Modular Verification of JML Contracts Using Bounded Model Checking

Bernhard Beckert[1,2], Michael Kirsten[1][0000−0001−9816−1504], Jonas Klamroth[2], and Mattias Ulbrich[1]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] FZI Research Center for Information Technology, Karlsruhe, Germany
{beckert, kirsten, ulbrich}@kit.edu, klamroth@fzi.de

Abstract. There are two paradigms for dealing with complex verification targets: Modularization using contract-based specifications and whole-program analysis. In this paper, we present an approach bridging the gap between the two paradigms, introducing concepts from the world of contract-based deductive verification into the domain of software bounded model checking. We present a transformation that takes Java programs annotated with contracts written in the Java Modeling Language and turns them into Java programs that can be read by the bounded model checker JBMC. A central idea of the translation is to make use of nondeterministic value assignments to eliminate JML quantifiers. We have implemented our approach and discuss an evaluation, which shows the advantages of the presented approach.

Keywords: Software verification · Modular design · Design by contract · Software bounded model checking

1 Introduction

Over the last decades, the reach and power of formal methods for program verification has increased considerably. However, at some point, one has to face the complexities of real-world systems. There are two paradigms for dealing with complex verification targets. (1) Modularization and (de-)composition using contract-based specifications: Components – typically methods or functions – are verified separately, and can then be replaced by their abstract contracts for verifying the overall system. (2) Whole-program analysis, where the search space is restricted by over- or under-approximating the set of reachable states. While modular verification is most often performed using a deductive verification engine relying on some form of theorem prover, whole-program verification applies techniques like predicate abstraction, abstract interpretation or bounded model checking to reduce the size of the state space.

Here, we focus on bounded model checking where the search space is restricted using bounds on the number of loop iterations and the size of data structures. While modularization requires user interaction to specify the components, software bounded model checking is fully automatic, but comes at the

cost of potential false negatives that miss program failures beyond the chosen bounds. In this paper, we present an approach that bridges the gap between the two paradigms by introducing concepts from the world of contract-based deductive verification [12,23] into the domain of software bounded model checking [5]. Our method enables a software bounded model checker to verify properties of components (methods) written in a contract-based specification language. This allows for modular proofs in a software bounded model checking context. The proofs can also be hybrid where only some parts are modular in an otherwise monolithic proof. We envision three main application areas: (1) The reach of software bounded model checking is extended. While many program parts can be dealt with using exhaustive search, other parts need to be decomposed in order to verify them for non-trivial bounds. This may even allow for an increase of the bounds for the non-modular parts to the point where the software bounded model checker can explore the full search space. (2) Software bounded model checking can be combined with deductive program verification, where those components that – even after decomposition – cannot be handled by a model checker can be verified using a deductive verification tool. (3) Our bridging approach has the potential of being a valuable tool during the engineering phase of a deductive proof. Typically, formulating contracts and constructing a proof in a deductive verification tool requires several iterations of adjusting either code or specification until a proof is found. A software bounded model checker that can handle contracts may be used to spot bugs in the specification and the code before a full deductive verification is started.

While the concepts behind our approach apply to a range of languages and tools, in the following we target the Java programming language, bringing together two important players in formal methods for Java: The Java Bounded Model Checker (JBMC) [10] meets the Java Modeling Language (JML) [19]. We present a transformation that takes Java programs annotated with contracts written in JML and turns them into Java programs that can be read by JBMC, i.e., the JML specifications are turned into Java code and annotations in the form of **`assume`** and **`assert`** statements understood by JBMC. A central idea of the translation is to make use of nondeterministic value assignments '`x = *`'[3] to eliminate (part of the) JML 'forall' and 'exists' quantifiers. Therefore, the resulting programs are not executable, but can be handled by JBMC more efficiently.

The rest of this paper starts with a brief introduction to software (bounded) model checking, deductive verification, as well as the syntax and semantics of JML in Sect. 2. Then, Sect. 3 shows the main ideas of our approach, and Sect. 4 illustrates our translation from Java with JML into Java with assertions, assumptions and nondeterministic assignments. In Sect. 5, we present a prototypical implementation[4] and evaluate our approach on multiple case studies in Sect. 6. We discuss related work in Sect. 7 and conclude in Sect. 8.

[3] The notation `x = *;` (and semantics) are borrowed from nondeterministic assignment in dynamic logic [15]. Boogie, e.g., often refers to this as the `havoc` statement.

[4] The source code is available at https://github.com/JonasKlamroth/JJBMC.

2 Background

Software Bounded Model Checking (SBMC) is a formal program verification technique that, given a program and a software property to be checked, verifies fully automatically whether the program satisfies the property [5]. In a nutshell, that question is translated into a reachability problem w.r.t. the given program. SBMC symbolically, i.e., without the need for concrete values, executes the program and exhaustively checks it for errors that could violate the given property within some given bounds that restrict the number of loop iterations and recursive method calls. Using these bounds, SBMC limits all runs through the program to a bounded length and can thereby unroll the control flow graph of the program and transform it into static single assignment form [8]. This bounded program is then translated into a formula in a decidable logic, e.g., an instance of the SAT problem. The formula is satisfiable if and only if a program run exists that violates the given software property within the given bounds. Modern SAT or SMT solvers [2, 13] can be used to check whether such a program run exists, in which case the SBMC tool constructs the corresponding problematic input and presents the counterexample to the user. If no such program run is found, that may be either because the property is actually satisfied, or because it is invalid only for runs exceeding the given bounds. In some cases, SBMC is also able to infer statically which bounds are sufficient, in order to come to a definitive conclusion. SBMC tools also permit to extend the program with nondeterministic value assignments and `assume` statements in order to restrict the values and states that are to be considered. The properties to be checked are given in the form of `assert` statements. Hence, SBMC checks whether there are any runs through the program that satisfy all encountered `assume` statements but violate an `assert` statement.

Deductive Program Verification is based on a logical (program) calculus to construct a proof for a formula expressing that a program satisfies its specification [12, 23]. Typically, deductive verification uses invariants and induction to handle loops. In order to mitigate complexity, most deductive approaches employ design by contract [21], where functions resp. methods are specified with formal pre- and postconditions. These additional annotations enable a modular verification [3], where each method is individually proved to satisfy its contract. To this end, each method – together with its contract – is translated into a formula, e.g., using some form of weakest precondition computation [11]. Method calls are replaced by the contract of the called method (instead of the method body), and loops are replaced by their invariants (instead of loop unwinding). The resulting formulas are either discharged using automatic theorem provers, e.g., SMT solvers [2], or shown to the user for interactive proof construction.

