# The Karlsruhe Java Verification Suite

Jonas Klamroth[1], Florian Lanzinger[2][0000−0001−8560−6324], Wolfram Pfeifer[2][0000−0002−9478−9641], and Mattias Ulbrich[2][0000−0002−2350−1831]

[1] FZI Research Center for Information Technology, Karlsruhe, Germany
<lastname>@fzi.de
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
<firstname>.<lastname>@kit.edu

**Abstract.** Thanks to the deductive verifier KeY, the formal verification of Java programs has a long-standing tradition at Karlsruhe. The design of KeY implies some properties that can restrict its use in real-world application cases: (1) Verifying long, code-intensive methods with many instructions or bit-wise operations is difficult even if their behaviour is not overly complex, and (2) tracking formal guarantees through unverified code is difficult if not impossible using KeY.
To mitigate these weak spots, we introduce the *Karlsruhe Java Verification Suite*, a collection of formal Java verification tools that work with the Java Modeling Language (JML). Complementing KeY, the suite comprises *JJBMC*, a bounded model checker for Java and JML and the *Property Checker*, a type checker for user-defined property types.
In this paper, we first discuss formally how tools sharing a common specification language can share distributed obligations in a general setting, and then specialise this to the case of Java and our tool suite.
In a case study, we show that the Karlsruhe Java Verification Suite can verify a program that none of the three components could have proved alone.

**Keywords:** Software verification · Modular design · Design by contract · Software bounded model checking · Pluggable type systems · Deductive verification · Refinement types

## 1 Introduction

The KeY project [1] was initiated more than 20 years ago at Karlsruhe University, with Reiner Hähnle one of the founders of the project. The deductive proof engine for the formal verification of the correctness of formally specified Java code has since been an important player in the world of formal analysis of Java programs. KeY is still an active project that is now co-developed in Karlsruhe, Darmstadt and Gothenburg.

Over the years we have observed that when applying formal Java verification closer to practical application cases, there are properties of KeY which make it hard to apply KeY easily in practical situations:

1. *Some parts of a program to be verified may not fall into the fragment that KeY can handle well.*
   While the symbolic execution of Java programs in KeY models the Java semantics very precisely, it does not scale very well and verifying larger sections of code may hamper the verification significantly even if the code is loop-free and has a simple specification. Since KeY models the bounded integer types in Java using unbounded mathematical integers and modulo operations, verification of programs using bit-wise operations (like XOR) are also difficult.
   It would be beneficial if the power of KeY could be complemented by verification routines that are particularly good on these domains.
2. *It may be necessary to track formal guarantees through code outside the verified core.*
   Usually one can identify a critical core of a program onto which formal verification is applied to guarantee its correctness. However, it is not unusual that data leaves this verified core, is processed in code areas with a lower criticality level (like a user interface) and then, later, reenters the verified core. Since KeY does not scale well enough to verify entire code bases, it would be beneficial to complement KeY with approaches which scale well and allow verified properties to be propagated through large code bases.

In this paper, we present how two approaches and their corresponding tools complement KeY with functionalities that fill precisely these two gaps. It is a deliberate decision that they are not tightly integrated into KeY, but collaborate with KeY using the *Java Modeling Language (JML)* [19] as their common specification language. The rationale behind this loose coupling is that thus no technical tool-specific encoding or implementation details must be considered outside each verification tool. Using the common specification language JML as the interface makes it possible to easily incorporate other tools than ones presented here into the approach.

*JJBMC* [2] complements the deductive verification engine in KeY by a component for bounded model checking. It translates JML specifications into assumptions and assertions in the code, which can then be analysed using bounded model checking. While bounded model checking can in general not fully prove properties about programs with loops or recursion, it is well well-suited to loop-free programs with many cases, which often slow KeY's proof search to a crawl. Unlike KeY, which models Java bounded integers using unbounded mathematical integers, JJBMC can deal well with bit-wise operators.

The *Property Checker* [18] brings together the expressive power of formal specification and verification with KeY and the scalability of lightweight verification using decidable type systems by translating type qualifiers whose correctness cannot be shown by the type system into JML annotations. We will show how this combination can be used to reduce the specification and verification overhead of proving program correctness.

Since KeY and these two approaches are currently actively developed at KIT in Karlsruhe, the combination of approaches and tools form the *Karlsruhe Java*

*Verification Suite*. The collaboration between the tools works by a *distribution of proof obligations* among them. We discuss for a simple while language with embedded assumptions and assertions, how the assertions to be proved can be distributed between different approaches and when such a distribution is correct. We formally prove this in Thm. 1 in Sec. 4. We then (in Sec. 5) informally lift this result from the while language to JML-annotated Java and show how the tool collaboration there can follow the same principles as in the simpler language. We discuss a few points necessary for an application of the approach in the field (semantic coherence, proof management).

We have implemented a small wine-store example illustrating the benefits of the collaboration within the Karlsruhe Java Verification Suite. The critical core is a sorting routine which is verified using KeY, supported by JJBMC for long linear code and bit-level operations within it. The application has a graphical user interface outside the verified core. The Property Checker is used to propagate the sortedness property through the non-core code. By joining forces, the three tools can together show that the contracts are satisfied. No tool would have been able to show this alone.

The main contributions of this paper are the following:

- an approach to combine verification tools that follow different formal analysis approaches but share a common specification language,
- a formalisation of this approach for a while language with embedded assumptions and assertions and a formal correctness proof,
- the description of an instantiation of the combination idea with three tools (KeY, JJBMC, Property Checker) that collaborate using JML,
- a case study for the JML combination which the tools can only verify collaboratively.

## 2 The Java Modeling Language

The Java Modeling Language (JML) [19] serves as lingua franca for the different tools in the Karlsruhe Java Verification Suite. Therefore, this section provides a short introduction to its concepts. JML is a behavioural specification language for sequential Java programs and the de facto standard in the Java verification community. It is designed to be close to the Java language and thus comparatively easy to write and understand for Java developers. JML follows the principle of *design by contract* [21]. This means that specifications are written at the method level using *contracts*, which abstract from the method's behaviour. A contract in general consists of a precondition and a postcondition with the semantics that *if* the precondition holds at the method call, the postcondition must hold after returning from the method. The use of contracts allows one to divide the complexity of large software into smaller parts, which can then be reasoned about individually (*modular reasoning*). However, to be able to verify the contracts using a deductive program verifier, it is often necessary to provide additional helper specifications. The most prominent ones are *loop invariants* and *framing clauses*: The former can be used to conduct induction proofs over the number of

```
/*@ normal_behavior
  @  requires 0 <= a < array.length;
  @  requires 0 <= b < array.length;
  @  ensures array[a] == \old(array[b]);
  @  ensures array[b] == \old(array[a]);
  @  assignable array[a], array[b];
  @*/
public static void swap(int[] array, int a, int b) {
  int tmp = array[a];
  array[a] = array[b];
  array[b] = tmp;
}
```

**Fig. 1.** A formally specified `swap` method as an example JML method contract.

loop iterations, while the latter are used to specify an upper bound of the heap locations written by the method. These auxiliary specifications are often difficult to write and error prone. In addition to contracts, JML also supports inlined specification statements like explicit assertions or assumptions.

