Karlsruhe Institute of Technology

Specifying and Verifying Real-World Java Code with KeY – Case Study java.math.BigInteger

Bachelor Thesis of

Wolfram Pfeifer

At the Department of Informatics
Institute of Theoretical Informatics

Responsible Advisor: Prof. Dr. Bernhard Beckert
Advisors: Dr. rer. nat. Mattias Ulbrich
Dipl.-Inform. Sarah Grebing

Submission Date: 5th May 2017

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, May 5, 2017

Abstract

The KeY system has been developed to verify Java methods and classes. However, very little examples exist where real-world Java code of more than a few lines is verified. This thesis takes a step to close that gap. We specify real-world Java code and afterwards verify the contracts with the KeY system. For that we take `java.math.BigInteger` from the Java Class Library. The purpose of this class is to provide arbitrary precision integers which behave similarly to Java's primitive type integers. Many security applications as well as Java's security packages make heavy use of BigInteger[1], as algorithms of public-key cryptography like RSA need very large primes to be considered as secure.

Deutsche Zusammenfassung

Das KeY-System wurde entwickelt, um Java-Methoden und Klassen zu verifizieren. Allerdings gibt es nur sehr wenige Beispiele, in denen Java-Code aus dem Praxiseinsatz mit mehr als ein paar Zeilen verifiziert wird. Das ist der Ansatzpunkt dieser Thesis. Wir spezifizieren „real-world“ Java-Code und verifizieren die Verträge anschließend mit dem KeY-System. Der Code stammt dabei aus der Klasse `java.math.BigInteger` der Java Class Library. Zweck dieser Klasse ist es, beliebig große Ganzzahlen darzustellen, die sich ähnlich wie Zahlen vom Datentyp `int` in Java verhalten. Viele Sicherheitsanwendungen sowie Java's Sicherheits-Packages nutzen die Klasse BigInteger[1], da Public-Key-Algorithmen wie RSA sehr große Primzahlen benötigen, um als sicher zu gelten.

[1] `https://docs.oracle.com/javase/7/docs/api/java/math/class-use/BigInteger.html`

Contents

1. Introduction

1.1. Motivation

Formal verification of software is a growing field of research. This may be associated with general increasing needs of privacy and security, which displays for example in the increasing amount of website and email encryption, but on the other hand with threatening scenarios arising from software faults in advanced technologies, like for example self-driving cars or planes. Furthermore, recent discoveries of implementation flaws like the Heartbleed bug as well as leaks of observation techniques (for example Vault 7) increase the awareness of security.

One of the tools that have been developed for formal verification is the KeY system[1]. It is a tool developed since 1998, mainly to verify Java source code, and since then has been greatly improved. However, the existing examples of verification usually consist of relatively little lines of code and are often written with verification in mind. Since the long-term objective of formal verification is to reason about real-world software, it is important to know how a verification tool measured in that. To provide a case study and to examine how KeY is able to deal with real-world code are the main goals of this thesis.

1.2. Approach

To test the capabilities of KeY, we take the class `java.math.BigInteger` from the Java Class Library. This class provides arbitrary precision integers which behave similarly to Java's primitive type integers. Since Java does not support overloading of operators for object types, the class contains public methods to provide the desired behavior, for example `add`, `subtract`, `mutliply`, and many more. As the BigInteger class consist of more than 4400 lines of code (about 1650 logical executable lines), we have to limit our verification approach to a little excerpt. For that, we take the public `add()` method and the methods called from there. As a specification language we use the Java Modeling Language (JML), which enables us to perform modular specification. This means, we do not have to describe the functionality of the whole source code at once, but instead are able to specify and afterwards verify each method separately. Because the public `add()` method relies on many other private methods, we take a bottom up approach and verify the most basic and short methods first before getting to the larger and more high level ones. This also has the advantage that we can rely on already proved contracts for the calling methods.

[1] `https://www.key-project.org/`

As testing KeY with real-world code is our main objective, we will alter the source code as little as possible. The only simplification[2] put into effect is limiting our scope to the code actually used, which is in the sense of modularity. Obviously, methods and constructors never called from the code we consider are not influencing our verification in any way.

1.3. Related Work

From the developers of the JML standard there is already a specification of the BigInteger class[3]. However, this is not a full functional specification, but just (mostly trivial) constraints the public methods. Therefore, it is of no use for our thesis.

In chapters 18 of [Ahrendt et al., 2016] an electronic voting system is specified and verified. Their approach using KeY and JML closely relates this work to ours. However, they go beyond functional verification and additionally conduct an information flow proof. This shows additional possibilities with the KeY system not covered by this thesis.

In chapter 19 of [Ahrendt et al., 2016], functional verification of a sorting algorithm is presented. Their conclusion states that the biggest amount of rules applied comes from reasoning about different heap states. This correlates with our results later on.

In [Huisman et al., 2001] another approach in verifying code from the Java Class Library is presented. The tools used in this work are the LOOP compiler and the Prototype Verification System (PVS), a higher-order theorem prover. However, no full functional verification is done, but it is only reasoned about the class invariant of Java's Vector class.

Full functional verification is realized in [Schmitt and Tonin, 2007]. In this work about 180 lines of JML are verified in KeY, while the source code consists of more than 300 lines of JavaCard code, which is a lighter version of Java.

All theses works have in common that JML is used as specification language.

1.4. Contributions

In the context of this thesis the following contributions are made:

- We will present a case study which tests the capabilities of KeY. By discovering bugs and usability weaknesses of KeY, it will support the development process.
- It will be shown that some contract parts which are impossible to verify in KeY can be translated to a bounded model checker like CBMC and afterwards shown with it very easily. Even an automatic translation for selected goals is conceivable. This fruitful interplay of the two tools may result in a future integration of a bounded model checker in KeY.
- In the context of JML we will show that JML's approach of assuming the invariant to hold after a RuntimeException or even Error in a constructor is questionable, as those are not supposed to be caught. They instead indicate a serious implementation or execution problem, which makes it impossible to create a valid instance of the class.
- Finally, we provide a general modeling of BigInteger's semantics. This enables us to specify and verify several methods of the class and provide the foundations for future proofs of others.

[2] As KeY is not able to deal with generic types, we also have to replace them in an appropriate way. However, since we use exactly the same way the Java compiler handles them, this is more an equivalent transformation than a simplification. The exact way how it is done can be found in Section 3.2.

[3] `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs/java/math/BigInteger.html`

1.5. Outline

Apart from this introduction, the thesis is composed of the following parts:

- Chapter 2 contains the preliminaries, which includes introductions to the specification language JML as well as to the KeY system, which we mainly use in this thesis. In addition, short notes on using CBMC for Java are given, as this tool is used for one single method contract.
- Chapter 3 describes how specification and verification of the methods is done. This includes an explanation of the general concepts of BigInteger, how they are modeled in JML, and method contracts for the called library methods outside the BigInteger class. After that, we continue with more technical sections: Necessary code adaptations of the BigInteger code for KeY, user-defined inference rules, and settings of KeY, before finally the actual method contracts for BigInteger methods an their proofs are described.
- Chapter 4 provides a summary of the verification with statistics and experience report, thoughts on future work, and a conclusion.
- The appendix contains the complete source code and specification used for this thesis as well as a list of the settings of KeY.

2. Preliminaries

In this chapter we provide a quick introduction to the specification language and the tools we will use for verification. This includes the Java Modeling Language (JML), the deductive verification system KeY and the bounded model checker CBMC.

2.1. A Brief Introduction to JML

Before we start verifying the code, we have to know against what we want to verify and thus write a formal specification. We derive this specification from the JavaDoc and comments attached to the code as well as from the code itself. As a language to write this formal specification we use the Java Modelling Language (JML), which brings all features required to specify the functional as well as non-functional aspects of our code. In this section we will give a brief introduction to JML. Because JML contains far more features (e.g. for specification of concurrent programs) than we need, we will limit this introduction to features actually needed for our work. Further information about JML can be found at [Leavens et al., 2013].

The behavior of a class or method can be specified with JML annotations directly in the Java source code file (.java). JML annotations are started using the Java comment beginning `//` or `/*` directly followed by an at-sign (`@`). Thus for the Java compiler the code is still unchanged and special tools like KeY can extract the JML specification from the same file. We have to mention that additional `@` symbols at the beginning of a line or before the closing `*/` may be inserted for better readability. Furthermore, it is possible to again use Java comments inside a JML annotation, for example to describe the specification directly in place.

2.1.1. Method Contracts

JML takes the approach of modular specification. This means, the code is split up into parts which are specified independently and can be verified in the same way. At different levels those parts are interfaces, classes and methods.

Our goal is to describe the behavior of methods of `java.math.BigInteger`, which is done via contracts. Those contracts have to state what the method is expected to do presuming the specified preconditions hold when the method is called. We can go even further and constrain what the method may do, for example which places it may access and which not.

2.1.1.1. Normal Termination

Every method contract contains three parts: a precondition, a postcondition, and a framing clause.

The precondition has to be guaranteed by the caller of the method, so that the method can rely on it. In return the method has to ensure that after its termination the postcondition holds. The `assignable` clause states which parts of the heap may be changed after the method's termination. All other locations are assumed to be unchanged. This deals with the so called frame problem [Borgida et al., 1993].

We take a look at a simple contract example:

```
1 /*@ normal_behavior
2   @ requires y != 0;
3   @ ensures \result == x/y;
4   @ assignable \nothing;
5   @*/
6 public static int divide(int x, int y) {
7     return x/y;
8 }
```

Listing 2.1: A very simple method contract

At the first line we start a new method contract with normal termination. This means, the method is expected not to throw an exception. We write our precondition after the `requires` clause. In this case, `y` must not be 0, because otherwise the result of the division would be undefined. If that precondition holds, our method has to ensure that the return value of the method, noted by `\result`, is `x` divided by `y` with integer division. It is to mention that all object parameters, which includes arrays, have an additional implicit precondition: They are non-null by default. If we want to allow null values, we have to explicitly prepend `/*@ nullable */` to the parameter.

In case one of this three clauses is missing, the defaults apply for that clause. Those are: `requires true`, `ensures true`, `assignable \everything`.

This little example shows that in general JML expressions can contain Java expressions as in `x!=0` or `x/y`. Precisely, every side effect free Java expression can be used. Side effect free means that the expression must not change the heap state, for example by changing member fields of an object. Apart from that there are several JML operators which are not part of Java: The most common ones are equivalence `<==>` and implication `==>`. Additionally we will make use of some quantifiers: `\forall` and `\exists`, which represent the well known quantifiers of predicate logics, and the `\sum` quantifier. In Listing 2.2 there is an example shown to demonstrate the syntax of those.

```
1 (\forall int i; 0 <= i && i < a.length; a[i] < 100;)
2 (\exists int i; 0 <= i && i < 100; 2*i == 100)
3 (\sum int i; 1 <= i && i < 100; i) == 100*101/2
```

Listing 2.2: Example use of quantifiers

Here we have to remark that the KeY system only deals with bounded sums of the mentioned form even if JML allows a much less restricted use of them, as for example unbounded sums. We will see however that this isn't an issue for our work.

If in the postcondition it is necessary to refer to the value of a field or parameter before the start of the method, the keyword `\old(...)` can be used.

Some methods may contain recursion. In such case it is necessary to ensure termination of the recursion. This is done via the `measured_by` keyword followed by a nonnegative expression strictly decreasing with recursion depth.

```
1 /*@ normal_behavior
2   @ requires ...
3   @ ensures ...
4   @ measured_by exp;
5   @*/
6 public static long power(int base, int exp) {
7     return exp == 0 ? 1 : base * power(base, exp-1);
8 }
```

Listing 2.3: Example of a method contract with recursion

2.1.1.2. Exceptional Termination

It may occur that instead of normal termination a method terminates throwing an exception. Therefore JML provides a possibility to specify exceptional termination as well. An example is given below.

```
1 /*@ exceptional_behavior
2   @ requires b == 0;
3   @ signals_only ArtihmeticException;
4   @ signals (ArithmeticException e) true;
5   @*/
6 public static int divide(int a, int b) {
7     throw new ArithmeticException("Division by 0!");
8 }
```

Listing 2.4: A very simple exceptional method contract

As we can see, we have a precondition here as well. In contrast to normal termination, there is no `ensures` clause. Instead, we have one (or more) `signals` clauses here. The above given means: If an `ArithmeticException` occurs, then the subsequent formula must hold. In our case, this formula is `"true"`, which always holds. So if the divisor is 0, our method must always throw an `ArithmeticException`. The additional `signals_only` clause means that no other kind of exception may occur.

2.1.1.3. Nontermination

Via the keyword `diverges` it is also possible to specify that a method may not terminate. Since it is not needed in this thesis, we won't give an example here.

2.1.1.4. Dependency Contracts

Sometimes it is useful to specify which locations the result of a method may depend on. Thus we may easily deduce whether two calls to the same method return the same value or not. The JML clause for this is `accessible` plus a location set. It is to mention that local values, method parameters and locations of newly created objects may always be accessed and therefore must not be added to the `accessible` clause. Besides, JML allows special location set expressions `\nothing` and `\everything`.

In our simple example in Listing 2.5 we see, that two calls to `roundedAvg()` give the same result, even if `arraySum()` is called in between. That is, because the result of `roundedAvg()` depends just on the elements of the array (short notation: `array[*]`) and not on `cache`, which is the only value potentially altered by `arraySum()`.

We have seen now different parts of method specifications. Even if it is possible to include all those parts into a one single method contract prepended by the keyword `behavior` or even no keyword, in this thesis we will write several independent contracts and connect them via the keyword `also`. This is a cleaner approach with much better readability.

```
1  int[] arr;
2  int cache;
3
4  /*@ normal_behavior
5    @ accessible arr[*];
6    @*/
7  public int roundedAvg() {
8      ...
9  }
10
11 /*@ normal_behavior
12   @ assignable cache;
13   @*/
14 public int arraySum() {
15     ...
16 }
```

Listing 2.5: Dependency contract in interplay with a framing clause

2.1.2. Specification-Only Class Members

JML offers additional elements for help in specification. For this thesis we will make use of model methods. Because they exist only in specification, they obviously must not have side effects. Apart from that they use a combination of JML and Java syntax as you would expect it.

In addition we use ghost fields in this thesis. Those are additional specification-only fields, which may abstract from the concrete program state or even add additional information to the specification. The value of ghost fields can be updated with a `set` statement.

```
1  int[] a;
2  //@ ghost \bigint sum;
3
4  /*@ public model \bigint calcSum() {
5    @     return (\sum int i; 0 <= i && i < a.length; a[i]);
6    @ }
7    @*/
8
9  public ArraySum(int[] a) {
10     this.a == a;
11     //@ set sum = calcSum();
12 }
```

Listing 2.6: Ghost field and model method

2.1.2.1. Data Types

While model and ghost elements may be of any Java type, JML provides an extension for arbitrary precision of integers and reals: the types `\bigint` and `\real`. In this thesis, we will make heavy use of the `\bigint` type. It acts like the well known primitive `int` type but has arbitrary precision, which means that overflows can not occur. So the `\bigint` type is able to represent mathematical integers. Apart from that, it may be used with comparison (==, >, <, ...) or math (+, -, *, ...) operators as expected.