The Java Modeling Language (JML) is a specification language for Java programs that follows the design-by-contract paradigm and enables the user to annotate Java programs with modular specifications, e.g., method contracts and

```
/*@ requires 0 <= x1;
  @ ensures \result == x1 * x2;
  @ assignable \nothing;
  @*/
public int mult(int x1, int x2) {
  int res = 0;
  /*@ loop_invariant 0 <= i && i <= x1 && res == i * x2;
    @ decreases x1 - i;
    @ assignable \nothing;
    @*/
  for (int i = 0; i < x1; ++i) res += x2;
  return res;
}
```

Listing 1. An example of a method specified with JML.

loop invariants [19]. JML annotations are written in Java comments that are initiated with the character sequence "`/*@`". The syntax and semantics for JML expressions are equivalent to those of Java expressions, which additionally permits universal and existential quantifiers as well as special keywords, e.g., **`\old`** that enables the postcondition to refer to expressions before executing the method.

Consider, for example, Listing 1, where the method `mult` multiplies two integers using repeated addition. The precondition (indicated by **`requires`**) requires that both integers are non-negative; the postcondition (indicated by **`ensures`**) demands that the returned value (indicated by **`\result`**) is the product of the two parameters `x1` and `x2`. Note that, even though this program may produce an integer overflow, the specification is still correct, as JML and Java have the same integer semantics. Moreover, the **`assignable`** clause restricts the heap locations which the method may change. The keyword **`\nothing`** requires that no heap location may be changed. In case we allow the method to change existing heap locations, we would specify a sequence of *storage references* (either field accesses `o.f`, object accesses `o.*` meaning that all fields of `o` may be written, or array access ranges `a[`i`..`j`]` meaning that any index between i and j in array `a` may be written). JML also permits to give auxiliary specifications, e.g., loop invariants to specify the behavior of a loop, that are specified inside the method body. The loop invariant in Listing 1 specifies that, for each loop execution, the currently computed result `res` is equal to the value of the loop variable multiplied by the second parameter `x2`. Loop invariants may also be extended by a **`decreases`** clause that specifies an integer expression which must strictly decrease in every loop iteration and never become negative. Since infinite strictly decreasing sequences are not possible within the domain of the natural numbers, this clause permits to prove termination of the loop. While JML encompasses many more concepts, we assume in the rest of this paper that method contracts are desugared, i.e., they adhere to the description from above [1, 19].

There are two deductive verification tools available for JML-annotated Java code: the KeY tool and OpenJML. The KeY tool supports both automatic and interactive verification [1]. KeY's support for user interaction permits deductive verification w.r.t. expressive specifications. OpenJML is an automatic verification tool for verifying JML annotations [9]. The JML proof obligations are first reduced to SMT formulas which are then discharged by SMT solvers.

The Java Bounded Model Checker (JBMC) is an extension of the C Bounded Model Checker (CBMC) and performs (software) bounded model checking on Java bytecode for a bit-accurate verification of Java programs by combining SAT/SMT solving with a full symbolic state-space exploration [10]. It includes an exact and verification-friendly model of standard Java library classes. Behavioral subtyping is handled by conducting a case distinction over all possible implementations expanding their respective method bodies. JBMC supports all control flow mechanisms of Java including exceptions. The tool is fully automatic and its scalability depends mainly on the complexity of string operations, loops, recursion and floating-point arithmetic in the analyzed code.

3 The Main Ideas Behind the Approach

At the base of our approach is the assumption that the reach of software bounded model checking is extended when modularization is added, which comes with three individual arguments as following. (1) While many program parts can be dealt with using exhaustive search, other parts need to be decomposed in order to verify them for non-trivial bounds. (2) For devising a formal program specification, it is often worthwhile to early on either gain trust in its validity or uncover its incorrectness already for a bounded domain or scope. Exploiting this *small scope hypothesis* [16] lets us do effective program verification within a bounded scope and mitigate the otherwise common state space explosion. However, prominent examples such as the TimSort algorithm show that the more labor-intensive deductive program verification within a universal scope is generally desirable [14]. (3) With the approach taken in this paper, we enable a powerful combination of both methodologies on a modular level, such that a verification engineer can avoid wasting time in labor-intensive interactive verification when guarantees within a bounded scope suffice. The bounded scope in our case does not only refer to unwindings and recursion inlinings, but also to data structures. With data always being finite, program verification becomes a theoretically decidable and in many cases practically manageable problem. Our approach gives the user a fine-grained control as to which degree or which parts and how much of the program to verify either within a bounded scope or deductively. The communication between both verification techniques happens on specification level via method contracts, loop invariants, or block contracts, making use of the design-by-contract paradigm (see Sect. 2).

Consider, e.g., the common case where the user develops a method together with some inner helper method. For the deductive verification scenario, the outer

method would have a method contract corresponding to its API. However, also the inner method would need a contract, which is not known yet when it is still being developed. In this early development stage, the user can rapidly gain confidence in possible contracts for this inner method by employing a modular proof within a bounded scope, as no user interaction is needed. Once the development of the method is finished, the user can opt for employing an unbounded modular proof, after gaining confidence that the proof will succeed. Often, the size of modules for SBMC can be considerably larger than that for deductive verification scenarios where every small method is individually specified. We automatically translate proof obligations induced by modular specification contracts into special code constructs that let the SBMC tool restrict the state space to the one defined by the precondition and insert assertions into the code that are equivalent to the postcondition. This relieves us from manually creating an execution harness for the whole-program approach that SBMC otherwise takes, which inlines method invocations. Similar to the technique of runtime assertion checking or runtime verification, the necessary abstractions from, e.g., method contracts are automatically encoded in assertions that are inserted into the program (see Sect. 7). However, unlike in runtime verification, we insert statements into the code that are only useful for static verification, namely **`assume`** statements and nondeterministic value assignments. These additional statements enhance the expressiveness and efficiency for static verification, but alter the execution semantics of the program. Quantifiers that cannot be represented by nondeterminism are translated into loops that iterate over the quantified domain. We evaluate such expressions in additional statements that implement side computations.