To be transparent to Java compilers, all JML annotations are embedded into Java comments that start with an 'at' sign after the comment delimiter (i.e. `//@` and `/*@`). JML intends to be precise on the one and concise on the other hand; therefore it imposes some useful defaults for specification, for example that fields, parameters and return values are non-null by default.

Figure 1 provides an example of a JML method contract. The keyword `normal_behavior` is used to specify that the method always terminates and does not throw an exception. Under the precondition (`requires`) that the given indices `a` and `b` are in the bounds of the array, the contract states that after execution of the method the two elements will be swapped (`ensures`). The operator `\old(...)` is used inside the postconditions to evaluate an expression in the method's pre-state rather than in its post-state. Furthermore, a framing clause (`assignable`) is given: The method is at most allowed to write to the heap locations of the two elements of the array.

A number of formal tools to reason about JML specifications has been implemented over the years, with KeY and *OpenJML* [8] the most actively developed tools today. In addition to deductive verification, OpenJML also supports run-time assertion checking. There are also other dynamic verification tools for JML like for example JMLUnitNG [24], which allows users to create unit tests with test oracles automatically generated from JML specifications.

To enable parts of the specification only for specific tools, JML brings the feature of *annotation markers*: If an annotation contains the tool name prior to the 'at' sign (e.g. `//+KeY@`), this annotation is only to be considered by the named tool, other tools have to ignore it. Likewise, specific JML clauses can be explicitly disabled for some tools via `//-<toolname>@`. Therefore, some assertions can be

discharged by specific tools, while other tools may assume them afterwards. Of course, one has to be careful not to conduct an unsound circular proof.

## 3 Tools in the Karlsruhe Java Verification Suite

The specification language JML is the common denominator and the communication means by which the components of the verification suite can interact and combine and exchange their verification results. In the following we represent the main three components of the suite (co-)developed in Karlsruhe: *KeY* as a full fledged deductive verification tool at the heart of the tool suite, *JJBMC* as a more versatile, flexible, well-scaling bounded verification tool for more lightweight static checking and the *Property Checker* as a means to check lightweight formal properties in a well-scaling type checker.

While the presentation in this paper and in the case study in Sec. 6 focus on these tools developed at KIT, the described approach is by no means limited to them. On the contrary, since the only requirement is support of the JML language, other tools that operate using this language can be naturally incorporated as well. In particular, the deductive JML verification engine OpenJML fits seamlessly into the tool suite.

### 3.1 KeY

KeY [1] is a tool for deductive verification of Java programs which are formally specified in JML. At its core is a sequent calculus working on Java Dynamic Logic (JavaDL) formulas. This logic features the modal operators $[p]$ and $\langle p \rangle$ ('box $p$' and 'diamond $p$') parametrised by a Java program $p$. The formula $[p]\psi$ is valid iff starting in any pre-state, either the program $p$ does not terminate or it does terminate and $\psi$ holds in the post-state of its execution. In contrast to that, $\langle p \rangle \psi$ is valid iff the program terminates and $\psi$ holds in the state afterwards. In dynamic logic, the Hoare triple $\{\phi\}p\{\psi\}$ with a precondition $\phi$, a program $p$, and a postcondition $\psi$ can be expressed as $\phi \rightarrow [p]\psi$. In general, dynamic logic is more expressive than Hoare logic, since the formulas can contain nested modalities again, which enables the specification of, for instance, the equivalence of two programs. In KeY, multiple modalities are used for example to formulate proof obligations for information flow in a very intuitive and natural fashion, whereas in Hoare logic additional constructs like Hoare Quadruples would have to be introduced for this.

Besides modalities, JavaDL extends first order dynamic logic by a type hierarchy suitable for Java. In particular, the types Heap, Object, Field, and LocationSet are included to be able to model and reason about memory properties of Java programs using the theory of arrays [20] and dynamic frames [16].

The usual workflow in KeY is as follows: After loading the method contract to be proven, KeY creates a JavaDL proof obligation whose validity entails the correctness of the method wrt. the contract. Next, the program is symbolically executed by applying a series of sequent calculus rules that transform the code

inside modalities into substitutions outside of them. Eventually, symbolic execution terminates in a proof tree with one or more branches which contain only first-order formulas without modalities. While in theory, the validity of formulas already in first-order logic (and thus also in JavaDL) is undecidable, in practice, it is possible to find a proof for many instances even automatically by using well-designed heuristics built into KeY. However, in case the automatic proof search fails, KeY provides the possibility to apply rules interactively, which further increases the number of provable instances.

### 3.2 JJBMC

JJBMC [2] is a command-line tool for the verification of JML-annotated Java code based on the bounded model checker JBMC [9]. The tool provides an automatic translation of JML specifications to pure Java code with additional assertions, assumptions and non-deterministic value assignments. This translation is a purely syntactical replacement function $\mathit{trans}\,(\cdot) : \mathit{JML} \cup \mathit{Java} \to \mathit{Java}$. It relies on the base idea of translating a method contract as first assuming the precondition, then executing the method body, and finally asserting the postcondition. This can be formally expressed as follows:

$$\mathit{trans}\left(\begin{array}{l}\texttt{/*@ requires } R\texttt{;}\\ \texttt{\ \ @ ensures } E\texttt{; */}\\ \texttt{\{ } B \texttt{ \}}\end{array}\right) = \begin{array}{l}\mathit{trans}(\texttt{assume } R\texttt{;})\\ \mathit{trans}(B\texttt{;})\\ \mathit{trans}(\texttt{assert } E\texttt{;})\end{array}$$

The transformation *trans* is recursively defined on all statement and expression constructors. While some Java expressions are their own translation directly, some JML-specific expressions like quantifiers require a more sophisticated translation which may involve additional code, like loops in the case of a quantification over an integer range. The JML example presented in Fig. 1 gets translated into the Java code in Fig. 2.

The bounded model checker JBMC is then able to analyse the result of the translation and thus verify each method wrt. its contract. By using JBMC as a back end, JJBMC inherits the bounded analysis semantics of JBMC:

The key idea of bounded verification is to consider only program runs which are bounded by a given threshold in loop iterations and recursive method calls. In particular, this allows the bounded analysis to unroll loops, inline method calls and thus create a finite program. While this brings along several advantages like the possibility to leave out auxiliary specification as well as being a fast and fully automatic approach, this also means that results obtained in this manner can only ever be valid up to the given threshold. If the program contains loops which may have more iterations or contains arrays which are bigger than the threshold, the result is only partial. By partial we mean that although the tool signals a successful verification, there may still be a violation of the specification for runs of the program that exceed the threshold.

```
public static void swapVerf(int[] array, int a, int b) {
  assume(array != null);
  assume(0 <= a && a < array.length);
  assume(0 <= b && b < array.length);
  int old0 = array[b];
  int old1 = array[a];
  int tmp = array[a];
  array[a] = array[b];
  array[b] = tmp;
  assert array[a] == old0;
  assert array[b] == old1;
}
```

**Fig. 2.** Result of JJBMC's JML-to-Java transformation for the `swap` method of Fig. 1. `assume` refers to a static verification-only method declared by JBMC.

In JBMC (and, hence, also in JJBMC), all data is modelled in a bit-precise fashion using bit vectors. This encoding has the advantage that bit-wise logical or shift operations can easily be formulated and reasoned about. When representing bounded Java integers using mathematical integers like in KeY, it is still possible to encode such operations, but makes reasoning significantly more difficult.