2.1.3. Invariants

In computer science, invariants are formulas that have to hold before and after the execution of a specific block of code. They serve as a pre- and a postcondition to this block simultaneously. In JML, two different kinds of invariants exist: type invariants and loop invariants.

2.1.3.1. Type Invariants

Type or class invariants can be used to describe a domain concept (like for example nonnegativity of a member field), which has to be ensured in every (visible) program state. Thus it is not necessary to include this constraint to the pre- and postconditions of all methods. Instead it is enough to specify the invariant once and the methods implicitly have to preserve it.

An invariant can be either a static or an instance invariant. Static methods have to consider just the static invariants whereas instance methods and constructors have to take care of instance invariants as well. It is to note here that we may omit the `instance` keyword since it is the implicit default. There is a way to make a method or constructor independent of the invariants: The keyword `helper`. A method or constructor specified as such can not assume the invariants in precondition, but in return does not have to ensure them after termination.

```
1 /*@ invariant (\forall int i; 0 <= i && i < a.length; a[i] >= 0);
2   @ accessible \inv: a.*;
3   @*/
4 //@ invariant sum >= 0;
```

Listing 2.7: Example invariants for the ghost field example from above

One final note on type invariants: Their `accessible` clause has the special syntax shown in line 2 of Listing 2.7.

2.1.3.2. Loop Invariants

Loop invariants follow a different concept: They are not directly needed for specification, but instead provide a help for verification. As we can see in the example below, each loop specification consists of three parts: The actual invariant formula, a `decreases` clause and an `assignable` clause. The actual invariant has to hold before and after each loop iteration and even if the loop terminates, which may also be due to a `return` or a `break` instruction. The `decreases` clause, sometimes called loop variant, is used to ensure termination of the loop. The term after the `decreases` keyword has to be nonnegative and strictly decreasing with each loop iteration.

```
1  /*@ normal_behavior
2    @ ensures \result == (\forall int i;
3    @                           0 <= i && i < a.length-1;
4    @                           a[i] <= a[i+1]);
5    @ assignable \nothing;
6    @*/
7  public static boolean isSorted(int[] a) {
8      /*@ loop_invariant (\forall int j;
9        @                         0 <= j && j < i;
10       @                         a[j] <= a[j+1]);
11       @ decreases a.length - i;
12       @ assignable \nothing;
13       @*/
14     for (int i = 0; i < a.length - 2; i++) {
15         if (a[i] > a[i+1])
16             return false;
17     }
18     return true;
19 }
```

Listing 2.8: Example of a loop invariant as a help for contract verification

2.1.4. Conclusion

This overview enables us now to write the specification for the BigInteger methods considered in this thesis. A more detailed view on JML can be found at [Leavens et al., 2013] or with special regard to the KeY system at [Ahrendt et al., 2016, Chapter 7].

2.2. Introduction to the KeY System

The KeY system is an interactive theorem prover using a sequent calculus. In this section, we will give an overview over the logic used by KeY and the features of Java and JML it supports. After that we describe how KeY performs symbolic execution of Java instructions and explain the powerful concept of taclets, KeY's inference rules. We provide some short hints on data types and the graphical user interface and finally give a quick outlook on further features and possibilities of the KeY system.

2.2.1. JavaDL

For reasoning about the behavior of programs KeY uses Java Dynamic Logic (in short JavaDL). Formulas in JavaDL may use the modalities $[p]$ ("box") and $\langle p \rangle$ ("diamond"), where p is an arbitrary[1] sequence of Java statements. The meaning of these modalities is as follows: $\langle p \rangle \psi$ states that p terminates and after that ψ holds. $[p]\psi$ is slightly different: If p terminates, then ψ holds.

Of course, Java program code may only occur in the box and diamond modalities and not in ψ, which has to be a formula of first-order predicate logic. However, this is not just classical first-order logic, but an extension of it called Java first-order logic (JFOL). JFOL basically adds a type hierarchy with integers, booleans, Java object types, heaps, fields and locations sets as well as function and predicate symbols for theses types to the basic first-order logic.

We can see that a method contract, given that the method terminates, can be expressed as $\phi \rightarrow \langle p \rangle \psi$, where ϕ is the precondition, p the code in the method body, and ψ denotes the postcondition of the method. In fact this is the general concept how JML method contracts are translated and expressed in JavaDL. However, in practice there are more details to consider, for example recursion, framing, implicit pre- and postconditions, and exceptions. A detailed view on creating JavaDL proof obligations from JML method contracts can be found at chapter 8 of [Ahrendt et al., 2016].

2.2.2. Sequent Calculus

Logical reasoning in KeY is done via the sequent calculus. A sequent is of the form $\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$. Intuitively, this has the following meaning: If all the formulas $\phi_1, \ldots, \phi_n$ hold, we have to show that at least one of the formulas $\psi_1, \ldots, \psi_m$ holds to prove the sequent valid. The sequent calculus works by decomposing complex formulas into simpler ones by the application of inference rules. As an example a rule to decompose a conjunction on the right side of sequent arrow is shown:

$$\text{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta} \qquad \begin{array}{l}(\Gamma \text{ and } \Delta \text{ are sets of formulas,} \\ \phi \text{ and } \psi \text{ single formulas})\end{array}$$

As we can see, this rule splits the proof into two branches. Of course, most of the rules (for example for dealing with modalities) are more complex than this basic one. By applying

[1] In fact there are limitations: JavaDL for example does not support concurrency, floating-point types, and dynamic class loading (reflection). A list of supported features of Java can be found at `https://www.key-project.org/applications/program-verification`.

inference rules successively a proof tree is constructed. If all branches of the tree are closed (proved), the sequent started from is valid.

How a sequent looks in KeY can be seen at Figure 2.1. We can see there the sequent arrow at the left with the preconditions above (A) and the statement to show below. Also we can see the program fragment inside modality brackets \<{ and }\>. The statement after the modality is the postcondition to show (B).

Figure 2.1.: Example sequent from a proof in KeY

2.2.3. Symbolic Execution and Updates

As explained above, a proof obligation for a method contract in JavaDL looks like this: $\phi \rightarrow \langle p \rangle \psi$, where p contains the Java Code from the method body. It is possible to transform this statement about two states (before and after execution of p) into a statement about a single state, the state before execution of p. For this, KeY successively decomposes the program p into a smaller one until the modality is empty. This process is called *symbolic execution*.

Decomposition rules for program fragments always apply to the first (if we ignore block prefixes like "try ...") statement of a modality, which is called *active statement*. The active statement gets highlighted in KeY's graphical user interface (GUI) as shown in Figure 2.1, marked with C.

Updates represent state transitions as well but have two important differences: They are side-effect free, and they are always terminating. Again, Figure 2.1 shows an example of

possible updates, marked with D. After the program fragment is completely decomposed, all updates can be applied in parallel to the formula q, which is then a statement about the pre-state of the method.

2.2.4. Taclets

Inference rules in KeY (*taclets*) are not hard coded, but written in a special declarative language, called the *taclet language*. This makes the proof search much more flexible, as additional rules can be added if necessary by developers as well as users. To maintain soundness, such additional rules should be proved against the basic rules of KeY, though this is not enforced.

Taclets are basically typed first-order rules as described in Section 2.2.2 with additional information like heuristics and display information. With such, it can be controlled when the taclet is used by automatic proof search strategies and how it is presented to the user. Moreover, different taclet sets can be loaded for different purposes, such as proving functional contracts, information flow, reasoning in Hoare logic, etc.

As an example, Listing 2.9 shows the taclet `andRight` already shown in textbook notation above.

```
1 andRight {
2     \find (==> b & c)
3     \replacewith(==> b); \replacewith(==> c)
4     \heuristics(beta)
5 };
```

Listing 2.9: The taclet `andRight`

Note that the `\find` clause states where the rule can be applied, in this case to a conjunction on the right side of the sequent arrow.

KeY has more than 1500 built-in rules of different categories:

- *Nonprogram Rules* are used to handle statements about a single state without modalities.
- *Symbolic Execution Rules* transform a program inside a modality into updates and case distinctions.
- *Update Simplification Rules* are used for transforming sequential to parallel updates and discarding unnecessary ones.
- The last category are *rules for program abstraction and modularization*, which can not be encoded in the taclet language, but are hard coded. This includes for example rules dealing with loop invariants and method contracts.

As already mentioned above, in KeY it is not only possible to perform rule applications manually, but also run automatic proof search strategies. With theses strategies KeY is usually able to perform the symbolic execution and update simplification automated and prove simple branches. For more difficult branches, interaction is needed afterwards with nonprogram rules.

Here we have to note that the automatic strategies transform (in-)equations and formulas into normal forms, which may be very counterintuitive and difficult for the user to read. For the user it is often convenient to prune already done normalization steps and restart from the basic formulas of that branch.

In chapter 3 we will see many examples of successful automatic proof search as well as cases where it fails.

2.2.5. JML*: Extensions of JML in KeY

In Section 2.1 we provided an overview over the concepts and features of JML. While KeY supports these main concepts, some extensions and changes are made compared to the official JML standard. This JML dialect is called as JML*. Additions to the official JML standard which are important for our thesis are:

- It is possible to add an `assignable` clause to loop specifications. This greatly simplifies reasoning about loops, since it then is known, which heap locations they may change.
- The keyword `\strictly_nothing` in an `assignable` clause denotes that not only existing objects and fields are not changed, but in addition no new objects are created on the heap. In the same manner there is a modifier `\strictly_pure`, which is syntactic sugar for `assignable \strictly_nothing`.
- Visible state semantics of invariants: An invariant may be broken by a method inside method body, but has to be restored by the method and hold afterwards. This is less restrictive than the original JML semantics.
- With the keywords `\static_invariant_for(`*`classname`*`)` a method contract may refer to the static invariant of a class and assume it in precondition or ensures it in postcondition.

Similarly, some features of official JML are not supported by KeY. In general, KeY supports the JML language level 0 (see section 2.9 of [Leavens et al., 2013]) and some parts of levels 1 and 2, but not the levels 3, C (for concurrency), and X (experimental). For example, history constraints, math modifiers, and `assume` statements are not supported. The `\real` type can be parsed, but not reasoned with in KeY.

2.2.6. Supported Java Features

While the KeY system supports a wide range of Java features, there are limitations though. For example, KeY is not able to deal with floating point numbers. Other features of Java not integrated in KeY include multithreading and Java 8 lambdas. Generic types are not directly supported, but a workaround exists: Like the Java compiler does, we can replace an unbounded generic type by `java.lang.Object`. Since the class BigInteger inherits from the generic interface `Comparable`, we will see an example of this in Section 3.2. There is also an Eclipse tool from the developers of KeY for doing this automatically[2].

We have to remark that although syntactically supported in KeY, there are no rules for reasoning about bitwise arithmetic operators. This is due to a design decision: Many excellent tools for bitvectors exist, so why reinvent the wheel? In this thesis we will therefore make use of the bounded model checker CBMC.

2.2.7. Data Types

In the specification language JML all Java types are allowed, which includes objects as well as the primitive types `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`. In addition, the specification only types `\bigint` and `\real` are allowed. On the other hand JavaDL's type hierarchy includes the types integers, booleans, Java object types, heaps, fields and locations (compare to Section 2.2.1). This leads to the question how the JML types are translated to JavaDL. This is done as follows: All integer types from Java (`byte`, `short`, `int`, `long`, `char`) as well as the `\bigint` type are translated to JavaDL's integer

[2]More information about supported Java features and the tool to remove generic types can be found at `https://www.key-project.org/applications/program-verification/`

type. For this type three semantics are available: Ignoring overflows, which treats integers as mathematical (unbounded) integers, Java semantics, in which overflows are calculated as in Java, and integer semantics with overflow check, which checks if an overflow occurs. If a semantic with bounds is chosen, the types are correctly checked for their ranges with the predicates inInt(), inLong(), etc.

For `float` and `double` as well as the `\real` type, no JavaDL counterpart exists.

The other JML types, booleans and objects (which includes arrays), are translated to JavaDL as expected.

2.2.8. KeY GUI

The graphical user interface (GUI) of KeY is an important part for working with it. We will briefly describe the workflow for functional verification, which is what is needed later.

After start of KeY, the main window opens. For loading a contract to prove we click on File → Load and open our file. The Proof Management Window appears, which shows all contracts found in this file (and additionally, in all files of the same folder). We select the contract we want to proof. After that, the proof obligation is shown in the Current Goal window (D). In Figure 2.2 a screenshot of such a situation is shown. On the left side, we have a list of proofs we started (A), as well as an outline of the current proof tree (B). Other tabs provide a list of open goals, settings for the proof search, and information about the available rules, functions and terms.

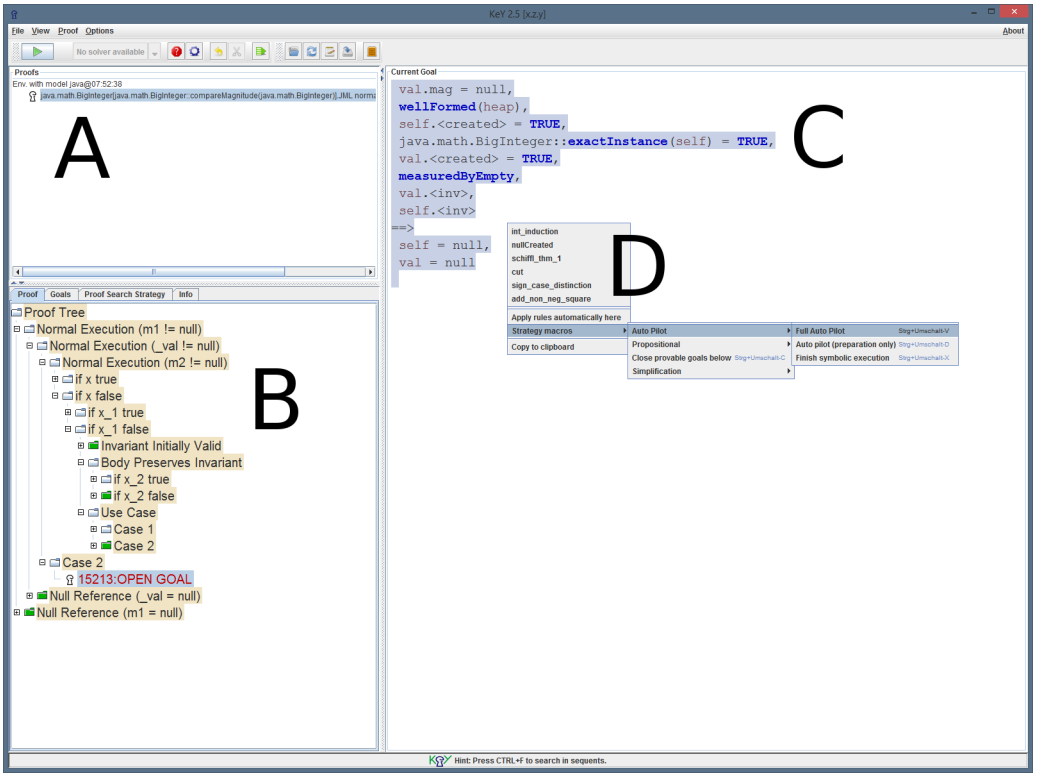

Figure 2.2.: A screenshot of KeY's GUI

To advance in proof, there are basically two possibilities: manual rule applications or automatic search. For manual application, we have to click on a formula or term in the current goal view. Depending on the selection, the available rules are shown in the context menu. Two other settings may influence the available rules: The taclet settings, and the One Step Simplification. The former are general settings for the proof which are well

explained in their settings window. The latter is an aggregation setting which combines different simplification steps into a single proof step. Advantages of it are that the user does not have to do all (often trivial and very numerous) simplification steps separately. Also it speeds up the automatic proof search.