Within our formal translation rules described in Sect. 4, we reflect the distinction of side computations and computing the value of an expression E by splitting the translation E into two parts: (a) a command translation $[\![E]\!]^{cmd}$ and (b) a value translation $[\![E]\!]^{val}$. We make use of nondeterministic assignment when translating quantifiers by using a form of skolemization – instead of translating into a loop with many assertions. In order to express, e.g., that all elements of an array `a` are positive, instead of

```
for (int i = 0; i < a.length; i++){ assert 0 < a[i]; },
```

we generate the following more efficient, yet equivalently valid translation:

```
int i := *; assert !(0 <= i && i < a.length) || 0 < a[i];
```

The latter encoding makes use of the builtin nondeterministic choice operation of SBMC to make sure that all possible valuations are covered, whereas the former translation makes this explicit by iterating over all possible values. For the latter encoding, the assertion is violated once there exists a value within the bounds which makes the assertion invalid. The advantage of nondeterministic choice is that the instantiation task is given to a SAT or SMT solver that is optimized to cover all cases in a more clever way than by naive explicit enumeration.

4 Translating JML Annotations

Basics. This section describes how a Java program with JML annotations (in particular method contracts and loop invariants) can be translated into Java code ready for the analysis with the bounded model checker JBMC. The target language is Java code without JML annotations as the annotations are replaced by additional Java statements. The additional Java code includes statements that are interpreted by the model checker in a particular way: `assume` statements and nondeterministic value assignments. While we present them as keyword statements in this paper, they are expressed as special method invocations in the actual implementation. The meaning of an assertion `assert` c`;` is the usual one of Java: A program run is considered failing if the assertion is reached and the evaluation of the `assert`ed proposition c evaluates to false. In contrast, if in the statement `assume` c`;` the condition c evaluates to false, then the program run is not considered failing, but irrelevant. One may think of this as a graceful, but abrupt termination of the program at this point. The nondeterministic assignment `x = *;` assigns an arbitrary, nondeterministically chosen, not further constrained value of `x`'s static type to `x`. When such an assignment is reached multiply during a program run, each time a different value may be chosen.

Formally, our translation is defined as a syntactical replacement function

$$[\![\cdot]\!] : \mathit{JML} \cup \mathit{Java} \to \mathit{Java}$$

that takes Java annotated with JML and returns Java constructs without JML (but with assumptions and nondeterministic assignments). The translation is recursively applied as a rewriting rule to the program in a top-down fashion. In the following subsections, we present the most noteworthy rewriting rules that define $[\![\cdot]\!]$, but refrain from providing a complete list due to space limitations. We focus on a subset of Java and JML, where

- method calls only appear standalone in the form lhs `=` o`.m(`a_1`,`a_2`,...)`,
- `break` and `continue` statements do not occur, and
- `try-catch` statements do not occur.

This is not a fundamental restriction; additional rules for handling these features can easily be added. Our implementation, in fact, supports already a considerably larger subset of Java and JML than shown in this paper (see Sect. 5).

4.1 Translating Method Contracts

As design by contract targets individual methods (and not the whole program) for a method-modular program analysis, we start with a translation pattern in (1) that covers blocks of pre- and postconditions for method contracts. This easily extends to classes by applying the translation to all methods in a class.

$$\left[\!\!\left[\begin{array}{l}\texttt{/*@ requires } R\texttt{;}\\ \texttt{\ \ @ ensures } E\texttt{;}\\ \texttt{\ \ @*/}\\ \texttt{\{ } B \texttt{ \}}\end{array}\right]\!\!\right] = \texttt{\{ } [\![\texttt{assume } R]\!]\texttt{; } [\![B]\!]\texttt{; } [\![\texttt{assert } E]\!]\texttt{; \}} \quad (1)$$

The translation of the precondition R is **assume**d before the block and the translation of the postcondition E is **assert**ed after its execution. The translation's goal is that any program that satisfies the block contract on the left does not fail any assert in the program on the right and vice versa. This encoding schema is also the basis for the translation schema for methods with contracts. However, it is important that the control flow in B does not bypass the assertion at the end of the block (e.g., by throwing an exception).

$$[\![\texttt{/*@ requires } R\texttt{;} \;\; \texttt{@ ensures } E\texttt{;} \;\; \texttt{@ assignable } M\texttt{;} \;\; \texttt{@*/} \;\; T\ m(P)\ \texttt{throws}\ S\ \texttt{\{}\ B\ \texttt{\}}]\!] = \left(\begin{array}{l} T\ m(P)\ \texttt{throws}\ S\ \texttt{\{} \\ \quad [\![\ \texttt{assume}\ R\ ]\!]\texttt{;} \\ \quad T\ \texttt{result;} \\ \quad \mathit{saveOld}(E, B)\texttt{;} \\ \quad \texttt{try \{}\ [\![B]\!]\ \texttt{\}} \\ \quad \texttt{catch (ReturnExc e) \{\}} \\ \quad [\![\ \texttt{assert}\ E\ ]\!]\texttt{;} \\ \quad \texttt{return result;} \\ \texttt{\}} \end{array} \right) \tag{2}$$

The specified method m is translated into a method with the same signature, embedding the pre- and postcondition as assumption and assertion, respectively. Moreover, we need additional statements and declarations. New variables are initialized in $\mathit{saveOld}(E, B)$ to enable the translation of **\old** that refers to values at method entry. We add the variable **result** for the return value of the method. Together with the exception **ReturnExc**, this encodes return statements. The correctness of the translation is captured by the following claim:

> **Correctness of Translation.** A JML-annotated Java method m satisfies its JML contract if and only if the translation of m does not fail[5] any of its assertions for any initial state, argument values, nondeterministically chosen values and bound on the number of loop iterations.

This claim has been shown for a simple while language, but remains to be proven for the full semantics of Java and JML [17]. Most Java statements s in a method body B are translated by the identity ($[\![s]\!] = s$), i.e., are left unchanged. The translation differs only for modularity-related aspects, e.g., modular handling of loops, assignable clauses, and abstractions of method calls using contracts, which are covered in Sect. 4.4 and 4.5.

