic simplification on the expression $\mathtt{(i+j)\text{-}j}$. Such a strategy of *eager* update parallelization and simplification helps to keep symbolic expressions small and is crucial for performance.

2.7 Forward Symbolic Execution of Straight-line Programs

To be able to verify straight-line programs with the JavaDL calculus, two more components are needed: Handling complex expressions and conditional statements. We start with the former. Typically, in a rich language such as Java is that an array assignment could be of the form `e[e'] = e''`, where each of `e`, `e'`, `e''` might be a complex expression. Moreover, evaluation of `e''` can incur side effects that may or may not influence evaluation of `e'` (`i++` vs. `++i`). Symbolic execution must respect Java's evaluation rules and record side effects at the correct place. Not surprisingly, a large number of rules are required. Luckily, all of these rules follow the same simple principles. We discuss one typical representative:

$$\frac{\Longrightarrow\ [T_{nse}\ \texttt{v; v = }nse\texttt{; v[e] = e'; }p]\phi}{\Longrightarrow\ [nse\texttt{[e] = e'; }p]\phi}\ \text{assignmentUnfoldLeftArrayRef}$$

This rule handles the case when the array reference nse is not a simple expression and possibly has side effects. First a fresh variable `v` is allocated that holds the reference expression nse. Subsequently, the original assignment is unfolded and nse replaced with `v`. The premise can now be symbolically executed, relying on `v` being simple. Of course, further rules must be applied to deal with `e`, `e'`.

All rules for complex assignments follow this simple schema: (i) Memorize a non-simple sub-expression, (ii) unfold a complex expression with the memorized value, (iii) arrange the sequence of assignments to reflect Java's evaluation rules.

The same principle is used to ensure that guards of conditionals and loops are side effect-free, simple expressions before they are symbolically executed. In consequence, the rule for conditionals is straightforward:

$$\frac{se \doteq TRUE \Longrightarrow [p\texttt{; }r]\phi \qquad se \doteq FALSE \Longrightarrow [q\texttt{; }r]\phi}{\Longrightarrow\ [\texttt{if (se) }p\texttt{ else }q\texttt{; }r]\phi}\ \text{ifElseSplit}$$

2.8 Procedure-modular Verification: Contracts and Method Calls

To verify the example in Listing 1, we need to handle recursive procedure calls (for loops, see Section 3). We focus on a simple case to avoid the main idea getting buried under technicalities: Assume a method signature `static T m(T' arg);` with a contract $(pre, post)$. We design a JavaDL rule that, instead of inlining `m`'s implementation, uses its contract (how to verify contracts is shown next). In the conclusion of the rule below, we assign the result of a call to `m` with a simple argument `se` compatible to `T'` to a simple location expression `v` compatible to `T`. Further, we assume that pre' and $post'$ are the desugared first-order translations of pre and $post$, respectively, where `res` corresponds to JML's `\result`.

$$\frac{\Longrightarrow\ \{\texttt{arg} := se\}\, pre' \qquad \Longrightarrow\ \{\texttt{res} := c_r\}\,(post' \rightarrow \{\texttt{v} := \texttt{res}\}\,[p]\phi)}{\Longrightarrow\ [\texttt{v = m(se); }p]\phi} \tag{4}$$

The left premise validates that `m`'s contract goes into effect by proving the precondition with se as the value of `arg`. The right premise uses the postcondition in the remaining proof. To this end, first `res` is initialized with an unknown value

(fresh Skolem constant c_r), then *post′* is added as an assumption. Whatever *post′* knows about `res` is propagated to `v` and can be used to establish $[p]\phi$.

The general case for method contract application can be more complicated. Specifically, for non-static method calls (dynamic dispatch), the implementation of `m` might be impossible to determine statically. In this case, the verification branches into different cases, one for each potential implementation. In addition, the caller expression must be correctly set up and possible side effects of the call, as described in the `assignable` clause, must be considered. Finally, `m` might terminate with an exception. We refer to [34] for a full treatment.

To formalize *verification* of a contract's correctness is easy in JavaDL, because the modal correctness formulas are closely aligned to the semantics of contracts. With the terminology from above, to verify a contract, we prove the following sequent (`arg` is the name of `m`'s parameter used in *pre′*):

$$pre' \implies [\texttt{res = m(arg);}]post'$$

To avoid a circular argument, rule (4) is *not* permitted, but `m` is *inlined*. Again, this does not yet account for the possibility that `m` may throw an exception. To exclude this case, one can simply wrap the method call in a `try` statement and add a check to the postcondition ensuring no exception was thrown.

We close the section observing that the expressiveness of dynamic logic permits to formalize method contract correctness and method contract usage as a single JavaDL formula resp. a JavaDL calculus rule. This is in contrast to VCG style verification based on Hoare logic, where this must be encoded with numerous `assert` statements dispersed throughout the program under verification.