In the context menu (D) in addition to single rules some strategy macros show up. These apply specific rule sets (rules of specific heuristics) until their purpose is served. Finish Symbolic execution, for example, tries to decompose all program fragments. The Close provable goals closes all branches, that are provable automatically (within the number of steps specified in Strategy Settings).

In addition, we provide some tips which can make proof search much easier:

- To get an overview over the proof tree the option hide intermediate proof steps can be selected from the context menu and afterwards expand goals only. That way the user can get an idea, which parts of the proof are difficult or which parts of the specification (for example loop invariants) require refinement.
- The option "View" → "Use pretty syntax" really makes the formulas better human readable by using shortcuts for accessing fields and objects and for displaying numbers. While this option should be activated by default, there is an additional one to display unicode symbols. This makes KeY use the well known mathematical symbols for conjunction ($\wedge$), quantifiers ($\forall, \exists$), and other operators.
- Formulas and terms can be dragged via the mouse and dropped to another one. After releasing the mouse button, the available taclets for combining them are shown. Drag & Drop can also be used if a taclet application opens a new window because instantiation of a variable is required.
- The context menu of the proof tree allows pruning of branches, for example to get rid of already performed normalization steps as explained above.
- The Full Auto Pilot macro structures the proof better than it is done by just starting proof search with the green button. In general, just applying the strategy may lead often to many erratic rule applications.

2.2.9. Other Frontends and Use Cases of KeY

Instead of the graphical user interface described in the previous section, KeY can be used via its command line interface. As it is not possible to apply rules interactively, this is a very limited feature. Approaches are made to include so called *proof scripts*, which describe the steps (single rules or macros), that shall be applied. However, currently for every rule the formula to apply it on has to be written completely, which usually results in very large and confusing scripts.

For teaching, there is KeY Hoare, a version of KeY focusing on Hoare logic. Another relatively new development is the *Symbolic Execution Debugger* (SED), which is able to visualize the execution paths through a program. The SED does not use the KeY GUI, but is an extension to the IDE Eclipse. There are other extensions for Eclipse like KeY 4 Eclipse Starter, which gives the ability to start the KeY GUI directly from Eclipse, or KeYIDE, an alternative user interface of KeY in Eclipse.

Other interesting uses of KeY are counterexample and unit test generation as well as information flow verification and verifying correct compilation.

2.3. Short Remarks on Using CBMC for Java

CBMC stands for C Bounded Model Checker. A bounded model checker is a tool that approaches the model checking problem by successively unwinding the state transition relation a program represents. However, since the state space grows exponentially, the bounded model checker only unrolls the state machine up to a given bound and encodes it into a Boolean formula. Then it runs a SAT solver on it and tries to find a counterexample. If such a counterexample is found, the model is proved valid. If not, the search may be restarted with a greater bound. Of course, this approach is limited if iteration of any kind is used, since the number of possible states and thus the Boolean formula grows exponentially with the bound.

While the encoding even of a smaller program may already result in a large Boolean formula (typically with many thousands of variables), normally the solution is found very fast due to the excellent heuristics of today's SAT-solvers. Although CBMC is a tool initially designed for the C language, the developers recently added support for Java Bytecode. It is still marked as experimental, but for our very simple use case it is completely enough. How we translate our JML contract into assertions such that CBMC can handle it will be explained in Section 3.3.2.1.

Because we just use a very small subset of CBMC's capabilities, we do not provide further explanations here and instead refer to the CProver Manual[3].

[3] `http://www.cprover.org/cprover-manual/`

3. Code Specification and Verification

In this chapter we will do the actual specification and verification of the selected methods. At first we will explain the general concepts of the class `java.math.BigInteger`. Afterwards we will provide an overview over our abstract modeling of these concepts for verification. Since our verification tool is the KeY system, we then have to introduce and proof the user defined rules used in this thesis. After that we are ready to go through the methods in a bottom up approach and explain the contracts and their verification.

The Java code is taken from OpenJDK8, but however it does not differ from that of Oracle's JDK8. Our version of KeY is a branched version (taken from the 2.7 master), which was adapted to allow the use of static invariants. The hash from the developers' git repository of the KeY version we use is 7954548a25a5ad9aa4aa6203da1449e0a121190f.

3.1. BigInteger Concepts

The class `java.math.BigInteger` provides arbitrary[1] large integers, which behave similar to Java's primitive type integers. For this purpose the value is stored as an array of integers, which has the size necessary to contain the desired value. This value array is interpreted as if it was one large number in dual system. Thereby the first element of the array is the most significant. In addition, the sign of the BigInteger is stored in a separate integer field, which may contain the values 0, 1, and -1. A BigInteger object once created is immutable. To ensure uniqueness of each BigInteger, the array must not have leading zeros and if the sign is 0, then the value of the BigInteger must be 0 and vice versa. Because a BigInteger with the value 0 is frequently used, it is created once and stored as the static constant ZERO.

To make calculations on BigInteger objects possible, the class provides public methods for many operations, for example addition, subtraction, multiplication, bitwise operations and many more. Often those public methods call private methods for the actual calculations. In this thesis, we limit our scope to the public add method and the methods and constructors called from there.

[1] Of course, in fact there is a limitation to the size due to Java's maximum array size. The BigInteger class itself performs checks (see method `checkRange()`) and limits the size of the integer array to `Integer.MAX_VALUE/Integer.SIZE` $+1 = (2^{31}-1)/2^5+1 = 2^{26}$. In addition, if the array has maximum size, the first bit of the first element has to be 0. So the maximum number represented by a BigInteger may be $2^{2^{26}\cdot 2^5-1}-1 = 2^{2^{31}-1}-1$, which in decimal system is still a number with more than 640 million digits.

3.2. Code Adaptations for KeY

Not all features of Java are currently supported by KeY. Because of that, some code adaptations are necessary such that we are able to load our code into KeY.

First of all, for simplification we use a shortened version of the BigInteger source code. This file contains only methods we will consider in this thesis. Due to the modular approach of JML as well as KeY this is possible without losing soundness of our proofs.

Since Java's generic types are not directly supported within KeY, it is necessary to rewrite them in `java.math.BigInteger` and `java.lang.Comparable`. For that, we take the interface `Comparable` and replace the generic type `T` by `java.lang.Object`. This does not change the semantics, since it is exactly the same way Java internally handles generic types.

```
1 public interface Comparable/*<T>*/ {
2     compareTo(/*T*/ Object o);
3 }
```

Listing 3.1: Rewritten version of the interface `java.util.Comparable`. The original version with generic types is commented out.

With this changes in the interface we are able to rewrite BigInteger without generics as well:

```
1 public class BigInteger extends Number implements  Comparable/*<BigInteger>*/ {...}
```

In addition to the adaptation for generic types, we have to provide all the code we are using in one folder (and its subfolders). This includes the class BigInteger, the implemented interface Comparable, and the classes Arrays and System, which BigInteger makes use of.

3.3. Modeling the Semantics of BigInteger

In this section we introduce a new abstraction layer on top of the actual implementation, which will allow us to write clean specifications later on. This includes a field encoding the actual value of a BigInteger object into a natural number, some model methods for calculating this value, and the type invariants of the class. An overview is given in Figure 3.1.

3.3.1. The Ghost Field for the Value

While the implementation stores the information in a sign-magnitude form, for our specification the semantics are better caught by a single `value` field. Therefore we introduce an additional specification-only field (in JML denoted by the `ghost` keyword) of the type `\bigint`. As explained in Section 2.1.2.1, the `\bigint` type can be used to represent mathematical integers of potential infinite range.

```
1 //@ ghost \bigint value = 0;
```

Of course, initially our new `value` field has no meaningful value. To link the field with the actual value of the BigInteger, we have to calculate it from `signum` and `mag` array upon creation of a new BigInteger. Because a BigInteger object is final, we never have to recalculate it after that.

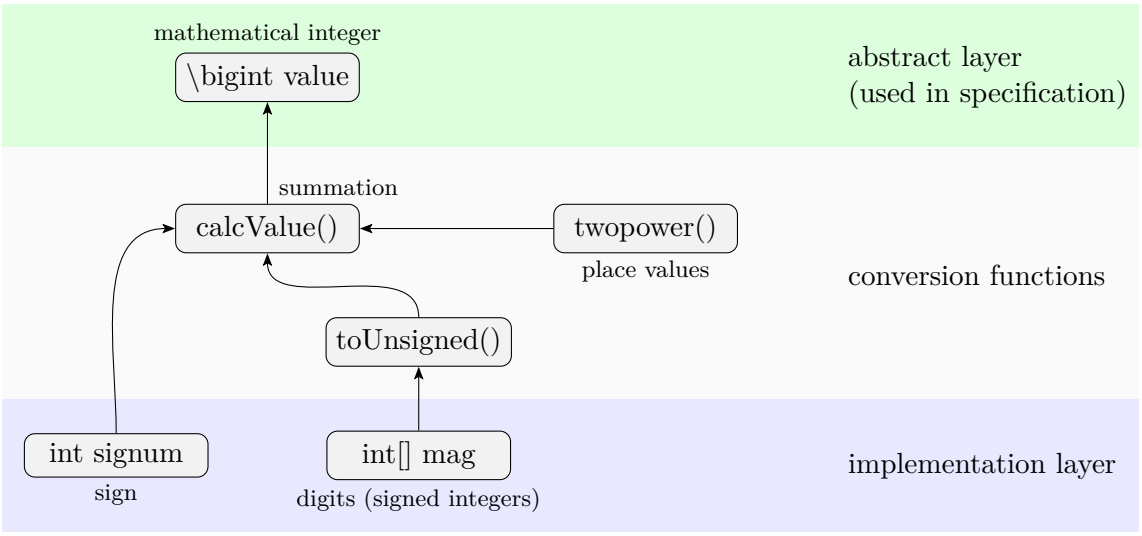

Figure 3.1.: Overview over the fields and methods introduced for modeling the semantics

3.3.2. Adding Helper Model Methods for Specification

The section below explains the functions for converting from implementation layer to the abstract specification layer. While there are other model methods used in this thesis, those in this section are used very frequently in many contracts and proofs. Those only used locally as abbreviations or to provide lemmas are explained in the corresponding sections of those methods.

3.3.2.1. `toUnsigned()`

A BigInteger object stores an array of single digits. As each digit is of type int, it can hold a value from `Integer.MIN_INT` $= -2^{31}$ to `Integer.MAX_INT` $= 2^{31} - 1$, both inclusive. In a place value notation, negative digits are not allowed. Therefore the negative values of the signed `int` are shifted up by 2^{31} to achieve an unsigned `int`. In code, this is done by (implicitly) casting to a long and then masking out the first 32 bits with a bitwise `and` operation.

For our specification, however, we decided to encapsulate this computation into a separate method. This is due to three reasons: At first, KeY is not good at dealing with bitwise operations. Second, the method name works as abbreviation. And third, we can annotate the method with a contract, which describes the behavior of our method. Especially this last thing will simplify the proofs very much.

```
1  /*@ model_behavior
2    @ ensures value == 0 ==> \result == 0;
3    @ ensures value != 0 ==> \result > 0;
4    @ ensures value > 0 ==> \result == value;
5    @ ensures value < 0 ==> \result == value + 0x100000000L;
6    @ ensures \result >= 0;
7    @ ensures \result < 0x100000000L;
8    @ ensures (\forall int i; value != i ==> \result != toUnsigned(i));    // injective function
9    @ accessible \nothing;
10   @ static helper model long toUnsigned(int value) {
11   @     return (long)value & 0xffffffffL;
12   @ }
13   @*/
```

Listing 3.2: Definition and contract of our model method `toUnsigned()`

Of course, we have to prove the contract in Listing 3.2. As mentioned above, dealing with bitwise operations in KeY is very difficult. Therefore, we use CBMC for our proof. Listing Listing 3.3 shows the contract prepared for CBMC.

```
1  public static long toUnsigned(int value) {
2      return (long)value & 0xffffffffL;
3  }
4
5  public static void main(int value, int other) {
6      long result = toUnsigned(value);
7      long otherRes = toUnsigned(other);
8
9      assert value != 0 || result == 0;
10     assert value == 0 || result > 0;
11     assert value <= 0 || result == value;
12     assert value >= 0 || result == value + 0x100000000L;
13     assert result >= 0;
14     assert result < 0x100000000L;
15     if (value != other) {
16         assert result != otherRes;
17     }
18 }
```

Listing 3.3: The contract of `toUnsigned()` prepared for proving in CBMC

As we can see, the first three lines simply contain the definition of our function, now of course without the special JML operators and keywords. In the main method we encoded all postconditions into assert statements, which will be checked by CBMC later on. We note that the universal quantifier disappeared. That is because we transfered the nondeterminism of the quantifier to a second input parameter. The implication within the scope of the quantifier is translated into a conditional expression (line 15). As we compile our source code to bytecode and run CBMC on it, we almost instantly get the results: All assertions valid.

The dependency contract (`accessible` clause) of this method can be easily shown in KeY.

3.3.2.2. `twopower()`

Since BigInteger uses place-value notation, we have to supply our `calcValue()` method with the power of two corresponding to each digit. Java has no built in power operator, in addition we need very large powers of type `\bigint`. Therefore we use our own recursive model method to calculate those values.

```
1  /*@ model_behavior
2    @ requires exp >= 0;
3    @ ensures \result >= 1;
4    @ ensures (\forall int i; 0 <= i && i < exp; twopower(i) < \result);   // strictly increasing
5    @ measured_by exp;
6    @ accessible \nothing;
7    @ static helper model \bigint twopower(int exp) {
8    @     return exp == 0 ? 1 : twopower(exp - 1) * 0x100000000L;
9    @ }
10   @*/
```

Listing 3.4: The model method `twopower()`

We have to mention that the actual multiplication factor (line 8) is not 2, but 2^{32}. That is because the digits have a maximum value of $2^{32} - 1$. Because it will come handy for some proofs later on, we provide an additional postcondition which states that the method's result is strictly increasing with `exp`.

As explained in Section 2.1.1.1, we use the `measured_by clause` to ensure termination of the recursion: The term `exp` is non-negative and strictly decreasing with recursion depth.

Just as with the previous method, the `accessible` clause is proved automatically by KeY.

