Karlsruhe Institute of Technology

Modular Verification of JML Contracts Using Bounded Model Checking

Master's Thesis of

Jonas Klamroth

at the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Dr. Mattias Ulbrich
Second advisor:	Dipl. Inform. Michael Kirsten

3 September 2018 – 3 March 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(Jonas Klamroth)

Abstract

In this thesis, we present an approach that allows the verification of Java programs with regard to JML annotations with a bounded model checker. We therefore translate a given Java program and its specification into a program using assumptions, assertions and nondeterministic values. The translation is proven correct for a while language and formalized for a subset of Java and JML. Additionally, a tool is presented that implements that approach and we show that the tool is capable of finding proofs in multiple case studies.

Zusammenfassung

In dieser Masterarbeit stellen wir einen neuen Ansatz zur Verifikation von mit JML spezifierten Java-Programmen mittels eines Bounded Model Checkers vor. Hierzu wurde eine Übersetzung von Java und der dazugehörigen Spezifikation in ein um Annahmen, Bestätigungen und nicht deterministische Werte erweitertes Java entwickelt. Wir beweisen, dass diese Übersetzung für eine while-Sprache korrekt ist und formalisieren sie für eine Teilmenge von Java und JML. Außerdem zeigen wir anhand mehrer Fallstudien, dass die prototypische Implementierung dieses Ansatzes in der Lage ist Beweise über Programme zu führen.

Contents

List of Figures

List of Tables

1 Introduction

Since the number of programs grows by the minute, the need for quality assurance in software development is as important as it has never been before. Due to increasing complexity and higher safety requirements, purely testing the software is often times not sufficient. This is where software verification comes into play. Software verification allows a program to be proven correct for a given specification. The problem here is that verifying software can be very time consuming. This thesis presents an approach which combines two different verification techniques, modular verification and bounded model checking, in order to decrease the user interaction necessary while still allowing complex properties to be verified.

We formalize and implement a translation of programs specified with the Java Modeling Language (JML) into a pure Java-program, only relying on two additional non-java operations: assumptions and nondeterministic values. Using this translation, we are able to apply the Java bounded model checker (JBMC) to our translated program and thus allow the checker to verify JML specifications. Enabling a Bounded Model Checker this way brings along two main advantages: 1) proofs may be split up between deductive verification tools and Bounded Model Checkers according to their respective strengths 2) The Bounded Model Checker may be used to provide confidence in the correctness of a specification (e.g., an invariant), by proving it for a bounded domain.

The translation we introduce is provided as formal rewrite rules and implemented in a prototype. With this prototype we conducted several case studies, which show that the approach is not only working but is advantageous (time and complexity wise) when conducting certain proofs.

This thesis is split up into six main chapters: First we lay theoretical foundations for the next chapters. Then we present the theory for our approach. We prove that the general translation we are introducing is correct and give a set of rules, which is able to translate a subset of Java and JML. In the third chapter we present our tool and talk about how we implement the previously presented approach. The fourth chapter discusses the case studies we did, to provide evidence that the presented approach is advantageous. Before we end with a conclusion, we present related work and discuss how our approach relates to existing tools and approaches.

2 Foundations

In this chapter introduce some basics which are necessary to then present our approach.
We first give a brief overview over syntax and semantics of first order dynamic logic (FODL), then lay the basics of program verification before we give a short introduction into the Java Modeling Language and eventually present the tools we are using for this thesis.

2.1 First Order Dynamic Logic (FODL)

In this section we give a brief overview over first order dynamic logic (FODL), which allows us to do reasoning with programs. An understanding of first order logic (FOL) is assumed. The here presented syntax and semantic is taken from [14].

We have a signature $\Sigma = (F, P, \alpha)$ where F is a set of function symbols, P is a set of predicate symbols and $\alpha : F \cup P \rightarrow \mathbb{N}$ an arity function for those functions and predicates. Additionally we have a set of variables Var. In the following section we will use the following notations: $var \in Var, f \in F, p \in P, prog_1, prog_2 \in Prog, fml_1, fml_2 \in Fml, term_x \in Term$:

Definition 2.1.1
$Term ::= var \mid f(term_1, \ldots, term_{\alpha(f)})$
$Fml ::= true \mid false \mid p(term_1, \ldots, term_{\alpha(p)}) \mid term_1 = term_2 \mid \neg fml_1 \mid fml_1 \wedge fml_2 \mid fml_1 \vee fml_2 \mid fml_1 \rightarrow fml_2 \mid \exists var.fml_1 \mid \forall var.fml_1 \mid [prog]fml_1 \mid \langle prog \rangle fml_1$
$Prog ::= prog_1 \cup prog_2 \mid var := term \mid prog_1 * \mid prog_1; prog_2 \mid var := * \mid ?fml$

This is the syntactical material for FODL. To define the semantics of these programs we first need to define first order structures and Kripke frames.

Definition 2.1.2 *A first order structure is a tuple (D, I) where D is the domain (a non emtpy set of objects) and I is an interpretation: $I(f) : D^{\alpha(f)} \rightarrow D$ for a $f \in F$ and $I(p) \subseteq D^{\alpha(p)}$ for a $p \in P$.*

This definition introduces a domain (as a set of objects) which are the entities over which reasoning can be done and an interpretation which defines the semantic of each predicate and function. Notice how the interpretation of a function is a function into the domain itself whereas the interpretation of the predicates is basically assigning truth values to the predicates for all possible arguments.

Definition 2.1.3 *A fixed Kripke-Frame (S_D, p_D) given a first order logic domain D is a tuple consisting of a set of states S_D and a function $p_D : Prog \rightarrow 2^{S \times S}$. Where $S_D = Var \rightarrow D$ is the set of assignments of elements of the domain to variables.*

This can be intuitively seen as a graph where the each node is one possible variable assignment and each edge is given by all programs which lead from one variable assignment to another. Thus by knowing the initial variable assignments and a program one can walk along the according edges of said graph and see in which state (and thus variable assignment) the programs terminates (if it terminates). With this idea we are now able to define the semantics of this logic.

The semantic of a term $t \in Term$ is the usual first order evaluation of t in (D, I). To allow us to define the semantics of programs we need one more notation:

Definition 2.1.4 *Given $s \in S_D, a \in D, x \in Var$ a function update is*
$$s[x/a](y) = \begin{cases} a \text{ if } y = x \\ s(y) \text{ else} \end{cases}$$

Which intuitively states that updating a variable assigning the value a to a variable x in state s will not change the value of any variable in s except for x which then has the value a.

With this, the semantic of a program $prog \in Prog$ may be defined as follows:

Definition 2.1.5 *Given a Kripke-Structure $K(S_D, p_D)$ the semantics of programs is:*
$$\begin{array}{lll} p(x := v) & ::= & \{(s, s_1) \mid s_1 = s[x/val_{D,I,s}(v)]\} \\ p(x := *) & ::= & \{(s, s_1) \mid \exists a : s_1 = s[x/a]\} \\ p(a; b) & ::= & \{(s, s_1) \mid \exists t \in S : (s, t) \in p(a) \wedge (t, s_1) \in p(b)\} \\ p(a \cup b) & ::= & p(a) \cup p(b) \\ p(a*) & ::= & \{(s_1, s_n) \mid \exists n, s_1, \ldots, s_n : \forall 1 \leq i < n : (s_i, s_{i+1}) \in p(a)\} \\ p(?F) & ::= & \{(s, s) \mid I, s \vDash F\} \end{array}$$
(Where $a, b \in Prog$, $x \in Var$, $v \in D$ and F is a program free FOL-formula.)

To understand this definition remember the idea of visualizing a Kripke-Structure as a graph. Now if we assign a value to a variable, as we said before, no other variable changes so we transition to the state where each variable has the same value except the variable we changed which has the value we just assigned. Accordingly if we assign a nondeterministic value we may transition to any state where each variable has the same value except the one we just changed. The composition of programs is defined as one would except: Starting in a state s, if there is a state s1 which can be reach be executing the first part of the program and executing the second part of the program in s1 leads to transitioning into state s2 then the composition of these two program parts transitions from state s to state s2. The nondeterministic branching allows to provide two programs where the decision of which one of them is executed is nondeterministic. The *-operator is basically the extension of self composition to an arbitrary number of times. And the test operator allows only transitions into states in which the given formula is valid.

The last missing definition is the semantic of formulas.

Definition 2.1.6 *Semantics of formulas:*
$$\begin{array}{lll} I, s \vDash p(t_1, \ldots, t_\alpha(p)) & \textit{iff} & (val_{I,s}(t_1), \ldots, val_{I,s}(t_{\alpha(p)})) \in I(p) \\ I, s \vDash v = t & \textit{iff} & val_{I,s}(t) = val_{I,s}(v) \\ I, s \vDash [\pi]F & \textit{iff} & \forall (s, s_1) \in p(\pi) : I, s_1 \vDash F \\ I, s \vDash \langle\pi\rangle F & \textit{iff} & \exists (s, s_1) \in p(\pi) : I, s_1 \vDash F \end{array}$$

$\models$ *is homomorphic for* $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall x, \exists x$

We write $I \models F$ *for* $\forall s \in S : I, s \models F$ *and* $I, S' \models F$ *for* $\forall s \in S' : I, s \models F$ *where* $S' \subseteq S$.

With the syntactical base material we defined in 2.1.1 we are able to define more complex operators as known from real life programming languages:

- `if` ϕ `then a else b` $\leftrightarrow (?\phi; a) \cup (?\neg\phi; b)$
- `while` ϕ `do a od` $\leftrightarrow (?\phi; a)*; ?\neg\phi$

The if-then-else construct is represented using the test operator twice (once negated) and combined nondeterministically all transition of this program are the ones which take the expected branch (if ϕ is true than the if branch otherwise the else branch). Accordingly the loop is represented using the * operator and the test operator. The negated test operator afterwards guarantees as that all the "loop" is not left until ϕ is not true.

Now we call a program a while-program if $*$ and $\cup$ only occur in if and while statements and var:= * does not occur in the program. Such programs are always deterministic (if the program terminates it terminates always in the same state if started in the same).

2.2 Program Verification

Program verification is the act of proving programs correct with respect to a given formal specification (e.g. JML see 2.3). In this thesis we focus on two different verification techniques: deductive verification and bounded model checking.

2.2.1 Deductive Verification (DV)

In deductive verification the program and its properties to be proven are translated into formulas of a logic (e.g. Hoare-Logic). Properties of programs are often times expressed in Hoare-triples of the form: $\{P\}S\{Q\}$ stating that a program S which is executed in a state that satisfies P is guaranteed to end (if it ends) in a state that satisfies Q. Hoare-Logic provides a set of rules which enables deduction over triples like this [15].

There are several tools supporting the user in translating a given program and its specification into a logic and allowing him to apply rules of a calculus, in order to prove the program correct. Depending on the tool, different logics and calculi may be used[1][5][23].

The main advantage of DV is that due to its nature of operating directly on the logic it is very expressive which allows the user to prove even complicated properties. On the other hand this leads to a high degree of needed user interaction since the decision of which rule of the calculus to apply at each step is hard to automate.

2.2.2 Bounded Model Checking (BMC)

The explanations and definitions of this paragraph are taken from [10]. For model checking the system under investigation is modeled as a Kripke structure. A Kripke structure is a triple $K =< S, R, L >$ where S is a set of states, L is a labeling function and $R \subset S \times S$ is a set of transitions. L assigns each state a set of atomic propositions. Using this notion a model checker can be defined as a decision procedure for $K, s \models f$ where K is a Kripke structure, s is a state and f is a state formula, if a state formula is a boolean combination of atomic propositions and CTL* formulas (Computation Tree Logic), whose outermost operator is a path quantifier.

A Kripke structure can be represented as a directed graph were the nodes correspond to the states, the edges correspond to the transitions and each node has a set of labels (the atomic propositions). Adding a set $I \subset S$ of initial states, a program run then corresponds to a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ where $(s_i, s_{i+1}) \in R$ for all i and $s_0 \in I$. If a property holds in every state of every possible run its a property of the program modeled by this graph.

Bounded model checking is the idea of searching for a program run with maximum length k, that violates a given property. By restricting the maximum length of a run it becomes possible to encode this problem as a formula in propositional logic. Consider as an example the property G p given in LTL establishing p as a global invariant. If we define propositional predicates $I(s)$, which evaluates to true if $s \in I$ and $R(s_1, s_2)$, which evaluates to true if $(s_1, s_2) \in R$ we can construct the following formula [6]:

$$\exists s_0, s_1, \ldots, s_k : I(s_0) \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg p(s_k)$$

This formula is satisfiable if and only if there is a run of length k which violates the property. With quantifier elimination and a Tseitin-Transformation this formula can be transformed into conjunctive normal form which is the standard input format for most SAT-solvers[6].