2.9 Proving the Contract of Binary Search

We prove the contract shown in Listing 1. Thus, we expect the following JavaDL formula to be provable:

$$\{\texttt{v} := v_0 \,||\, \texttt{low} := l \,||\, \texttt{up} := u\}\ (pre' \rightarrow \langle \texttt{res = binSearch(a,v,low,up);} \rangle post')$$

Observe that this is a *total correctness* formula while the rules so far were formulated with *partial correctness* operators. Fortunately, the calculus for partial and total correctness is exactly the same, except for Java constructs with potentially unbounded behavior. These are recursive calls and loops. To deal with the former, a check for the measure to decrease must be added to rule (4). When provable, total correctness of all method contracts in a given program implies total correctness of any program. This follows from a result proved in [49].

Before we can prove the DL formula above with KeY, there is one last loose end to tie up: It concerns how assignments involving array types are handled. Due to the considerations in Section 2.7 we can assume that all locations are simple and side effect-free. Yet the assignment rules—below the one for array access on the right of the sequent—are relatively complex:

$$\frac{\begin{array}{c}\texttt{a} \neq \texttt{null},\ 0 \leq e < \texttt{a.length} \implies \{\texttt{v} := \texttt{a}[e]\}\,[p]\phi \\ \texttt{a} \doteq \texttt{null} \implies [\texttt{throw new NullPointerException();}\ p]\phi \\ \texttt{a} \neq \texttt{null},\ 0 > e \vee e \geq \texttt{a.length} \implies [\texttt{throw new AIOoBException();}\ p]\phi\end{array}}{\implies [\texttt{v = a[e];}\ p]\phi}$$

This rule (as other array rules) reflects that in Java an array access can throw a `NullPointerException` or an `ArrayIndexOutOfBoundsException`, which cannot be statically excluded (for symbolic execution of exceptions, see Section 3.3). The actual update happens in the first premise. Array updates $\texttt{v} := \texttt{a}[e]$ constitute a new class of elementary updates with a dedicated set of update application and simplification rules, reflecting the semantics of Java arrays. In particular, these rules take into account that in Java array references might be aliased.

3 Towards Real Java

So far we learned how to specify and verify a simple program, but the preceding section left some gaps. The symbolic execution rules discussed above only consider updates on local variables without aliasing and, for the most part, without exceptional behavior. Furthermore, just like in Hoare calculus, KeY's JavaDL calculus requires loop invariants. In this section, we introduce the concepts necessary to specify and verify the iterative version of binary search in Listing 2: the heap model, exceptions and other abnormal termination, and loop invariants.

3.1 Aliasing: State Updates on the Heap

A major difficulty in verifying object-oriented programs is aliasing on the heap. Consider an assignment to a field `o.f`. Then, the assignment rule from Section 2 no longer suffices because changing the value of `o.f` might also change the value of `o2.f` if $\texttt{o} \doteq \texttt{o2}$. To accommodate aliasing, JavaDL models the heap as an array with indices (o, f) (called heap locations), where o is a first-order expression of type *Object* and f is a first-order expression of type *Field*, the type of field references. The axiomatization is based on the theory of arrays [50], but it is extended by axioms specific to JavaDL. The list of these axioms is found in [57], here we explain the functions defined through these axioms informally.

Given a program variable h of type *Heap*, a heap location (o, f), and an expression e, the expression $\text{store}(h, o, f, e)$ evaluates to a heap identical to h except that the value of location (o, f) is e. For any Java type A, there is a function select_A such that $\text{select}_A(h, o, f)$ evaluates either to the value at the location (o, f), if that value has type A, or to an underspecified value otherwise.

Now we can give an update rule for field assignments. If either side of an assignment is a complex expression, we first apply unfolding rules similar to the rule (assignmentUnfoldLeftArrayRef) from Section 2.7.For a field assignment where both sides are simple expressions, we have the following rule. Similar to the rule seen in Section 2.9, we need a premise to deal with a possible `NullPointerException`. The first premise translates the assignment to an update using the store function on the heap.

```
/*@ private normal_behavior
  @   requires   (\exists int idx; 0 <= idx < a.length; a[idx] == v);
  @   requires   (\forall int x, y; 0 <= x < y < a.length; a[x] <= a[y]);
  @   ensures    0 <= \result < a.length;
  @   ensures    a[\result] == v;
  @   assignable \nothing;
  @ also private exceptional_behavior
  @   requires   !(\exists int idx; 0 <= idx < a.length; a[idx] == v);
  @   assignable \nothing;
  @   signals_only NoSuchElementException;
  @*/
private int binSearch(int[] a, int v) {
  int low = 0;
  int up = a.length;
  /*@ loop_invariant 0 <= low <= up <= a.length;
    @ loop_invariant (\forall int x; 0 <= x < low; a[x] != v);
    @ loop_invariant (\forall int x; up <= x < a.length; a[x] != v);
    @ assignable \nothing;
    @ decreases up - low;
    @*/
  while (low < up) {
    int mid = low + ((up - low) / 2);
    if (v == a[mid])        { return mid; }
    else if (v < a[mid])    { up = mid; }
    else                    { low = mid + 1; }
  }
  throw new NoSuchElementException();
}
```