3.3.2.3. `calcValue()`

With the two methods from the previous sections providing the input, we can now specify the method for calculating the value (compare Figure 3.1).

```
1  /*@ model_behavior
2    @ requires m.length > 0 ==> m[0] != 0;
3    @ requires s == 0 <==> m.length == 0;
4    @ ensures \result
5    @         == s*(\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
6    @ ensures s < 0 ==> \result < 0;
7    @ ensures s == 0 ==> \result == 0;
8    @ ensures s > 0 ==> \result > 0;
9    @ accessible m[*];    // m.length is a function in KeY, not a field, and thus not included here
10   @ static helper model \bigint calcValue(int s, int[] m) {
11   @     return s*(\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
12   @ }
13   @*/
```

Listing 3.5: The model method `calcValue()`

With the knowledge from the previous sections, the actual sum calculation now is straightforward. The only point to mention here is that while we start with the first element of the array, we have to multiply with the largest power. This is due to the big-endian order of the magnitude array. The precondition ensures the uniqueness of the BigInteger as explained in Section 3.1.

3.3.3. Writing Type Invariants for BigInteger

In this section we show how the general concepts of BigInteger explained in Section 3.1 are modeled as constraints. For that we use static and instance invariants.

3.3.3.1. Static Invariants

We use static invariants to describe the constant BigInteger `ZERO`. Since we do modular verification, we will not create invariants for static fields not needed for this thesis.

It is a design decision of the KeY developers that the static invariants are not automatically included in pre- and postcondition of a method. Instead, if we want to use it during a proof, we have to add it explicitly.

`ZERO` In the Java code, this name refers to a static BigInteger constant with the value 0. Because this object is a valid BigInteger, its invariant always has to hold.

```
1  //@ static invariant BigInteger.ZERO.<inv>;
2  //@ static invariant BigInteger.ZERO.value == 0;
```

3.3.3.2. Instance Invariants

In this section we describe the invariants which have to hold for each specific instance of BigInteger.

Value of the ghost field is correct: For establishing the connection between abstract and implementation layer, we have to ensure that the value is always exactly the same as computed by the `calcValue()` model method.

Possible `signum` values: There are only three values allowed for the signum field: 0 for the BigInteger with value 0, 1 for positive BigIntegers, and -1 for negative ones.

Uniqueness of BigInteger 0: This invariant prevents that different BigInteger objects with value 0 have differing mag fields by having a different amount of leading zeros. Each BigInteger with sign 0 must have an empty magnitude array and vice versa.

First element of the magnitude array is not 0: As explained above, this invariant is essential for uniqueness of the BigInteger object.

```
/*@ invariant value == calcValue(signum, mag);
  @ invariant signum == 0 || signum == 1 || signum == -1;
  @ invariant signum == 0 <==> mag.length == 0;
  @ invariant mag.length > 0 ==> mag[0] != 0;
  @ accessible \inv: signum, mag, mag[*], value;
  @*/
```

Listing 3.6: Instance invariants that have to hold for a valid BigInteger

KeY is nearly able to prove the dependency contract automatically, only very little simple interaction steps are required.

3.4. User-defined Taclets

As explained in Section 2.2.4, it is possible to extend KeY with User-defined taclets. In this subsection we will explain the two custom rules used in our verification.

3.4.1. `bsum_all_summands_gez`

This taclet represents the rule that a sum is greater or equal to zero if all elements are.

```
\schemaVariables {
  \term int i0,i1,t;
  \variables int uSub;
}
bsum_all_summands_gez {
  \find(bsum{uSub;}(i0,i1,t))
  \varcond(\notFreeIn(uSub, i0, i1))
  "Precondition": \add(==> \forall uSub; (i0 <= uSub & uSub < i1 -> t >= 0));
  "Use Case": \add(bsum{uSub;}(i0,i1,t) >= 0 ==>)
};
```

The `\find` clause enables us to apply the rule to every `bsum` (bounded sum). With application of the rule, two branches are added: In the "Precondition" case we have to show that all elements are greater or equal to zero. After that we may use our statement about the sum in the "Use Case". Of course our sum variable `uSub` must be chosen such that it does not appear in lower and upper bounds of the sum, which is ensured by the `\varcond` clause.

Proof: The KeY system generates a proof obligation which after some simple proof normalization steps in mathematical notation[2] reads as follows:

$$\forall_{i=i0}^{i1} a[i] \geq 0 \quad \Longrightarrow \quad \sum_{i=i0}^{i1} a[i] \geq 0$$

While the proposition is intuitively clear, we make an induction over the length of the array to show it in KeY:

hypothesis: $$\forall_{i=i0}^{i0+nv} a[i] \geq 0 \quad \Longrightarrow \quad \sum_{i=i0}^{i0+nv} a[i] \geq 0$$

basis (nv = 0): $$\forall_{i=i0}^{i0} a[i] \geq 0 \quad \Longrightarrow \quad \sum_{i=i0}^{i0} a[i] = 0 \geq 0$$

step: $$\forall_{i=i0}^{i0+nv+1} a[i] \geq 0 \quad \Longrightarrow \quad \sum_{i=i0}^{i0+nv+1} a[i] = \underbrace{a[i0+nv+1]}_{\geq 0 \text{ (quantifier)}} + \underbrace{\sum_{i=i0}^{i0+nv} a[i]}_{\geq 0 \text{ (hypothesis)}} \geq 0$$

□

In KeY, even though the induction is a bit more technical, the proof looks basically the same. When we have finished our induction, in the "Use Case" branch we make a cut with $i1 \geq i0$ (otherwise the sum is empty) and then instantiate nv with $i1 - i0$ in the quantifier. Now KeY is able to perform a normalization of the inequations and close all remaining branches.

3.4.2. `bsum_estimation`

This taclet provides an estimation for our frequently used sum term. Thus it looks more difficult due to index shifting, it basically states that in a place-value system the order of magnitude of digit n is greater than the maximum value that can be represented with $n-1$ digits. Since we use `twopower()` and `toUnsigned()`, the base of our numeral system is 2^{32}.

```
1  \schemaVariables {
2    \term int t,a,b;
3    \term Heap h;
4    \variables int uSub;
5  }
6  bsum_estimation {
7    \find(bsum{uSub;}(a+1,b,(java.math.BigInteger::toUnsigned(h,t)
8                          * java.math.BigInteger::twopower(h,b-uSub-1))))
9    \varcond(\notFreeIn(uSub, a,b,h))
10   "Valid Indices": \add(==> b >= a+1 & a+1 >= 0);
11   "Use Case": \add( bsum{uSub;}(a+1,b,java.math.BigInteger::toUnsigned(h,t)
12                                     * java.math.BigInteger::twopower(h,b-uSub-1))
13                  < java.math.BigInteger::twopower(h,b-a-1) ==>)
14 };
```

Proof: We just provide a paper version of this proof. This is due to two reasons: At first in KeY during the proof of a taclet the definition of a method can not be used and therefore the taclet can not be verified. Second, the proof contains reversing the order of the sum elements, which would be very difficult to perform in KeY.

[2] We use mathematical notation here as it is more pleasant to read. However, the terms can be translated one-to-one to KeY's ASCII based language.

We abbreviate $w := 2^{32}$, such that twopower(i) $\equiv w^i$. In addition, we write $|k|$ short for toUnsigned(k), where k is an arbitrary int value. Now the contract of toUnsigned() provides the following statement for the elements of the magnitude array m:

$$\forall i \in [0, m.length) : 0 \leq |m_i| < w$$

Now we consider our sum:

$$\sum_{i=a+1}^{b} |j_i| \cdot w^{b-i}$$

At first we shift the index down to zero:

$$= \sum_{i=0}^{b-a-1} |j_{i+a+1}| \cdot w^{b-i-a-1}$$

Instead of iterating over the array from first to last element, we want to go backwards:

$$= \sum_{i=0}^{b-a-1} |j_{b-i}| \cdot w^i \quad \overset{!}{<} \quad w^{b-a}$$

We set $n := b - a - 1$. Now the proposition can be proved with a simple induction over n:

hypothesis:	$\sum_{i=0}^{n} \lvert j_{b-i}\rvert \cdot w^i$	$<$	w^{n+1}
base case (n=0):	$\sum_{i=0}^{0} \lvert j_{b-i}\rvert \cdot w^i = 0$	$<$	w^1
step:	$\sum_{i=0}^{n+1} \lvert j_{b-i}\rvert \cdot w^i$		
	$= \sum_{i=0}^{n} \lvert j_{b-i}\rvert \cdot w^i + \lvert j_{b-i}\rvert \cdot w^{n+1}$		
	$\leq \sum_{i=0}^{n} (w-1) \cdot w^i + (w-1) \cdot w^{n+1}$		
	$< w^{n+1} + (w-1) \cdot w^{n+1}$		
	$= w \cdot w^{n+1}$		
	$= w^{n+2}$		

□

In addition to this taclet we also use a convenience version of it with indices shifted by one. Apart from the slightly different indices, the proof can be repeated the same way.

3.5. Method Contracts for Required Library Methods

As some parts of the code we want to verify use library methods from the classes `java.util.Arrays` and `java.lang.System`, we have to provide contracts for them as well. The methods are assumed to fulfill their contracts as verifying them was beyond the scope of this thesis. In the following section these contracts are explained.

3.5.1. `java.lang.System.arrayCopy()`

The `arrayCopy()` method of the class System can be used to copy a specific amount of elements from one array to another. We notice that `src` and `dest` are of type Object. Because every array type in Java is a subtype of Object, this makes it possible to copy arrays of any type. However, since we just want to copy arrays of `int`, we avoid unnecessary complexity and limit our contract by putting that into the precondition. Also we assume the array bounds to be correct as well as src and dest to be different. This assumptions allow us to write a much simpler contract without considering exceptions, but of course they have to be proved at every method call.

```
1  /*@ public normal_behavior
2    @ requires src instanceof int[] && dest instanceof int[];
3    @ requires src != dest;
4    @ requires srcPos >= 0 && destPos >= 0;
5    @ requires length >= 0;
6    @ requires srcPos + length <= ((int[])src).length && destPos + length <= ((int[])dest).length;
7    @ ensures (\forall int i; 0 <= i && i < length;
8    @             ((int[])dest)[destPos + i] == ((int[])src)[srcPos + i]);
9    @ assignable \strictly_nothing;
10   @*/
11 public static native void arraycopy(Object src,  int  srcPos, Object dest, int destPos, int length);
```

Listing 3.7: Contract of `arrayCopy()`: As can be seen, it is relatively simple due to limitation to our specific use case.

3.5.2. `java.util.Arrays.copyOfRange()`

This second library method has a similar function to the one recently explained. The main differences are: This one returns a new array to which the elements are copied. It may even expand the array and fill it up with zeros. Furthermore it is limited to int arrays. Listing 3.8 shows the full contracts for normal and exceptional behavior of the method.

```
1  /*@ public normal_behavior
2    @ requires 0 <= from && from <= original.length && from <= to && original != null;
3    @ ensures \result != original;                                    // always returns a new array
4    @ ensures \result.length == to - from;
5    @ ensures (to <= original.length)
6    @      ==> (\forall int i; from <= i && i < to; \result[i - from] == original[i]);
7    @ ensures (to > original.length)
8    @      ==> ((\forall int i; from <= i && i < original.length; \result[i - from] == original[i])
9    @          && (\forall int i; original.length <= i && i < to; \result[i - from] == 0));
10   @ assignable \nothing;
11   @
12   @ also
13   @
14   @ public exceptional_behavior
15   @ requires from < 0 || from > original.length || from > to || original == null;
16   @ signals (IllegalArgumentException e) from > to;
17   @ signals (ArrayIndexOutOfBoundsException e) from < 0 || from > original.length;
18   @ signals (NullPointerException e) original == null;
19   @ signals_only IllegalArgumentException, ArrayIndexOutOfBoundsException, NullPointerException;
20   @ assignable \nothing;
21   @*/
22 public static int[] copyOfRange(int[] original, int from, int to) { ... }
```

Listing 3.8: Contract of `copyOfRange()`

3.6. KeY's Taclet Settings and Proof Search Strategy

Before we start with the actual method contracts, we have to make some remarks on the settings of the KeY system.

3.6.1. Taclet Settings

Integer Semantics While there is a setting for taking integer overflows into account, it makes the proofs much more complicated by introducing modulo operations. Therefore, we treat integers as if they were unbounded. Of course this is not the way Java handles the `int` type, but for proving the functionality of the methods it should be enough, as long as no overflow may occur.

Implicit Exceptions In a similar way we simplify from Java's runtime exceptions and assume that an unchecked runtime exception results in a program crash. This is sound if no runtime exceptions occur during execution, which we ensure with suitable preconditions.

Static Initialization Finally, we consider the static initialization as finished at proof start, in contrast to Java where it is done when a class is first used. A complete list of settings can be found at Section B.

3.6.2. Proof Search Strategy

As a proof search strategy we use the predefined strategy for Java verification. An list of all settings can be found in Section B. However, it may sometimes be necessary to change a settings, for example to deal with a system of inequations. This will be noted at the respective text position.

3.7. Writing and Proving the Method Contracts

In this section we explain the actual method contracts and their verification. We intend to verify as much as possible from the call hierarchy shown in Figure 3.2. To ensure soundness of our proof, we approach verification bottom up, starting with very simple methods providing and going on to more complex ones. This enables us to use the already verified contracts of the small methods when proving the larger ones.

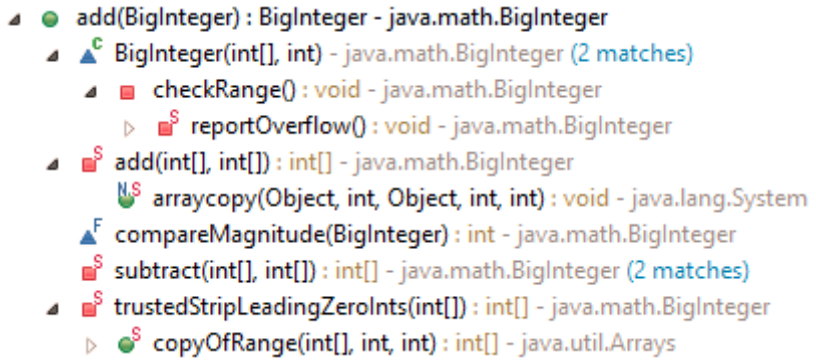

Figure 3.2.: Call hierarchy of the `public add()` method from Eclipse.

3.7.1. reportOverflow()

This method provides very simple error reporting functions. Its only purpose is to signal that the resulting BigInteger would be too big for the supported length of the magnitude array.

```
1  /*@ private exceptional_behavior
2    @ signals (ArithmeticException e) true;
3    @ signals_only ArithmeticException;
4    @ assignable \nothing;
5    @*/
6  private static /*@ helper @*/ void reportOverflow() {
7    throw new ArithmeticException("BigInteger would overflow supported range");
8  }
```

Listing 3.9: The very simple method `reportOverflow()`

Because this method always throws an `ArithmeticException` and no other exception, we use the `signals_only` clause and the `signals` clause to specify exactly this. Of course the method must not assign any existing heap location. We don't use `strictly_nothing` here because the method may assign the newly created object for the return value.