For the concrete translation of programs into SAT formulas there are several intermediate steps. The approach described here is taken from CBMC [11]. At first loop constructs are replaced by equivalent while loops. Then functions are inlined. Recursive functions calls are inlined n times where n is specified by the user. Loops are now unwound, meaning the body is copied n times each time guarded by an if statement using the same condition as the original while loop (n is specified by the user). All remaining goto statements pointing backwards are unwound in a similar manner. At this point the program only consists of assignments and possibly branching goto statements which only point forward. At the end the program is transformed into static single assignment form[2]. Now the translation into a SAT problem is straight forward: We construct 2 bit-vector equations, one where for each variable the assignment is encoded and one which encodes the assertion we want to check. The conjunction of the first and the negated second equation gives us the desired SAT formula.

The described transformation is given for C programs but is also applicable to Java programs with some additional preprocessing concerning exceptions and polymorphism[13].

Bounded model checking is inherently incomplete since it just searches for errors in program runs up to the given bound. Errors which occur in runs longer than that do not get detected and thus programs which allow such runs may not be proven correct with bounded model checking. This lack of expressiveness is compensated by the fact that

BMC is naturally highly automated (the translation into SAT is automated and SAT solvers don't need any user interaction).

2.3 Java Modeling Language (JML)

The Java Modeling Language (JML) is a specification language for Java programs [21]. It follows the design by contract paradigm [19] and enables the user to specify several properties of programs such as pre- post and frame-conditions. The specification is written in Java comments where each line is preceded by an '@'. This way programs containing JML specifications are always valid Java programs. JML syntax is close to Java syntax with the addition of quantifiers (forall and exists) and some special keywords. JML has no universally accepted semantic but for this work we will use the one given in [7]. A small example of a JML-specified program is given in Lst. 2.1.

Listing 2.1: An example of a method specified with JML

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

In this example, a method which multiplies two integers is specified. The precondition is that both integers are positive since otherwise the calculation as it is done would not work. The postcondition is straight forward that the returned value equals the product of the two input parameters. Notice the special JML-keyword *result* which allows us to refer to the returned value of a method. Additionally an assignable clause is specified which allows us to limit the fields and variables to which this method may assign new values. In our case the method itself should not write to any fields or parameters. To specify this we add the assignable clause with another JML-keyword *nothing* which describes what is expected: stating that no variables or field other than local ones may be written to. In the method body itself a loop invariant is used to specify the behavior of the loop. In this case the invariant is pretty simple: A each loop execution the currently computed result is equal to the value of the loop variable times the second parameter. Since the loop iterates up to the first parameter and adds the second parameter to the result each time it is executed this should lead straight to the expected postcondition. In addition to the loop invariant decreases clause is given which is an expression evaluating to an integer. This integer is proven to be always greater or equal to zero and getting lesser each loop iteration. Thus it can be proven, that the loop terminates.

2.4 Tools

We will make extensive use of the tools JBMC and openJML for this thesis. In this section we are going to present these tools and give a brief introduction on how they operate.

2.4.1 JBMC

JBMC is a Bounded Model Checker (BMC) for Java[13]. It is developed parallel to CBMC which is a BMC for C. JBMC takes a compiled Java file (.class file) as input and is able to prove several properties of programs (e.g. absence of Null-Pointers or index out of bounds exceptions). Additionally JBMC provides two functions which extend normal Java: Assumptions and Nondeterministic values. Both are implemented as static methods, which are provided by a class (CProver.java), which has to be imported if these functions are used in a program. In order to understand why these two functionalities are essential for proofs with a BMC, remember how BMC work (see 2.2.2). We see programs as all possible state sequences that may occur when executing a certain source code. In Java normally variables may not be used, as long as they are not initialized. Initializing them leads to them having a certain value. Nondeterministic values allow the programmer to assign an non specific value to a variable and thus, if verifying a program with a BMC to consider all possible values for a certain variable. Accordingly an assume statement (which takes a boolean expression as argument) allows the programmer to restrict the considered states/traces at a certain point in the code.

Consider the code snippet given in 2.2:

Listing 2.2: An example demonstrating the use of JBMC specific operations

```java
int x = CProver.nondetInt();
CProver.assume(x > 0);
assert(x >= 0);
```

Lets for a moment imagine that the type int had only 2 bits and the possible values would be -1 to 2. So since we have only 1 variable in this miniature example and this variable may take 4 different values our state space consists of only 4 states. After the first line of code we could be in every single one of them since we assigned a nondeterministic value. However the assume statement limits the possible states to two of them, namely the states where x = 1 and x = 2. Now at the assert statement we check whether the condition of the assert is true in all states that we may be in. In our example this is obviously true.

2.4.2 OpenJML

OpenJML is a commandline-tool which is built on OpenJDK [12]. It provides a parser and typechecker for JML-specifications which can be used to get an abstract syntax tree (AST) from a given Java file. Additionally OpenJML supports several modes which allow you to:

generate counter examples for violated specifications, verify specifications via SMT-solvers, generate relevant test cases for a program using its specification, create runtime checks for a given specification and create documentation taking JML-specifications into account. An Eclipse Plug-In exists as a front-end which can graphically present the results openJML obtained. For example showing counterexamples in the source code where the paths that were taken are marked and the variable assignments at the time of the error are displayed.

For this thesis the most interesting part of OpenJML is the API it provides to parse JML annotated Java code into an AST and manipulate it. OpenJML therefore extends visitors and utils classes provided by the JDK to support not only Java but JML as well. In our implementation we use OpenJML to parse the input files and manipulate the AST.

3 Translating Java and JML to JAVA!?

In this chapter we an approach how to translate a JML annotated method into a method with no specification but added assume and assert statements, so that an assertion is violated if and only if the original program violates its specification.

Definition 3.0.1 *We will call Java with assumptions and nondeterministic values JAVA!?.*

In the first section we introduce our basic idea and prove it to be correct for a simple while language. Than we extend the ideas to fit a real world programming language and refine it over the following sections.

3.1 Translating a While-Language

In this section we want to present a very general approach on how pre- and post-conditions of a while-program may be translated into a program using assertions and assumptions (which we will define soon) and prove that this translation is correct.

For this section we use first order dynamic logic (as presented in 2.1) as our foundation of argumentation. As a first step we extend the usual FODL-syntax to include two new operations: assume and assert.

Definition 3.1.1
$Term ::= var \mid f(term_1, \ldots, term_{\alpha(f)})$
$Fml ::= true \mid false \mid p(term_1, \ldots, term_{\alpha(p)}) \mid term_1 = term_2 \mid \neg fml_1 \mid fml_1 \wedge fml_2 \mid fml_1 \vee fml_2 \mid fml_1 \rightarrow fml_2 \mid \exists var.fml_1 \mid \forall var.fml_1 \mid [prog]fml_1 \mid \langle prog \rangle sfml_1$
$Prog ::= prog_1 \cup prog_2 \mid var := term \mid prog_1* \mid prog_1;prog_2 \mid var := * \mid ?fml \mid !fml$

As you can see, in addition to the standard FODL programs we added only one new possible syntactic element: $!fml$. This is because we can use the test operation as an assume. So the new syntax element ! is meant to be an assert operation. The semantic of these programs now has to be adapted to suite this new operation:

Definition 3.1.2 *Given a Kripke-Structure $K(S_{D_\epsilon}, p_D)$ which in addition to the normal states (implicitly given through D and Var) contains one special error state ϵ the semantics of programs is as follows:*

$$
\begin{array}{lll}
p(x := v) & ::= & \{(s, s_1) \mid s \neq \epsilon \wedge s_1 = s[x/val_{D,I,s}(v)]\} \cup \{(\epsilon, \epsilon)\} \\
p(x := *) & ::= & \{(s, s_1) \mid s \neq \epsilon \wedge \exists a \in D : s_1 = s[x/a]\} \cup \{(\epsilon, \epsilon)\} \\
p(a; b) & ::= & \{(s, s_1) \mid \exists t \in S : (s, t) \in p(a) \wedge (t, s_1) \in p(b)\} \\
p(a \cup b) & ::= & p(a) \cup p(b) \\
p(a*) & ::= & \{(s_1, s_n) \mid \exists n, s_1, \ldots, s_n : \forall 1 \leq i < n : (s_i, s_{i+1}) \in p(a)\} \\
p(?F) & ::= & \{(s, s) \mid s \neq \epsilon \wedge I, s \models F\} \cup \{(\epsilon, \epsilon)\} \\
p(!F) & ::= & \{(s, \epsilon) \mid s \neq \epsilon \wedge I, s \models \neg F\} \cup \{(s, s) \mid s \neq \epsilon \wedge I, s \models F\} \cup (\epsilon, \epsilon)
\end{array}
$$

(Where $a, b \in Prog$ and F is a program free FOL-formula.)

Notice how this definition of the semantic of assume (as the ?-operation) is slightly different from the semantic we normally see in programs like JBMC. In our definition an assume(false) would lead to the program having no possible states after that statement (unless we were in the error state before that). Normally a program with an assume(false) keeps running just that all assertions succeed afterwards. For us however this definition is useful as we are only interested whether all asserts hold or not (as you will see in the coming paragraphs).

Definition 3.1.3 *We call the language with the presented syntax and sematic while!?-language (as it is a normal while language with assertions and assumptions).*

So now assume we have a program P where $P \in Prog$ with precondition ϕ and postcondition γ. We want to show that if P is executed in a state which satisfies ϕ than after the execution of P γ holds. Expressed in the dynamic logic we just defined we want to show that $\phi \rightarrow [P]\gamma$ holds. In order to allow bounded model checkers like JBMC to verify such a formula we have to translate it in a manner that only a program remains (into while!?). So our proposed top level translation is (we write $A \Rightarrow B$ for A is translated to B):

$$\phi \rightarrow [P]\gamma \Rightarrow ?\phi; P; !\gamma$$

Our claim is that proving $I \models \phi \rightarrow [P]\gamma$ is equivalent to showing that $?\phi; P; !\gamma$ can not terminate in ϵ. More formally:

Theorem 3.1.1 $(I \models \phi \rightarrow [P]\gamma) \quad \leftrightarrow \quad (\neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon)$

In order to prove this theorem we introduce the following notation: We write S_{first} for the set $\{s \mid (s, s_1) \in S\}$ (S_{second}) accordingly).

Proof: First consider the set of states $T \subseteq S$ in which $\phi \leftrightarrow false$. If that is the case, then $I, T \models \phi \rightarrow [P]\gamma \leftrightarrow I, T \models false \rightarrow [P]\gamma$ which is valid and from the definition of p we also know that $p(?\phi; P; !\gamma) \leftrightarrow \{s, s_1 \mid \exists t : (s, t) \in p(?\phi) \wedge (t, s_1) \in p(P; !\gamma)\}$ but we know that $p(?\phi)_{first} \cap T = \emptyset$ since $\forall s \in T : I, s \models \phi \leftrightarrow false$ and $\neg\exists s \in S : I, s \models false$ so we know that $p(?\phi; P; !\gamma)_{first} \cap T = \emptyset$ and thus $\neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon$ is valid as well.

So now consider the opposite and say $S' \subseteq S$ is the set of states in which $\phi \leftrightarrow true$. Now $I \models \phi \rightarrow [P]\gamma \leftrightarrow I, S' \models [P]\gamma$ and similarly $S' = \{s_1 \mid \exists s \in S : (s, s_1) \in p(?\phi)\}$. We now introduce a second set $S'' \subseteq S'$ defined as $S'' := \{s_1 \mid \exists s \in S' : (s, s_1) \in p(P)\}$ and thus we know that $S'' = p(?\phi; P)$ and $I, S' \models [P]\gamma \leftrightarrow I, S'' \models \gamma$. From the definition of the !-operator now follows directly that $I, S'' \models \gamma \leftrightarrow \neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon$. So now we know that $(I, T \models \phi \rightarrow [P]\gamma) \leftrightarrow (\neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon)$ and $(I, S' \models \phi \rightarrow [P]\gamma) \leftrightarrow (\neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon)$. And since $T \cup S' = S$ we know: $(I \models \phi \rightarrow [P]\gamma) \leftrightarrow (\neg\exists (s, s_1) \in p(?\phi; P; !\gamma) : s_1 = \epsilon)$ q.e.d.