For methods, we furthermore need to translate **assert**, **assume** and **return** statements. The former two occur directly in the method's translation, and the latter one is required for a control flow that contains the explicit cast of an exception. In order to evaluate conditions in assertions and assumptions, we need to know their "polarity", which depends on whether they occur within an assumption or an assertion, and changes within negated expressions. As an example, the translation of quantifiers requires to distinguish their polarity. For expressions, there are several different translation functions for the contexts in which

[5] Failure means that an exception is thrown when evaluating the assertion.

the expression occurs. Depending on the polarity of the expression (i.e., whether it occurs negated or not and whether it is **assume**d or **assert**ed), we translate expressions differently (indicated by the *assert* or *assume* subscript). Moreover, some expressions require that code is executed before their evaluation. There is, hence, for both modes another translation function that gives the code to evaluate the expression (it is denoted by the superscript *cmd* while the superscript *val* indicates the code for the expression itself). This distinction enables a more efficient treatment of quantifiers as shown in Sect. 4.3.

$$\begin{aligned}
[\![\texttt{assert}\ A]\!] &= \{[\![A]\!]^{cmd}_{assert}\ \texttt{;}\ \texttt{assert}\ [\![A]\!]^{val}_{assert}\ \texttt{;}\} \\
[\![\texttt{assume}\ A]\!] &= \{[\![A]\!]^{cmd}_{assume}\ \texttt{;}\ \texttt{assume}\ [\![A]\!]^{val}_{assume}\ \texttt{;}\} \\
[\![\texttt{return}\ E]\!] &= \{\texttt{result = } E\texttt{; throw new ReturnException();}\}
\end{aligned}$$

4.2 Translating JML Expressions

The expression language in JML extends the side-effect-free expressions in Java. In most cases, the translation operator is simply propagated to all sub-expressions. For literals and local variables, the translation is the identity. We hence give rules for the majority of all binary operations $\circ$, such as `+` or `==`:

$$[\![A \circ B]\!]^{val} = [\![A]\!]^{val} \circ [\![B]\!]^{val} \qquad [\![A \circ B]\!]^{cmd} = [\![A]\!]^{cmd}\ \texttt{;}\ [\![B]\!]^{cmd}$$
$$[\![x]\!]^{val} = x \qquad [\![x]\!]^{cmd} = \texttt{\{\}}$$

The translation of unary operators, field and array accesses, etc., follows the same principle. Special attention must be given to the case of binary Boolean connectives that have a short-circuit semantics in Java, i.e., the second operand is only evaluated if the result is not determined by the value of the first one. This applies to Java operators such as "`&&`" and "`||`", but also to the implication "`==>`" in JML (see Sect. 4.6). The rules are the same both for *assert* and *assume*:

$$[\![A\ \texttt{\&\&}\ B]\!]^{val} = [\![A]\!]^{val}\ \texttt{\&\&}\ [\![B]\!]^{val} \qquad [\![A\ \texttt{||}\ B]\!]^{val} = [\![A]\!]^{val}\ \texttt{||}\ [\![B]\!]^{val}$$
$$[\![A\ \texttt{==>}\ B]\!]^{val} = [\![\texttt{!}A]\!]^{val}\ \texttt{||}\ [\![B]\!]^{val}$$
$$[\![A\ \texttt{\&\&}\ B]\!]^{cmd} = [\![A]\!]^{cmd}\ \texttt{; if(}\ [\![A]\!]^{val}\texttt{)\{}\ [\![B]\!]^{cmd}\ \texttt{\}}$$
$$[\![A\ \texttt{||}\ B]\!]^{cmd} = [\![A]\!]^{cmd}\ \texttt{; if(}\ [\![\texttt{!}A]\!]^{val}\texttt{)\{}\ [\![B]\!]^{cmd}\ \texttt{\}}$$
$$[\![A\ \texttt{==>}\ B]\!]^{cmd} = [\![A]\!]^{cmd}\ \texttt{; if(}\ [\![A]\!]^{val}\texttt{)\{}\ [\![B]\!]^{cmd}\ \texttt{\}}$$

An additional twist occurs with operators that modify polarity, most notably negation. In that case, *assert* gets switched to *assume* and vice versa:

$$[\![\texttt{!}A]\!]^{val}_{assert} = \texttt{!}\,[\![A]\!]^{val}_{assume} \qquad [\![\texttt{!}A]\!]^{cmd}_{assert} = [\![A]\!]^{cmd}_{assume}$$
$$[\![\texttt{!}A]\!]^{val}_{assume} = \texttt{!}\,[\![A]\!]^{val}_{assert} \qquad [\![\texttt{!}A]\!]^{cmd}_{assume} = [\![A]\!]^{cmd}_{assert}$$

The ternary conditional operator (C `?` T `:` E) is special, since the condition C occurs both positive (as a guard for T in case C is true) and negative (as a

guard for E in case C is false). Furthermore, we introduce another mode *demonic* which makes sure that the optimizations proposed in Sect. 4.3 are not applied (there are also the dual rules to the following ones for *assert*):

$$\begin{aligned}
&[\![C \texttt{ ? } T \texttt{ : } E]\!]^{val}_{assume} = [\![C]\!]^{val}_{demonic} \texttt{ ? } [\![T]\!]^{val}_{assume} \texttt{ : } [\![E]\!]^{val}_{assert} \\
&[\![C \texttt{ ? } T \texttt{ : } E]\!]^{cmd}_{assume} = \\
&\quad [\![C]\!]^{cmd}_{assume} \texttt{; if(} [\![C]\!]^{val}_{assume} \texttt{)\{} [\![T]\!]^{cmd}_{assume} \texttt{\} else \{} [\![E]\!]^{cmd}_{assume} \texttt{\}}
\end{aligned}$$

Apart from pure Java, also JML-specific constructs, e.g., implications as shown in the beginning of this subsection, may occur within specifications. We support the `\old(`E`)` construct which can be used to refer to the value of an expression E in the state at the beginning of the current method invocation. This semantics is achieved by storing the prestate value of all expressions used as arguments for this operator in fresh variables before executing the method (as done for *saveOld* in (2)). The keyword `\result` can be used in postconditions to refer to the result of the method invocation. We translate it into the new variable `result` during the translation of the method body in (2).

$$\begin{aligned}
[\![\texttt{\textbackslash result}]\!]^{val} &= \texttt{result} & [\![\texttt{\textbackslash result}]\!]^{cmd} &= \texttt{\{\}} \\
[\![\texttt{\textbackslash old(}E\texttt{)}]\!]^{val} &= oldVar(E) & [\![\texttt{\textbackslash old(}E\texttt{)}]\!]^{cmd}_{x} &= [\![E]\!]^{cmd}_{x}
\end{aligned}$$

The symbol x is used as a placeholder for either *assume* or *assert* mode. Moreover, we require special treatment when `\old(`E`)` occurs within a quantified expression if it contains the quantified variable.