As we can see, this method contract is very simple, which enables KeY to verify it fully automated. We just have load the contract and start the verification.

3.7.2. checkRange()

This method checks if the BigInteger is in the supported range (see footnote[1] on page 17). If the formula in the `if` clause holds, the method throws an exception, otherwise not. We use that formula in the precondition of our `exceptional_behavior` contract and the formula negated for `normal_behavior`.

```
1  /*@ private normal_behavior
2    @ requires mag.length <= MAX_MAG_LENGTH
3    @      && (mag.length != MAX_MAG_LENGTH || mag[0] >= 0);
4    @ assignable \nothing;
5    @
6    @ also
7    @
8    @ private exceptional_behavior
9    @ requires mag.length > MAX_MAG_LENGTH
10   @       || mag.length == MAX_MAG_LENGTH && mag[0] < 0;
11   @ signals (ArithmeticException e) mag.length > MAX_MAG_LENGTH
12   @                                 || mag.length == MAX_MAG_LENGTH && mag[0] < 0;
13   @ signals_only ArithmeticException;
14   @ assignable \nothing;
15   @*/
16 private void checkRange() {
17   if (mag.length > MAX_MAG_LENGTH || mag.length == MAX_MAG_LENGTH && mag[0] < 0) {
18     reportOverflow();
19   }
20 }
```

Listing 3.10: Contract and implementaion of `checkRange()`

As this method is still very simple, both contracts can be proved by KeY automatically.

3.7.3. stripLeadingZeroInts()

As explained in the BigInteger concept section Section 3.1, for uniqueness it is necessary for the magnitude array not to have any leading zeros. Therefore this method strips any

leading zero elements of a given array. We distinguish between three different cases: The array is empty, it contains just zeros, or it contains at least one nonzero element. The specification for each of these cases can be seen in listing Listing 3.11. In addition we specify that the order of the values in the array must not change, that the resulting array must have a less or equal number of elements, and that the result must not be a reference to the given array.

```
1  /*@ public normal_behavior
2    @ ensures \result != val;                          // difference to trustedStripLeadingZeroInts
3    @ ensures \result.length <= val.length;
4    @ ensures val.length == 0 ==> result.length == 0;              // case1: arr = {}
5    @ ensures (\forall int i; 0 <= i && i < val.length; val[i] == 0)
6                    ==> \result.length == 0;                       // case2: arr = {0 .. 0}
7    @ ensures (\exists int i; 0 <= i && i < val.length; val[i] != 0)
8                    ==> \result[0] != 0;                           // case3: arr = {x1 .. xn}
9    @ ensures (\forall int i; 0 <= i && i < \result.length;
10                 \result[i] == val[firstNonZeroIndex(val) + i]);   // copy of old values, same order
11   @ assignable \nothing;
12   @*/
13 private static int[] stripLeadingZeroInts(int val[]) {...}
```

Listing 3.11: Contract of `stripLeadingZeroInts()` with the three different cases as explained above

As we have to specify that nothing except from cropping the zeros happens (lines 9-10), we have to know the index of the first nonzero element. That part is done by the helper model method `firstNonZeroIndex()`. Of course for this method the array must contain such an element, because otherwise the index would not exist. This is ensured by excluding the cases 1 and 2 in the precondition.

```
1  /*@ public normal_behavior
2    @ requires arr.length > 0;                                    // case 1: arr != {}
3    @ requires (\exists int i; 0 <= i && i < arr.length; arr[i] != 0);   // case 2: arr != {0 .. 0}
4    @ ensures \result >= 0;
5    @ ensures \result < arr.length && arr[\result] != 0
6          && (\forall int i; 0 <= i && i < \result; arr[i] == 0);       // case 3: arr = {x1 .. xn}
7    @ public static helper model int firstNonZeroIndex(int[] arr);
8    @*/
```

Listing 3.12: Helper method for specification which "calculates" the index of the first nonzero element

Now we can start with the proving part. Even though the method looks still very simple, it turns out that it is much more difficult to prove than the preceding ones. This is caused by the loop: Since the loop bounds are not known during compilation time, KeY can not simply unroll the iterations. Instead, we must help KeY through providing a loop specification. As outlined in Section 2.1.3.2, this specification consists of a loop invariant, an `assignable` clause, and a termination witness. The interesting loop invariant part states, that up to the current index of the loop all preceding elements of the array are zeros. The rest of the loop specification is pretty self-explanatory.

In proof, KeY uses an induction-like technique: When applying the loop specification, it creates three branches. The first one is the base case, the situation before the first execution of the loop code. The second one is the induction step, we have to prove here that the loop maintains the loop invariant. In addition, it has to be shown that the loop variant is really nonnegative and strictly decreasing as well as it is not written to any heap location. If we have proved those two branches, we may continue our proof with the third branch, called Use Case. This branch represents the situation after termination of the loop. We may now use the loop invariant to show the postcondition of our method.

```
1  private static int[] stripLeadingZeroInts(int val[]) {
2      int vlen = val.length;
3      int keep;
4
5      // Find first nonzero int
6      /*@ loop_invariant 0 <= keep && keep <= vlen && (\forall int i; 0 <= i && i < keep; val[i] == 0);
7        @ assignable \strictly_nothing;
8        @ decreases vlen - keep;
9        @*/
10     for (keep = 0; keep < vlen && val[keep] == 0; keep++)
11         ;
12     return java.util.Arrays.copyOfRange(val, keep, vlen);
13 }
```

Listing 3.13: Implementation of `stripLeadingZeroInts()` with loop specification

After we have written this, KeY is able to prove the method automatically. We can see that the prover applies the loop specification and proves all three branches as explained above.

3.7.4. `trustedStripLeadingZeroInts()`

While the previous method served as an intermediate step, the current is the one we actual need (see Figure 3.2). It does nearly the same as stripLeadingZeroInts. The difference is in the last line: Instead of always returning a new array, if the array already has no leading zeros the input array itself is returned without creating a new one and copying it.

```
12     return keep == 0 ? val : java.util.Arrays.copyOfRange(val, keep, vlen);
```

Listing 3.14: The only line different to `stripLeadingZeroInts()`

To specify its behavior, we simply take the contract of `stripLeadingZeroInts()` (Listing 3.11) and omit line 2. As was the case with the previous method, this one can be proved by KeY automatically if we provide the exact same loop specification.

3.7.5. `compareMagnitude()`

With this method, we get to the first non-static method. It takes another BigInteger as a parameter and returns 0, −1 or 1 if the absolute value of that is equal, greater or less than the one of the BigInteger on which the method is called. Of course the invariant for the given BigInteger has to hold at method call, as well as implicitly the invariant of the `this` object holds.

```
1  /*@ normal_behavior
2    @ requires val.<inv>;
3    @ ensures calcMagSum(val.mag) == calcMagSum(mag) ==> \result == 0;
4    @ ensures calcMagSum(val.mag) > calcMagSum(mag) ==> \result == -1;
5    @ ensures calcMagSum(val.mag) < calcMagSum(mag) ==> \result == 1;
6    @ assignable \nothing;
7    @*/
8  final int compareMagnitude(BigInteger val) { ... }
```

Listing 3.15: Contract of `compareMagnitude()`

As can be seen, for a clean specification we introduced the model method `calcMagSum()`. This method is the subject of the next section.

3.7.5.1. The Helper Model Method `calcMagSum()`

In contrast to `calcValue()` this model method ignores the sign and calculates just the absolute value. In addition, the contract contains two lemmas which we will use in our proof of `compareMagnitude()`. Those lemmas are denoted with A and B in Listing 3.16.

```
1  /*@ private model_behavior
2    @ requires m.length > 0 ==> m[0] != 0;
3    @ ensures \result
4    @         == (\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
5    @ ensures m.length > 0 ==> \result > 0;
6    @ ensures \result >= 0;
7    @ ensures (\forall int[] m2;                  // A: a.length < b.length -> a.result < b.result
8    @             0 <= m2.length && m2.length < m.length && (m2.length > 0 ==> m2[0] != 0);
9    @             calcMagSum(m2) < \result);
10   @ ensures (\forall int[] m3;                  // B: equal lengths and elements -> equal results
11   @             m3.length == m.length && (\forall int j; 0 <= j && j < m.length; m3[j] == m[j]);
12   @             \result == calcMagSum(m3));
13   @ accessible m[*];
14   @ private static helper model \bigint calcMagSum(int[] m) {
15   @     return (\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
16   @ }
17   @*/
```

Listing 3.16: Contract of the helper model method `calcMagSum()`. A and B are lemmas.

Of course we have to proof the contract of the model method as well, especially the two lemmas.

Proof The first three ensures clauses are proved very easily by application of method contracts and use of the precondition.

Lemma A This lemma is really important for the proof of `compareMagnitude()`. It states that the result of `calcMagSum()` increases strictly monotonically with the size of the given array. This is true because we use place value notation and the first (most significant) elements of both arrays are assumed to be non-zero. For the same reason a 5-digit number always is greater than a 4-digit number. Because the proof gets relatively technical due to the base of our number system, model methods, and KeY itself, we will not describe it in details.

Lemma B If two sums have equal length and elements, they are equal. This very basic statement can be seen as a convenience lemma for abbreviation in the proof and is shown simply by skolemization and instantiation of the inner quantifier.

3.7.5.2. Proof of `compareMagnitude()`

With the model method proved, we now turn towards the proof of the Java method. As in the Java methods of the previous sections the implementation of `compareMagnitude()` contains a loop. So we have to provide a loop specification. It simply states that the two arrays are equal up to the current loop index (excluded).

After running the macro "Full Auto Pilot", 12 branches are still open. Outside the branches related to the loop, we can see that the different cases of array lengths result in 4 open branches (if clauses in lines 6 and 8). These goals can be shown by deducing a contradiction from the relation between array lengths and `calcMagSum()` results on the left hand side of

```
final int compareMagnitude(BigInteger val) {
  int[] m1 = mag;
  int len1 = m1.length;
  int[] m2 = val.mag;
  int len2 = m2.length;
  if (len1 < len2)
    return -1;
  if (len1 > len2)
    return 1;

  /*@ loop_invariant 0 <= i && i <= len1 && (\forall int j; 0 <= j && j < i; m1[j] == m2[j]);
    @ assignable \nothing;
    @ decreases len1 - i;
    @*/
  for (int i = 0; i < len1; i++) {
    int a = m1[i];
    int b = m2[i];
    if (a != b)
      return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
  }
  return 0;
}
```

Listing 3.17: Implementation of `compareMagnitude()` with loop specification.

the sequent arrow. For that, we need lemma A from our model method calcMagSum. The base case of the loop specification is proved automatically by KeY.

In the use case branch 3 goals are still open: Two of them can be shown by deriving a contradiction with the help of lemma B and for the remaining one we use the dependency contract of the invariant of the object "this".

The interesting part of the proof is showing that the loop body preserves the loop invariant. Here 5 branches are open. On can be shown just by using the dependency contract of `this.<inv>`, another one can be closed automatically by KeY. The remaining 3 branches require much user interaction. The relevant part of the sequence contains a relation between the (magnitude) sums and the elements at current loop index.

```
calcMagSum(val.mag) < calcMagSum(self.mag);
toUnsigned(self.mag[i_3]) < toUnsigned(val.mag[i_3]);
...
==>
...
val.mag[i_3] = self.mag[i_3];
```

Listing 3.18: Shortened version of the sequent of the most difficult proof part. i_3 stands for the current loop index. The two other open goals are very similar to this and require the same interactive steps.

Since we know that the arrays (and hence the sums) are equal up to current index, we can subtract that part of the sum from the (in)equation. Then we pull out the first element of the sums and use our taclets for estimations on the remaining sum terms. After that, KeY should be able to proof the inequation system automatically. However, we have to use the Strategy Setting "Model Search" and lead KeY to the right direction. This is done by hiding all unused formulas and pulling out (i.e. replacing them by symbols) the sum and method terms to prevent the system by further inserting definitions. With this help, finally all branches get closed.

In summary, we divided our problem into different lemmas (in postcondition and a taclet), which reduced its complexity drastically. With this, we could prove our contract, even if we needed interaction to push KeY to the focus of the problem.

3.7.6. `subtract()`

We will not provide a proof in KeY for this method. This is due to two reasons: First, since the method creates an array on the heap and assigns its elements, there are different heap states to reason about. Typically, for this we could use our framing clauses (`assignable`) and dependency contracts of the methods. However, in KeY there is a limitation on the latter: They may only be used if the exact same method call but referring to another heap appears on the sequent. In our case, not the method call appears again, but the information is provided in a slightly different form. Therefore we can not use the dependency contracts. The second reason is that the method makes use of bitwise operations, for which no reasoning rules exist in KeY.

There are workarounds (manual cuts and using CBMC) for both issues, but they are laborious and too time-consuming for this thesis. Instead, we provide a proof on paper, which may be transferred to KeY in future work, perhaps in an improved version of KeY.

3.7.6.1. Overview

As the name states, the method subtracts the two arrays given as parameters. For that, it imposes the restriction that the value represented by the first parameter (via the place value notation introduced in Section 3.3) is (strictly) greater than that represented by the second one. Thus the value represented by the result is always positive. Overall the subtraction is very similar to the method taught at school: The arrays are run through from the last (least significant) digits to the most significant ones. By doing that, the current two digits are subtracted with regard to a possible borrow from the previous places. All this is done by the first loop. At some point, the end of the little array is reached. If there is still a borrow, the second loop propagates it as long as there are zeros in the big array, until it finally disappears. Due to the precondition, we know that this actually happens. Finally the third loop just copies the remaining digits of the bigger number to the result.

3.7.6.2. Preconditions

As already stated above, the method `subtract()` may be called only if the first parameter represents a greater value than the second one. Other cases have to be checked first by the calling methods. For modeling this precondition we use our already introduced model method `calcMagSum()`, which in addition provides a lemma about the lengths of the parameters (lemma A in Section 3.7.5.1).

The second precondition ensures that the arrays have no leading zeros. This is a simplification, since the method fulfills its purpose otherwise as well. However, it simplifies the proof and is ensured by all methods of BigInteger calling `subtract()`.

```
1 /*@ private normal_behavior
2   @ requires calcMagSum(big) > calcMagSum(little);
3   @ requires (big.length > 0 ==> big[0] != 0) && (little.length > 0 ==> little[0] != 0);
4   @ ensures simpleSum(\result) == simpleSum(big) - simpleSum(little);      // may have leading zeros
5   @ assignable \nothing;
6   @*/
7 private static /*@ helper @*/ int[] subtract(int[] big, int[] little) { ... }
```

Listing 3.19: Contract of the method `subtract()`

3.7.6.3. Postconditions

The postcondition just states that the result is the difference of the parameters. The model method `simpleSum()` just serves as an abbreviation for our usual place-value sum term. The assignable clause states that no heap locations of existing objects are changed.