Quantified expressions Now only one problem remains: In the logic we presented above: by using one of the following operators ! or ?, formulas may be introduced into the program and formulas may contain quantifiers. These quantifiers are not part of the program syntax and thus we have to find a way to translate them as well. We propose the following translations:

1. $!(\forall x : \phi) \Rightarrow x := *;\ !\phi$
2. $?(\exists x : \phi) \Rightarrow x := *;\ ?\phi$
3. $?(\forall x : j \leq x \leq k \rightarrow \phi) \Rightarrow$
   ```
   $b := true$;
   $x := j$;
   $while(x \leq k)\ do$
   $b := b \wedge \phi$;
   $x := x + 1$
   $od$;
   $?\phi(b)$
   ```
4. $!(\exists x : j \leq x \leq k \rightarrow \phi) \Rightarrow$
   ```
   $b := true$;
   $x := j$;
   $while(x \leq k)\ do$
   $b := b \vee \phi$;
   $x := x + 1$
   $od$;
   $!\phi(b)$
   ```

Now we will prove that these translations maintain our claim made in Theorem 3.1.1. For the first translation it suffices to show that if in the first program the error state may be reached, it may as well be reached in the second one.

$$\epsilon \in p(!(\forall x : \phi))_{second} \leftrightarrow$$

$$\epsilon \in (\{(s, \epsilon) \mid I, s \models \neg(\forall x : \phi)\} \cup \{(s, s) \mid I, s \models (\forall x : \phi)\})_{second} \leftrightarrow$$

$$\epsilon \in (\{(s, \epsilon) \mid \exists x : I, s \models \neg\phi\} \cup \{(s, s) \mid \forall x : I, s \models \phi\})_{second} \leftrightarrow$$

$$\epsilon \in \{(s, s_1) \mid \exists t \in S : (s, t) \in (\{(v, w) \mid v \neq \epsilon \wedge \exists a : w = v[x/a]\} \cup \{(\epsilon, \epsilon)\}) \wedge (t, s_1) \in (\{(j, \epsilon) \mid I, j \models \neg\phi\} \cup \{(j, k) \mid I, j \models \phi\})\}_{second} \leftrightarrow$$

$$\epsilon \in \{(s, s_1) \mid \exists t \in S : (s, t) \in (\{(v, w) \mid v \neq \epsilon \wedge \exists a : w = v[x/a]\} \cup \{(\epsilon, \epsilon)\}) \wedge (t, s_1) \in (\{(j, \epsilon) \mid I, j \models \neg\phi\} \cup \{(j, k) \mid I, j \models \phi\})\}_{second} \leftrightarrow$$

$$\epsilon \in p(x := *;\ !\phi)_{second}$$

Now to the translation of an assumed existential quantifier. This proof is parallel to the previous:

$$\epsilon \in p(?\exists x : \phi)_{second} \leftrightarrow$$

$$\epsilon \in (\{(s,s)|\ I, s \vDash \exists x : \phi\} \cup \{(\epsilon,\epsilon)\})_{second} \leftrightarrow$$

$$\epsilon \in \{(s,s)|\ \exists x : I, s \vDash \phi\}_{second} \leftrightarrow$$

$$\epsilon \in \{(s,s_1)|\ \exists t \in S : (s,t) \in (\{(v,w) \mid v \neq \epsilon \wedge \exists a : w = v[x/a]\} \cup \{(\epsilon,\epsilon)\}) \wedge (t,s_1) \in \{(j,j)|\ I, j \vDash \phi\} \cup \{(\epsilon,\epsilon)\}\}_{second} \leftrightarrow$$

$$\epsilon \in p(x := *;\ ?\phi)_{second}$$

For the two last translations first notice that we only translate quantifiers which a) quantify over integers and b) are bounded (explicitly given upper and lower bound for x). This is important for our translation to work. Now we want to provide proves for these two rules in a little less formal way. First we prove the following lemma:

Lemma 3.1.1
$[b := true;$
$x := j;$
$while(x \leq k)\ do$
$b := b \wedge \phi;$
$x := x + 1$
$od;]$
$b \leftrightarrow \forall x : j \leq x \leq k \rightarrow \phi$
is valid.
(if b is fresh variable and does not occur in ϕ, j and k)

First we notice that if $j > k$ then the loop will never get executed and since b is initially set to true it still will be at the end of the program which is expected. So now we assume $j \leq k$. In this case the loop will be executed at least once. In each iteration of the loop b will stay true only if $\phi \leftrightarrow true$. If at one iteration $\phi \leftrightarrow false$ than b will become and stay equivalent to false. So b will be equivalent to false if and only if there is at least one x for which $\phi \leftrightarrow false$. Since x takes every value from j to k (its initialized with j and incremented until its greater than k) if there was any value between j and k that did lead to $\phi \leftrightarrow false$ it would be found and thus if $b \leftrightarrow \forall x : j \leq x \leq k \rightarrow \phi$

So with lemma 3.1.1 the correctness of the third translation rule follows instantly. A proof for the last translation rule can be done correspondingly to the one just done for the third rule.

3.2 Translating real Java and JML

We now present our translation from Java annotated with JML contracts to JAVA!? (see 3.0.1).

The main idea of the translation is to have rewriting rules which are applied recursively in a top down manner meaning the program gets broken down into smaller and smaller parts, which are then translated independently from each other. The translation for now is given for methods but could easily be extended to classes.

Our rewriting rules are given as a table where the first column is the original program and the second one the transformed version of it. To keep the translation tables as compact as possible the method is expected to be compiling Java code and have a normalized specification (as described in [1]), which essentially means that there is exactly one ensures-clause, one requires clause, one assignable-clause and one signals clause (although we leave out the signals clause since we do not support the translation of that).

Before we dive into the rules themselves, hare are some basic formatting decisions to make the rules more readable.

- Java-code is written in typewriter-font
- other rewriting rules which are used as subrules are written in italic
- the syntax [transformation()] is used to signal that the result of "transformation" is inserted at this position

So now we can look at the rule for the transformation of a method:

m	methodT(m)
	`M RT method(P) throws T {`
	[$assumeT(R)_c$]
	`assume(`[$assumeT(R)_v$]`)`
	[$saveOld(E, B)$]
`/*@ requires R;`	`RT resultVar =`
`@ ensures E;`	
`@ assignable A;`	[$defaultValue(RT)$]`;`
`@*/`	`try {`
`M RT method(P) throws T {`	[$bodyT(B, P, A)$]
`B`	`} catch(ReturnExc e) {}`
`}`	[$assertT(E)_c$]
	`assert(`[$assertT(E)_v$]`)`
	`return resultVar;`
	`}`

As you can see, on the left side the untransformed method has a specification containing the mentioned clauses but is otherwise a fully generic Java method. Modifiers (M), return type (RT) and throwable (T) are not changed in the translation as they do not affect the specification. Furthermore the translated method is using other translations for its body, its requires and its ensures clause. The basic idea is to assume the requires-clause then execute the body (with some transformations) and assert the ensures-clause (same as in the previous section for while!?). Beyond this base idea there are a few little details that we want to take a closer look at.

First off is the $saveOld(E, S, B)$ line. This is due to the fact that JML allows referencing the value of variables before executing the method with the \old keyword. So to be able to use this keyword in our translation we save all values of the variables that appear as argument in a \\old-call to new variables, which we then can use instead of the function call.

One line below that you can see a variable with the same type as the return type (RT) being declared and initialized. This is necessary since the body may contain return statements which would lead to skipping the part where the postcondition is asserted. So in order to avoid this, instead of returning a value we save it to the variable that we just declared. To have a compilable Java program this variable has to have a value when its first used. Thats why we defined a default value for each return type and this value is assigned to our return variable. For the same reason the transformed body is enclosed by a try-catch-statement which basically "filters" the ReturnException since this exception should not be thrown outside this method.

Translation of the method body As mentioned before, the body has to be translated as well. This translation is described by the rewriting rules given in table 3.1.

This transformation not only returns one value but two. The first column is the transformed code as we have seen it before for the first rule (therefore the index c). The second column represents the value of the transformed code (thus index v). As you can see, for some rules this value column is empty. This is because we handle two different kinds of code fragments here: statements and expressions. While expressions have values, statements do not. This differentiation between the transformed code and the value of the transformed code is necessary because of rules like the if-rule. As you can see the condition is basically copied in the transformation. But the transformed code potentially adds statements before the if-statement itself. This added code for example may contain assertions that are necessary due to the condition having side effects. So at this point we need the expression itself (as condition for the if) and the code that was added because of this expression. Hence the two columns.

One simple but essential rule is the last one which handles blocks. This rule allows us to transform blocks (which in Java are a single statement), by transforming each statement of the block body separately. The try-catch-rule and the return-rule are needed because of the problem mentioned in the last paragraph, being that return would alter the control flow to skip over the assertions we want to make at the end of the method. So as described above, we replace each return statement by assigning the return-value to a previously created variable and throwing an exception (which we then can catch and progress with the assertion of the postconditions). To prevent this exception from being caught anywhere else but by our intended catch, we replace each try catch statement inside the method body with a similar try catch statement which basically forwards each ReturnException and thus prevents it from being caught to early. For the sake of oversight we left out some rules at this point. For a list of the complete rule set refer to A.1.

3.2.1 Nothing is changing - How to handle loop invariants

One rule we want to specifically touch on is the loop-rule. Bounded model checking inherently has issues with loops since more often than not loops have no upper bound on how many times they may be executed. So the standard approach of BMC to unroll the loop, does not work in this situation. A possible solution to this problem is using loop

b	$bodyT(b)_c$	$bodyT(b)_v$

return expr;	$[bodyT(expr)_c]$
	resultVar = $[bodyT(expr)_v]$;
	throw new ReturnExc();	

lexpr $\circ$ rexpr ($\circ \notin \{=, += , -= , *=, /=\}$)		$[assumeT(lexpr \circ rexpr)_v]$
expr;	$[bodyT(expr)_c]$ $[bodyT(expr)_v]$;	

try stmt
catch (ET e)
cstmt
	try {
	$[bodyT(stmt)_c]$
	} catch (ReturnExc e) {
	throw e;
	} catch (ET e) {
	$[bodyT(cstmt)_c]$
	}	

if(c) thenStmt
else otherStmt
	$[bodyT(c)_c]$
	if($[bodyT(c)_v]$) {
	$[bodyT(thenStmt)_c]$
	} else {
	$[bodyT(otherStmt)_c]$
	}	

{
stmt1;
stmt2;
stmt3;
...
}
	{
	$[bodyT(stmt1)_c]$
	$[bodyT(stmt2)_c]$
	$[bodyT(stmt3)_c]$
	...
	}	

Table 3.1: Rewriting rules for the method body

invariants, as normal in tools for deductive verification (e.g. KeY [4]). As we discussed previously (2.1) a loop in dynamic logic has the following form: $(?\phi; a)*; ?\neg\phi$. Now if we want to show that a post condition of that loop Δ holds ($[(?\phi; a)*; ?\neg\phi]\Delta$) it suffices to show that for an arbitrary formula I: $I \wedge (I \wedge \phi \rightarrow [a]I) \wedge (I \wedge \neg\phi \rightarrow \Delta)$. So a loop invariant is a formula that has to hold before and after each loop-iteration. Showing that such an invariant holds before entering the loop and that if the invariant holds before a loop iteration, it will hold afterwards as well, allows us to prove programs containing loops independently of how many times they are executed. This is basically an induction over the loop iterations.

So for our rule we assert the invariant, then we havoc (see 3.2.7) the assignables of the loop, assume the invariant, execute the loop-body and assert the invariant again. Additionally in our case we want to provide a possibility to prove termination of the loop. In JML this is done with a decreases-clause. This clause states, that the given expression decreases in each loop iteration and is always greater 0. This way the loop is guaranteed to terminate at some point. Last but not least we add one more statement at the end of the loop body: assume(false). This is necessary because we chose the a random loop iteration but all assertions after the loop only have to hold if the loop was fully executed. So as long as the loop body is executed, we assume(false) which prevents any assertions after that point from failing.

b	$bodyT(b)_c$	$bodyT(b)_v$

`/*@ loop_invariant I;`
`  @ loop_modifies A;`
`  @ decreases D;`
`  @*/`
`for(Type v = i; c; inc) {`
`        body`
`}`
	`Type v = i;`
	`int oldD = D;`
	[$assertT(I)_c$]
	`assert(`[$assertT(I)_v$]`);`
	`havoc(v, A);`
	[$assumeT(I)_c$]
	`assert(`[$assumeT(I)_v$]`);`
	`if(c) {`
	    [$bodyT(body)_c$]
	    `inc;`
	    [$assertT(I)_c$]
	    `assert(`[$assertT(I)_v$]`);`
	    `assert(D > 0`
	      `&& D < oldD);`
	    `assume(false);`
	`}`	

3.2.2 Translating expressions

As mentioned before we use two different translation rules for assume and assert statements. This is mainly due to different handling of quantifiers (for detailed explanation see 3.2.3).

In this translation we again use two different columns where the first one is a list of added statements inserted before the expression may be evaluated and the second one is the translated expression itself. So for example consider the following translation of a implies-expression:

a	$assumeT(a)_c$	$assumeT(a)_v$
`expr1 <==> expr2`	$[assumeT(expr1)_c]$ $[assumeT(expr2)_c]$	$[assumeT(expr1)_v]$ == $[assumeT(expr2)_v]$

As you can see the rule is used recursively for each subexpression to created the necessary code for it and then transforms the equivalence-expression into an equivalent equals-expression. So this rule does not add any code itself but only passes on the code of its subexpressions. We wont discuss any further translation rules for expressions but you can find a complete table with all of them at A.1.