Listing 2. Iterative implementation of binary search with JML specification

$$\frac{\begin{array}{r} \mathtt{v} \neq \mathtt{null} \implies \{\mathtt{heap} := \mathrm{store}_A(\mathtt{heap}, \mathtt{o}, \mathtt{f}, \mathtt{v})\}\,[p]\phi \\ \mathtt{v} \doteq \mathtt{null} \implies [\mathtt{throw\ new\ NullPointerException();}p]\phi \end{array}}{\implies [\mathtt{o.f\ =\ v;}p]\phi}\ \text{assignmentToField}$$

To support modular verification as presented in Section 2, we need a way to model the effects of a method call on the heap. KeY uses a variant of dynamic frames [43,58], an approach which uses sets of heap locations as first-class logical variables. To model the heap after a method call, we use a function which takes a heap h and a location set s and replaces the value of any location in s by an unknown value. This is accomplished by the anonymization function anon: The expression $\mathrm{anon}(h, s, h')$ evaluates to a heap equal to h except that all values of locations in s are taken from h'. If h' occurs nowhere else in the sequent, then these values are unknown. Then, exactly the information in the postcondition is what is known about the new values. Our anonymization is related to the "havoc" notion in Boogie [11].

3.2 Loop Invariants in JML and JavaDL

To verify unbounded loops, KeY requires a manually specified loop invariant. A loop invariant is a formula that holds before entering the loop and after every loop iteration. Additionally, we need a termination witness (called loop variant) to prove total correctness.

The loop invariant in Listing 2 consists of three clauses: The first limits the range of the index variables `low` and `up`, like in the precondition of the recursive version. The other two clauses differ from the recursive contract. The recursive contract states that the searched value is between the indices `low` and `up`. When using a loop invariant, we must instead state that the searched value is *not* between the indices `0` and `low` nor between `up` and `a.length`. (KeY also permits contract-like, recursive loop specifications [62], but this is beyond the scope of this tutorial).

The loop variant, specified by `decreases`, is an expression whose value is always at least `0` but strictly decreases with every loop iteration. Finally, the loop needs an `assignable` condition to prove the surrounding method's `assignable` condition.

When encountering a loop in JavaDL's calculus, one must prove three claims: (i) The loop invariant holds when entering the loop; (ii) the loop invariant is preserved by the loop body; (iii) after the loop terminates, the invariant ensures that the postcondition holds after executing the rest of the program. These claims are captured in the three premises of the following rule (a simplified version that only applies to loops without side effects in the loop guard and without abnormal termination; it also does not consider the loop variant):

$$\frac{\begin{array}{c}\Gamma \implies \mathcal{U} inv \\ \Gamma \implies \mathcal{U}\mathcal{A}((inv \land \texttt{cond} \doteq \text{TRUE}) \to [body] inv \land frame) \\ \Gamma \implies \mathcal{U}\mathcal{A}((inv \land \texttt{cond} \doteq \text{FALSE}) \to [\pi\omega]\phi)\end{array}}{\Gamma \implies \mathcal{U}\,[\pi\ \texttt{while (cond) \{}\ body\ \texttt{\}}\ \omega]\phi}\ \text{simpleInv}$$

Here, we drop the notational convention established in Section 2.6 and write the update $\mathcal{U}$ and antecedent Γ explicitly. We also write ω for the *rest of the program* and π for the *inactive prefix*, which may include a sequence of opening braces `{` and initial try blocks "`try {`". The *initial update* $\mathcal{U}$ captures the state of symbolic execution before the loop. The first premise ensures that the invariant *inv* holds upon entering the loop. The second and third premise contain the update $\mathcal{A} = \{\texttt{heap} := \text{anon}(\texttt{heap}, mod, a_h) \,||\, \texttt{l}_1 := c_1 \,||\, \cdots \,||\, \texttt{l}_n := c_n\}$, Here, *mod* corresponds to the `assignable` clause, a_h is an unknown heap (i.e., a heap which occurs nowhere else in the sequent) and $\texttt{l}_i := c_i$ are updates which set any local variable $\texttt{l}_i$ written in the loop body to an unknown value c_i. The two updates $\mathcal{U}\mathcal{A}$ are applied sequentially to transfer that part of the symbolic state that is unchanged by the loop across the loop boundary. If, in that partially anonymized state, the invariant and loop guard both hold, executing the loop body must preserve the invariant and the *frame condition* which ensures that any heap location outside *mod* is unchanged. If the invariant holds but the loop guard

does not (the loop terminates), the postcondition must hold after executing the program rest ω.

3.3 Exceptions in JML and JavaDL

In Section 2 we considered programs that terminate normally. But the version of `binSearch` in Listing 2 throws an exception if the element is not found (instead of returning `-1`). To specify this, we add a second contract using the keyword `also`. That contract starts with `exceptional_behavior`, which specifies that the method terminates with an exception if the precondition holds. The keyword `signals_only` followed by a list of exception types states that the method throws no other exceptions except those listed.