3.7.6.4. Proof structure

We subdivide the proof into three parts corresponding to the three loops of the method. For each loop we provide a loop specification and afterwards justify the choice of it. As with the loops in previous methods, we have to show three propositions for each of the loops:

- Invariant initially Valid: The invariant holds before the first iteration.
- Body Preserves Invariant (induction step): If the loop invariant holds for an (arbitrary) iteration, it also holds for the following one.
- Use Case: After loop termination the remaining method body establishes the method's postcondition. In this proof part we may assume the loop invariant as well as the negated loop condition.

In all proof parts we will concentrate on the interesting and difficult statements and omit trivial parts, for example showing that the indices are in the bounds given by the invariant.

3.7.6.5. Proof Part 1: First Loop

The first loop subtracts the two arrays starting at the last (least significant) digits with decreasing indices until all elements of the smaller array (`little`) have been considered. The subtraction (lines 19-21) is done by interpreting the digits as unsigned integers (see Section 3.3.2.1) and taking a potentially existing borrow of the previous place into account. This borrow is calculated by shifting `difference` (see line 12 of Listing 3.20). If the current element of `big` is strictly smaller than that of `little` plus the possible borrow (0 or 1) from the previous iteration, then `diff` $\gg 32$ evaluates to -1, otherwise to 0.

```
1  private static /*@ helper @*/ int[] subtract(int[] big, int[] little) {
2    int bigIndex = big.length;
3    int result[] = new int[bigIndex];
4    int littleIndex = little.length;
5    long difference = 0;
6
7    // Subtract common parts of both numbers
8    /*@ loop_invariant 0 <= littleIndex && littleIndex <= little.length
9      @              && 0 <= bigIndex && bigIndex <= big.length
10     @              && big.length - bigIndex == little.length - littleIndex
11     @              && partialSum(result, bigIndex)
12     @                 + (difference >> 32) * twopower(big.length - bigIndex) // borrow from next digit
13     @                == partialSum(big, bigIndex)
14     @                 - partialSum(little, littleIndex);
15     @ assignable result[(result.length-little.length)..(result.length-1)];
16     @ decreases littleIndex;
17     @*/
18   while (littleIndex > 0) {
19     difference = (big[--bigIndex] & LONG_MASK) -
20                  (little[--littleIndex] & LONG_MASK) +
21                  (difference >> 32);
22     result[bigIndex] = (int)difference;
23   }
24   ...
```

Listing 3.20: Section of the method body relevant for the first proof part

The first three lines of the loop invariant state that the indices are valid and inside bounds. The last part of the invariant is the interesting one. It states that the partial sum of the result array up to the current loop index equals the difference of the partial sums of the parameter arrays plus a possible borrow. This is expressed via the model method `partialSum()`, which just serves as an abbreviation for the place-value sum starting from the given index. Since the borrow is taken from the next place, it has to multiplied with the corresponding place value calculated by `twopower()`. Note that the borrow is placed on the left side of the equation, because it may be -1 or 0.

The assignable clause exactly specifies which elements are written by the loop. The bound are included.

Abbreviations To improve the readability we will use the following abbreviations:

$$\begin{aligned}
\texttt{diff} &\mathrel{\widehat{=}} \texttt{difference} \\
\sum_r &\mathrel{\widehat{=}} \texttt{partialSum(result,bigIndex)} \\
\sum_b &\mathrel{\widehat{=}} \texttt{partialSum(big,bigIndex)} \\
\sum_l &\mathrel{\widehat{=}} \texttt{partialSum(little,littleIndex)} \\
w^i &\mathrel{\widehat{=}} \texttt{twopower}(i)
\end{aligned}$$

The last one means that w equates to 2^{32}, which is exactly the base of the function represented by `twopower()`.

We want to abstract from the fact that the indices of the two arrays are different (`bigIndex` and `littleIndex`) and that we start from the end of the arrays. Therefore we use the following abbreviation for the index of the current loop iteration:

$$i := \texttt{big.length} - \texttt{bigIndex} = \texttt{little.length} - \texttt{littleIndex}$$

A prime after a sum or variable (e.g. $\sum_r{}'$, $\texttt{diff}'$) indicates the next loop iteration (index increased by 1). Finally, R, B and L denote the first summand pulled out of the according sum with prime:

$$\begin{aligned}
R &\mathrel{\widehat{=}} \texttt{toUnsigned(result[bigIndex])} \cdot \texttt{twopower}(i) \\
B &\mathrel{\widehat{=}} \texttt{toUnsigned(big[bigIndex])} \cdot \texttt{twopower}(i) \\
L &\mathrel{\widehat{=}} \texttt{toUnsigned(little[littleIndex])} \cdot \texttt{twopower}(i)
\end{aligned}$$

Putting all this together, our first loop invariant looks like this:

$$\sum_r + (\texttt{diff} \gg 32) \cdot w^i = \sum_b - \sum_l$$

Note that $\gg$ denotes Java's arithmetic right shift operator. An arithmetic right shift by n has the effect of a division by 2^n with the result rounded towards negative infinity[3].

[3]For reference see `http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-15.19`

Invariant Initially Valid With $\texttt{diff} = 0$ and empty partial sums:

$$\underbrace{\sum_{r}}_{=0} + \underbrace{(\texttt{diff} \gg 32)}_{=0} \cdot w^i = \underbrace{\sum_{b}}_{=0} - \underbrace{\sum_{l}}_{=0} \quad \Leftrightarrow \quad 0 = 0$$

□

Body Preserves Invariant

Here we assume that our invariant holds for an arbitrary index i:

$$\sum_{r} +(\texttt{diff} \gg 32) \cdot w^i = \sum_{b} - \sum_{l} \tag{*}$$

We show now that this statement also holds for the next loop iteration with index $i + 1$. Therefore, we have to proof the following statement:

$$\sum_{r}{}' + (\texttt{diff}' \gg 32) \cdot w^{i+1} = \sum_{b}{}' - \sum_{l}{}'$$

We split off the first elements of all three sums (using the abbreviations given above):

$$\sum_{r} +R \cdot w^i + (\texttt{diff}' \gg 32) \cdot w^{i+1} = \sum_{b} - \sum_{l} +B \cdot w^i - L \cdot w^i$$

The sums on the right can be replaced with our assumption (*):

$$\sum_{r} +R \cdot w^i + (\texttt{diff}' \gg 32) \cdot w^{i+1} = \sum_{r} +(\texttt{diff} \gg 32) \cdot w^i + B \cdot w^i - L \cdot w^i$$

Now we subtract the sum from both sides:

$$R \cdot w^i + (\texttt{diff}' \gg 32) \cdot w^{i+1} = (\texttt{diff} \gg 32) \cdot w^i + B \cdot w^i - L \cdot w^i$$

Since we know it is greater than 0, we can divide by w^i:

$$R + (\texttt{diff}' \gg 32) \cdot w = (\texttt{diff} \gg 32) + B - L$$

We insert the assignment $R := (int)\texttt{diff}'$ from the code (line 22), where (int) is the Java cast operator:

$$\underbrace{(int)\texttt{diff}'}_{\text{lower 32 bits of } \texttt{diff}'} + \underbrace{(\texttt{diff}' \gg 32) \cdot w}_{\text{upper 32 bits of } \texttt{diff}'} = B - L + (\texttt{diff} \gg 32)$$

As stated in the brace above, the cast[4] to an int just takes the lower 32 bits of $\texttt{diff}'$. The second brace is tricky: The arithmetic shift right shifts the upper half of the long to its lower half while performing sign extension. This means, the upper half of the long consists now either only of zeros or only of ones. Now we multiply with $w = 2^{32}$, which is equivalent to shifting back left by 32, filling the lower half with zeros. Obviously by adding these two results, we just get $\texttt{diff}'$ and thus the following equation:

$$\texttt{diff}' = B - L + (\texttt{diff} \gg 32)$$

This is exactly the assignment from lines 19-21 of the Java code above and thus holds.

□

[4]See `http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1.3`

3.7.6.6. Proof Part 2: Second Loop

We now continue our proof with the "Use Case" part of the first loop, which leads us to the second loop. After the termination of the first loop, there may still be a borrow. This borrow propagates as long as the according digits of the array `big` are zeros. The second loop iterates until the borrow disappears.

```
1  ...
2  // Subtract remainder of longer number while borrow propagates
3  boolean borrow = (difference >> 32 != 0);
4
5  /*@ loop_invariant 0 <= bigIndex && bigIndex <= big.length
6    @       && partialSum(result, bigIndex) == partialSum(big, bigIndex)
7    @                                        - partialSum(little, 0)
8    @                                        + (borrow ? twopower(big.length-bigIndex) : (\bigint)0);
9    @ assignable result[*];
10   @ decreases bigIndex;
11   @*/
12 while (bigIndex > 0 && borrow)
13   borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);
14   /* --bigIndex;
15    * result[bigIndex] = big[bigIndex] - 1;
16    * borrow = (big[bigIndex] == 0);
17    */
18 ...
```

Listing 3.21: Section of the method body relevant for the second proof part.

Line 13 of Listing 3.21 is composed of multiple statements. For better understanding, the comments in lines 14-16 contain this line broken down into simple statements. It can be seen that `borrow` remains `true` until a nonzero element of `big` is reached.

The loop invariant is very similar to that of the first loop. We have to note that the partial sum of `little` now is constant and comprises all elements array. Contrary to the first loop, the borrow stands now on the right side of the equation. This is simply because the sign of the borrow term now is positive.

We have to mention that the `assignable` clause does not specify the bounds for changes exactly. That is, because to give an exact specification we would need an additional model method for calculating the lower bound. Since for the paper proof the assignable clause is not needed, we will not provide such a method. Of course this problem also persist for the next loop, this time for the upper bound.

Assumptions We can now presume the negated loop condition of the first loop:

$$\neg(\texttt{littleIndex} > 0)$$

Because we also know that an array index has to be nonnegative, this becomes:

$$\texttt{littleIndex} = 0$$

In addition we assume the loop invariant of the previous loop:

$$\sum_{r} + (\texttt{diff} \gg 32) \cdot w^i = \sum_{b} - \sum_{l}$$

Initially Valid We have to show the following statement:

$$\sum_r = \sum_b - \sum_l + (\texttt{borr}\ ?\ w^i : 0)$$

We insert the invariant of the previous loop:

$$\sum_r = \sum_r + (\texttt{diff} \gg 32) \cdot w^i + (\texttt{borr}\ ?\ w^i : 0)$$

Now we can subtract the sum from both sides:

$$0 = (\texttt{diff} \gg 32) \cdot w^i + (\texttt{borr}\ ?\ w^i : 0)$$

Again we divide by w^i:

$$0 = (\texttt{diff} \gg 32) + (\texttt{borr}\ ?\ 1 : 0)$$

This results in two different cases:

Case 1: borr = true:

$$(\texttt{diff} \gg 32) = -1$$

Case 2: borr = false:

$$(\texttt{diff} \gg 32) = 0$$

This looks contradictory, but with the definition of the first borrow

$$\texttt{borr} := (\texttt{diff} \gg 32 \neq 0)$$

it is clear that the statements in both cases hold.

□

Body Preserves Invariant We now have a small change in the definition of a sum abbreviation. Since we reached the end of the array,

$$\sum_l = \texttt{partialSum}(\texttt{little},\ 0)$$

is constant now for each loop iteration of the second loop. We again assume that the loop invariant holds for an arbitrary index i:

$$\sum_r = \sum_b - \sum_l + (\texttt{borr}\ ?\ w^i : 0) \qquad (**)$$

The statement to show looks as follows (note that the sum of `little` has no prime):

$$\sum_r' = \sum_b' - \sum_l + (\texttt{borr}'\ ?\ w^{i+1} : 0)$$

We again pull out the first elements of the sums with prime, insert the assumption (**), and subtract common parts from the equation:

$$\sum_r +R \cdot w^i = \sum_b - \sum_l +B \cdot w^i + (\texttt{borr}' \ ? \ w^{i+1} : 0)$$
$$\sum_b - \sum_l +(\texttt{borr} \ ? \ w^i : 0) + R \cdot w^i = \sum_b - \sum_l +B \cdot w^i + (\texttt{borr}' \ ? \ w^{i+1} : 0)$$
$$(\texttt{borr} \ ? \ w^i : 0) + R \cdot w^i = B \cdot w^i + (\texttt{borr}' \ ? \ w^{i+1} : 0)$$

Afterwards follows a division by w^i:

$$(\texttt{borr} \ ? \ 1 : 0) + R = B + (\texttt{borr}' \ ? \ w : 0)$$

We know that `borr` is `true`, because otherwise we would not have reached the next loop iteration:

$$1 + R = B + (\texttt{borr}' \ ? \ w : 0)$$

Here we have to be very careful with the integer semantics. The single digits of the arrays use the unsigned integer semantics explained in Section 3.3.2.1. In our proof, integers have the semantics of mathematical integers. This means, we have to convert our value if we want to insert the assignment of $R := B - 1$ from the code. Therefore, we use $|x|_u$ as an abbreviation for $\texttt{toUnsigned}(x)$. With that, we now are allowed to insert our assignment $R := |B - 1|_u$:

$$1 + |B - 1|_u = B + (\texttt{borr}' \ ? \ w : 0)$$

From that we get two cases, in which we have to respect the assignment between $\texttt{borr}'$ and B as explained in line 16 of Listing 3.21 :

$$\texttt{borr}' := (B == 0)$$

Case 1: $\texttt{borr}' = \texttt{true}$: This means B equals 0.

$$1 + |-1|_u = w$$

We use the following formula to translate the signed integer to unsigned:

$$|-1|_u = w - 1$$

Now we can insert and simplify the formula with mathematical integer semantics:

$$1 + (w - 1) = w$$
$$w = w$$

Case 2: $\texttt{borr}' = \texttt{false}$: This means B is greater than 0.

$$1 + |B - 1|_u = B + 0$$

Since $B - 1$ is nonnegative, its unsigned value equals the signed one:

$$1 + B - 1 = B$$
$$B = B$$

□

3.7.6.7. Proof Part 3: Final Loop

The "Use Case" part of the second loop now again contains the next loop. This simply copies the remaining digits of `big` to `result`.

```
1   ...
2   // Copy remainder of longer number
3   /*@ loop_invariant bigIndex >= 0
4     @             && partialSum(result, bigIndex) == partialSum(big, bigIndex)
5     @                                              - partialSum(little, 0);
6     @ assignable result[*];
7     @ decreases bigIndex;
8     @*/
9   while (bigIndex > 0)
10    result[--bigIndex] = big[bigIndex];
11
12  return result;
13 }
```

Listing 3.22: Remaining method body for the third proof part

Assumptions We may now assume the negated loop condition of the second loop:

$$\neg(\texttt{bigIndex} > 0 \land \texttt{borr})$$

With De Morgan's law:

$$(\texttt{bigIndex} \leq 0) \lor \neg\texttt{borr}$$

Since `bigIndex` is nonnegative (see invariant of previous loop), this becomes:

$$(\texttt{bigIndex} = 0) \lor \neg\texttt{borr}$$

Due to the fact that `big` represents a greater number than `little`, we know that the case

$$(\texttt{bigIndex} = 0) \land \texttt{borr}$$

can not happen. So in summary the statement

$$\neg\texttt{borr}$$

can be achieved via simple transformations.