3.2.3 Translating Quantifiers

Quantifiers are an essential part of JML. There are two different types of JML-quantifiers: universal-quantifiers and existential-quantifiers. Quantifiers may be unbound (e.g. \forall int i; i ≤ i + 1;) and may quantify over objects (e.g. \forall Object o; list.contains(o); o == null). However in this work we only support quantifiers over a range of integers (with an explicitly given lower and upper bound). So in short we allow only quantified expressions of the form:

\forall/exists int i; i ≥ n && i ≤ j; $\phi(i)$;

Other conditions to the variable bound by the quantifier may be included in the last part of the quantified-expression (for condition cond: \forall/exists int i; i ≥ n && i ≤ j; cond(i) $\longrightarrow \phi(i)$;)

For the following chapter we call the variable bound by a quantifier the quantifier variable, the condition to the quantifier variable the quantifier condition and the expression which is quantified over the inner expression of a quantifier. (In the example above i is the quantifier variable, "i ≥ n && i ≤ j" the quantifier condition and "cond(i) $\longrightarrow \phi(i)$" is the inner expression). A quantifier whose quantifier condition evaluates to false is called an empty quantifier.

In JBMC, due to the nature of the tool, assert statements are implicitly all quantified and assume statements are implicitly exists quantified (3.1). So the JML-expression (\forall int i; i >= 0 && i < 10; i >= 0;) could be asserted using the following translation:

Listing 3.1: Translation of an assertion of a forall quantifier

```
int i = nondetInt();
assert(!(i >= 0 && i < 10; i >= 0) || i >= 0);
```

Notice how the quantifier variable is translated as a nondeterministic integer, which is not bounded at all. The inner expression however is only asserted for the given range (if i is not in range the asserted expression is trivially true due to the first part of the or). An important detail is the order of the quantifier condition and the inner expression in this case since the inner expression should only be evaluated for values of i which are valid according to the condition. For a detailed discussion of this issue see 4.1. Similarly the following expression (\exists int i; i >= 0 && i < 10; i >= 0;) may be assumed as follows:

Listing 3.2: Translation of an assumption of an existantial quantifier

```
int i = nondetInt();
CProver.assume(i < 10 && i >= 0);
CProver.assume(i >= 0);
```

Note how the translations differ in the second line. In contrast to the assume in the second translation, we use the negated quantifier condition as an additional asserted expression in the first one. The more intuitive way is the second translation, where the restrictions to the quantified variable are assumed. This has the advantage that no value that the quantifier excluded is used in the translation and it limits the paths JBMC has to check, such that we can expect that JBMC runs faster. However in the first translation this would lead to wrong results when translating empty quantifiers. An empty universal quantifier evaluates always to true, but its quantifier condition evaluates to false. So assuming the quantifier condition in this case would cut all remaining execution paths and thus result in JBMC not finding any errors after that line. In the second translation this effect is intended. Since an empty existantial quantifier should evaluate to false and we want to assume this expression, the result of assuming false is the expected behavior.

More problematic are the quantifiers which can not be translated using the implicit semantics of the used statement (assume and forall, assert and exists; we call these quantifiers "demonic quantifiers"). In this case we decided to use an explicit loop to translate the given quantifier. Using this idea the JML-expression (\exists int i; i >= 0 && i < 10; i >= 0;) may be asserted as follows:

Listing 3.3: Translation of an assertion of an existanial quantifier

```
boolean b = false;
for(int i = 0; i < 10; ++i) {
  b = b || i >= i;
}
assert b;
```

and accordingly (\forall int i; i >= 0 && i < 10; i >= 0;) may be assumed with:

Listing 3.4: Translation of an assumption of a forall quantifier

```
boolean b = true;
for(int i = 0; i < 10; ++i) {
  b = b && i >= i;
}
assume b;
```

So the formal translation rule would look like this:

a	$assumeT(a)_c$	$assumeT(a)_v$
`(\exists int i; i >= min && i <= max; expr)`	`int i = nondetInt(); CProver.assume(i <= max && i >= min);` $[assumeT(expr)_c]$	*expr*
`(\forall int i; i >= min && i <= max; expr)`	`freshBool(true, b); for(int i = min; i <= max; ++i) {` $[assumeT(expr)_c]$ `b = b &&` $[assumeT(expr)_v]$ `}`	*b*

Notice how these rules introduce new code as discussed in the previous section. For example the boolean variable b (forall-translation) can not be evaluated (or at least is not equivalent to the translated expression) if the new code is not executed before.

These translations are exactly the ones we presented in the translation of a while-language earlier this chapter (see 3.1). Since the argumentation why this is a sound is equivalent to the ones for the while-language we wont go into details at this point.

3.2.3.1 Dealing with Negated Quantifiers

Additionally we have to take care of negated quantifiers. There are two problems: The first one being that the easiest translation for angelic quantifiers namely translating

Listing 3.5: Assert a negated quantified expression

```
assert(!(\forall int i; i >= 0 && i < 10; i >= 0;))
```

as:

Listing 3.6: Assert a negated quantified expression (wrong translation)

```
int i = nondetInt();
assert(!(i >= 0) || !(i >= 0 && i < 10; i >= 0));
```

is obviously wrong. To solve this problem we use the duality of the two quantifiers:
$\neg(\exists i : \phi(i)) \leftrightarrow \forall i : \neg\phi(i)$
and
$\neg(\forall i : \phi(i)) \leftrightarrow \exists i : \neg\phi(i)$

Knowing that we can eliminate negated quantifiers and such avoid needing a translation for them. The second problem is that quantifiers may appear as subterms in a negated term. For example how do we assert:

Listing 3.7: Assert a negated expression containing a quantifier

```
assert(!(5 > i || (\forall int i; i >= 0 && i < 10; i >= 0;)));
```

Here the solution is to bring the formula into negation normal form (NNF). In NNF negations are only allowed before variables and literals. For example $\neg(A \wedge B)$ may be rewritten to $\neg A \vee \neg B$ (De Morgans law).

Lemma 3.2.1 *Each term consisting of variables, literal, quantifiers and the logical operations* $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ *can Be brought into NNF.*

To do this transformation we use the following table:

A	NNF(A)
$\neg(A \wedge B)$	$\neg NNF(A) \vee \neg NNF(B)$
$\neg(A \vee B)$	$\neg ANNF(A) \wedge \neg NNF(B)$
$\neg(A \rightarrow B)$	$NNF(A) \wedge \neg NNF(B)$
$\neg(A \leftrightarrow B)$	$(\neg NNF(A) \wedge NNF(B)) \vee (NNF(A) \wedge \neg NNF(B))$
$\neg\neg A$	$NNF(A)$
$\neg(\exists i : A(i))$	$\forall i : \neg NNF(A(i))$
$\neg(\forall i : A(i))$	$\exists i : \neg NNF(A(i))$

We can use this theory for JML expressions as well, since we support exactly these boolean operators (and, or, implies, equivalence). So before applying our translation we always bring each expression into NNF as a preprocessing step. This way we do not have to deal with the two problems mentioned above.

3.2.4 Framing everybody - or how to handle assignables

In this section we discuss how assignable-clauses (see 2.3) may be translated.

There are three different syntactic entities in Java, to which an assignment can be made: a (local) variable, a class-field or a field of an array.

In order to allow only legal assignments according to a given specification, before each statement which includes a write operation (e.g. =, ++, −, +=, -=,...) we assert that the assignment is legal. To define what "legal assignment" means we use the following notions. First we call the left side of an assignment lhs and the right side rhs. Each method may have exactly one (wlog) assignable clause consisting of several LocationSets. A LocationSet may be:

- a parameter: passed to a method
- a field: of a class
- an arrayRange (`a[i], a[i..j], a[..i], a[i..], a[*]`): this is not Java but JML specific
- all fields of an object (fields of object o: o.*) (we call that a wildcard-field): again this is JML specific

We define a predicate conforms(l, s), which is true if and only if an assignment to l is allowed with an assignable clause only consisting of s. For the following definition we assume that "this." is never left out (so this.f is never written as f). Additionally we assume that index-ranges are always given in the form i..j meaning that if the upper bound is not given we set it to the length of the indexed array minus 1 and if the lower bound is not given we set it to 0 (e.g. arr[..i] becomes arr[0..i], arr[*] becomes arr[0..arr.length] and so on) and if only 1 index is given, upper and lower bound are set to this index (e.g. arr[i] becomes arr[i..i]). Last but not least we introduce a function val(x) which gives us the value of a given variable/field/arrayAccess. We define conforms(l, s) as follows:

Definition 3.2.1
$conforms(a[idx], b[i..j]) ::\leftrightarrow val(a) = val(b) \wedge val(idx) \leq val(j) \wedge val(idx) \geq val(i)$
$conforms(a[i], b) ::\leftrightarrow val(a) = val(b)$
$conforms(t.f, o.*) ::\leftrightarrow val(t) = val(o)$
$conforms(t.f, o.f) ::\leftrightarrow val(t) = val(o)$
$conforms(v, s) ::\leftrightarrow val(v) = val(s)$

This definition intuitively states that:

- its legal to assign to a array field if either the array itself is writable or an array range of this array is writable and the index of the write assignment is between the upper and lower bound of the array range
- its legal to assign to a field of object o if either all elements of object o are assignable or exactly this field of this object is assignable
- its legal to assign to a variable if the referenced object is assignable itself or aliases with an object which is assignable

And with this definition we are now able to define what a legal assignment is:

Definition 3.2.2 *An assignment to lhs l with corresponding assignable clause c (consisting of one or more locationSets s) is called legal if* $\exists s \in c : conforms(l, s)$ *or l is a local variable with primitive type.*

This is the intuitive way to define a legal assignment as it states that we need some location set in the assignable-clause which allows us to perform said assignment. Assignments to local variables of primitive type are always legal as they cannot alias with other objects and thus can not manipulate heap objects.

To clarify why we define conforms(x, y) as we did, we illustrate the heap as a table (see Fig. 3.1) where each column is an object on the heap and each row is a field that an object may or may not have. Now consider an assignment to *.f is made. First thing we notice is that any locationSet that does not end with .f cannot allow such an assignment since all fields are rows and as such are pairwise disjunct (exception .* which allows writing to any field). Now all we have to check, is whether the object is the same. Since Java variables however are not objects but references to objects we can not distinguish on a syntactical level whether two variables P and T reference the same object or not. Hence the comparison of the values of the variables. In the graphic this is illustrated as the two variables P and T which point to the same object.

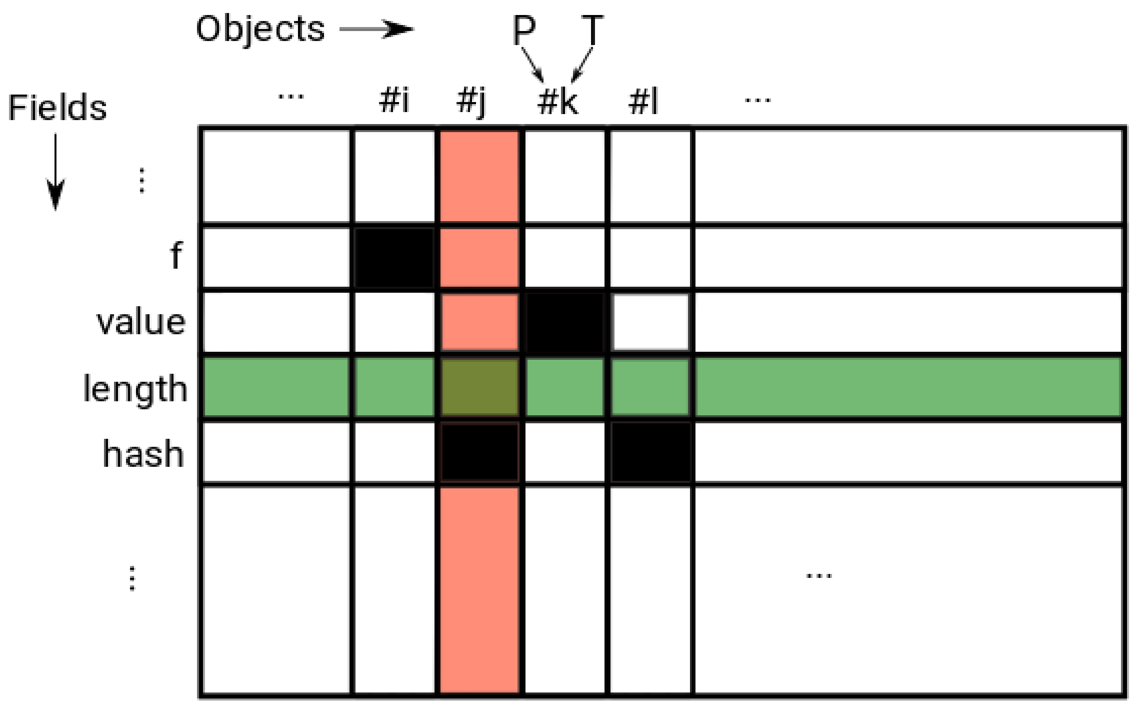

Figure 3.1: Illustration of a heap in Java

This approach however can not handle newly created objects. For example the snippet of code shown in Lst. 3.8 could not be verified.