4.3 Translating Quantifiers

JML also supports universally and existentially quantified expressions, and although JML permits to quantify over objects and unbounded ranges, the following rules only cover bounded integer ranges, where for an integer variable i bounded by L and H, and the quantified expression E, expressions are as follows:

$$\begin{aligned}
&\texttt{(\textbackslash forall int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)} \\
&\texttt{(\textbackslash exists int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)}
\end{aligned}$$

In JBMC's semantics, `assert` statements can be seen as implicitly universally quantified and `assume` statements as implicitly existentially quantified (see Sect. 2). Hence, we translate the JML clause

```
ensures (\forall int i; 0 <= i && i < 10; 0 <= a[i]);
```

by assigning a nondeterministic value to `i` and eliminating the quantifier:

```
int i = *; assert !(0 <= i && i < 10) || 0 <= a[i];
```

Note that this also works for unbounded quantifiers. By the duality of the quantifiers, an equivalent translation exists for the assumption of existentially

quantified expressions. We denote such quantifiers that may be translated in this way as "angelic" quantifiers, since they are the "easy" case regarding translation.

$$[\![\texttt{(\forall int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)}]\!]^{cmd}_{assert} = \texttt{int } i\texttt{=*; } [\![E]\!]^{cmd}_{assert}$$

$$[\![\texttt{(\forall int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)}]\!]^{val}_{assert} = [\![\texttt{(}L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{) ==> } E]\!]^{val}_{assert}$$

$$[\![\texttt{(\exists int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)}]\!]^{cmd}_{assume} = \texttt{int } i\texttt{=*; } [\![E]\!]^{cmd}_{assume}$$

$$[\![\texttt{(\exists int } i\texttt{; } L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{; } E\texttt{)}]\!]^{val}_{assume} = [\![\texttt{(}L \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } H\texttt{) \&\& } E]\!]^{val}_{assume}$$

The integer expressions for the bounds L and H of the index variable i are not subject to the translation $[\![\cdot]\!]^{cmd}$ and must not contain quantifiers. Special care must be given to quantifiers that cannot be translated by implicit semantics, i.e., universal quantifiers within **assume** and existential quantifier within **assert**. We call these quantifiers "demonic quantifiers", as these are more problematic and we need an explicit loop within our translation:[6]

$$\left[\!\!\left[\begin{array}{l}\texttt{(\exists int i;}\\ \quad L \texttt{ <= i \&\& i < } H\texttt{;}\\ \quad E\texttt{)}\end{array}\right]\!\!\right]^{cmd}_{assert} = \left(\begin{array}{l}\texttt{b = false;}\\ \texttt{for (int i = } L\texttt{; i < } H\texttt{; ++i) \{}\\ \quad [\![E]\!]^{cmd}_{assert}\\ \quad \texttt{b = (b || } [\![E]\!]^{val}_{assert}\texttt{)}\\ \texttt{\}}\end{array}\right)$$

$$\left[\!\!\left[\begin{array}{l}\texttt{(\forall int i;}\\ \quad L \texttt{ <= i \&\& i < } H\texttt{;}\\ \quad E\texttt{)}\end{array}\right]\!\!\right]^{cmd}_{assume} = \left(\begin{array}{l}\texttt{b = true;}\\ \texttt{for (int i = } L\texttt{; i < } H\texttt{; ++i) \{}\\ \quad [\![E]\!]^{cmd}_{assume}\\ \quad \texttt{b = (b \&\& } [\![E]\!]^{val}_{assume}\texttt{)}\\ \texttt{\}}\end{array}\right)$$

$$[\![\texttt{(\exists int i; } L \texttt{ <= i \&\& i < } H\texttt{; } E\texttt{)}]\!]^{val}_{assert} = \texttt{b}$$

$$[\![\texttt{(\forall int i; } L \texttt{ <= i \&\& i < } H\texttt{; } E\texttt{)}]\!]^{val}_{assume} = \texttt{b}$$

In this translation, `b` is a fresh Boolean variable that does not occur in the program, and is assumed to be declared at the beginning of the program. The translations of **\forall** and **\exists** differ in the initialization value of `b` and

[6] Using the demonic translation also for angelic quantifiers would be sound, yet less efficient. Hence, we use it in the *demonic* mode of the ternary operator.

the Boolean operation in the loop body. The requirement of bounded integer ranges is crucial (the loop must terminate). Although this translation may be more intuitive, it is significantly less efficient for verification and hence only used when necessary. Note that the quantified range must not only be bounded, but must also be of the expected form. Consider, e.g., the set of even integers smaller than 10. This set is clearly bounded, but it does not fit the expected form, as there is an additional constraint in the guard. This can, however, always be fixed by moving the additional constraint to the inner expression within the quantifier.

4.4 Translating Frame Conditions

So far, we only considered the translation of pre- and postconditions. In JML, however, method contracts also contain frame conditions, which are specified within **assignable** clauses (see Sect. 2). The basic idea is to add an assertion for each assignment that fails if and only if the assignment violates the frame condition (for the sake of simplicity, we only consider assignments, but our approach also applies to other state-changing operations). Note that these rules only cover assignments to arrays and object fields, as assignments to local variables are always permitted. If '**assignable** a_1**,** a_2**,** $\ldots$**,** a_n' is the assignable clause for the enclosing method, the translation rules for an assignment to a left-hand side of the form $O.f$, where O is of type OT, as well as a left-hand side of the form A`[`I`]`, where A is of array type AT`[]`, are as follows:

$$[\![O.f\texttt{=}E\texttt{;}]\!] = \begin{pmatrix} OT\ \texttt{nO = } O\texttt{;} \\ \texttt{assert } mc(\texttt{nO.}f\texttt{,\old(}a_1\texttt{))} \texttt{ || } \ldots \texttt{ ||} \\ \qquad\quad mc(\texttt{nO.}f\texttt{,\old(}a_n\texttt{));} \\ \texttt{nO.f = } E\texttt{;} \end{pmatrix}$$

$$[\![A\texttt{[}I\texttt{]=}E\texttt{;}]\!] = \begin{pmatrix} AT\texttt{[] nA = } A\texttt{;} \\ \texttt{assert } mc(\texttt{nA[}I\texttt{],\old(}a_1\texttt{))} \texttt{ || } \ldots \texttt{ ||} \\ \qquad\quad mc(\texttt{nA[}I\texttt{],\old(}a_n\texttt{));} \\ \texttt{nA[}I\texttt{] = } E\texttt{;} \end{pmatrix}$$

The predicate $mc(l, a)$ determines whether an assignment to location l (a field access $o.f$ or an array access $a[i]$) is justified by a storage reference a in the assignable clauses. This predicate is defined as follows, where for all other combinations not explicitly mentioned, mc is `false`:

$$mc(o\texttt{.f}, p\texttt{.g}) \Leftrightarrow \texttt{false} \qquad mc(o\texttt{.f}, p\texttt{.f}) \Leftrightarrow o\texttt{==}p \qquad mc(o\texttt{.f}, p\texttt{.}*) \Leftrightarrow o\texttt{==}p$$
$$mc(a[i], b[l..h]) \Leftrightarrow (a \texttt{ == } b \texttt{ \&\& } l \texttt{ <= } i \texttt{ \&\& } i \texttt{ < } h)$$

The above translation is sound, but produces false positives for newly created objects. Consider, e.g., a method annotated by '**assignable \nothing;**' that starts by creating a new object, then stores this object in a local variable, and

finally assigns a new value to one of the object's fields. Our translation would lead to a frame-condition violation being reported as there is no storage reference in the assignable clause that justifies this assignment. In order to fix this problem, we introduce a predicate that we call *newObj*, which we **assume** for each new object. We are then able to adapt our assertions by requiring that an assignment is permitted by the assignable clause (as before) *or* the *newObj* predicate is true for the left-hand side of the assignment.