Initially Valid We assume the previous loop invariant

$$\sum_r = \sum_b - \sum_l + (\texttt{borr}\ ?\ w^i : 0)$$

and have to show the following formula:

$$\sum_r = \sum_b - \sum_l$$

This can be directly deduced with the assumption $\neg\texttt{borr}$ from the previous section.

□

Body Preserves Invariant This induction proof is very simple. With

$$\sum_r = \sum_b - \sum_l \tag{***}$$

as as assumption and we have to show the following statement:

$$\sum_r' = \sum_b' - \sum_l$$

We use the same strategy as above, pull out the first sum elements of the sums with prime, insert the assumption (***), and divide by w^i:

$$\begin{aligned}
\sum_r + R \cdot w^i &= \sum_b - \sum_l + B \cdot w^i \\
\sum_r + R \cdot w^i &= \sum_r + B \cdot w^i \\
R \cdot w^i &= B \cdot w^i \\
R &= B
\end{aligned}$$

The last line displays exactly the assignment of R from the code.

□

Use Case We have to show here that from the loop invariant after the final iteration and the negated loop condition the method's postcondition follows. The postcondition part about the array lengths holds instantly, since the array is only assigned once with exactly that length. Now the negated loop condition

$$\neg(bigIndex > 0)$$

becomes (we again use the fact that an array index is nonnegative):

$$bigIndex == 0$$

With that, our partial sums from the loop invariant

$$\sum_r = \sum_b - \sum_l$$

are sums over the total array range and thus equal to the definition of `simpleSum()`. So our postcondition

$$\texttt{simpleSum(r)} = \texttt{simpleSum(b)} - \texttt{simpleSum(l)}$$

holds.

□

We have seen that this method uses sophisticated bitshifting to calculate the borrow. In addition, the heap is updated frequently. The rest of the proof is relatively simple. If convenient techniques are found to deal with these problems, one should be able to perform the proof from this section in the KeY system.

3.7.7. `add()`

Though unfortunately it was not possible during this thesis to verify the following methods, we provide contracts here. The explanations why it was impossible to prove them are postponed here and can be found in Section 4.1.3.

The private `add()` method is very similar to the `subtract()` method. As that, it is a static helper method which assigns nothing on the heap (expect the parameters, which do not have to be explicitly mentioned). Of course the value represented by the resulting array has to be the sum of the values of the two parameter arrays. Differences to subtract are that the length of the result may be by 1 greater than that of the bigger parameter. Additionally, we do not have to presume something about the input arrays. The contract modeling this behavior is shown in Listing 3.23.

```
1 /*@ private normal_behavior
2   @ requires \static_invariant_for(BigInteger);
3   @ ensures simpleSum(\result) == simpleSum(x) + simpleSum(y);
4   @ ensures \result.length <= (x.length > y.length ? x.length : y.length) + 1;
5   @ assignable \nothing;
6   @*/
7 private static /*@ helper @*/ int[] add(int[] x, int[] y) { ... }
```

Listing 3.23: Method contract of the `add()` method

3.7.8. The Constructor `BigInteger()`

As explained in the comments provided by the developers in Listing 3.24, this is an internal constructor that assumes that its arguments are correct. Because of that, we have to include several preconditions: To make it possible to establish the instance invariants the given signum parameter has to be one of the three possible values, the magnitude array must not contain leading zeros, and a sign of 0 implies that the array is empty. Since the constructor succeeds only if the magnitude array is in the permitted bounds, we need to add this as a precondition. In exchange, the constructor ensures that the assignments of the parameters are done correctly (explicit part of postcondition), and that the instance invariant for the newly created object holds afterwards (implicit). In addition, it implicitly guarantees that the newly created object is fresh, i.e. that it does not coincide with any other object on the heap. Because only fields of the object are assigned, the constructor is considered as pure. The case when the array is too large to fit into a BigInteger and an `ArithmeticException` is thrown is modeled by the `exceptional_behavior` contract.

In the constructor we need to set our ghost field to a meaningful value. This is done in line 28 of Listing 3.24 via the `set` keyword and the model method `calcValue()`.

It at first seems to be a good idea to add an additional invariant about correct ranges of the magnitude (the statement shown in line 7). Since a BigInteger's value is final (neither signum nor mag are ever changed after construction) and exactly these bounds are checked at creation of each BigInteger object, this would be a suitable invariant. However, the JML standard and KeY prevent us from doing so because after an exception is thrown (actually any instance of `Throwable`, even `Error` and `RuntimeException`) the invariant has to hold also, even if the object is not completely constructed. In our case, there is no way to recover from the once thrown `ArithmeticException`, since the BigInteger object is not correctly created. However, Errors and RuntimeExceptions in Java are not meant to be caught as they indicate serious system or implementation problems. So the way JML specifies to treat those is questionable at least for constructors.

```
1  /*@ normal_behavior
2    @ requires (   magnitude.length < MAX_MAG_LENGTH
3    @           || magnitude.length == MAX_MAG_LENGTH && magnitude[0] >= 0)
4    @          && (signum == 0 || signum == 1 || signum == -1)
5    @          && (magnitude.length > 0 ==> magnitude[0] != 0)
6    @          && (signum == 0 ==> magnitude.length == 0);
7    @ ensures this.mag == magnitude;
8    @ ensures (magnitude.length == 0 ==> this.signum == 0) &&
9    @         (magnitude.length != 0 ==> this.signum == signum);
10   @ assignable \nothing;                              // special purity rules for constructors
11   @
12   @ also
13   @
14   @ exceptional_behavior
15   @ requires     magnitude.length > MAX_MAG_LENGTH
16   @           || magnitude.length == MAX_MAG_LENGTH && magnitude[0] < 0;
17   @ signals_only ArithmeticException;
18   @ signals (ArithmeticException e) magnitude.length > MAX_MAG_LENGTH
19   @                              || magnitude.length == MAX_MAG_LENGTH && magnitude[0] < 0;
20   @ assignable \nothing;                              // special purity rules for constructors
21   @*/
22 BigInteger(int[] magnitude, int signum) {
23   this.signum = (magnitude.length == 0 ? 0 : signum);
24   this.mag = magnitude;
25   if (mag.length >= MAX_MAG_LENGTH) {
26     checkRange();
27   }
28   /*@ set value = calcValue(this.signum, this.mag); @*/
29 }
```

Listing 3.24: Contracts and implementation of the constructor. Note the `set` statement in line 28.

3.7.9. `public add()`

This method is the connection between all the previous ones. It consists mainly of case distinctions and method calls and is only 13 lines long, since the real addition is done by the add and (if one summand is negative) subtract methods. Accordingly the contract (shown in Listing 3.25) is relatively simple.

```
1  /*@ public normal_behavior
2    @ requires val.<inv> && \static_invariant_for(BigInteger);
3    @ requires val.mag.length < MAX_MAG_LENGTH
4    @       || val.mag.length == MAX_MAG_LENGTH && val.mag[0] >= 0;
5    @ requires this.mag.length < MAX_MAG_LENGTH
6    @       || this.mag.length == MAX_MAG_LENGTH && this.mag[0] >= 0;
7    @ ensures \result.value == val.value + this.value;
8    @ assignable \nothing;
9    @*/
10 public BigInteger add(BigInteger val) { ... }
```

Listing 3.25: Contract of the `public add()` method

The actual functionality is now, thanks to our modeling, expressed very easily via the ghost fields (line 7). Line 2 contains the invariants as precondition, which of course have to hold, but must be included explicitly, since only the instance invariant of the object the method is called on is implicitly included. Line 3 to 6 contain additional information known for correct BigIntegers, which we (due to the reasons explained in the previous section) unfortunately can not include into our invariants.

4. Results and Conclusion

4.1. Verification Summary

In this section we provide a summary of the verification. At first we will display statistics to obtain an overview over the difficulty levels of the methods. After that we will summarize our experiences with the KeY system, which includes some suggestions for future development. Finally, we evaluate the class BigInteger itself and mention what made specification and verification of the real-world code difficult.

4.1.1. Proof Statistics

In this section we provide statistics for each method we considered. Based on that we will examine which properties of the methods have an impact on the degree of difficulty of the verification.

In general, the measure used for counting the proof steps are the numbers of total and interactive (abbreviated by tot. and int.) rule applications as displayed by KeY. Of course, this allows only an approximate evaluation of how difficult a proof is, since there are great differences in the difficulty of the rules. For example, an induction and a commutation of an equation count both as a single rule application, even though the latter is very simple while the former requires a large amount of considerations. Therefore, some additional measures like the number of loops as well as notes are given.

For each method or taclet we include the lines of code (LOC) of the definition as well as the lines of code needed for each single contract (abbreviated by LOC spec.). The specification lines are counted as given, i.e. each line is counted as long it does not solely contain comments. We have to note that the contracts may be split and parts of the specification may be counted twice. This happens for example if the same precondition clause is used for a functional as well as for a dependency contract. Then the line count of the precondition is added to the line count of both contracts.

4.1.1.1. User-Defined Taclets

As explained in Section 3.4, two user-defined taclets (inference rules of KeY) have been written and used in this thesis. While the proof of the second taclet is done on paper, the first taclet can be proved in KeY. Even if it looks like the proof is mainly done automatically, the opposite is the case: During the proof many nontrivial interactions such as induction and

quantifier instantiations are required. KeY afterwards automatically performs inequation normalization and closes the branches. We can already see in this simple example that reasoning about sums is difficult in KeY and requires a large amount of interaction.

	LOC definition	proof steps tot./int.	notes
`bsum_all_summands_gez`	4	523/30	
`bsum_estimation`	4	-/-	proof on paper (see Section 3.7.6)

Table 4.1.: Statistics of the user-defined taclets

4.1.1.2. Model Methods and Instance Invariant Dependency Contract

Due to a limitation in KeY method bodies of model methods may only consist of one single return statement. Therefore, it is meaningless to provide the number of lines in method body.

	LOC spec.	static/ helper	proof steps tot./int.	notes
`toUnsigned()`		✓/✓		
model	7		-/-[1]	verified with CBMC
model (injectivity)	2		146/-	for comparison, already proved in CBMC
dependency	1		55/-	
`twopower()`		✓/✓		
model	5		513/-	recursive method
dependency	2		101/-	
`calcValue()`		✓/✓		
model	7		17.705/52	custom taclets needed for sum estimation
dependency	3		624/3	
`firstNonZeroIndex()`	7	✓/✓	826/-	
`calcMagSum()`		✓/✓		
model	7		1.821/103	custom taclets needed for sum estimation
dependency	2		512/-	
`partialSum()` dep.	0	✓/✓	549/-	
`simpleSum()` dep.	1	✓/✓	-/-	just serves as abbreviation
instance inv. dep.	1		1.751/5	

Table 4.2.: Statistics of model methods and instance invariant

We can see that the dependency contracts always require much less steps than the functional contracts. Additionally, the two proofs where we needed our custom sum estimation taclet are very large. This is a general result: If we had to reason about sums in the proof, it was usually difficult and required many interactive steps.

The verification of the bitvector arithmetics in `toUnsigned()` was very easy and fast.

[1] The SAT-Solver gets a formula consisting of 2316 variables and 5475 clauses, which it is able to solve in a split second.

4.1.1.3. Java Methods

For the lines of Java code we count each line meaningful for the program. This includes code lines and brackets, but not empty lines and comments. We count code in the method body only.

Library Methods These library method are used from the code of BigInteger we consider. The first method as native and thus there is no Java code. Intentionally we did not want to prove the library methods but just write contracts for the use in BigInteger. However, as we tried to check whether the specification was correct, the proof of `copyOfRange()` arose as side product.

	LOC spec.	static	loops	proof steps tot./int.	notes
`System.arraycopy()`	9	✓	-	-/-	native, no proof
`Arrays.copyOfRange()`		✓	-		
normal	10			3.202/6	
exceptional	7			-/-	

Methods of BigInteger Here we can see that a loop notably increases the amount of proof steps needed. In addition to that, if we have to deal with sums, this highly increases the number of necessary proof steps (`compareMagnitude()`).

	LOC Java/JML	static/ helper	loops	proof steps tot./int.	notes
`reportOverflow()`	1/4	✓/✓	-	367/-	
`checkRange()`		X/X	-		
normal	3/3			398/-	
exceptional	3/5			505/4	
`strLZI()`[a]	5/8	✓/✓	1	7.856/-	
`tStrLZI()`[b]	5/7	✓/✓	1	8.840/-	
`comMag()`[c]	15/6	X/X	1	23.858/570	needs leammas from `calcMagSum()`
`subtract()`	16/6	✓/✓	3	-/-	loops are consecutive
`BigInteger()`	5/14	X/X	-	-/-	no proof
`add()`	31/5	✓/✓	3	-/	no proof
`public add()`	13/6	X/X	-	-/-	no proof

[a] `stripLeadingZeroInts()`
[b] `trustedStripLeadingZeroInts()`
[c] `compareMagnitude()`

Newly created objects in the method body make verification much more complicated due to different heap states. To sum up the results, we have specified 11 Java methods with a total of 216 specification lines written. In comparison, we verified 5 methods of BigInteger in KeY and therefore needed more than 66.500 verification steps of which were 773 interactive steps (about 1%). This includes the steps from model and library methods as well as the user-defined taclet, as they are needed for the verification of the BigInteger methods. Is is remarkably that nearly a third of the total steps is needed for the method `calcMagSum()`. Also, this method needs most of the interactive steps. This is due to complicated sum estimations and inequation system.

4.1.2. BigInteger Evaluation

In this section we provide a short evaluation, how compatible the BigInteger class is with formal specification and verification. However, the statements refer just to the parts of BigInteger considered in this thesis.

Since it is a math library class providing calculation functionality, the code is already written highly modular. None of the considered methods consists of more than 35 lines of code. This in general simplifies verification. On the other hand, there are drawbacks: Since the code was not written with formal verification in mind, there are some statements that are hard to read, especially regarding the actual subtraction and borrow calculation, which made it difficult for example to find appropriate loop invariants in Section 3.7.6. In addition, because the first digit of the magnitude array is the most significant, the index calculations for place value notation are relatively complicated. This became visible for example in Section 3.4, where we had to reorder the elements of the array for our induction proof. In general, the BigInteger code is reasonably well documented (comments in code and JavaDoc), even if some confusing or simply wrong comments exists. The documentation was of great help for specification.

4.1.3. Experiences with KeY and Development Ideas

This section summarizes the experiences with KeY and points out what was easy in KeY and where problems appeared.

The first to mention is the symbolic execution: It works very well and nearly fully automatically, regardless of the complexity of the Java statements. There was just one single case during the proofs when it could not be performed without interaction: In the constructor proof, when a set statement appeared in code. The simple reason of this is that there had been no rule for translating the model method in code correctly to JavaDL. This problem was recently fixed by the developers and can be expected to reach the master branch soon.