Listing 3.8: An example showing a case which can not be handled with the first approach

```java
//@ assignable \nothing;
private void method(SomeObject o) {
  SomeObject obj = new SomeObject();
  obj.someField = value;
}
```

Since obj is a newly created object assigning to its fields is legal but by the definition above there is no assignable clause, which allows writing to this object so this could not be verified.

To avoid this weakness we define an additional predicate $fresh(o)$ which we assume for each newly created object. So in the translation after each assignment of an new object o to a local variable we assume $fresh(o)$.

Now we can refine our definition 3.2.2 of a legal assignment to:

Definition 3.2.3 *An assignment is legal if* $\exists s \in c : conforms(l, s) \vee fresh(l)$ *or* l *is a local variable with primitive type.*

So now that we know what an legal assignment is, we can use a translation rule to ensure that only legal assignments are allowed.

b	$bodyT(b)_c$	$bodyT(b)_v$
lexpr $\circ$ rexpr ($\circ \in \{=, += , -=, *=, /=\}$)	assert ($conforms(lexpr, A)$); $[bodyT(rexpr)_c]$	lexpr $\circ$ $[bodyT(rexpr)_v]$

As you can see, if any expression occurs that assigns a value to a lhs, than an according assert statement is added to ensure that its legal.

3.2.5 Ensuring Well-definedness

The well-definedness of a specification is defined as whether all expressions in the specification can be evaluated without throwing an exception. In our case this is straight forward since we translate the specification to normal Java anyway. So all we have to do is to ensure that the translation has the same semantic as the original specification. For most operations this is naturally given. For example a specification only consisting of the following ensures clause: ensures 1/0 > 0 is not well defined but since we translate this to: assert 1/0 > 0 we don't have to do any special treatment. The only exception to this is the treatment of shortcut evaluation in combination with demonic quantifiers. Consider the specification clause in Lst. 3.9.

Listing 3.9: An example of a well defined ensures-clause

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

In Java the evaluation of an or-expression is stopped after evaluating the first operand, if it evaluates to true (since the result is going to be true independently of the evaluation of the second operand). So the above specification is well-defined since there is no case in which an exception could be thrown. But the translation as we proposed it would translate Lst. 3.9 to the code shown in Lst. 3.10 example to:

Listing 3.10: An example of a well defined ensures-clause translated

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

As you can see the value of the quantifier is basically precomputed and thus this specification would result in an error. The solution of this problem is, to translate the quantifier only if it is needed. So for each operator with a possible shortcut (and, or, implies) we define a function sc(op), which gives us the value for which the shortcut is taken.

op	sc(op)
&&	false
||	true
==>	false

Now we can adapt the translation for these operators as follows:

a	$assertT(a)_c$	$assertT(a)_v$
`expr1 ∘ expr2` $\circ \in \{\&\&, \|\|, +, -, *, /, \%, ==, != \}$	$[assertT(expr1)_c]$ $[assertT(expr2)_c]$	$[assertT(expr1)_v] \circ [assertT(expr2)_v]$

becomes

a	$assertT(a)_c$	$assertT(a)_v$
`expr1 ∘ expr2` $\circ \in \{+, -, *, /, \%, ==, != \}$	$[assertT(expr1)_c]$ $[assertT(expr2)_c]$	$[assertT(expr1)_v] \circ [assertT(expr2)_v]$
`expr1 ∘ expr2` $\circ \in \{\&\&, \|\|\}$	$[assertT(expr1)_c]$ `freshBool(false, b);` `if(`$[assertT(expr1)_v]$ `!= sc(`$\circ$`)) {` $[assertT(expr2)_c]$ `b =` $[assertT(expr1)_v]$ `}`	$[assertT(expr1)_v] \circ b$

Notice how the second part of the operation is only executed if it is necessary and thus can only throw an exception if the in standard Java semantics an exception would have been thrown anyway.

Using this adapted translation the example given in Lst. 3.9 gets translated to Lst. 3.11.

Listing 3.11: An example of a well defined ensures-clause adapted translation

```java
boolean b1 = false;
if(true != true) {
  boolean b2 = false;
  for(int i = 0; i < 1; ++i) {
    b2 = b2 || 1/i == 0;
  }
  b1 = b2;
}
assert(true || b1);
```

Note how the critical operation (i/0) is never executed and thus no error occurs and the expression is evaluated as expected.

3.2.6 Method Calls - Making it modular

The standard treatment for method calls in JBMC is inlining them. Now that we want to have a modular tool we have to provide a possibility to avoid this behavior. If the called method has a specification this is however straight forward: We assert that the precondition of the called method is satisfied when we call it, then we havoc all assignables of this method and assume its postcondition. So the rule we use is as follows:

m	methodCallT(m)

`/*@ requires R;`
`  @ ensures E;`
`  @ assignable A;`
`  @*/`
`M RT method(P) throws T {`
`        B`
`}`
	`M RT methodC(P) throws T {`
	$[assertT(R)_c]$
	$assert([assertT(R)_v])$
	$saveOld(E, B)$
	$havoc(A)$
	`RT resultVar;`
	$[assumeT(E)_c]$
	assume $([assumeT(E)_v])$
	`return resultVar;`
	`}`

As in the "normal" translation of the method we use the method *saveOld* to store values of variables which are later referred to via $\backslash old$. Note that we again introduce a variable for the return value. We have to do this before assuming the postcondition since the postcondition may contain restrictions to the returned value.

3.2.7 Wreaking havoc - or how to anonymize Java objects

At several points in the translation it is necessary to anonymize certain values or objects. We call this operation havoc. Havocs may be applied to locationSets (which are, just a reminder):

- a variable
- a field
- an arrayRange (a[i], a[i..j], a[..i], a[i..], a[*])
- all fields of an object (field of object o: o.*) (we call that wildcard-field)

For variables havocing is equivalent to assigning a nondeterministic value of the same type. For arrays we treat each field of the array as an independent value and thus can assign nondeterministic values as for variables. It may be necessary to create a for loop to iterate over all indices since they might be determined at runtime (e.g. a[i..j]). A special case is a[*] where we currently assign a new nondeterministic array and assume that it has the same length. This is not sound since it is not the same object as before. The same is true for wildcard-fields where we currently use f = nondetObject as havoc for f.*. As mentioned this translation suffers from the same problem as the havoc method for array-wildcards.

3.2.8 Object Invariants

JML allows to specify object invariants. Informally object invariants are invariants that have to hold whenever an object may be accessed from outside the class itself (e.g. after each method call of this object, after a public constructor, ...). For a formally complete definition on when object invariants have to hold and differences between static and instance invariants please refer to [21].

As a simplification consider object invariants to be normal JML-expressions as they may occur in method pre- or postconditions. If thats the case to support object invariants we additionally assume the object invariant at the start of a method and after a method invocation and in return show that it still holds after executing the method body and before invoking a method.

This is a rude simplification and is just thought to demonstrate that supporting object invariants with the presented approach is theoretically possible.

4 Implementation

In this chapter we are going to present our tool, which implements most parts of the theoretical approach presented in chapter 3. We will therefore describe how the tool is working and what features we implemented. We then discuss the features we left out as of now and which of them are theoretically plausible to implement at a later stage. In the last section of this chapter we describe how we tested our translation in order to guarantee a certain degree of stability.

The tool is written as a Java console application. Input for it is a file with Java code and JML specification. Additionally some options may be provided. For a more detailed description of the available options and their effects see 4.3. The tool internally translates the file and calls JBMC for the specified method.

4.1 Design Decisions

The basic design decision for the translation was that it should be able to be used like a runtime checker. So given a well-defined specification (see) that is not violated by the program the result of the original program and the translated version should be equivalent (exceptions are thrown for the same inputs and if no exception is thrown the return value is the same). The well-definedness is a necessary limitation to that claim since the specification is translated to JAVA!? (see 3.0.1) and may thus throw exceptions if its not well-defined. If the program does not conform to the specification, then assertions will fail or exceptions are thrown so the result may also differ from the original program.

Knowing that we would not be able to support all Java/JML features for our prototype, it was important to us to make clear to the user which features we support which we don't, to prevent unsound behavior. So our solution was to use a white list approach where whenever a feature of either JML or Java is present in a file that our tool does not support, we throw an according exception. This way our tool may support a smaller subset of Java/JML than it is theoretically capable of, but we are sure that no the proofs our tool produces are unsound due to unsupported features not being detected.

Last but not least we wanted to make sure that the tool brings everything it needs to be executed. So we decided to ship included JBMC and OpenJML versions so that there is no confusion on which version of OpenJML or JBMC we are using and that bugs due to conflicting versions being used are eliminated. Everything the tool needs is copied automatically.

4.2 Discussion of supported features

Since our tool is a prototype, not all features presented in the last chapter are already implemented. In this section we would like to discuss which features we implemented and which are still missing and why.

Method Contracts For method contracts we support exactly one specification case for normal behavior. Furthermore we only support the most fundamental keywords in `requires`, `ensures` and `assignable` to support pre-, post- and frame-conditions. Other clauses like `signals` or `signals_only` are currently not supported but should be relatively easy to implement since they require one more catch statement for the try which surrounds the body anyway. For `signals` an additional assertion is added for `signals_only` all listed exception types are caught so JBMC would only fail if an exception type not listed it thrown. Other clauses like `diverges` or `measured_by` may be harder to implement but discussing all of them here would go beyond the scope of this work.

Java-Expressions Java expressions should be fully supported however since the white list approach of our implementation requires to list all supported operations explicitly we may have missed some. This is no fundamental problem but only a matter of seconds to add them to the list of supported operations.

JML-Expressions JML expressions may contain either Java expressions or special JML expressions. For special JML expressions we only support a small subset, since normally supporting these expressions need explicit implementations for each of them. Most importantly we support quantifiers over bound integer ranges. In general supporting quantification over unbound ranges is, at least for demonic quantifiers, impossible due to the nature of BMC. Quantification over bounded sets of objects is certainly possible[9] but requires some more implementation.

Additionally we support the `\old` keyword however only for primitive types. To refer to the old value of a variable the obvious way is to save its value before executing the code. However this is not trivial for non primitive types due to the complexity of creating a deep copy of objects in Java[18]. Furthermore we support implications and equivalence as the are not part of the normal Java syntax but easy to implement (see 3.2.3.1). Last but not least `\result` is supported as it is a crucial tool in specifying method behavior.

Loop specification Providing loop invariants is supported. For the exact handling of loop invariants refer to 3.2.1. We also support `decreases` statements and assignables for loops. As of now this is only supported for while and (standard) for loops however extending the implementation to support do-while or for-each loops is possible.

Assignables Our implementation allows assignable clauses with all types of locations sets including the special keywords `\nothing` and `\everything`. Our approach is sound but not complete since we are not able to handle assignments of newly created objects.

The approach of using uninterpreted functions as presented in 3.2.4 is possible to realize, since JBMC allows the use of uninterpreted functions, but is currently not implemented.

Object Invariants Object invariants are currently not supported but are not theoretically out of scope (see 3.2.8).

4.3 Tool Options

The tool supports the following options:

- fileName (needed): The path to the file that contains the method to be verified.
- methodName: The name of the method to be verified (currently only methods of the class with the same name as the file are supported)
- -verifyAll/-va: may be given instead of a methodName. If given the tool verifies all methods (in the class with the same name as the file).
- -dontFilter/-df: The tool as default filters the jbmc output to provide a more readable result. If this is not wanted the filter may be turned off using this option.
- -keepTranslation/-kp: The tool internally creates a new java file containing the translation of the given specification. This file gets deleted after calling jbmc. Given -keepTranslation this temporary file will not be deleted.
- -jbmcOpt/-j: This option takes jbmc options to be given to jbmc (may be used multiple times to pass multiple options).
- -help/-h: Shows a help message for the program which lists all options and explains the general usage.

4.4 Testing

A test for our scenario is not a unit test of the implementation but a Java method with specification which is annotated as to whether this method with its given specification should be verifiable with our approach or not. The expected result is provided using a custom annotation. Additionally its possible to provide the number of times JBMC should unwind loops and recursions via another annotation. The test framework translates these tests and runs JBMC on each of them asserting that the result of JBMC is the expected one. This way writing tests to test new behavior is very fast and easy.