The translation to JavaDL combines both contracts: The JavaDL precondition is the disjunction $pre \vee pre'$ of the two preconditions, and the postcondition is

$$(pre \rightarrow \texttt{exc} \doteq \texttt{null} \wedge post) \wedge (pre' \rightarrow \text{instanceOf}_{\texttt{NSEE}}(\texttt{exc}))$$

where *post* is the translation of the `ensures` clause and `exc` is a reserved program variable set when a `throw` statement is symbolically executed.

3.4 Integer Semantics

Recall that in Example 1, we glossed over the issue of arithmetic overflows. We treated Java's `int` type as the mathematical integers $\mathbb{Z}$ and all arithmetic operations on `int` as their mathematical counterpart (our discussion focuses on integers, but similar considerations apply to `byte`/`long`). Clearly, it is unsound to disregard overflows. Consider the DL formula

$$\texttt{i} \geq 0 \rightarrow [\texttt{i = i + 1;}](\texttt{i} > 0)$$

At first glance, it seems to be valid. But in case `i`'s value is the maximal `int` value, there is an overflow resulting in a negative value of `i`. To render the formula valid, we can strengthen the precondition by `i < Integer.MAX_VALUE`.

To permit flexibility in the choice of the arithmetic model, KeY translates operations `+`, `-`, `*`, etc., to *abstract* JavaDL functions during symbolic execution. For example, `a + b` becomes javaAddInt(`a`, `b`) (assuming that `a`,`b` are of type `int`). The interpretation of these abstract functions can be configured in the KeY tool (option "intrules"). Three options for integer semantics are available:

(I) The default integer semantics, arithmeticSemanticsIgnoringOF, translates to $\mathbb{Z}$, as we did in Example 1. This semantics allows for easy prototyping and teaching—also specifications tend to be much simpler—but it is unsound. Nor is this semantics complete, as some valid formulas cannot be proven, such as $\texttt{i} \doteq \texttt{Integer.MAX\_VALUE} \rightarrow [\texttt{i = i + 1;}](\texttt{i} < 0)$. (II) To verify a program that does not rely on overflows, the semantics checkedOverflow is suitable. It checks that for all abstract functions, the result is in the value range of `int`, i.e. it proves the *absence* of overflows. While checkedOverflow is sound, it is

not complete. If an intentional overflow occurs, the proof cannot be finished. Both proof and specification efforts tend to be bigger with this option than for the mathematical semantics. (III) The javaSemantics accurately models most operations on Java's `int` and provides soundness and completeness. All abstract functions are translated to accurate calculations for `int`, at the cost of even more complex proofs.

Integer semantics options let the user trade off the complexity of proofs and specifications against the accuracy of the modeling: Is an exact model of Java's `int` required which will complicate the proof? Is showing the absence of overflows sufficient? Is the limited accuracy of mathematical integers acceptable? The answer will depend on the specific case.

Floating Point Numbers Over the last years, KeY has added support for floating point numbers, using a combination of theories in taclets and SMT solvers. See [1] for details and limitations.

4 Inside KeY's Core

4.1 Prover Architecture

As discussed in Section 2.5, KeY does not have a VCG architecture. Unlike such tools as OpenJML [25] or Dafny [47], KeY comes with a built-in theorem prover, but can also use external SMT solvers. It works directly on Java source code avoiding an intermediate representation. Instead, it utilizes updates to achieve forward symbolic execution, relying on its JavaDL calculus and automatic prover to close goals. The latter is strong enough in many complex situations.

In addition to avoiding the limitations of VCG discussed in Section 2.5, this approach has four main advantages: (I) Proofs generated by KeY are self-contained without a reference to—or trust placed in—external tools. It is always possible to examine the current proof state in KeY without the need to understand, for example, the SMTLIB format [12]. (II) The user of the KeY prover and the tool itself work on the same structure and goals. This simplifies understanding of proofs, the underlying calculus, and potential errors. (III) The automation capabilities of KeY enable it to simplify any JavaDL formula, not only quantifier-free first-order expressions, during symbolic execution. Since the automation strategies aggressively simplify updates, first-order formulas, and terms while symbolically executing a program, many branches in a proof tree are closed early or are not created in the first place. Simplification is crucial to lessen the impact of *path explosion*—a well-known issue in symbolic execution [9]. (IV) KeY generates an explicit, self-contained *proof object*. A KeY proof can be saved and reloaded, even when it is incomplete. A proof consists of the claim to be proven plus a series of rule applications. This permits to share and re–play proofs, increasing trust in KeY artifacts and enabling reproducible results. Hence, the trusted code base is only KeY and its 25 years of experience.

The downside of the KeY architecture is that, when verifying exceptionally complex code, KeY's automatic capabilities may be insufficient. In this case, KeY can export a (first-order) goal to SMTLIB format and hand it to an SMT solver, such as Z3 [29] or cvc5 [10]. This is especially useful for floating point numbers (see Section 3.4). In this manner, KeY can still profit from the advances in SMT-solving technology, albeit at the cost of sacrificing self-contained proof objects.