4.5 Translating Method Invocations

So far, we have seen how to translate JML expressions and use that to translate method contracts. However, in order to achieve a truly modular approach, we need to replace parts of Java code by their contracts, namely method calls. In this section, we moreover show how loops can be replaced by loop invariants, as the general idea for both method calls and loops is very similar: First, the precondition (or the invariant) is **assert**ed, then the parts of the state that are modifiable by the method call (or the loop) according to the JML assignable clause are anonymized, and finally the post condition is **assume**d. The same translation technique can be applied for block contracts and statement contracts (not shown here). The standard treatment for method calls in JBMC is to inline the method body. We can exploit this behavior, as we replace the original definition of the method by a "symbolic" definition, which contains the method contract instead of the method body. Once the symbolic method body gets inlined, it takes care of all necessary assertions and assumptions. The transformation of the method definition is contained in the following rule:

$$
[\![
\begin{array}{l}
\texttt{/*@ requires } R\texttt{;} \\
\quad \texttt{@ ensures } E\texttt{;} \\
\quad \texttt{@ assignable } A\texttt{;} \\
\quad \texttt{@*/} \\
T\ m\texttt{(}P\texttt{) throws } S\ \texttt{\{} \\
\quad B \\
\texttt{\}}
\end{array}
]\!] =
\left(
\begin{array}{l}
T\ m\texttt{Contract(}P\texttt{) throws } S\ \texttt{\{} \\
\quad [\![\ \texttt{assert } R\ ]\!]\texttt{;} \\
\quad T\ \texttt{result;} \\
\quad \mathit{saveOld}(E, B)\texttt{;} \\
\quad \mathit{havoc}(A)\texttt{;} \\
\quad [\![\ \texttt{assume } E\ ]\!]\texttt{;} \\
\quad \texttt{return result;} \\
\texttt{\}}
\end{array}
\right)
$$

The "normal" translation of a method contract, as shown in the beginning of this section, is used to prove that a method satisfies its contract, while the above translation uses the contract and assumes its correctness. Both translations use the method *saveOld* to store values of variables which may be referred from thereon via the keyword **\old**. We introduce a variable for the return value before the postcondition is assumed, since the postcondition may contain restrictions to the returned value. Moreover, we introduce the translation method $\mathit{havoc}(A)$, which anonymizes the values of all location sets of the assignable clause A. "Havocing" for primitive types is equivalent to assigning a nondeterministic value of that type. We must also assume and enable that, if A contains

locations of object type, then $havoc(A)$ may nondeterministically generate new objects, which may be used in the assignments and cannot be easily reduced to a nondeterministic assignment for JBMC (see Sect. 5). Remember that, as to provide a compact representation for our rules, we do not permit method calls to occur as sub-expressions but only in assignments to local variables. Thus, the translation rule for method calls extends the rule for Java statements (see Sect. 4.1) and is as follows:

$$[\![var\texttt{=}m\texttt{(}P\texttt{);}]\!] \;=\; var\texttt{=}m\texttt{Contract(}[\![P]\!]\texttt{);}$$

Therein, the assignment to the local variable *var* is optional. The same as for method calls can be applied to loops. Verifying loops is inherently challenging, in particular if no bound on the number of loop iterations is known beforehand. For this matter, we can use loop invariants, which can be understood as the contract for a loop, as a loop invariant must be guaranteed and can be assumed for the whole loop execution. A loop invariant acts both as a pre- and a postcondition for the loop, as it must hold before and after each loop iteration.

$$\left[\!\!\left[\begin{array}{l}\texttt{/*@ loop\_invariant } I\texttt{;}\\ \texttt{\quad @ assignable } A\texttt{;}\\ \texttt{\quad @ decreases } D\texttt{;}\\ \texttt{\quad @*/}\\ \texttt{while (}C\texttt{) \{ }B\texttt{ \}}\end{array}\right]\!\!\right] = \left(\begin{array}{l}\texttt{int oldD = }D\texttt{;}\\ [\![\ \texttt{assert } I\ ]\!]\texttt{;}\\ \quad havoc(A)\texttt{;}\\ [\![\ \texttt{assume } I\ ]\!]\texttt{;}\\ \texttt{if (}C\texttt{) \{}\\ \quad [\![B]\!]\texttt{;}\\ \quad [\![\ \texttt{assert } I\ ]\!]\texttt{;}\\ \quad \texttt{assert } D\texttt{ < oldD \&\& 0 <= }D\texttt{ ;}\\ \quad \texttt{assume false;}\\ \texttt{\}}\end{array}\right)$$

On the right side of the above rule (a similar rule is defined for **for**-loops), we (1) **assert** the invariant, then (2) we "havoc" the assignable set of the loop that replaces all loop iterations that may already have occurred, then (3) **assume** the invariant, (4) execute the loop body once, and finally, (5) **assert** the invariant again. Steps (1) to (3) make use of the invariant to replace multiple loop iterations, while steps (3) to (5) prove the inductive invariant. Proving the invariant for a single loop execution suffices to establish its validity. Additionally, we prove that the loop terminates by asserting that the decreases clause does indeed decrease and is still greater than zero.

Finally, we append the statement "**assume false;**" to the loop body, as we chose an arbitrary loop iteration, but all assertions after the loop must only hold in case the loop is fully executed. Essentially, as long as the loop body is executed, we prevent the model checker from reporting any assertion violations, since this is not a valid program run.