The test suite provided with the tool contains over 100 tests covering all major features of the tool. Three aspects of the tool were tested. First the soundness of the tool. This is normally done with tests that are expected to be not verifiable. The second aspect is the completeness of the supported features. Last but not least there are some tests which ensure that unsupported features throw exceptions.

As an example consider the following two tests:

Listing 4.1: An example for a test case

```java
//@ ensures !(\forall int i; i >= 0 && i < 3;
(\exists int j; j >= 0 && j < 3; j > i));
@Verifiable
private void negatedQuantifierTest1() {}

//@ ensures !(\forall int i; i >= 0 && i < 3;
(\exists int j; j >= -1 && j < 3; i > j));
@Fails
private void negatedQuantifierTest3() {}
```

As seen in 4.1, the tests are annotated as to whether they should be verifiable or not. In this case the translation of negated nested quantifiers is tested.

5 Evaluation

To evaluate the presented approach we conducted multiple case studies proving real world programs taken from different contexts to show that the presented approach is not only working but provides significant improvements for prove finding. The full source code for all case studies can be found in the appendix of this thesis (see A.2).

5.1 Bubble Sort

As a first test for our approach we wrote a simple bubble-sort implementation. The top level specification states that the array is sorted in the usual way (each element is smaller or equal to the following one). It contains the usual two nested loops which iterate over the array and a swap method which swaps the position of two array elements. This swap method does an in place swap with bitwise xors.

We proved this example with our approach for arrays up to size 5. To examine our approach we tweaked this case study in two directions: Firstly we proved it with and without loop invariants provided, secondly we proved it inlining the swap function compared to using its specification instead. Our goal was to compare the different versions according to runtime. In table 5.1 you can see the results of this experiment.

5.2 Big Integer Conversion

Wolfram Pfeiffer as his Bachelor thesis proved the basic arithmetic operations of the BigInteger class of Java correct[24]. This case study was conducted with KeY which lead to the problem that one particular method was difficult to verify due to its nature of containing several bit operations on integers. The functionality of this method is that it converts a signed integer into an unsigned integer (see Fig. 5.1). To prove this method Pfeiffer et al. translated their JML contract in a equivalent assert statements and proved this translation with CBMC. Our approach was now capable of proving the method to be correct without further changes. This example shows that we are able to prove methods

	/w invariants	w/o invariants
inlining	28s	30s
no inlining	33s	-

Table 5.1: Different runtimes of BubbleSort case study

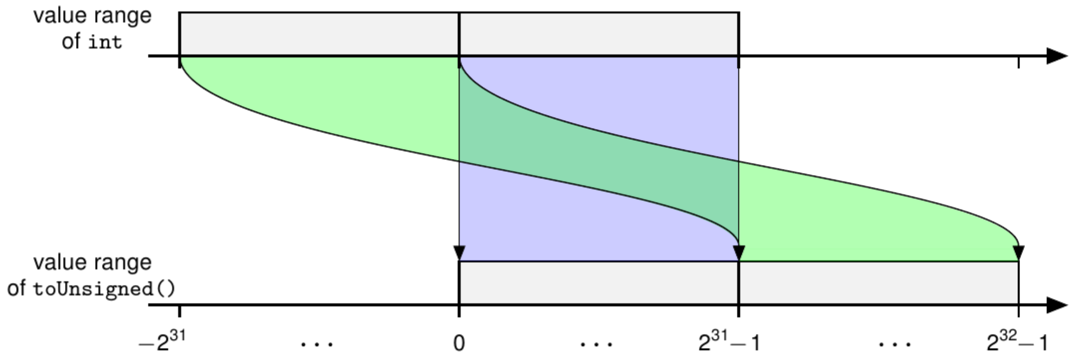

Figure 5.1: The idea of the to unsigned method (taken from [24])

that are not provable in KeY and such facilitates the idea that a JML supporting bounded model checker can be an asset to deductive verification at certain points.

5.3 HammingWeight

The hamming weight of an integer is the number of 1 bits in its binary representation. So to determine the hamming weight of an integer, the naive approach is to iterate over its bits and count the ones that are not zero. A more sophisticated approach is to use some bit magic to calculate the hamming weight in constant time. This can we done as shown in Lst. 5.1:

Listing 5.1: One way to calculate the Hammingweight of an integer

```
x = x - ((x >>> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >>> 2) & 0x33333333);
x = (x + (x >>> 4)) & 0x0f0f0f0f;
x = x + (x >>> 8);
x = x + (x >>> 16);
result = (x & 0x3f);
```

Now for our case study we extended this to integer arrays in the natural way, where the hamming weight of an integer array is the sum of the hamming weights of the integers in the array. We proved that the hamming sum of an integer array does not exceed 32 times the length of the array. This is a very simple example but demonstrates 2 strengths of our approach. If we use the "bit magic"-approach we take advantage of JBMCs ability to handle bit operations very well and if we take the naive approach we have a bounded loop which we can simply unroll without needing an invariant. Both approaches are verified using our tool taking less than a second each.

5.4 Dual-Pivot-Quicksort

Schiffl et al. proved JDKs implementation of Dual-Pivot-Quicksort correct (although they found one invariant that does not hold)[4]. Since this algorithm operates differently depending on the size of the array to verify the algorithm as a whole we would have to do this for arrays of size bigger than 46, which is not feasible for our approach. However again in this paper CBMC was used to prove a method which contained bit operations. We were able to reproduce the proof for this method and some of the auxiliary methods that were used in the proof. Our tool was able to verify the given specification without further adaption fully automatically.

Interestingly we found a minor error regarding the well-definedness (see 4.1) of one verified method. Consider the loop invariant in Lst. 5.2.

Listing 5.2: The original loop invariant of the Dual-Pivot-Quicksort proof

```
@ loop_invariant
@     k <= great && 0 <= great
@     && (\exists int i; left <= i && i <= great;
    a[i] <= pivot2)
@     && (\forall int i; great < i && i <= \old(great);
    a[i] > pivot2)
@     && great <= \old(great);
@ assignable great;
@ decreases great;
```

Notice that the variable *great* maybe changed during the loop body (as it is part of the assignable-clause). This means that to show the invariant we have to show it for an arbitrary value for *great*. Now notice that great has given an upper and lower bound in the invariant itself, however the upper bound is only given after the two quantified expressions. So assuming the invariant for an arbitrary value of great is not well defined since the index of the array access in the exists quantifier has basically no upper bound.

To correct this specification it suffices to change the order of the lines so that the upper bound is given before the quantified statements are reached as seen in Lst. 5.3

Listing 5.3: The corrected loop invariant of the Dual-Pivot-Quicksort proof

```
@ loop_invariant
@     k <= great && 0 <= great
@     && great <= \old(great)
@     && (\exists int i; left <= i && i <= great;
    a[i] <= pivot2)
@     && (\forall int j; great < j && j <= \old(great);
    a[j] > pivot2);
@ assignable great;
@ decreases great;
```

5.5 PairInsertionSort

PairInsertionSort is the counterpart to DualPivotQuickSort which is used to sort arrays with less than 47 elements. It is part of the Java standard library. We verified this algorithm using our tool for arrays up to length 5. To do so we removed all given loop invariants and block contracts and proved only the sortedness of the resulting array not that it is a permutation of the original array. This proof was successful without any adaptations of the source and was conducted fully automatically in approximately 10s.

Again we experimented with adding known loop invariants and removing them to see whether or not and how fast our tool can verify these loop invariants. This time the loop invariants where more complex than in the previous examples and our tool failed to proof any of them in reasonable time. We tried several configurations but it seems that all of them are hard to proof for our approach.

5.6 Multiplication

The last case study reveals one weakness of our approach. We tried to prove the program shown in Lst. 5.4.

Listing 5.4: Method which multiplies two integers by multiple additions with specification

```
/*@
  @ requires x1 >= 0 && x2 >= 0;
  @ requires x1 < N && x2 < N; //limit x1, x2 somehow
  @ ensures \result == x1 * x2;
  @ assignable \nothing;
  @*/
public static int mult(int x1, int x2) {
  int res = 0;
  //@ loop_invariant res == i * x2;
  //@ loop_invariant i >= 0 && i <= x1;
  //@ loop_modifies res;
  //@ decreases (x1 - i) + 1;
  for(int i = 0; i < x1; ++i) {
    res += x2;
  }
  return res;
}
```

As you might have noticed, this is the same example as in 2.3 where we multiply two integers by multiple additions. Note that this is an example containing a loop where we are theoretically able to eliminate the loop using the loop rule with the given invariant, such that an unbounded verification is possible. But here comes the weakness of JBMC into play. If x1 and x2 are not limited, JBMC takes hours to verify this little example. The problem here is that x1 and x2 are nondeterministic values (since they are parameters) and the multiplication of two nondeterministic values apparently is very hard to handle for JBMC. This problem can be boiled down to the code shown in Lst. 5.5:

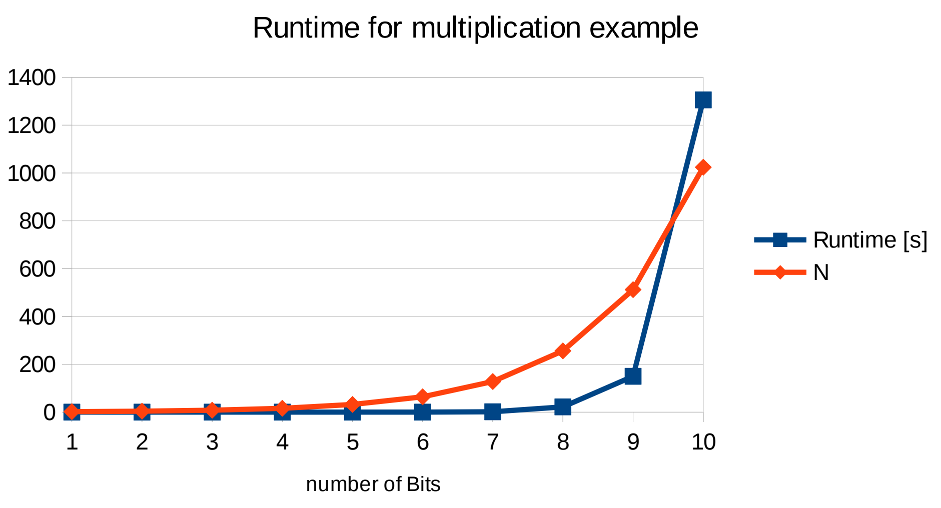

Figure 5.2: Different runtimes of CBMC depending on input size

Listing 5.5: A minimal code example exposing the weakness of JBMC concerning multiplication of nondeterministic values

```
//@ requires x1 < N &&  x2 < N;
//@ requires x2 >= 0&& x1 >= 0;
//@ requires res == x1 * x2;
//@ ensures res + x2 == (x1 + 1) * x2;
public void test(int x1, int x2, int res) {
}
```

We conducted an experiment where we ran this example with different values for N. The results of this experiment are shown in Fig. 5.2. As you can see, the runtime of CBMC grows way faster than the input and becomes so large that verifying programs that rely on this is basically infeasible.

6 Related Work

The idea of automated software verification is not new. There are several ideas, tools and approaches on how to handle different aspects but there is no silver bullet. On a top level view there are three main quality criteria that are essential at this field:

- the complexity of the properties that can be verified
- the scalability of the approach
- the amount of needed user interaction

For each quality there are tools which focus on that criterion. For rather simple properties like dereferencing null pointers or dead locks there are tools like JayHorn[16] and JBMC[13]. These tools are highly automated and mostly run fast even on big code samples but are not suitable to verify complex (object related) properties of programs. Other tools like OpenJML[12] and InspectJ[22] provide the possibility to specify (via JML) and verify such properties but may timeout or return "unkown". In contrast to these approaches there are tools which focus on proofs of complex properties without restrictions to loop iterations or heap size but in turn need a lot of user interaction in order to find proofs (for example KeY[1] or Coq[5]).

6.1 Combining Deductive Verification and BMC

There are several papers which present approaches based on combining deductive verification and BMC. For example in [25] finite state systems are verified relative to linear time logic constraints or in [26] where many connections between DV and BMC are shown and examples how these two approaches can interact are presented. Another paper that discusses this synergy and shows how BMC and DV can be used to create an advantageous proof system for C is shown in [3].

6.2 KeY

KeY[1] is a tool for deductive verification developed at the Karlsruher Institute of Technology and the TU Darmstadt. It allows verifying Java programs annotated with JML contracts. As the underlying logic a sequence calculus is used. The Java program gets translated into a sequence to which the user can apply rules of the calculus to prove the generated proof verification conditions. KeY provides an auto pilot which automatically searches for rule applications which close a proof and offers a translation of a sequent to SMT. This translation allows to generate counterexamples when verifications fails.

Additionally a symbolic execution debugger integrated into Eclipse as a plugin allows to debug programs using extra information obtained by symbolic execution.