4.2 Taclets

As a proof assistant, KeY allows significant flexibility regarding its underlying calculus. Most rules of the JavaDL calculus are not hard-coded but written in a simple, but expressive, language for such rules called *taclets*. We provide a succinct description of taclets. For a more in-depth coverage of taclets, their features, and correctness, see [55].

Taclets are very versatile and permit axiomatization of data structures, definition of symbolic execution rules, rules for propositional and first-order logic, etc. They allow users to define their own rules to accommodate a specific verification purpose. To ensure soundness of first-order taclets, KeY generates a proof obligation expressing the soundness of the taclet, which is proven in KeY itself.

For simplicity, we only present one form of taclets: *rewrite taclets*. Recall the rule in Section 2.6 for symbolically executing assignments. Listing 3 defines the same rule in form of a taclets. The `assignment` rule has four parts:(i) A definition of *schema variables* matching formulas (`post`), program variables (`#loc`), and side effect-free expressions (`#se`); (ii) a `\find` clause, defining the formula "in focus," i.e., to be replaced in the premise—in this case a modality of any kind containing an assignment; (iii) `\replacewith` providing the formulas in the premises; (iv) a `\heuristics` clause, instructing the automatic prover when this rule should be applied.

```
assignment {
  \formula post;  \program Variable #loc;  \program SimpleExpression #se;
  \find(\modality{#allmodal}{.. #loc = #se; ...}\endmodality(post))
  \replacewith({#loc:=#se}\modality{#allmodal}{.. ...}\endmodality(post))
  \heuristics(simplify_prog)
};
```

Listing 3. A taclet defining the rule for symbolically executing an assignment.

Observe the opening and closing ellipses (`..` and `...`) in the modality. These stand for the inactive prefix π and the rest of the program ω, respectively, as introduced in Section 3.2.

5 Advanced Concepts for Object-Orientation

For the verification of non-trivial object-oriented programs, two specification features are important: (i) *Data abstraction* by which the content of data structures is represented using mathematical values thus hiding implementation details and

(ii) *data encapsulation* that allows reasoning about data structures locally provided that any structure operates only on memory locations belonging to itself.

5.1 Ghost and Model Fields, Model Methods

Abstraction is relevant for programs operating on non-trivial data structures, as dealing with the details and memory layouts of data structure implementations unnecessarily increases proof complexity. So it is important (and often an enabling factor) to possess means to abstract from implementation details and to work with abstract values describing and capturing the state of data structure objects. For object-oriented programs, the state of an object is often best captured abstractly in form of one or more values in mathematical data types.

The canonical abstraction of the state of a doubly linked list implementation, for example, is a sequence of its entries. The expected behavior of list operations can be described in contracts using this abstraction. A client using such lists does not need to know anything about the data structure's actual implementation. KeY supports three means to introduce abstract values as JML annotations into class files: *Ghost fields*, *model fields*, and *model methods*.

Ghost fields (and variables) are fields (and variables) that only exist for verification purposes. Since JML annotations are written in comments, they are ignored during compilation. For verification, however, ghost entities are treated like normal Java fields and variables. In particular, ghost fields give rise to heap locations as outlined in Section 3.1. Ghost entities in JML may have types which are only available in JML but not in Java. In assignments, expressions that go beyond the expressiveness of Java (like quantifiers) can be used with ghost variables. Ghost fields and variables are often used to store redundant information or intermediate results, which are not required for computations at run time, but can considerably simplify deductive verification.

The example in Listing 4 illustrates how a ghost field is used to abstract from a concrete data structure. The `List` interface declares the ghost field `content` holding the list's abstraction, which is a sequence of values. The abstraction suffices to specify the contract of method `get`, which obtains the integer value stored at index `idx`. The method `add` ensures that a value is appended to the `content`. It is specified using the sequence operator "`+`" in KeY's JML. The implementing class `ArrayList` uses an array that actually holds the list's values. The connection between the abstract list and its implementation is established via a *coupling invariant*. In this case the function `\array2seq` can be used to read the sequence of values from an array.

```
interface List {
  //@ instance ghost \seq content;

  //@ requires 0 <= idx < content.length;
  //@ ensures \result == (int)content[i];
  int get(int idx);

  //@ ensures content == \old(content) + \seq(value);
  void add(int value);
```

```
}
class ArrayList implements List {
  int[] array;
  //@ invariant content == \array2seq(array);
  ...
}
```

Listing 4. Implementation and specification of a list with model entities

Modifications of ghost fields must be made explicitly using assignments in contracts. For example, the contract of method `add` (not shown here, but available in the tutorial sources), must set `content` explicitly to the new value.

Model fields are, like ghost fields, only visible during verification and not at compile time. However, unlike ghost fields, model fields do not have a state of their own but are *observer symbols* whose value is computed from the current heap state. They are more like side effect-free Java query methods than Java fields. A model field is declared by adding the JML modifier `model`.

The benefit of model fields is that they need not (and cannot) be updated explicitly since they "automatically" change their value. However, verification of programs with model fields usually needs significantly more interactions than programs with ghost fields, and proofs tend to be larger and more complex.