4.6 Ensuring Correct Behavior for Boolean Operators

As seen in Sect. 4.2, binary Boolean connectives that have a short-circuit semantics in Java need special consideration during the translation. According to JML's semantics, if an exception is raised during the evaluation of an expression, then the whole clause is considered to have failed, i.e., the program does not satisfy that clause [7]. Thus, no method can satisfy the ill-defined specification '`ensures 1/0 == 0;`'. However, the definition uses the short-circuit semantics of Java operators, so that every method satisfies '`ensures true || (1/0 == 0);`' In most cases, our translation easily leads to the right behavior of the resulting code. For example, '`ensures 1/0 == 0;`' becomes '`assert 1/0 == 0;`', which brings the invalid postcondition directly to the Java code in form of an assertion whose evaluation throws an exception.

However, we need to be careful when translating short-circuit behavior in combination with "demonic" quantifiers. Consider, for example, the following postcondition:

```
ensures (true || (\exists int i; 0 <= i && i < 1; 1/i == 0));
```

Due to the short-circuit semantics of `||`, and since the first operand is `true`, we never evaluate the second operand, and the whole expression evaluates to `true`, which is trivially satisfied by any method. If the special behavior of `||` were not considered, our translation would produce the following (wrong) result:

```
b = false;
for (int i = 0; i < 1; ++i) { b = (b || 1/i == 0); } // WRONG
assert true || b;
```

However, this translation is wrong, since the `for`-loop throws an exception when `i` equals `0` in the first iteration, which would falsely indicate a failure. Hence, in order to deal with such behavior, our translation adheres to the rules presented in Sect. 4.2, and the code $[\![B]\!]^{cmd}$ for the second operand B of a disjunction is only evaluated if the first operand A evaluates to `false`. Consequently, our translation produces the following code with the desired behavior, where the loop is not executed and no exception is thrown:

```
if (!true) {
  b = false;
  for (int i = 0; i < 1; ++i) { b = (b || 1/i == 0); }
}
assert true || b;
```

5 Implementation

We provide a prototypical implementation of our approach in form of the command-line application JJBMC.[7] It translates a Java source file annotated with JML specifications into a Java file to be read by JBMC. The implementation makes use of the OpenJML back end (see Sect. 2) to parse and manipulate the given Java/JML. The user can choose to either verify only a single method or all methods in the given Java file. Furthermore, they can pass any JBMC options for a customized behavior, e.g., concerning various bounds for objects, arrays, or object structures, as well as the handling of exceptions, the employed SAT or SMT solver, or the output format. If JBMC is able to find a counterexample for the given specification and program, the counterexample is parsed and provided as a program trace, i.e., the sequence of program states up to the violated assertion (with concrete instantiations for the nondeterministic values). The output is optimized from the original JBMC output such that the user may (relatively) easily understand and analyze the semantics. We provide additional options for the user to choose whether auxiliary specifications (contracts of called methods and loop invariants) or inlining shall be used. Even though ignoring contracts seems to contradict our modular approach, it is sometimes useful to try verification that way first, so that unnecessary modularization may be avoided. Inlining also allows the user to flexibly distinguish between errors in the top-level specification and individual auxiliary specifications. Note that our implementation is still prototypical and does currently not support full JML and Java. However, we provide a clear user feedback whenever unsupported features are used, in order to maintain the soundness of our approach. For full (sequential) Java, everything except for catching exceptions, `break` and `continues` statements, and inheritance is supported. For JML, we currently support preconditions, postconditions, loop invariants, frame conditions (limited to fields and array ranges) for contracts and loops, assertions, assumptions, `\old` (with similar restrictions as for frame conditions), and universal and existential quantifiers.

Given that both SBMC and runtime assertion checking do not involve the full abstraction from JML contracts, tasks that require to distinguish different heap states or specify object anonymization are new and challenging for JBMC's semantics. Consider, e.g., the keyword `\old` that "remembers" a variable's value before method execution. Java lacks support for deep copies of objects, which hinders the implementation of such a concept. Furthermore, JBMC's semantics of nondeterministic value assignments for objects is not sufficient to implement anonymization of heap locations: JBMC interprets a nondeterministic object as a new object with nondeterministic values assigned to its fields. JML, however, demands that this "anonymous" object may or may not be new, and its fields may or may not point to existing or new objects. For implementing such semantics, we would need an explicit model of all objects within the Java program, such that we can do a nondeterministic selection among all those objects.

[7] The source code is available at https://github.com/JonasKlamroth/JJBMC.

```
/*@ requires a != null && a.length <= 5;
  @ ensures \result <= a.length * 32;
  @ assignable \nothing;
  @*/
int naiveHammingWeight(int[] a) {
  int result = 0;
  /*@ loop_invariant result <= i * 32;
    @ loop_invariant 0 <= i && i <= a.length;
    @ assignable result;
    @*/
  for (int i = 0; i < a.length; i++) {
    int x = a[i];
    while (x != 0) { result += x&1; x = x >>> 1; }
  }
  return result;
}
```

Listing 2. Calculation of the hamming weight for an array.

6 Evaluation

We evaluated our translation and its implementation on a selection of JML-annotated Java examples[8] that are shipped with the KeY tool [1], which illustrate a variety of JML's and KeY's features. The goal of our evaluation was to demonstrate correctness and feasibility of our approach, i.e., that JML annotations are translated into programs which are correctly read and verified by JBMC. Using a bound of 5 on the number of loop iterations and array sizes, all verification tasks were successfully performed by JBMC within a few seconds. Besides simple examples, our evaluation included algorithms that perform array manipulations, e.g., sorting algorithms, and algorithms with bit-operations.

Let us first consider the program given in Listing 2, which calculates the hamming weight of an integer array by iterating over all array elements and adding together their respective hamming weights. Each hamming weight is – very inefficiently – calculated by iterating over every bit and checking whether it is zero or not. The program contains two loops, but a loop invariant is only provided for the outer one. This is what an engineer may do in practice, as the inner loop is guaranteed to run at most 32 times (for each bit of the integer value) and is thus a prime suspect for loop unrolling, since it does not necessarily require a loop invariant. In contrast, the outer loop iterates over array elements, where the number of iterations is unknown. Using our translation, JBMC verifies this program for our default upper bound with an array size of 5. Note that, for very large arrays, the postcondition is actually not satisfied, as `a.length * 32` may overflow. The fact that this is not discovered by JBMC is due to the inherently bounded nature of *bounded* model checking and does not mean that our transla-

[8] All case studies are available at https://github.com/JonasKlamroth/JJBMC.

tion is incorrect. In addition to this inefficient hamming weight calculation, we also implemented and verified a more efficient version that uses a sequence of bit operations without the need for an inner loop.