6.3 JML Runtime Checker

Gary Leavens et al. presented a tool which is able to translate JML into runtime assertions checks for Java programs[9]. In this paper they discuss similar topics as we did in this thesis like for example the well-definedness of specifications and the translation of quantifiers. At this point they offer quantification over iterable collections and offer implicit quantifiers like `\sum`. Additionally the paper discusses specification inheritance and model specifications. This approach is quite similar to the one we presented here having the main difference is that while Leavens et al. use their approach to actually perform runtime checks we use it as the input for a static verification tool (thus they translated JML to Java and we translate JML to Java!?). Our advantage is that the performance of the generated code does not matter (since the generated code should never actually be executed) which they describe to be one of the main disadvantages. Similar approaches have been presented by [8], [20] and [17].

7 Conclusion

In this thesis we presented an approach which allows the verification of Java programs annotated with JML contracts using a BMC. We discussed how we are able to translate JML contracts into Java!?, an augmented Java with assumes, asserts and nondeterministic values. We proved that the basic idea of our translation is correct for a while-language and presented formal rules for the translation of a subset of Java/JML. Additionally we implemented our approach as a command line tool in Java. This tool allows the verification of JML contracts using our translation and the Bounded Model Checker JBMC. To evaluate our approach we conducted several case studies which show that our tool is able to proof contracts which were difficult to proof in other tools and even showed a minor well definedness error in one specification. As this approach uses Java/JML as input it is possible to use it in combination with other proof systems.

We see several ways to extend our research in this area: The First one being that the supported subset of JML could be extended to support more features of JML (e.g. Object invariants, signals-clauses, ...). Secondly the translation could possibly be improved concerning the runtime of JBMC. As our focus was the soundness of our translation, the number of asserts and whether there are possibilities to allow JBMC to verify our translated program faster, were not yet fully explored. Furthermore the counterexamples that are generated by JBMC for a failed proof attempt are very hard to read. A more readable presentation of those counterexamples, at best with line numbers and error descriptions in the original source, would help a great deal when trying to understand why a proof is not closing.

Overall we consider this thesis a success as the goal laid out in the proposal of this work was achieved: We were able to verify JML contracts which were difficult to prove in a DV-tool alone and thus showed that the combination of DV and BMC is advantageous.

Bibliography

[1] Wolfgang Ahrendt et al., eds. *Deductive Software Verification – The KeY Book: From Theory to Practice.* en. Programming and Software Engineering. Springer International Publishing, 2016. ISBN: 978-3-319-49811-9. URL: `https://www.springer.com/de/book/9783319498119` (visited on 02/06/2019).

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting Equality of Variables in Programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '88. event-place: San Diego, California, USA. New York, NY, USA: ACM, 1988, pp. 1–11. ISBN: 978-0-89791-252-5. DOI: `10.1145/73560.73561`. URL: `http://doi.acm.org/10.1145/73560.73561` (visited on 02/07/2019).

[3] Bernhard Beckert et al. "Integration of Bounded Model Checking and Deductive Verification". en. In: *Formal Verification of Object-Oriented Software.* Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 86–104. ISBN: 978-3-642-31762-0.

[4] Bernhard Beckert et al. "Proving JDK's Dual Pivot Quicksort Correct". en. In: *Verified Software. Theories, Tools, and Experiments.* Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Cham: Springer International Publishing, 2017, pp. 35–48. ISBN: 978-3-319-72307-5 978-3-319-72308-2. DOI: `10.1007/978-3-319-72308-2_3`. URL: `http://link.springer.com/10.1007/978-3-319-72308-2_3` (visited on 02/01/2019).

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* en. Texts in Theoretical Computer Science. An EATCS Series. Berlin Heidelberg: Springer-Verlag, 2004. ISBN: 978-3-540-20854-9. URL: `https://www.springer.com/gp/book/9783540208549` (visited on 02/06/2019).

[6] Armin Biere and Daniel Kröning. "SAT-Based Model Checking". en. In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 277–303. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_10`. URL: `https://doi.org/10.1007/978-3-319-10575-8_10` (visited on 02/07/2019).

[7] Daniel Bruns. "Formal Semantics for the Java Modeling Language". en. Dipl. KIT, June 2009.

[8] Patrice Chalin and Frédéric Rioux. "JML Runtime Assertion Checking: Improved Error Reporting and Efficiency Using Strong Validity". en. In: *FM 2008: Formal Methods*. Ed. by Jorge Cuellar, Tom Maibaum, and Kaisa Sere. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 246–261. ISBN: 978-3-540-68237-0.

[9] Yoonsik Cheon and Gary T Leavens. "A Runtime Assertion Checker for the Java Modeling Language (JML)". en. In: (Apr. 2004), p. 10.

[10] "Introduction to Model Checking". en. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8`. URL: `http://link.springer.com/10.1007/978-3-319-10575-8` (visited on 02/07/2019).

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-24730-2.

[12] David R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". en. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5.

[13] Lucas Cordeiro et al. "JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode". en. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 183–190. ISBN: 978-3-319-96145-3.

[14] David Harel, Dexter Kozen, and Jerzy Tiuryn. "Dynamic Logic". In: *Handbook of philosophical logic*. Springer, 2001, pp. 99–217.

[15] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: `10.1145/363235.363259`. URL: `http://doi.acm.org/10.1145/363235.363259` (visited on 02/07/2019).

[16] Temesghen Kahsai et al. "JayHorn: A Framework for Verifying Java programs". en. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 352–358. ISBN: 978-3-319-41528-4.

[17] Nikolai Kosmatov and Julien Signoles. "Runtime Assertion Checking and Its Combinations with Static and Dynamic Analyses". en. In: *Tests and Proofs*. Ed. by Martina Seidl and Nikolai Tillmann. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 165–168. ISBN: 978-3-319-09099-3.

[18] Charlie Lai and Sun Microsystems. "Java insecurity: Accounting for subtleties that can compromise code". In: *IEEE Software* (2008).

[19] Gary T Leavens and Yoonsik Cheon. "Design by Contract with JML". en. In: (2006), p. 12.

[20] Gary T Leavens et al. "How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification". en. In: (), p. 27.

[21] Gary T Leavens et al. *JML Reference Manual*. 2008.

[22] M. Nagel, M. Taghdiri, and T. Liu. "Bounded Program Verification Using an SMT Solver: A Case Study". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation(ICST)*. 2012, pp. 101–110. ISBN: 978-0-7695-4670-4. DOI: `10.1109/ICST.2012.90`. URL: `doi.ieeecomputersociety.org/10.1109/ICST.2012.90` (visited on 02/06/2019).

[23] David von Oheimb. "Hoare logic for Java in Isabelle/HOL". en. In: *Concurrency and Computation: Practice and Experience* 13.13 (Nov. 2001), pp. 1173–1214. ISSN: 1532-0626, 1532-0634. DOI: `10.1002/cpe.598`. URL: `http://doi.wiley.com/10.1002/cpe.598` (visited on 02/07/2019).

[24] Wolfram Pfeiffer. "Specifying and Verifying Real-World Java Cod with Key - Case Study java.math.BigInteger". en. Bachelor. KIT, May 2017.

[25] Amir Pnueli and Elad Shahar. "A platform for combining deductive with algorithmic verification". en. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas A. Henzinger. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 184–195. ISBN: 978-3-540-68599-9.

[26] Natarajan Shankar. "Combining Model Checking and Deduction". en. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 651–684. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_20`. URL: `https://doi.org/10.1007/978-3-319-10575-8_20` (visited on 02/06/2019).

A Appendix

A.1 Formal Rules

m	methodT(m)
	M RT method(P) **throws** T {
	try {
	$[assumeT(R)_c]$
	$assume([assumeT(R)_v])$
	} **catch** (Exception e) {
	throw **new** SpecException();
	}
/@ requires R;*	$[saveOld(E,B)]$
@ ensures E;	RT resultVar;
@ assignable A;	**try** {
@/*	$[bodyT(B,P,A)]$
M RT method(P) **throws** T{	} **catch** (t ∈ T) {}
B	**catch**(ReturnExc e) {
}	**try** {
	$[assertT(E)_c]$
	assert($[assertT(E)_v]$)
	return resultVar;
	} **catch** (Exception e) {
	throw **new** SpecException();
	}
	}
	}

Table A.1: Rule to translate a method contract

m	methodCallT(m)

/*@ *requires R;*
  @ *ensures E;*
  @ *assignable A;*
  @*/
M RT meth(P) **throws** T {
  B
}
	M RT methC(P) **throws** T {
	
	  try {
	    $[assertT(R)_c]$
	    $assert([assertT(R)_v])$
	  } **catch** (Exception e) {
	    throw new SpecException();
	  }
	  RT resultVar;
	  $[saveOld(E, B)]$
	  $havoc(A)$
	  try {
	    $[assumeT(E)_c]$
	    assume($[assumeT(E)_v]$)
	    return resultVar;
	  } **catch** (Exception e) {
	    throw new SpecException();
	  }
	}

Table A.2: Rule to translate a method that represents the effects of its contract

a	$assumeT(a)_c$	$assumeT(a)_v$
expr1 ∘ expr2 $\circ \in \{\&\&, \|\|, +, -, *, /, \%. ==, != \}$	$[assumeT(expr1)_c]$ $[assumeT(expr2)_c]$	$[assumeT(expr1)_v] \circ$ $[assumeT(expr2)_v]$
∘expr $\circ \in \{!\}$	$[assumeT(expr)_c]$	$\circ[assumeT(expr)_v]$
expr1 <==> expr2	$[assumeT(expr1)_c]$ $[assumeT(expr2)_c]$	$[assumeT(expr1)_v] ==$ $[assumeT(expr2)_v]$
expr1 ==> expr2	$[assumeT(expr1)_c]$ $[assumeT(expr2)_c]$	$![assumeT(expr1)_v]$ $\|\| [assumeT(expr2)_v]$

(\forall **int** i;
  i >= min &&
  i <= max &&
  cond;
  expr)
	freshBool(**true**, b);
	for(**int** i = min;
	  i <= max;
	  ++i) {
	  if(cond) {
	    $[assumeT(expr)_c]$
	    b = b && $assumeT(expr)_v$;
	  }
	}
		b

(\exists **int** i;
  i >= min &&
  i <= max &&
  cond;
  expr)
	int i = nondetInt();
	assume(i >= min &&
	    i <= max);
	$[assumeT(expr)_c]$
		expr

\old(expr)	$[assumeT(expr)_c]$	*getOld(* $[assumeT(expr)_v]$)
array[expr]	$[assumeT(expr)_c]$	array[$[assumeT(expr)_v]$]
literal		literal
variable		variable

Table A.3: Rule to translate expressions for assume-statements

a	$assertT(a)_c$	$assertT(a)_v$
`expr1 ∘ expr2` $\circ \in \{\&\&, \Vert, +, -, *, /, \%. ==, != \}$	$[assertT(expr1)_c]$ $[assertT(expr2)_c]$	$[assertT(expr1)_v] \circ$ $[assertT(expr2)_v]$
`∘expr` $\circ \in \{!\}$	$[assertT(expr)_c]$	$\circ[assertT(expr)_v]$
`expr1 <==> expr2`	$[assertT(expr1)_c]$ $[assertT(expr2)_c]$	$[assertT(expr1)_v] ==$ $[assertT(expr2)_v]$
`expr1 ==> expr2`	$[assertT(expr1)_c]$ $[assertT(expr2)_c]$	$![assertT(expr1)_v]$ $\Vert\, [assertT(expr2)_v]$

`(\forall int i;`
`  i >= min &&`
`  i <= max &&`
`  cond;`
`  expr)`
	`int i = nondetInt();`
	$[assertT(expr)_c]$
		`!(i >= min &&`
		`i <= max) ||`
		`expr`

`(\exists int i;`
`  i >= min &&`
`  i <= max &&`
`  cond;`
`  expr)`
	`freshBool(false, b);`
	`for(int i = min;`
	`  i <= max;`
	`  ++i) {`
	`  if(cond) {`
	$\quad\quad [assertT(expr)_c]$
	`    b = b ||` $assertT(expr)_v$`;`
	`  }`
	`}`
		`b`

`\old(expr)`	$[assertT(expr)_c]$	$getOld($ $[assertT(expr)_v])$
`array[expr]`	$[assertT(expr)_c]$	`array[` $\quad [assertT(expr)_v]$`]`
`literal`		`literal`
`variable`		`variable`