Model methods are a generalization of model fields in the sense that they have arguments. They are side effect-free methods declared in JML annotations.

5.2 Dynamic Frames

Data encapsulation is closely related to data abstraction: If a data structure is well encapsulated, then its abstract value does not depend on memory areas outside the data structure. This is known as the *framing problem*: How to specify and verify that the abstract state of an object does not interfere with another unrelated object? Framing is usually addressed by requiring that the memory locations of data structures do not overlap. Over the last two decades, mainly three concepts to solve the framing problem have emerged: *Separation logic* [54], *ownership type systems* [30], and *dynamic frames* [44].

The KeY tool implements dynamic frames [58], where the set of locations that "belong" to a data structure, i.e. those locations that can be read or written by its operations are explicitly modeled as a set of memory locations, often called the *footprint* of an object. A ghost field is used to model this location set.

Revisiting the `List` example, in Listing 5 we specify at the *interface* level that the `get` query method may at most read memory locations in the footprint (using the keyword `accessible`). The function `add` may modify at most these locations (specified using `assignable`). When the footprint grows in `add`, only fresh locations that were not yet allocated prior to the call may be added to ensure that footprints remain separate. This is a typical specification pattern used when specifying and verifying object-oriented programs with dynamic frames. The `List` example is covered in the tutorial material (see Part II in Appendix **??**).

```
interface List {
  //@ instance ghost \locset footprint;
  //@ accessible footprint;
  //@ assignable \nothing;
  int get(int index);
  //@ assignable footprint;
  //@ ensures \new_elems_fresh(footprint);
  void add(int);
}
```

Listing 5. Specification pattern using dynamic frames for the list interface

The value of a query invocation can only change if an element in the footprint is modified. The following axiom is available in KeY:

$$\big(\forall o, f.\ (o, f) \in \texttt{list.footprint} \rightarrow \mathit{select}(h_1, o, f) \doteq \mathit{select}(h_2, o, f)\big) \rightarrow \\ \texttt{get}(h_1, \texttt{list}, \mathit{idx}) \doteq \texttt{get}(h_2, \texttt{list}, \mathit{idx}) \quad (5)$$

It expresses that the `get` function computes the same result in heaps h_1, h_2 if all locations in `footprint` hold the same values in h_1, h_2. When lists are known to have disjoint footprints, then the dynamic frame axiom (5) allows to infer that adding an element to one list has no influence on a query to the other list.

6 KeY as a Tool for the Community

Due to its maturity and openness, KeY is a valuable tool for the community. This includes the use of KeY as a tool for verification projects or for teaching, but also the use of KeY in research projects for building new tools on top of it.

6.1 KeY as a Tool to Verify Real-World Software

Over the years, a plethora of case studies has been conducted, where KeY was used to verify real-world algorithms and data structures. We present a selection; a more comprehensive list is on the KeY project website.

A verification case study that received much attention is TimSort, an algorithm combining merge and insertion sort. It is prominently used as Java's default for sorting collections of objects. However, that implementation had a bug and crashed for certain large collections. This issue was detected and explained in [28], a fixed version has been presented and verified with KeY in [27].

While the JDK uses TimSort to sort collections of objects, collections of primitive types are sorted using Dual Pivot Quicksort, which is a standard quicksort that partitions into three instead of into two parts. The implementation provided by the JDK has been proven correct in [19], which includes the sortedness property, the permutation property, and the absence of integer overflows.

In [23], the core of the JDK's Identity Hash Map was specified and verified. In that case study, a novelty is the use of several JML tools: KeY, the bounded

model checker JJBMC [16], and OpenJML [24], to exploit the strengths of each of them and jointly verify a large project.

Researchers at CWI showed that Java's LinkedList implementation breaks when lists with more than 2^{31} elements are created [40]. They propose a fixed version and verified it successfully with KeY. This case study shows the capability of KeY to reason about bounded integer data types and handle overflows.

The most recent large case study performed with KeY is the verification of the sorting algorithm in-place super scalar sample sort [18]. This algorithm is efficient on modern machines, as it avoids branch mispredictions, allows high instruction parallelism by reducing data dependencies in the innermost loops, and it is very cache-efficient. This case study shows that with KeY it is possible to verify state-of-the-art sorting algorithms of considerable size (in this case about 900 lines of Java) and complexity *without having to modify the source code.*

6.2 KeY for Teaching

KeY is well suited for teaching. It comes with a GUI that provides context-specific actions, such as the rules that are applicable to the specific selected term. It provides means to inspect partial proofs and to explore the state of the prover interactively. The approach and the tool are very mature, and a lot of material exists that describes them in great detail (e.g., [4,5,14]). For these reasons, KeY is used in many courses at various universities, a list can be found on the website of the project.[5] There is also a plethora of course notes and slides.