JJBMC can be used to fully verify loop-free code. But it can also be used as a means to gain confidence about a specification before conducting a full formal proof. The fully automatic bounded model checking approach allows one to check specifications early on even when auxiliary specifications like loop invariants or method contracts of subroutines are still missing. This provides a early feedback opportunity while engineering JML specifications.

### 3.3 Property Checker

*Pluggable type systems* [7] are type systems which extend a language's existing type system without changing its run-time semantics. The *Checker Framework* [10,22] is a framework for the creation of pluggable Java type systems using Java's annotation mechanism. For example, the annotation *@NonNull* and the base type `Object` can be combined into the type *@NonNull* `Object` of all objects which are not `null`. A type consisting of an annotation and a base type is called a *qualified type*, and an annotation which occurs in a qualified type is called a *qualifier*.

The advantages of pluggable type systems as a verification tool are that they are simple to use and that the type checker's run time generally scales very well with program size. On the other hand, they only provide conservative estimations of the property they are designed to show. So a nullness type checker

[3] See the original file at `https://github.com/codespecs/daikon/blob/a62c452bf4a5818271f87bd0d2ba322a18e197ee/java/daikon/PptTopLevel.java#L2087`

```
boolean is_less_equal(@NonNull VarInfo v1, @NonNull VarInfo v2) {
  @Nullable Invariant inv = null; @Nullable PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    inv = instantiate(slice);
  }
  if (inv != null) {
    @SuppressWarnings("nullness")
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

**Fig. 3.** Example of a false positive of the Checker Framework's Nullness Checker.[3]

will reject all programs in which a `NullPointerException` may occur, but it may also reject some NPE-free programs. Consider the excerpt in Fig. 3 from the Daikon Invariant Generator [11], which uses the Checker Framework to avoid `NullPointerException`s at run time. Our presentation of this example is taken from [18, Sec. 2]. The Checker Framework reports that the variable `slice` may be null when `slice.is_inv_true` is called. This is a false positive, because the implementation ensures that if the variable `inv` is non-null, the variable `slice` is also non-null.

Deductive verification, as provided e.g. by KeY, has the exact opposite advantages and disadvantages: It is rare that the correctness of a correct program cannot be proven using KeY's calculus. On the other hand, using KeY requires more expertise than using a type checker and both KeY's run time and the time required to write a correct specification scale badly with program size and complexity.

The *Property Checker* [18] is a generic framework for pluggable type systems with user-defined properties developed in the Checker Framework. This checker can be instantiated with a hierarchy of *property qualifiers*, which are qualifiers whose semantics is defined by a single Boolean expression, which may depend on the typed variable (called the *subject*). For example, *@NonNull* could be made into a property qualifier via the property $\mathit{subject} \neq \mathit{null}$. More elaborate examples for property type qualifiers can be found in the case study in Sec. 6.2. The Property Checker also supports qualifier hierarchies and parametrised qualifiers. For instance, a qualifier *@GreaterEq(int a)* can be defined by the property $\mathit{subject} \geq a$. A suitable subtyping hierarchy can be defined via $@\mathit{GreaterEq}(a) \preceq @\mathit{GreaterEq}(b) :\Longleftrightarrow a \geq b$. However, these features are not needed in the case study.

A type which is qualified with a property qualifier is called a *property type*. Property types can thus be seen as a kind of refinement types, a *refinement*