Let us further consider bubble sort[9], which performs array manipulations. The JML contract demands that the result array is sorted, i.e., each entry is less than or equal to the consecutive entry. The program contains two nested loops that iterate over the array and move the greatest remaining element to the end of the unsorted part of the array. Element swaps are carried out by a method `swap(int[] a, int i, int j)` which swaps `a[i]` and `a[j]` and is implemented as an in-place xor-operation. For this example, we evaluated different levels of modularity. In the first step, we verified the (translation of the) top-level specification of bubble sort with JBMC by unrolling the loops and inlining the swap method, i.e., a (non-modular) whole-program verification. In the next step, we used the contract of the `swap` method instead of its implementation and then, in the final step, we also used two loop invariants. This demonstrates that our approach may support finding both the right specification and implementation without the need of having everything ready from the beginning.

7 Related Work

Pnueli and Shahar present a verification system that combines both deductive verification and bounded model checking, where they verify finite-state systems w.r.t. constraints in linear temporal logic (LTL) [22]. Moreover, Shankar examines the interrelations between the two paradigms by exploring various examples for their combination, and illustrates the advantages [24]. The synergy of such a combination is also discussed by Beckert et al. on the verification of C programs, who focus, however, on combining two tools rather than doing program transformation [3]. Lourenço et al. present a minimal model as a combined theoretical basis for the two paradigms [4]. Similar to our work, they capture both concrete loop unrollings and abstract loop invariants. Whereas their model works on a simple while language, we target the Java programming language that comes with a richer semantics. Furthermore, the field of runtime verification also translates program specifications into assertions that are checked at runtime. Burdy et al. present a tool that translates JML annotations into runtime assertion checks for Java programs [6]. They encounter similar challenges as we do, e.g., interpreting well-definedness of specifications and translating quantifiers. Their translation covers the quantification over iterable collections and other forms of quantifiers such as `\sum`. While Burdy et al. focus on runtime checks, we use the translation as input for static verification, and instead of translating into pure Java, our output is extended by assertions and assumptions. The underlying idea, however, namely that "JML accommodates both runtime assertion checking and formal verification" is the same [20]. Chalin et al. discuss this approach for the strong-validity semantics of JML [7]. Similar work has been conducted by Kosmatov et al. for C programs and ACSL specifications [18].

[9] We do not print the code, as we used the well-known standard implementation.

8 Conclusion and Future Work

We presented a translation of JML-annotated Java code into Java programs that can be read by the software bounded model checker JBMC, which enables JBMC to check JML annotations at an early stage when developing the specification. This extends JBMC's reach such that it may use method contracts and loop invariants additionally to (mere) method body inlining or unwinding loops. Finally, we presented a prototypical implementation which we evaluated on first case studies that can be read and verified by JBMC.

As future work, we plan to extend our approach to support further features of Java and JML, e.g., full exception handling and abrupt termination within loops. We also plan to extend our translation to fully capture JML's semantics of heap anonymization or "havocing", as we are currently restricted by JMBC's default semantics of nondeterminism that excludes previously created objects. Moreover, we plan to evaluate and improve both the performance and the usability of our implementation, e.g., the readability of reported counterexamples. Finally, we performed first experiments for the verification of stability properties for floating-point operations such as addition, which we plan to extend.

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, LNCS, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
3. Beckert, B., Bormer, T., Merz, F., Sinz, C.: Integration of bounded model checking and deductive verification. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011). LNCS, vol. 7421, pp. 86–104. Springer (2012). https://doi.org/10.1007/978-3-642-31762-0_7
4. Belo Lourenço, C., Frade, M.J., Sousa Pinto, J.: A generalized program verification workflow based on loop elimination and SA form. In: 7th International Workshop on Formal Methods in Software Engineering (FormaliSE 2019). pp. 75–84. IEEE / ACM (2019). https://doi.org/10.1109/FormaliSE.2019.00017
5. Biere, A., Kröning, D.: SAT-based model checking. In: Handbook of Model Checking, pp. 277–303. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_10
6. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer **7**(3), 212–232 (2005). https://doi.org/10.1007/s10009-004-0167-4
7. Chalin, P., Rioux, F.: JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In: 15th International Symposium on Formal Methods (FM 2008). LNCS, vol. 5014, pp. 246–261. Springer (2008). https://doi.org/10.1007/978-3-540-68237-0_18
8. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: 40th Design Automation Conference (DAC 2003). pp. 368–371. ACM (2003). https://doi.org/10.1145/775832.775928

9. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Third International Symposium on NASA Formal Methods (NFM 2011). LNCS, vol. 6617, pp. 472–479. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_35
10. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: 30th International Conference on Computer Aided Verification (CAV 2018). LNCS, vol. 10981, pp. 183–190. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
12. Filliâtre, J.: Deductive software verification. International Journal on Software Tools for Technology Transfer **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
13. Gomes, C.P., Kautz, H.A., Sabharwal, A., Selman, B.: Satisfiability solvers. In: Handbook of Knowledge Representation, FAI, vol. 3, pp. 89–134. Elsevier (2008). https://doi.org/10.1016/S1574-6526(07)03002-7
14. 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: 27th International Conference on Computer Aided Verification (CAV 2015). LNCS, vol. 9206, pp. 273–289. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_16
15. Harel, D.: Dynamic logic. In: Handbook of Philosophical Logic, vol. 165, pp. 497–604. Springer (1984). https://doi.org/10.1007/978-94-009-6259-0_10
16. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: International Symposium on Software Testing and Analysis (ISSTA 2000). pp. 14–25. ACM (2000). https://doi.org/10.1145/347324.383378
17. Klamroth, J.: Modular Verification of JML Contracts Using Bounded Model Checking. Master's thesis, Karlsruhe Institute of Technology (KIT) (2019). https://doi.org/10.5445/IR/1000122228
18. Kosmatov, N., Signoles, J.: Runtime assertion checking and its combinations with static and dynamic analyses - tutorial synopsis. In: 8th International Conference on Tests and Proofs (TAP 2014). LNCS, vol. 8570, pp. 165–168. Springer (2014). https://doi.org/10.1007/978-3-319-09099-3_13
19. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
20. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming **55**(1-3) (2005). https://doi.org/10.1016/j.scico.2004.05.015
21. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
22. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: 8th International Conference on Computer Aided Verification (CAV 1996). LNCS, vol. 1102, pp. 184–195. Springer (1996). https://doi.org/10.1007/3-540-61474-5_68
23. Shankar, N.: Automated deduction for verification. ACM Computing Surveys **41**(4), 20:1–20:56 (2009). https://doi.org/10.1145/1592434.1592437
24. Shankar, N.: Combining model checking and deduction. In: Handbook of Model Checking, pp. 651–684. LNCS, Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_20