6.3 KeY as Library and Research Platform

In addition to the use of KeY as a standalone GUI-centric tool, it is possible to use KeY as a platform for research or to include it in a project as a library employing its symbolic execution and automated reasoning capabilities. One tool that uses KeY in such a way is CorC [56], which is an Eclipse-based tool that allows users to construct correct programs by stepwise refinement. To verify that the Java statements adhere to their "contracts" (pre- and postconditions created via refinement from the top-level specifications), CorC calls KeY as a backend.

KeYmaera [52] is an offspring of KeY that can be used to prove properties about cyber-physical systems, which are systems that have continuous behavior as well as discrete state changes (for example cars or planes). However, its successor KeYmaera X [33] is a green-field implementation and does not share a common code base with KeY anymore.

The Symbolic Execution Debugger [38] can be used to symbolically execute a program and obtain a tree of possible program paths. This helps to understand program and specification and to detect bugs, for example when unexpected paths are present or expected ones are missing. More recently, the Refinity tool [59] extends KeY by *abstract execution* [60] and lets one prove the correctness of refactorings. Both tools make use of KeY as a library.

[5] https://www.key-project.org/applications/key-for-teaching/

6.4 Open Source and Open Development

KeY has been open source since the inception of the project in 1999. In February 2023 the sources were moved to a public repository on GitHub.[6] The open development model facilitates bug reports and feature requests. GitHub also provides the possibility to contact the developers.

The annual KeY Symposium takes place since 2002. With an international field of participants, it has been a breeding ground for new ideas and features for KeY. Growing over the years, the most recent edition has been the largest ever with about 40 attendees. To transfer knowledge from experienced to newer developers, two hackathons have been organized (in 2018 and 2024). Both events were a great success and led to multiple new features and bug fixes.

Acknowledgments. This work was supported by the DFG projects BE 2334/9-1, BU 2924/3-1, HA 2617/9-1, and UL 433/3-1 as well as the Helmholtz topic Engineering Secure Systems (KASTEL) and the Helmholtz pilot program KiKIT.

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point Java programs in KeY. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. Proceedings. pp. 242–261. No. 12652 in LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_13
2. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (Aug 1996)
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool: Integrating object oriented design and formal verification. Software and System Modeling **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x
4. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book: From Theory to Practice. No. 10001 in Lecture Notes in Computer Science, Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
5. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives. No. 12345 in LNCS, Springer (2020)
6. Ahrendt, W., Gladisch, C., Herda, M.: Proof-based test case generation. In: Ahrendt et al. [4], chap. 12, pp. 415–451. https://doi.org/10.1007/978-3-319-49812-6_12
7. Ahrendt, W., Grebing, S.: Using the KeY prover. In: Ahrendt et al. [4], chap. 15, pp. 495–539. https://doi.org/10.1007/978-3-319-49812-6_15
8. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. Computing Research Repository (CoRR) (Sept 2020)
9. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 50 (2018). https://doi.org/10.1145/3182657

[6] https://github.com/KeYProject/key

10. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022, Proceedings, Part I. pp. 415–442. No. 13243 in LNCS, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO. pp. 364–387. Springer, Berlin, Heidelberg (2006)
12. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org (2016)
13. Beckert, B., Hähnle, R., Hentschel, M., Schmitt, P.H.: Formal verification with KeY: A tutorial. In: Ahrendt et al. [4], chap. 16, pp. 541–570. https://doi.org/10.1007/978-3-319-49812-6_16
14. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software: The KeY Approach. No. 4334 in LNCS, Springer (2006)
15. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach – Foreword by K. Rustan M. Leino. No. 4334 in LNCS, Springer (2007), https://doi.org/10.1007/978-3-540-69061-0
16. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 60–80. No. 12476 in LNCS, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
17. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Ahrendt et al. [4], chap. 3, pp. 49–106. https://doi.org/10.1007/978-3-319-49812-6_3
18. Beckert, B., Sanders, P., Ulbrich, M.: Formally verifying an efficient sorter. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th Intl. Conf. TACAS, Luxembourg City, Luxembourg. LNCS, Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
19. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 35–48. No. 10712 in LNCS, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
20. 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 (2004)
21. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Int. Conf. on integrated Formal Methodes. No. 10510 in LNCS, Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_7
22. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011)
23. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods. pp. 45–62. No. 13274 in LNCS, Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-07727-2_4

24. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 472–479. No. 6617 in LNCS, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
25. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) 1st Workshop on Formal Integrated Development Environment, F-IDE, Grenoble, France. pp. 79–92. No. 149 in EPTCS (2014)
26. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: VCC: Contract-based modular verification of concurrent C. In: Intl. Conf. on Software Engineering – Companion Volume. pp. 429–430 (2009)
27. De Gouw, S., De Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Automated Reasoning **62**(6) (2019)
28. De Gouw, S., Rot, J., De Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C. (eds.) Proc. 27th Intl. Conf. on Computer Aided Verification (CAV), San Francisco. pp. 273–289. No. 9206 in LNCS, Springer (Jul 2015)
29. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
30. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. J. Object Technol. **4**(8), 5–32 (2005). https://doi.org/10.5381/JOT.2005.4.8.A1
31. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
32. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. Springer-Verlag, New York, 2 edn. (1996)
33. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) 25th Intl. Conf. on Automated Deduction (CADE), Berlin, Germany. pp. 527–538. No. 9195 in LNCS, Springer (2015)
34. Grahl, D., Bubel, R., Mostowski, W., Schmitt, P.H., Ulbrich, M., Weiß, B.: Modular specification and verification. In: Ahrendt et al. [4], chap. 9, pp. 289–351. https://doi.org/10.1007/978-3-319-49812-6_9
35. Grahl, D., Ulbrich, M.: From specification to proof obligations. In: Ahrendt et al. [4], chap. 8, pp. 243–287. https://doi.org/10.1007/978-3-319-49812-6_8
36. Hähnle, R., Huisman, M.: Deductive verification: From pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science: State of the Art and Perspectives, pp. 345–373. No. 10000 in LNCS, Springer, Cham, Switzerland (2019)
37. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing, MIT Press (Oct 2000)
38. Hentschel, M., Bubel, R., Hähnle, R.: Symbolic execution debugger (SED). In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification, 14th International Conference, RV, Toronto, Canada. pp. 255–262. No. 8734 in LNCS, Springer (2014)
39. Hentschel, M., Bubel, R., Hähnle, R.: The Symbolic Execution Debugger (SED): A Platform for Interactive Symbolic Execution, Debugging, Verification and More. STTT **21**(5), 485–513 (Oct 2018)
40. Hiep, H.A., Maathuis, O., Bian, J., Boer, F.S.D., van Eekelen, M.C.J.D., Gouw, S.D.: Verifying OpenJDK's LinkedList using KeY. In: Biere, A., Parker, D. (eds.)

Tools and Algorithms for the Construction and Analysis of Systems, 26th Intl. Conf. TACAS, Dublin, Ireland, Part II. pp. 217–234. No. 12079 in LNCS, Springer, Cham (2020)
41. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. of the ACM **12**(10), 576–580, 583 (Oct 1969)
42. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (Aug 2008), http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf
43. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. pp. 268–283. Springer, Berlin, Heidelberg (2006)
44. Kassios, I.T.: The dynamic frames theory. Formal Aspects Comput. **23**(3), 267–288 (2011). https://doi.org/10.1007/S00165-010-0152-5
45. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)
46. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 2013), http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf, draft revision 2344
47. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference, Dakar, Senegal, Revised Selected Papers. pp. 348–370. No. 6355 in LNCS, Springer (2010)
48. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE 2014. pp. 3–15. No. 149 in EPTCS (2014)
49. Lidström, C., Gurov, D.: An abstract contract theory for programs with procedures. In: Guerra, E., Stoelinga, M. (eds.) Fundamental Approaches to Software Engineering, 24th Intl. Conf. FASE, Luxembourg City, Luxembourg. pp. 152–171. No. 12649 in LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_8
50. McCarthy, J.: Towards a mathematical science of computation. In: 2nd IFIP Congress. pp. 21–28. North-Holland (1962)
51. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. No. 2283 in LNCS, Springer-Verlag (2002)
52. Platzer, A., Quesel, J.D.: KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning. pp. 171–178. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_15
53. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: 17th Annual Symp. on Foundations of Computer Science, Houston, TX, USA. pp. 109–121. IEEE Computer Society, Los Alamitos, CA (1976). https://doi.org/10.1109/SFCS.1976.27
54. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science (LICS) 2002. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
55. Rümmer, P., Ulbrich, M.: Proof search with taclets. In: Ahrendt et al. [4], chap. 4, pp. 107–147. https://doi.org/10.1007/978-3-319-49812-6_4
56. Runge, T., Schaefer, I., Cleophas, L., Thüm, T., Kourie, D.G., Watson, B.W.: Tool support for correctness-by-construction. In: Hähnle, R., van der Aalst, W.M.P. (eds.) Fundamental Approaches to Software Engineering, 22nd Intl. Conf. FASE, Prague, Czech Republic. pp. 25–42. No. 11424 in LNCS, Springer (2019)

57. Schmitt, P.H.: First-order logic. In: Ahrendt et al. [4], chap. 2, pp. 23–47. https://doi.org/10.1007/978-3-319-49812-6_2
58. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java Dynamic Logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010, Paris. pp. 138–152. No. 6528 in LNCS, Springer (2010). https://doi.org/10.1007/978-3-642-18070-5_10
59. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: d. S. Oliveira, B.C. (ed.) Programming Languages and Systems, 18th Asian Symp., APLAS, Fukuoka, Japan. LNCS, vol. 12470, pp. 311–319. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_16
60. Steinhöfel, D., Hähnle, R.: Schematic Program Proofs with Abstract Execution: Theory and Applications. Journal of Automated Reasoning **68**(7) (2024)
61. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: AutoProof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems – 21st International Conference, TACAS, London, UK. pp. 566–580. No. 9035 in LNCS, Springer (2015)
62. Tuerk, T.: Local reasoning about while-loops. In: VSTTE Theory Workshop (VSTheory) (May 2012)