```
//@ requires v1 != null && v2 != null;
boolean is_less_equal(VarInfo v1, VarInfo v2) {
  Invariant inv = null; PptSlice slice = null;
  slice = findSlice(v1, v2);
  if (slice != null) {
    //@ assume slice != null;
    inv = instantiate(slice);
  }
  if (inv != null) {
    //@ assume inv != null;
    //@ assert slice != null;
    boolean found = slice.is_inv_true(inv);
    return found;
  }
  return false;
}
```

**Fig. 4.** (Simplified) translation of Fig. 3.

*type* being a subtype which restricts its base type by demanding that all of its instances fulfil some property [12,23].

Defining qualifiers using Boolean expressions allows the Property Checker to translate occurrences of qualifiers in a program to JML specifications, which in turn allows us to combine the scalability of type systems with the power of deductive verification: The Property Checker checks for a conservative estimation of the desired properties using simple subtyping rules over the declared hierarchy. Any occurrences of property qualifiers whose correctness cannot be established using this estimation are translated to JML assertions, to be discharged by some other JML tool. In addition, any occurrences of property qualifiers whose correctness can be established are translated to JML assumptions to aid in the proof search. The translation of our example from Fig. 3 is seen in Fig. 4: All qualifier occurrences proven by the type checker have been translated into assumptions, and all occurrences not proven into assertions. Thus, we can discharge most proof obligations using the scalable, easy-to-use type checker, but still rely on the full power of deductive verification for the trickier proof obligations.

## 4 Distributing Proof Obligations

For multiple verification tools to be able to collaboratively prove a program correct, we must first clarify how proof obligations for a program can be distributed between different tools while keeping a sound verification approach.

To this end, in this section we will not work with Java and JML, but study a simpler while language with assertion and assumption statements. For this language, we will prove that it suffices for a program to be correct that any assertion in the program be proven by any one tool while all other assertions

$$
\begin{aligned}
C ::= {} & x := T \mid \mathit{if}\ B\ \mathit{then}\ C\ \mathit{else}\ C \mid \mathit{while}\ B\ \mathit{do}\ C \mid C\ ;\ C \mid \\
& \mathit{assert}\ L : B \mid \mathit{assume}\ B \\
T ::= {} & x \mid T + T \mid T * T \mid T - T \mid 0 \mid 1 \\
B ::= {} & \neg B \mid B \wedge B \mid T = 0 \mid T > 0
\end{aligned}
$$

**Fig. 5.** Grammar for the while language with asssertions and assumptions.

can be taken as assumptions (i.e. their statements can be used in the verification process without having to be shown).

The considered while language is according to the grammar in Fig. 5, in which $x \in \mathit{Var}$ is a placeholder for a variable name from the set of variables $\mathit{Var}$. Assertions are garnished with a label $L$, which must be a unique character string within the entire program. The set $\mathit{labels}(P)$ collects all assertion labels that occur in $P$. The set $\mathit{Programs}$ is the set of all syntactically correct programs that can be produced from the non-terminal $C$ in the above grammar.

This deterministic language has the usual intuitive semantics with the set of states of an execution being $\mathit{State} = \mathbb{Z}^{\mathit{Var}}$, the set of all variable assignments. Intuitively, whenever the execution reaches an assumption whose condition fails, execution silently halts. Whenever the execution reaches an assertion whose condition fails, it raises an error. A program is correct if it does not raise an error for any initial state.

For the purposes of this paper, we only consider programs that always terminate, but assume termination silently without showing it.[4] We formally define the semantics of such programs using *assertion/assumption-traces* (short *aa-traces*): An aa-trace is a finite sequence of pairs in $A = \{\mathit{assert}, \mathit{assume}\} \times \mathit{Bool}$. Each element in the sequence denotes that an assertion/assumption has been reached and whether its condition evaluated to true or false. This is encoded as a function $[\![\cdot]\!](\cdot) : \mathit{Programs} \times \mathit{State} \to A^* \times \mathit{State}$ whose definition is shown in Fig. 6.

This definition is well-founded (despite the recursive definition for loops) since we silently only consider terminating programs. Note that failing assertions and assumptions do not end an aa-trace but let the execution continue. The formal definition of a correct program capturing the intuitive notion takes this into account by requiring that no assertion fails before an assumption has failed:

**Definition 1 (Correct Program).** *A program $P \in \mathit{Programs}$ is called correct if the trace $[\![P]\!](\sigma)$ for each initial state $\sigma \in \mathit{State}$*

1. *does not contain the pair $(\mathit{assert}, \mathit{false})$* or
2. *contains the pair $(\mathit{assert}, \mathit{false})$ only at a position after an occurrence of $(\mathit{assume}, \mathit{false})$.*

[4] It would have been possible to extend the following definitions also to non-terminating programs, but would have reduced readability without adding much insight.

$$[\![x := T]\!](\sigma) = (\epsilon, \sigma[x \mapsto T^\sigma]) \qquad (T^\sigma \text{ evaluates expression } T \text{ in } \sigma)$$

$$[\![c_1; c_2]\!](\sigma) = (s_1 \_ s_2, \sigma_2) \text{ with } (s_1, \sigma_1) = [\![c_1]\!](\sigma), \quad (s_2, \sigma_2) = [\![c_2]\!](\sigma_1)$$

$$[\![\mathit{if}\ b\ \mathit{then}\ t\ \mathit{else}\ e]\!](\sigma) = \begin{cases} [\![t]\!](\sigma) & \text{if } \sigma \models b \\ [\![e]\!](\sigma) & \text{otherwise} \end{cases}$$

$$[\![\mathit{while}\ b\ \mathit{do}\ c]\!](\sigma) = \begin{cases} (s_1 \_ s_2, \sigma_2) & \text{if } \models b \text{ with } (s_1, \sigma_1) = [\![c]\!](\sigma), \\ & \qquad (s_2, \sigma_2) = [\![\mathit{while}\ b\ \mathit{do}\ c]\!](\sigma_1) \\ (\epsilon, \sigma) & \text{otherwise} \end{cases}$$

$$[\![\mathit{assert}\ L : b]\!](\sigma) = \begin{cases} ([(\mathit{assert}, \mathit{true})], \sigma) & \text{if } \sigma \models b \\ ([(\mathit{assert}, \mathit{false})], \sigma) & \text{if } \sigma \not\models b \end{cases}$$

$$[\![\mathit{assume}\ b]\!](\sigma) = \begin{cases} ([(\mathit{assume}, \mathit{true})], \sigma) & \text{if } \sigma \models b \\ ([(\mathit{assume}, \mathit{false})], \sigma) & \text{if } \sigma \not\models b \end{cases}$$

where $s \_ u$ denotes the concatenation of two sequences $s$ and $u$.

**Fig. 6.** Definition of program semantics using assertion/assumption-traces.

Remember that in this section, we want to show that it suffices that each assertion is covered by one verification approach while all other verification engines are allowed to consider it an assumption. Therefore, we introduce the concept of *program variants*. A variant $P'$ of a program $P \in \mathit{Programs}$ is a program that can be produced from $P$ by rewriting arbitrarily many assertions into assumptions of the same conditions. For instance, `assert a; assume b` is a variant of `assert a; assert b` (but not vice versa). Since executions in variants take the same path as in the original program, but encounter potentially fewer assertions, every variant $P'$ of a correct program $P$ is also correct. In the following, we also want to refer to a verification tool $t$, which we consider to be a partial function $t : \mathit{Programs} \nrightarrow \mathit{Bool}$ which returns whether the argument is a correct program. We assume all tools are sound wrt. Def. 1.

**Theorem 1 (Distributed Proof Obligations).**
*Given a program $P \in \mathit{Programs}$ and a set of sound verification tools $T$, let for any $t \in T$ the program $P_t$ denote the variant assigned to tool $t$.*

*If $t(P_t) = \mathit{true}$ for all $t \in T$ and $\bigcup_{t \in T} \mathit{labels}(P_t) = \mathit{labels}(P)$, then $P$ is correct.*

This theorem formally captures the goal of this section: The condition $\bigcup_{t \in T} \mathit{labels}(P_t) = \mathit{labels}(P)$ encodes that every assertion (identified by its label in $\mathit{labels}(P)$) has not been rewritten into an assumption in at least one variant $P_t$. If all variants can be proven correct by their respective sound tool $t$, the joint verification effort proves that the input program is correct.

*Proof (of Thm. 1).* Let $P$ be a program according to the requirements of Thm. 1, i.e. every tool reports that its respective variant $P_t$ is correct. Let us assume

that $P$ is incorrect. There would then be an initial state $\sigma_x \in State$ with trace $s = [\![P]\!](\sigma_x)$, which is a counterexample to the correctness of $P$, i.e. the first failing verification statement is an assertion, not an assumption. Let us call the label of that assertion $L_x$ and its position in the trace $x$. All entries in the trace before $x$ are either $(assert, true)$ or $(assume, true)$.

Let us inspect a variant $P_t$ with $L_x \in labels(P_t)$ covering the assertion under observation, and its trace $s_t = [\![P_t]\!](\sigma_x)$. (Variant $P_t$ must exist as every label must be covered by some tool.) We notice that throughout the execution of the programs $P$ and $P_t$, the same operations have been executed and the same path has been taken. This implies that the conditions checked in assertions are the same, the only possible difference between $s$ and $s_t$ is that some 'assert' elements in the aa-trace $s$ have been replaced by 'assume' in $s_t$. This implies that the $x$th entry in $s_t$ must also be the first failing entry in that aa-trace, the same as in $s$. But failing this assertion entails that $P_t$ is not a correct program which is a contradiction against the assumed soundness of the tool $t$ which reported $P_t$ to be correct. So, while $P_t$ contains additional assumptions, all failing assertions will still be reported, as the assumptions are justified by other tools and, thus, do not restrict the traces of the program. □

This result allows us to distribute proof obligations in form of assertions in a while program among a set of verification approaches. This principle of distributed assertions is not limited to while programs and it remains as future work to extend the formal setting to (recursive) function invocations, their abstraction for a formal modular analysis following design-by-contract and to the more sophisticated features of the Java language (exceptions, non-standard control flow, etc.).

## 5 Tool Interaction and Integration

The central feature that allows the tools in the Karlsruhe Java Verification Suite to be integrated is the use of a common specification language: JML. In Sec. 4, we have seen how the proof obligations dispersed throughout a while program in the form of assertions (in a common language) can be distributed among a set of verification tools by building a variant for each tool. In a modular context following the design-by-contract principle (e.g. when using JML), there are fewer explicit assertions in the code and, usually, the specification goes into method contracts. Clauses in method contracts can most naturally also serve as such specification distribution points. This ties in with the concept of assumption variants from Sec. 4, as the clauses in contracts between caller and callee have both a nature of assertion and of assumption that go hand in glove. The schematic sequence diagram in Fig. 8 sketches this relationship for a method call to $m()$ with a contract with precondition *pre* (asserted by caller, assumed by callee) and postcondition *post* (asserted by callee, assumed by caller). If we now verify the two methods $n$ and $m$ with separate tools we can see that the tool for $n$ assumes the conditions verified by the tool covering $m$ and vice versa. Covering different