Table A.4: Rule to translate expressions for assert-statements

b	$bodyT(b)_c$	$bodyT(b)_v$

return expr;	$[bodyT(expr)_c]$
	resultVar = $[bodyT(expr)_v]$;
	throw new ReturnExc();	

method(P);	$assertAssignables(A1, A2)$; methodC(P);	
lexpr ∘ rexpr $(\circ \in \{=, +=, -=, *=, /=\})$	assert($lexpr \in A$); $[bodyT(rexpr)_c]$	lexpr ∘ $[bodyT(rexpr)_v]$
∘expr $(\circ \in \{++, --\})$	assert($expr \in A$); $[bodyT(expr)_c]$	$\circ[bodyT(expr)_v]$
expr∘ $(\circ \in \{++, --\})$	assert($expr \in A$); $[bodyT(expr)_c]$	$[bodyT(expr)_v]\circ$
expr;	$[bodyT(expr)_c]$ $[bodyT(thenStmt)_v]$;	

try stmt
catch (ET e) cstmt
	try {
	  $[bodyT(stmt)_c]$
	} **catch** (ReturnExc e) {
	  throw e;
	} **catch** (ET e) {
	  $[bodyT(cstmt)_c]$
	}	

if(c) thenStmt
else otherStmt
	$[bodyT(c)_c]$
	if($[bodyT(c)_v]$) {
	  $[bodyT(thenStmt)_c]$
	}
	else {
	  $[bodyT(otherStmt)_c]$
	}	

Table A.5: Rule to translate the statements of a method body (Part1)

b	$bodyT(b)_c$	$bodyT(b)_v$
	Type v = i;	
	int oldD = D;	
	assert(I);	
/@ loop_invariant I;*	$havoc(v, A)$	
@ loop_modifies A;	assume(I);	
@ decreases D;	**if**(c) {	
@/*	$[bodyT(body)_c]$	
for(Type v = i; c; inc) {	inc	
body	assert(I);	
}	assert(D >= 0 &&	
	D < oldD);	
	assume(**false**);	
	}	
{	{	
stmt1;	$[bodyT(stmt1)_c]$	
stmt2;	$[bodyT(stmt2)_c]$	
stmt3;	$[bodyT(stmt3)_c]$	
...	...	
}	}	

Table A.6: Rule to translate the statements of a method body (Part2)

A.2 Case Studies

Listing A.1: Case Study BubbleSort

```
package CaseStudy;

import TestAnnotations.Unwind;
import TestAnnotations.Verifyable;

/**
 * Created by jklamroth on 10/26/18.
 */
class BubbleSort {
    /*@
      @ requires arr != null && arr.length <= 5;
      @ ensures (\forall int v; v >= 0 && v <= \result.length - 1; (\
         forall int w; w >= 0 && w <= v - 1; \result[v] >= \result[w
         ]));
      @ assignable arr[*];
      @*/
    @Verifyable
    @Unwind(number = 7)
    static int[] sort(int arr[]) {
        for(int j = arr.length - 1; j >= 0; --j) {
            for (int i = 0; i < j; ++i) {
                if (arr[i] > arr[i + 1]) {
                    swap(arr, i, i + 1);
                }
            }
        }
        return arr;
    }

    /*@
      @ requires array != null && array.length >= 2;
      @ requires first < array.length && first >= 0;
      @ requires second < array.length && second >= 0;
      @ requires first != second;
      @ ensures \old(array[first]) == array[second] && \old(array[
         second]) == array[first];
      @ assignable array[first], array[second];
      @*/
    @Verifyable
    static void swap(int array[], int first, int second) {
        array[first] = array[first] ^ array[second];
        array[second] = array[second] ^ array[first];
        array[first] = array[first] ^ array[second];
    }
}
```

Listing A.2: Case Study DualPivotQuicksort

```java
package CaseStudy;

import TestAnnotations.Unwind;
import TestAnnotations.Verifyable;

class DualPivotQuicksort {

    static int less, great;
    static int e1,e2,e3,e4,e5;

    /*@
      @ requires a != null && a.length <= 5;
      @ requires 0 <= left && right < a.length;
      @ requires left == less && less < great && great < a.length;
      @ requires (\exists int j; less+1 <= j && j < great; a[j] >= pivot1);
      @ ensures less < great;
      @ ensures (\forall int i; left < i && i < less; a[i] < pivot1);
      @ ensures a[less] >= pivot1;
      @ ensures \old(less) < less;
      @ assignable less;
      @*/
    @Verifyable
    @Unwind(number = 7)
    static void  move_less_right(int[] a, int left, int right, int pivot1) {
        /*@
          @ loop_invariant
          @     0 <= less && less <= great && great < a.length
          @  && (\forall int i; left < i && i < less+1; a[i] < pivot1)
          @  && (\exists int j; less+1 <= j && j < great; a[j] >= pivot1)
          @  && \old(less) <= less;
          @  loop_modifies less;
          @  decreases great - less;
          @*/
        while (a[++less] < pivot1) {
        }
    }

    /*@
      @ requires a != null && a.length <= 5;
      @ requires 0 <= left && left <= less && less < great && great == right && right < a.length;
      @ requires (\exists int i; less <= i && i < great; a[i] <= pivot2);
      @ ensures less <= great;
      @ ensures (\forall int i; great < i && i < right; a[i] > pivot2);
      @ ensures a[great] <= pivot2;
      @ ensures great < \old(great);
      @ assignable great;
      @*/
    @Verifyable
    @Unwind(number = 7)
    static void move_great_left(int[] a, int left, int right, int pivot2) {
      /*@
        @ loop_invariant great > 0;
        @ loop_invariant left <= great && great <= right;
        @ loop_invariant less <= great;
        @ loop_invariant (\forall int i; great-1 < i && i < right; a[i] > pivot2);
        @ loop_invariant (\exists int j; less <= j && j < great; a[j] <= pivot2);
        @ decreases great;
        @ loop_modifies great;
        @*/
        while (a[--great] > pivot2) {
        }
    }

    /*@
      @ requires a != null && a.length <= 5 && left >= 0;
      @ requires 0 <= k && k <= great && great < a.length;
      @ requires (\exists int i; left <= i && i <= great; a[i] <= pivot2);
      @ ensures 0 <= great;
      @ ensures (\forall int i; great < i && i <= \old(great); a[i] > pivot2);
      @ ensures a[great] <= pivot2 || great == k;
      @ ensures k <= great && great <= \old(great);
      @ assignable great;
      @*/
    @Verifyable
    @Unwind(number = 7)
    static void move_great_left_in_loop(int[] a, int k, int left, int right, int pivot2) {
        /*@
          @ loop_invariant
          @     k <= great && 0 <= great
          @  && (\exists int i; left <= i && i <= great; a[i] <= pivot2)
          @  && (\forall int j; great < j && j <= \old(great); a[j] > pivot2)
          @  && great <= \old(great);
          @ decreases great;
          @ loop_modifies great;
          @*/
        while (a[great] > pivot2 && great != k) {
            --great;
        }
    }

    /*@
```

```
      @ requires 0 <= left && left < right && right - left >= 46 &&
         left <= 10000 && right <= 10000;
      @ ensures left < e1 && e1 < e2 && e2 < e3 && e3 < e4 && e4 < e5
          && e5 < right;
      @ assignable e1,e2,e3,e4,e5;
      @*/
    @Verifyable
    static void calcE(int left, int right) {
        int length = right - left + 1;
        int seventh = (length >> 3) + (length >> 6);
        seventh++;
        e3 = (left + right) >>> 1; // The midpoint
        e2 = e3 - seventh;
        e1 = e2 - seventh;
        e4 = e3 + seventh;
        e5 = e4 + seventh;
    }
}
```

Listing A.3: Case Study Hamming Weight

```java
package CaseStudy;

import TestAnnotations.Unwind;
import TestAnnotations.Verifyable;

class HammingWeight {

    /*@  requires a != null;
      @  requires a.length <= 5;
      @  ensures \result <= a.length * 32;
      @  assignable \nothing;
      @*/
    @Verifyable
    @Unwind(number = 6)
    int weight(int[] a) {
        int result = 0;
        //@ loop_invariant result <= i * 32;
        //@ loop_invariant i >= 0 && i <= a.length;
        //@ loop_modifies result;
        for(int i = 0; i < a.length; i++) {
            int x = a[i];
            x = x - ((x >>> 1) & 0x55555555);
            x = (x & 0x33333333) + ((x >>> 2) & 0x33333333);
            x = (x + (x >>> 4)) & 0x0f0f0f0f;
            x = x + (x >>> 8);
            x = x + (x >>> 16);
            result += (x & 0x3f);
        }
        return result;
    }

    /*@  requires a != null;
      @  requires a.length <= 5;
      @  ensures \result <= a.length * 32;
      @  assignable \nothing;
      @*/
    @Verifyable
    @Unwind(number = 6)
    int weight3(int[] a) {
        int result = 0;
        for(int i = 0; i < a.length; i++) {
            int x = a[i];
            x = x - ((x >>> 1) & 0x55555555);
            x = (x & 0x33333333) + ((x >>> 2) & 0x33333333);
            x = (x + (x >>> 4)) & 0x0f0f0f0f;
            x = x + (x >>> 8);
            x = x + (x >>> 16);
            result += (x & 0x3f);
        }
        return result;
    }

    /*@  requires a != null;
      @  requires a.length <= 5;
      @  ensures \result <= a.length * 32;
      @  assignable \nothing;
      @*/
    @Verifyable
    @Unwind(number = 33)
    int weight2(int[] a) {
        int result = 0;
        //@ loop_invariant result <= i * 32;
        //@ loop_invariant i >= 0 && i <= a.length;
        //@ loop_modifies 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 A.4: Case Study Multiplication of Nondeterministic Values

```
package CaseStudy;

import TestAnnotations.Unwind;
import TestAnnotations.Verifyable;

/**
 * Created by jklamroth on 2/1/19.
 */
public class MultExample {
    /*@
      @ requires x1 >= 0 && x2 >= 0;
      @ requires x1 < 256 && x2 < 256;
      @ ensures \result == x1 * x2;
      @ assignable \nothing;
      @*/
    public static int mult(int x1, int x2) {
        int res = 0;
        //@ loop_invariant res == i * x2;
        //@ loop_invariant i >= 0 && i <= x1;
        //@ loop_modifies res;
        //@ decreases (x1 - i) + 1;
        for(int i = 0; i < x1; ++i) {
            res += x2;
        }
        return res;
    }

    //@ requires x1 < 256 &&  x2 < 256;
    //@ requires x2 >= 0&& x1 >= 0;
    //@ requires res == x1 * x2;
    //@ ensures res + x2 == (x1 + 1) * x2;
    @Verifyable
    public void test(int x1, int x2, int res) {
    }
}
```

Listing A.5: Case Study PairInsertionSort

```java
package CaseStudy;

import TestAnnotations.Unwind;
import TestAnnotations.Verifyable;

/**
 * This Pair Insertion Sort in which two elements are handled at a time
 * is used by Oracle's implementation of the Java Development Kit (JDK)
 * for sorting primitive values, where a is the array to be sorted, and
 * the integer variables left and right are valid indices into a that
 * set the range to be sorted.
 * This was the first challenge from the VerifyThis competition @ ETAPS 2017
 * organized by M. Huisman, R. Monahan, P. Mueller, W. Mostowski,
 * and M. Ulbrich.
 * The specification considers only sortedness, the permutation property
 * is yet to be done.
 * @author Michael Kirsten <kirsten@kit.edu>
 */
public class PairInsertionSort {

    /*@ public normal_behaviour
      @ requires a != null && a.length < 5;
      @ requires 0 < left && left <= right && right < a.length;
      @ //requires right - left + 1 < 47;
      @ requires (\forall int i; left - 1 <= i && i <= right; a[left - 1] <= a[i]);
      @ assignable a[left..right];
      @ ensures (\forall int i; \old(left) - 1 <= i && i < \old(right); a[i] <= a[i + 1]);
      @*/
    @Unwind(number = 7)
    @Verifyable
    public static void sort(int[] a, int left, int right) {


        for (int k = left; ++left <= right; k = ++left) {
            int a1 = a[k];
            int a2 = a[left];

            if (a1 < a2) {
                a2 = a1;
                a1 = a[left];
            }

            while (a1 < a[--k]) {
                a[k + 2] = a[k];
            }
```

```
                a[++k + 1] = a1;
                while (a2 < a[--k]) {
                    a[k + 1] = a[k];
                }

                a[k + 1] = a2;
            }
            int last = a[right];
            while (last < a[--right]) {
                a[right + 1] = a[right];
            }

            a[right + 1] = last;
        }
}
```

Listing A.6: Case Study BitInteger toUnsinged

```java
package CaseStudy;

import TestAnnotations.Verifyable;

/**
 * Created by jklamroth on 12/18/18.
 */
public class BigInt {
    int[] result;

    final static long LONG_MASK = 0xffffffffL;

    // returns the value of the input integer as if it was unsigned
    /*@ ensures value == 0 ==> \result == 0;
    @ ensures value != 0 ==> \result > 0;
      @ ensures value > 0 ==> \result == value;
      @ ensures value < 0 ==> \result == value + 0x100000000L;
      @ ensures \result >= 0;
      @ ensures \result < 0x100000000L;
      @*/
    @Verifyable
    public static long toUnsigned(int value) {
        return (long)value & 0xffffffffL;
    }
}
```