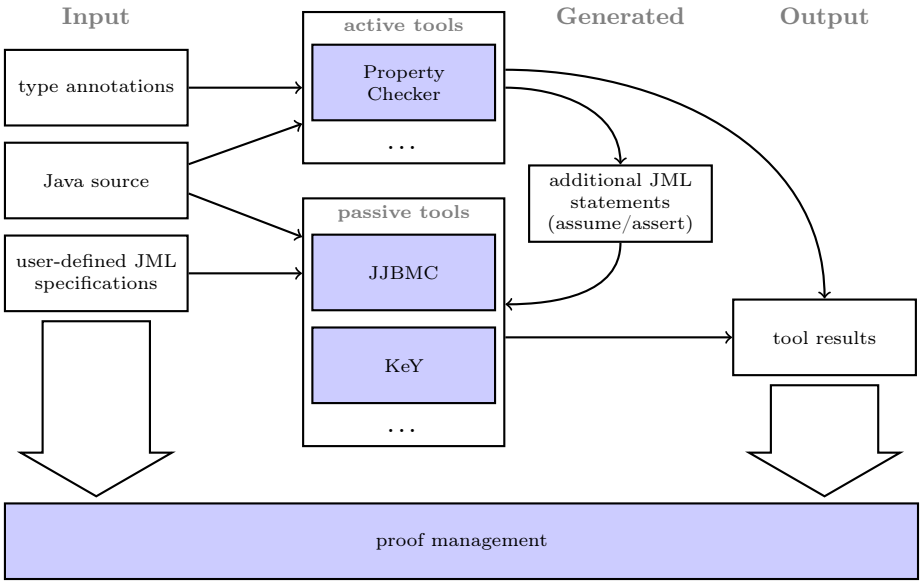



**Fig. 7.** Vision for interaction between the tools.

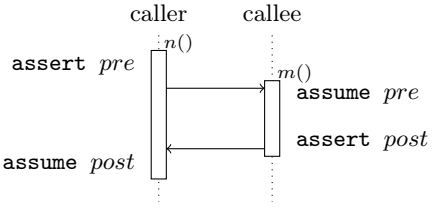



**Fig. 8.** Dual nature of specification clauses in design-by-contract

methods modularly with different JML verification tools is thus a special case of the proof obligation distribution sketched in Sec. 4.

Approaches for modular deductive verification often require extensive specifications, which in many cases is considered a major downside. But for once, the specification overhead can also be considered an advantage, as the explicit contracts allow one to distribute the verification overhead between different tools without requiring even more specification.

There are two ways in which a formal verification tool can interact via its specification language: either *passively* by interpreting specifications or *actively* by emitting specifications. Figure 7 shows the basic workflow and interaction between the tools. While JJBMC and KeY passively read and interpret JML specifications, the Property Checker does not itself digest JML specifications, but produces additional JML annotations to be verified or assumed by other tools. In either case, the tools integrate by distributing the proof responsibilities

between them. In the passive case, it is the user who decides which tool has to cover which assertion of the specification. In the active case, it is the tool that decides if other tools must verify a property (if it cannot be discharged) or may assume it (if it can be discharged).

### 5.1 JML Semantics

One major challenge for the collaboration of the different tools in the suite is the potentially different semantics that different tools might implement for annotations in the common language JML. There are a few points in the semantics of JML which have a canonical answer within each approach, but not necessarily the same one for all systems. Two such semantics questions shall illustrate the challenge:

*When do which object invariants have to hold?* This is still an active research area and the answer to the question heavily depends on the technique the tool implements to deal with heap framing (e.g. separation logic, region logic, dynamic frames, ownership, ...). While JML originally proposed an ownership approach, KeY adopted dynamic frames. It is very difficult to bring these two concepts together in general.

*What is the meaning of arithmetic operations in specifications?* Due to the nature of encoding data using bit vectors, JJBMC naturally uses strict Java 32-bit integer semantics for arithmetic operations in specifications while JML by default assumes integer arithmetic to be performed on non-overflowing mathematical integers. Fortunately, KeY has switches that allow it to treat integers compatibly to the bounded model checker.

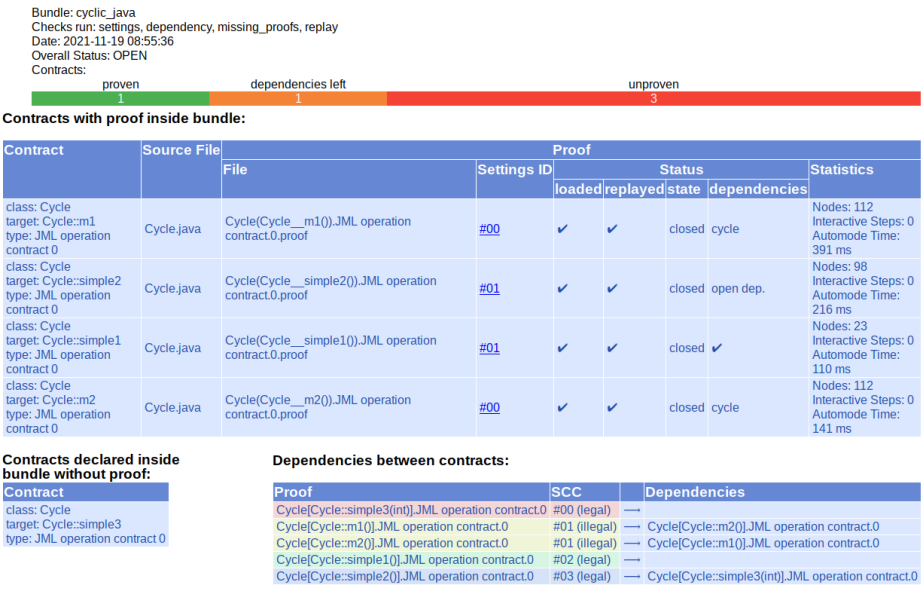
We have made sure (by manual review) that such differences do not compromise the validity of our case study. It remains as future work to either base all tools on the same semantic footing, even out differences in the specifications or at least to detect and report discrepancies. The envisioned Proof Management system (see Sec. 5.3) seems to be the ideal point to integrate this into the tool suite.

### 5.2 Formulating Program Variants Using JML

While assigning method contracts to different tools provides a natural process to distribute proof obligations between tools, it is also convenient to be able to follow the idea of assertion distribution from Sec. 4 more closely in such a setting.

Indeed, the JML annotation marker feature, which allows users to assign a JML annotation to specific tools, serves as a perfect means to express this distribution. If there are two JML-based verification tools $A$ and $B$, then the annotation `/*+A@ assert` $\phi$`;*//*+B@ assume` $\phi$`;*/` makes sure that the same condition $\phi$ is interpreted as an assertion by $A$ and as an assumption by $B$.

In the case study in Sec. 6, we have used this feature to combine bit-precise reasoning provided by JJBMC with more sophisticated reasoning in KeY. Fig. 12 shows the `swap` method that exchanges the $a$th and $b$th element of an array,



Bundle: cyclic_java
Checks run: settings, dependency, missing_proofs, replay
Date: 2021-11-19 08:55:36
Overall Status: OPEN
Contracts:

| proven | dependencies left | unproven |
|---|---|---|
| 1 | 1 | 3 |

**Contracts with proof inside bundle:**

| Contract | Source File | Proof | | | | | |
|---|---|---|---|---|---|---|---|
| | | File | Settings ID | Status | | | Statistics |
| | | | | loaded | replayed | state | dependencies | |
| class: Cycle<br>target: Cycle::m1<br>type: JML operation contract 0 | Cycle.java | Cycle(Cycle__m1()).JML operation contract.0.proof | #00 | ✔ | ✔ | closed | cycle | Nodes: 112<br>Interactive Steps: 0<br>Automode Time: 391 ms |
| class: Cycle<br>target: Cycle::simple2<br>type: JML operation contract 0 | Cycle.java | Cycle(Cycle__simple2()).JML operation contract.0.proof | #01 | ✔ | ✔ | closed | open dep. | Nodes: 98<br>Interactive Steps: 0<br>Automode Time: 216 ms |
| class: Cycle<br>target: Cycle::simple1<br>type: JML operation contract 0 | Cycle.java | Cycle(Cycle__simple1()).JML operation contract.0.proof | #01 | ✔ | ✔ | closed | ✔ | Nodes: 23<br>Interactive Steps: 0<br>Automode Time: 110 ms |
| class: Cycle<br>target: Cycle::m2<br>type: JML operation contract 0 | Cycle.java | Cycle(Cycle__m2()).JML operation contract.0.proof | #00 | ✔ | ✔ | closed | cycle | Nodes: 112<br>Interactive Steps: 0<br>Automode Time: 141 ms |

**Contracts declared inside bundle without proof:**

| Contract |
|---|
| class: Cycle<br>target: Cycle::simple3<br>type: JML operation contract 0 |

**Dependencies between contracts:**

| Proof | SCC | | Dependencies |
|---|---|---|---|
| Cycle[Cycle::simple3(int)].JML operation contract.0 | #00 (legal) | ⟶ | |
| Cycle[Cycle::m1()].JML operation contract.0 | #01 (illegal) | ⟶ | Cycle[Cycle::m2()].JML operation contract.0 |
| Cycle[Cycle::m2()].JML operation contract.0 | #01 (illegal) | ⟶ | Cycle[Cycle::m1()].JML operation contract.0 |
| Cycle[Cycle::simple1()].JML operation contract.0 | #02 (legal) | ⟶ | |
| Cycle[Cycle::simple2()].JML operation contract.0 | #03 (legal) | ⟶ | Cycle[Cycle::simple3(int)].JML operation contract.0 |

**Fig. 9.** Example of a report generated by our proof management tool.

implemented using XOR operations. JJBMC considers the swapping property in the JML annotation as an assertion, whereas KeY is allowed to assume it. JJBMC can prove this property easily while proving it is out of scope for KeY (which models Java integers using mathematical integers and cannot deal with bit-level operations). But with KeY being allowed to assume this property, it can then proceed and use it to prove a more sophisticated permutation property which would have been out of scope for JJBMC.

### 5.3 Proof Management

In the current state of the Karlsruhe Java Verification Suite, it has to be ensured by the user that the verification responsibilities are distributed soundly among the tools according to Thm. 1. While for simple cases, the proof coverage and the absence of cycles can still be checked manually, for large projects, it is essential to have some kind of proof management. For multiple KeY proofs, this gap is already closed by a command line tool which can be used to check that (a) the proofs can be reloaded and checked, and also match the Java source code and JML specifications, (b) there are no illegal[5] cyclic dependencies in the proofs, (c) all contracts a proof depends on are proven as well, and (d) the settings are compatible. This last point ensures that the semantics, for instance the meaning of arithmetic operations, is the same for each proof.

[5]To be able to reason about (mutually) recursive methods, cyclic dependencies are allowed as long as a termination witness is provided for each of them.

The proof management tool generates an HTML report from a set of input proofs that contains all the information described above. Fig. 9 shows an example of such a report for a source file containing 5 contracts, of which 4 are proven. However, one contract still has open dependencies, which means that the proof uses another contract that is still not discharged. In addition, there is an illegal cycle detected; therefore the two contracts in the cycle are considered unproven.

At the moment, this tool manages only KeY proofs. For the future however, we envision a proof management tool that also integrates results from JJBMC and the Checker Framework and possibly other tools such as OpenJML, and which therefore serves as a connection between the tools of the Karlsruhe Java Verification Suite. The goal is that if the proof management report indicates that all proofs have been done, then the proof obligations have been distributed successfully and the project has been correctly verified.

## 6 Case Study

We implemented a small Java-based shop GUI (shown in Fig. 10) to illustrate how the different components of the Karlsruhe Java Verification Suite can collaborate to prove a system correct. In the application, the user can buy different types of wine. Whenever they click on the button beside a wine, a number representing its price is added to the shopping basket at the top of the GUI. The items in the shopping basket are always kept in sorted order. The code contains a sorting routine whose verification falls clearly in the domain of deductive verification with KeY. However, there is a base case which is better handled using JJBMC. The client program uses GUI-specific code and could not even be loaded into KeY. Luckily, the sortedness property of the basket can be propagated through the GUI code using property types. All components of the Karlsruhe Java Verification Suite have to join forces to prove this program correct.

Figure 11 shows a class diagram for the case study. In addition to the `GUI` class, we have a `Shop` class containing the products and prices as well as a `Basket` class which encapsulates the user's shopping basket in an instance of `ImmutableArray`. `ImmutableArray` uses a sorting algorithm provided by the class `Quicksort`.

### 6.1 Library code: Quicksort With Explicit Base Case

As a library function for the web shop, we provide a method to sort the elements of an array. We follow a common practice in algorithm engineering by efficiently sorting the given array with Quicksort until we reach a small enough array (less than six elements) for which we employ a specialised sorting network. This approach proves to be faster than Quicksort in practice. We also use the in-place `swap` method using repeated XOR operations from Fig. 12 to swap elements in the array. The interplay between KeY and JJBMC for this method has already been explained in Sec. 5.2.

The sorting network implementing the base case with at most 5 elements (taken from [6]) is free of loops, but contains 12 consecutive `if` statements: It is

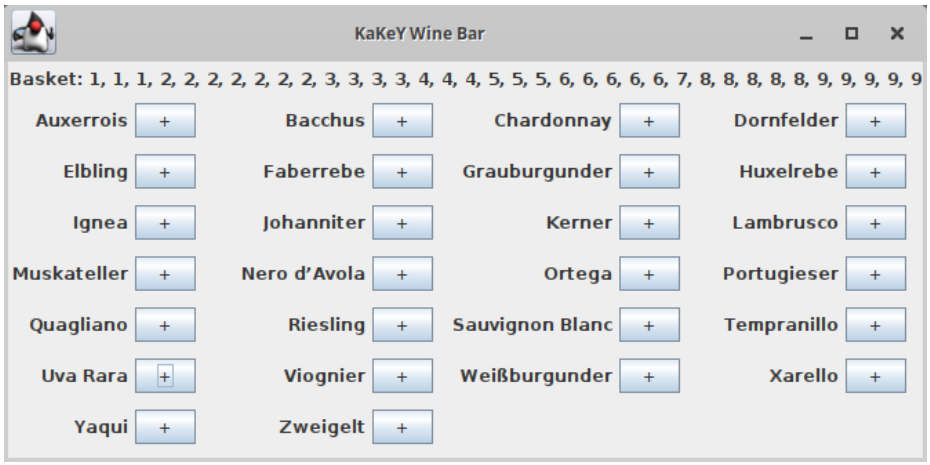



**Fig. 10.** Graphical user interface for the case study.

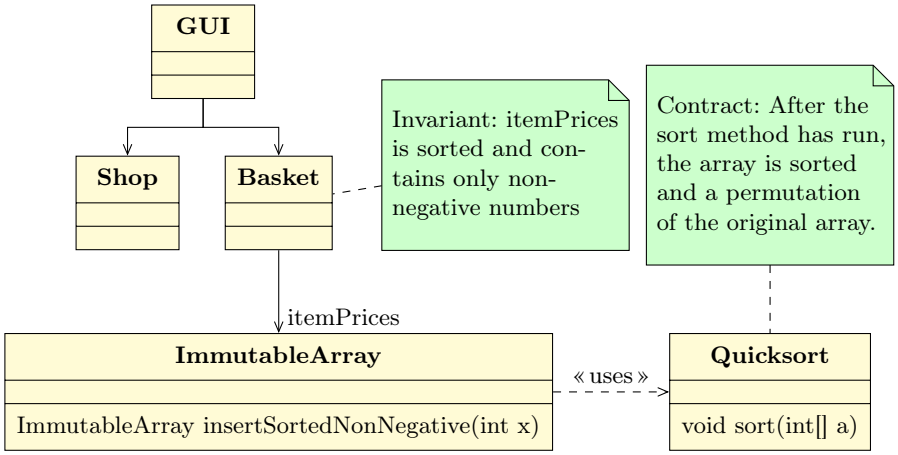



**Fig. 11.** Class diagram for the case study.

```
void swap(int[] arr, int a, int b) {
  if(a != b) {
    arr[a] ^= arr[a];
    arr[a] ^= arr[b];
    arr[b] ^= arr[a];
    //+KeY@   assume arr[a]==\old(arr[b]) && arr[b]==\old(arr[a]);
    //+JJBMC@ assert arr[a]==\old(arr[b]) && arr[b]==\old(arr[a]);
  }
}
```

**Fig. 12.** In-place swap of integers using XOR operations. (Its JML contract is the one shown in Fig. 1.)

$$@Sorted \triangleq subject \neq null \land subject.arr \neq null \land \texttt{Util.isSorted}(subject.arr)$$
$$@NonNegArray \triangleq subject \neq null \land subject.arr \neq null \land \texttt{Util.isNonNeg}(subject.arr)$$
$$@NonNeg \triangleq subject \geq 0$$

**Fig. 13.** Property qualifier definitions. The identifier *subject* refers to the variable receiving the type annotation.

thus a natural fit for bounded model checking, whereas KeY chokes on the input due to the large number of paths it has to traverse during symbolic execution.[6] In contrast, the actual Quicksort implementation with several unbounded loops and arrays is where KeY shines. This routine is a perfect example of how different tools can play together in order to conduct proofs that neither of them would be able to find alone.

The base case verifies in JJBMC within 13 minutes.[7] Sortedness is shown for bound k=6, the permutation property for k=5 (greater bounds resulted in a timeout). The analysis of the swap method is instantaneous. For the verification of the different parts of the Quicksort algorithm, KeY runs in automatic proof search mode for about 11 minutes in total[7]. For the permutation property, a few manual rule applications (which can be captured in a proof script) are needed to guide quantifier instantiation in the prover.

### 6.2 Library code: Immutable Arrays With Sortedness and Non-negativity

In the application, we want to be able to obtain immutable array instances which are known to only contain non-negative numbers in sorted order. Since references to arrays with such properties will also be handed around in GUI code, the plan

[6] There are verification tools which avoid this exponential blowup of proof obligations, but other tools relying on symbolic execution would suffer under the same problem.

[7] on a PC with an AMD Ryzen 7 PRO 4750U (8x1.7GHz) CPU and 32GB RAM

```
public static
@NonNegArray ImmutableArray
insertSortedNonNegative(@Sorted @NonNegArray ImmutableArray ia,
                        @NonNeg int newElement) {
  int[] newArr = new int[ia.arr.length + 1];
  System.arraycopy(arr, 0, newArr, 0, ia.arr.length);
  newArr[ia.arr.length] = newElement;
  Quicksort.sort(newArr);
  return new ImmutableArray(newArr);
}
```

**Fig. 14.** The `ImmutableArray` methods which need to be proven in KeY.

is to capture these conditions as property types to not have to load GUI code into the deductive verifier. The required qualifiers are shown in Fig. 13, where the qualifiers *@Sorted* and *@NonNegArray* can be applied to instances of the class `ImmutableArray`, and *@NonNeg* can be applied to int values. The type property definitions make use of the helper methods `Utils.isSorted`, which returns `true` if and only if the array is sorted, and `Utils.isNonNeg`, which returns `true` iff all elements in the array are $\geq 0$.

Next, we use these qualifiers to specify the method `insertSortedNonNegative()` (shown in Fig. 14) which takes a non-negative array and a non-negative number, returns a new sorted non-negative array. Since the implementation of property types is currently limited to immutable objects, the method returns a new object instead of modifying the argument array.

To obtain the well-typedness of the method, its return type must be correct, i.e. the returned array must (a) be sorted, and (b) contain only non-negative elements. The type-checking algorithm behind the Property Checker cannot look inside the type qualifier definitions and take their semantics into account. Instead, it only checks whether syntactic typing rules are respected; e.g., the right-hand side of an assignment must evaluate to a subtype of the left-hand side's type. Thus, the Property Checker is unable to prove the well-typedness of this method. The fact that the returned array is sorted could be shown by the checker if `Quicksort.sort()` were specified using property types. But since it was specified directly in JML instead, the checker cannot use its specification. The fact that the returned array contains only non-negative elements follows from the fact that `Quicksort.sort()` returns a permutation of the original array, and also cannot be established by the checker. Hence, it translates the type qualifiers of the returned value into JML assertions. The types of the method parameters, on the other hand, are guaranteed and are translated into JML assumptions. KeY is then able to discharge the proof obligations that arise from the assertions. This requires 85 seconds in KeY's automatic mode[8] and two manual quantifier instantiations.

[8] on a PC with an AMD Ryzen 7 PRO 4750U (8x1.7GHz) CPU and 32GB RAM

```
(ActionEvent e) -> {
  basket.prices = basket.prices.insertSortedNonNegative(price);
  ...
}
```

**Fig. 15.** An assignment which passes along a type property and can be verified by the Property Checker.

### 6.3 Client code: The web shop GUI

The method in the class `Quicksort` has now been formally verified, (Sec. 6.1) and a formal connection between type qualifiers and their defining predicates has been established (Sec. 6.2). The remaining classes in the case study do not establish any type properties, but only use them and pass objects having the property along. The well-typedness of these classes can be proven by the Property Checker. For example, consider the action listener in Fig. 15, which is executed whenever the user presses a button. It updates the shopping basket with the price of the newly chosen product. The fact that this assignment preserves the type properties of the shopping basket – i.e. that `basket.prices` is sorted and non-negative – follows directly from the well-typedness of `insertSortedNonNegative()` and is verified by the Property Checker. The total run time of the Property Checker for this case study is 8 seconds.[8]

Unlike KeY, the Property Checker supports code using features of Java 8 and later like lambda functions, making it well-suited to this kind of client code, which would otherwise have to be rewritten for KeY and specified in JML just for this relatively superficial formal treatment.

### 6.4 Conclusion

This case study demonstrated how multiple verification tools can be combined to prove the correctness of a program that is not wholly accessible to either of the tools. While sorting routines generally fall in KeY's domain, highly optimised routines like the one analysed here often have base cases with long `if` cascades, which are onerous to prove in KeY. JJBMC has no problem with this kind of code, but can only show bounded correctness in code with loops. But since the base case is only used for arrays with bounded size, JJBMC can give us a total correctness guarantee. Our program also has a graphical user interface, which we can only analyse in KeY or JJBMC after writing lots of boilerplate JML specifications for the GUI code. Even then, we still have to do a lot of work just to prove how the sortedness property is propagated through the GUI methods. Using a pluggable type system and the Property Checker, this task is less burdensome: We simply annotate all variables which should be sorted with an appropriate qualifier and run the Property Checker. On the other hand, the Property Checker is unable to reason about the methods which establish the sortedness property, which we

instead have to prove in KeY or JJBMC. Thus, all components of the Karlsruhe Java Verification Suite have to join forces to prove this program correct.

## 7 Related Work

An overview over existing approaches to combine multiple verifiers is discussed by Beyer and Wehrheim [5]. In their classification system, our approach would fit into the category 'cooperation of tools viewed as black box objects', since our tools only communicate via an interface (JML in our case), do not need to know anything about their internals, and cooperate on intermediate results of the verification (for instance, well-typedness information from type checker is passed via additional assumptions to KeY).

In [14], Jacobs presents a technique to construct a correctness witness from multiple partial analysis results. As opposed to our work for combining a type system, an interactive deductive verifier, and a bounded model checker, the technique presented there is targeted towards model checkers and static analysers which have an explicit notion of visited and checked states and record these states via so called abstract reachability graphs. On a practical level, as they implemented their technique using the CPAChecker framework [4], their technique works for programs written in C, while our approach works for Java.

There are many other deductive verification approaches which combine different tools to check the correctness of one program.

Type checkers for refinement type systems like *LiquidHaskell* [23] use SMT solvers internally, which allows them to be more powerful than conventional stand-alone type checkers. *LiquidJava* [13] applies this idea to Java: Refinements similar to our property types are specified using Java annotations and the proof obligations arising from them are translated to SMT.

*Hybrid type checking* [17] combines static type checks at compile time with dynamic checks at run time. The static type checking process has three possible results: 1. *definitely well-typed*, where the program's type safety was able to be established at compile time, 2. *definitely ill-typed*, where the compile-time checks found a definitive error, and 3. *unknown*, where some parts of the program were proven to be type-safe, and the other parts had dynamic run-time casts automatically inserted where necessary. JJBMC can distinguish between definitive incorrectness and bounded correctness. KeY too can in some cases generate counterexamples for incorrect programs using SMT. In contrast to hybrid type checking, which combines a compile-time type checker with run-time checks, our approach combines multiple compile-time tools, but it could also be extended to include run-time tools where the compile-time verification fails.

*RustBelt* [15] is an approach which proves the soundness of Rust's ownership type system. Programs written in a safe subset of Rust are proven to always be sound, and for library code using unsafe features, verification conditions for Coq [3] are generated. Thus, the correctness of a Rust program using such a library is proven by a combination of the Rust type checker and Coq. In contrast

to RustBelt, which is focused on ownership properties, our approach is based on the Java Modeling Language, allowing us to prove general functional properties.

## 8 Conclusion and Future Work

We introduced the Karlsruhe Java Verification Suite, a collection of Java verification tools (co)-developed in Karlsruhe built around the deductive verifier KeY, which next to KeY includes the bounded model checker JJBMC and a type checker called the Property Checker. We showed how proof obligations can be soundly distributed between different verification tools and how this distribution can be implemented for JML. We also demonstrated using a small case study how the Karlsruhe Java Verification Suite can be used to verify the correctness of a program which would have required a large refactoring and specification overhead if we had only used KeY.

In the future, we plan to investigate how other kinds of verification tools can be used to expand the Karlsruhe Java Verification Suite. We also plan to refine an existing proof management tool such that it can be used to orchestrate proofs with proof obligations distributed over the tool suite. In particular, the management tool has to be able to deal with differences in the interpretation of JML between different verification tools.

## References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (Dec 2016). https://doi.org/10.1007/978-3-319-49812-6
2. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2020). Lecture Notes in Computer Science, vol. 12476, pp. 60–80. Springer (Oct 2020). https://doi.org/10.1007/978-3-030-61362-4_4
3. 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). https://doi.org/10.1007/978-3-662-07964-5
4. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
5. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 143–167. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_8

6. Bingmann, T., Marianczuk, J., Sanders, P.: Engineering faster sorters for small sets of items. Software: Practice and Experience **51**(5), 965–1004 (2021). https://doi.org/10.1002/spe.2922, `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2922`
7. Bracha, G.: Pluggable type systems. In: OOPSLA'04 Workshop on Revival of Dynamic Languages (Oct 2004)
8. 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. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_35
9. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Computer Aided Verification (CAV). LNCS, vol. 10981, pp. 183–190. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
10. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.: Building and using pluggable type-checkers. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 681–690. ICSE 2011, Association for Computing Machinery (05 2011). https://doi.org/10.1145/1985793.1985889
11. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1–3), 35–45 (Dec 2007). https://doi.org/10.1016/j.scico.2007.01.015
12. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. p. 268–277. PLDI '91, Association for Computing Machinery, New York, NY, USA (1991). https://doi.org/10.1145/113445.113468
13. Gamboa, C., Santos, P.A., Timperley, C.S., Fonseca, A.: User-driven design and evaluation of liquid types in Java. CoRR **abs/2110.05444** (2021), `https://arxiv.org/abs/2110.05444`
14. Jakobs, M.C.: $\text{PART}_{\text{PW}}$: From partial analysis results to a proof witness. In: Cimatti, A., Sirjani, M. (eds.) Software Engineering and Formal Methods. pp. 120–135. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66197-1_8
15. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL) (Dec 2017). https://doi.org/10.1145/3158154
16. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 268–283. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11813040_19
17. Knowles, K., Flanagan, C.: Hybrid type checking. ACM Transactions on Programming Languages and Systems **32**(2) (Feb 2010). https://doi.org/10.1145/1667048.1667051
18. Lanzinger, F., Weigl, A., Ulbrich, M., Dietl, W.: Scalability and precision by combining expressive type systems and deductive verification. Proceedings of the ACM on programming languages **5**(OOPSLA), Article no: 143 (Oct 2021). https://doi.org/10.1145/3485520
19. 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//refman/jmlrefman.pdf`, revision 2344

20. Mccarthy, J.: Towards a mathematical science of computation. In: In IFIP Congress. pp. 21–28. North-Holland (1962). https://doi.org/10.1007/978-94-011-1793-7_2
21. Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (oct 1992). https://doi.org/10.1109/2.161279
22. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: International Symposium on Software Testing and Analysis. pp. 201–212. ISSTA, ACM, Association for Computing Machinery (2008). https://doi.org/10.1145/1390630.1390656
23. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton Jones, S.: Refinement types for Haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 269–282. ICFP '14, Association for Computing Machinery (September 2014). https://doi.org/10.1145/2628136.2628161
24. Zimmerman, D.M., Nagmoti, R.: JMLUnit: The next generation. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software. pp. 183–197. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_13