Another feature to point out is the update and statement simplification with the One Step Simplifier. It enables the user to perform multiple (often very many, in this thesis sometimes around 200) simplification steps for updates or first order formulas. One interesting point would be to enable the use of it not only for single formulas, but for multiple formulas at once. This could be a great time saver.

During the work on this thesis, we were frequently faced with situations when we would have wanted to apply a dependency contract to a bsum. It would be a nice feature to include a generic rule for bsum, i.e. checking whether the bsum result depends on a given location set, which can be reduced to checking whether any of the summands depends on it.

As seen in Section 3.7.5, KeY uses normalization and Gröbner bases to solve systems of inequations. This is a technique a human can not reasonably perform, as it requires a huge amount of steps and is very counterintuitive. A good idea could be to add rules that make interactive reasoning easier, for example an estimation rule. Such a version of KeY especially targeting interactive use is currently in development.

For some model methods, lemmas have been proved helpful, for example the lemma that `toUnsigned()` is injective in Section 3.3.2.1. If such a lemma is included in the method contract, it is necessary to insert the whole method contract and instantiate a quantifier every time it shall be used. An alternative approach to avoid that would be to create a user-defined taclet. However, this makes it impossible to prove the taclet, as in the taclet proof Java method definitions are not available (compare to Section 3.4). A possible

future improvement of KeY may be to provide the method definition axiom and make such proofs possible. This certainly requires some thoughts to not accidentally introduce circular dependencies into proofs.

During this case study, multiple bugs showed up, for example with saving and loading proofs, with the user interface, and with taclet side proofs.

If multiple dependent methods shall be proved, at small changes in specification of them often many proofs have to be repeated. At the moment, this is very inconvenient, because all invalidated proofs have to be found and replayed manually. A suggestion would be to implement a proof manager that takes respect of this dependencies and allows some kind of batch proving. Such a manager could even allow different tools for different contracts, for example it could include CBMC. Actually, there has been a batch prover for KeY in Eclipse (MONKeY), but it seems that this is not developed any more.

4.2. Future Work

BigInteger Verification With the corrections and improvements described in the previous paragraph, the proofs done by hand in this thesis could be transferred to a more formal basis and revised in KeY. Another big step would be moving from the overflow ignoring semantics of this thesis to checking overflows.

In this thesis the general modeling of the BigInteger semantics was done. With this as a foundation, future verification may include more complex methods of BigInteger as for example the multiplication method, which implements Karatsuba and Toom-Cook algorithms. There are many other methods of the BigInteger class which are frequently used and could be verified: For example the power method, division and modulo methods or primality tests (especially interesting for security applications). To conduct complete formal proofs of those would increase the security, as Javas security packages heavily rely on them. Since the BigInteger class is a very basic part of the Java Class Library, it can be expected to remain virtually unchanged in future Java versions. This additionally justifies to spend some time to its verification.

4.3. Conclusion

In this thesis real-world code from the Java Class Library has been specified and verified. Further verification of BigInteger is supposed to be possible in general, since no reason prohibiting it showed up. As became visible during this thesis, unfortunately some bugs and problems exist, which require workarounds and make the verification difficult and time-consuming. Additionally, it has been shown that it could be of great use to integrate reasoning about bitvectors to the KeY system, maybe by adding the possibility to interact with an existing tool like CBMC.

At the moment, KeY's main advantages are its flexibility and possibilities, while its main drawbacks reasoning about real-world code are usability and stability.

Bibliography

[Ahrendt et al., 2016] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M., editors (2016). *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer.

[Borgida et al., 1993] Borgida, A., Mylopoulos, J., and Reiter, R. (1993). "... and nothing else changes": The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 303–314, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Huisman et al., 2001] Huisman, M., Jacobs, B., and van den Berg, J. (2001). A case study in class library verification: Java's vector class. *International Journal on Software Tools for Technology Transfer*, 3(3):332–352.

[Leavens et al., 2013] Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D. M., and Dietl, W. (2013). Jml reference manual. http://www.eecs.ucf.edu/ leavens/JML/jmlrefman/jmlrefman.html. Draft Revision 2344.

[Schmitt and Tonin, 2007] Schmitt, P. H. and Tonin, I. (2007). Verifying the mondex case study. In *SEFM*, pages 47–58. IEEE Computer Society.

Appendix

A. Java Source Code

A.1. Arrays.java

```
1  package java.util;
2
3  public class Arrays {
4      /*@ public normal_behavior
5        @ requires 0 <= from && from <= original.length && from <= to && original != null;
6        @ ensures \result != original;                                    // always returns a new array
7        @ ensures \result.length == to - from;
8        @ ensures (to <= original.length)
9        @      ==> (\forall int i; from <= i && i < to; \result[i - from] == original[i]);
10       @ ensures (to > original.length)
11       @      ==> ((\forall int i; from <= i && i < original.length; \result[i - from] == original[i])
12       @          && (\forall int i; original.length <= i && i < to; \result[i - from] == 0));
13       @ assignable \nothing;
14       @
15       @ also
16       @
17       @ public exceptional_behavior
18       @ requires from < 0 || from > original.length || from > to || original == null;
19       @ signals (IllegalArgumentException e) from > to;
20       @ signals (ArrayIndexOutOfBoundsException e) from < 0 || from > original.length;
21       @ signals (NullPointerException e) original == null;
22       @ signals_only IllegalArgumentException, ArrayIndexOutOfBoundsException, NullPointerException;
23       @ assignable \nothing;
24       @*/
25     public static int[] copyOfRange(int[] original, int from, int to) {
26         int newLength = to - from;
27         if (newLength < 0)
28             throw new IllegalArgumentException(from + " > " + to);
29         int[] copy = new int[newLength];
30         System.arraycopy(original, from, copy, 0,
31                          Math.min(original.length - from, newLength));
32         return copy;
33     }
34 }
```

A.2. BigInteger.java

```
1  package java.math;
2
3  import java.util.Arrays;
4
5  public class BigInteger extends Number implements Comparable/*<BigInteger>*/ {
6      /**
7       * The signum of this BigInteger: -1 for negative, 0 for zero, or
8       * 1 for positive.  Note that the BigInteger zero <i>must</i> have
9       * a signum of 0.  This is necessary to ensures that there is exactly one
10      * representation for each BigInteger value.
11      *
12      * @serial
13      */
14     final int signum;
15
16     /**
17      * The magnitude of this BigInteger, in <i>big-endian</i> order: the
18      * zeroth element of this array is the most-significant int of the
19      * magnitude.  The magnitude must be "minimal" in that the most-significant
20      * int ({@code mag[0]}) must be non-zero.  This is necessary to
21      * ensure that there is exactly one representation for each BigInteger
22      * value.  Note that this implies that the BigInteger zero has a
23      * zero-length mag array.
24      */
25     final int[] mag;
26
27     //@ static invariant BigInteger.ZERO.<inv>;
28     //@ static invariant BigInteger.ZERO.value == 0;
29
30     //@ ghost \bigint value = 0;
31
32     /*@ invariant value == calcValue(signum, mag);
33       @ invariant signum == 0 || signum == 1 || signum == -1;
34       @ invariant signum == 0 <==> mag.length == 0;
35       @ invariant mag.length > 0 ==> mag[0] != 0;
36       @ accessible \inv: signum, mag, mag[*], value;
37       @*/
38
39     // no valid invariant (after RuntimeException inv. has to hold)
40     // @ invariant mag.length < MAX_MAG_LENGTH || mag.length == MAX_MAG_LENGTH && mag[0] >= 0;
41
42     // returns the value of the input integer as if it was unsigned
43     /*@ model_behavior
44       @ ensures value == 0 ==> \result == 0;
45       @ ensures value != 0 ==> \result > 0;
46       @ ensures value > 0 ==> \result == value;
47       @ ensures value < 0 ==> \result == value + 0x100000000L;
48       @ ensures \result >= 0;
49       @ ensures \result < 0x100000000L;
50       @ ensures (\forall int i; value != i ==> \result != toUnsigned(i));    // injective function
51       @ accessible \nothing;
52       @ static helper model long toUnsigned(int value) {
53       @     return (long)value & 0xffffffffL;
54       @ }
55       @*/
56
57     // returns 2^(32*exp) = (2^(32))^exp
58     /*@ model_behavior
59       @ requires exp >= 0;
60       @ ensures \result >= 1;
61       @ ensures (\forall int i; 0 <= i && i < exp; twopower(i) < \result);   // strictly increasing
62       @ measured_by exp;
63       @ accessible \nothing;
```

```
64        @ static helper model \bigint twopower(int exp) {
65        @     return exp == 0 ? 1 : twopower(exp - 1) * 0x100000000L;
66        @ }
67        @*/
68
69      // returns the value of the BigInteger as \bigint
70      /*@ model_behavior
71        @ requires m.length > 0 ==> m[0] != 0;
72        @ requires s == 0 <==> m.length == 0;
73        @ ensures \result
74        @         == s*(\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
75        @ ensures s < 0 ==> \result < 0;
76        @ ensures s == 0 ==> \result == 0;
77        @ ensures s > 0 ==> \result > 0;
78        @ accessible m[*];    // m.length is a function in KeY, not a field, and thus not included here
79        @ static helper model \bigint calcValue(int s, int[] m) {
80        @     return s*(\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
81        @ }
82        @*/
83
84      /**
85       * This constant limits {@code mag.length} of BigIntegers to the supported
86       * range.
87       */
88      private static final int MAX_MAG_LENGTH = Integer.MAX_VALUE / Integer.SIZE + 1; // (1 << 26)
89
90      /**
91       * This mask is used to obtain the value of an int as if it were unsigned.
92       */
93      final static long LONG_MASK = 0xffffffffL;
94
95      /**
96       * The BigInteger constant zero.
97       *
98       * @since   1.2
99       */
100     public static final BigInteger ZERO = new BigInteger(new int[0], 0);
101
102     /**
103      * This internal constructor differs from its public cousin
104      * with the arguments reversed in two ways: it assumes that its
105      * arguments are correct, and it doesn't copy the magnitude array.
106      */
107     /*@ normal_behavior
108       @ requires (   magnitude.length < MAX_MAG_LENGTH
109       @           || magnitude.length == MAX_MAG_LENGTH && magnitude[0] >= 0)
110       @          && (signum == 0 || signum == 1 || signum == -1)
111       @          && (magnitude.length > 0 ==> magnitude[0] != 0)
112       @          && (signum == 0 ==> magnitude.length == 0);
113       @ ensures this.mag == magnitude;
114       @ ensures (magnitude.length == 0 ==> this.signum == 0) &&
115       @         (magnitude.length != 0 ==> this.signum == signum);
116       @ assignable \nothing;                              // special purity rules for constructors
117       @
118       @ also
119       @
120       @ exceptional_behavior
121       @ requires     magnitude.length > MAX_MAG_LENGTH
122       @           || magnitude.length == MAX_MAG_LENGTH && magnitude[0] < 0;
123       @ signals_only ArithmeticException;
124       @ signals (ArithmeticException e) magnitude.length > MAX_MAG_LENGTH
125       @                              || magnitude.length == MAX_MAG_LENGTH && magnitude[0] < 0;
126       @ assignable \nothing;                              // special purity rules for constructors
127       @*/
```

```
128     BigInteger(int[] magnitude, int signum) {
129         this.signum = (magnitude.length == 0 ? 0 : signum);
130         this.mag = magnitude;
131         if (mag.length >= MAX_MAG_LENGTH) {
132             checkRange();
133         }
134         /*@ set value = calcValue(this.signum, this.mag); @*/
135     }
136 
137     /**
138      * Returns a BigInteger whose value is {@code (this + val)}.
139      *
140      * @param  val value to be added to this BigInteger.
141      * @return {@code this + val}
142      */
143     /*@ public normal_behavior
144       @ requires val.<inv> && \static_invariant_for(BigInteger);
145       @ requires val.mag.length < MAX_MAG_LENGTH
146       @       || val.mag.length == MAX_MAG_LENGTH && val.mag[0] >= 0;
147       @ requires this.mag.length < MAX_MAG_LENGTH
148       @       || this.mag.length == MAX_MAG_LENGTH && this.mag[0] >= 0;
149       @ ensures \result.value == val.value + this.value;
150       @ assignable \nothing;
151       @*/
152     public BigInteger add(BigInteger val) {
153         if (val.signum == 0)
154             return this;
155         if (signum == 0)
156             return val;
157         if (val.signum == signum)
158             return new BigInteger(add(mag, val.mag), signum);
159 
160         int cmp = compareMagnitude(val);
161         if (cmp == 0)
162             return ZERO;
163         int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
164                            : subtract(val.mag, mag));
165         resultMag = trustedStripLeadingZeroInts(resultMag);
166 
167         return new BigInteger(resultMag, cmp == signum ? 1 : -1);
168     }
169 
170     /**
171      * Adds the contents of the int arrays x and y. This method allocates
172      * a new int array to hold the answer and returns a reference to that
173      * array.
174      */
175     /*@ private normal_behavior
176       @ requires \static_invariant_for(BigInteger);
177       @ ensures simpleSum(\result) == simpleSum(x) + simpleSum(y);
178       @ ensures \result.length <= (x.length > y.length ? x.length : y.length) + 1;
179       @ assignable \nothing;
180       @*/
181     private static /*@ helper @*/ int[] add(int[] x, int[] y) {
182         // If x is shorter, swap the two arrays
183         if (x.length < y.length) {
184             int[] tmp = x;
185             x = y;
186             y = tmp;
187         }
188 
189         int xIndex = x.length;
190         int yIndex = y.length;
191         int result[] = new int[xIndex];
```

```
192         long sum = 0;
193         if (yIndex == 1) {
194             sum = (x[--xIndex] & LONG_MASK) + (y[0] & LONG_MASK) ;
195             result[xIndex] = (int)sum;
196         } else {
197             // Add common parts of both numbers
198             while (yIndex > 0) {
199                 sum = (x[--xIndex] & LONG_MASK) +
200                       (y[--yIndex] & LONG_MASK) + (sum >>> 32);
201                 result[xIndex] = (int)sum;
202             }
203         }
204         // Copy remainder of longer number while carry propagation is required
205         boolean carry = (sum >>> 32 != 0);
206         while (xIndex > 0 && carry)
207             carry = ((result[--xIndex] = x[xIndex] + 1) == 0);
208 
209         while (xIndex > 0)
210             result[--xIndex] = x[xIndex];
211 
212         // Grow result if necessary
213         if (carry) {
214             int bigger[] = new int[result.length + 1];
215             System.arraycopy(result, 0, bigger, 1, result.length);
216             bigger[0] = 0x01;
217             return bigger;
218         }
219         return result;
220     }
221 
222     // no assumptions for zeros, parameter and result may have arbitrary amount of leading zeros
223     /*@ private static helper model \bigint simpleSum(int[] m) {
224       @     return (\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
225       @ }
226       @*/
227 
228     // partial sum of the array, starting from startIndex (inclusive)
229     /*@ private model_behavior
230       @ accessible m[startIndex .. (m.length-1)];
231       @ private static helper model \bigint partialSum(int[] m, int startIndex) {
232       @     return (\sum int i;
233       @                 startIndex <= i && i < m.length;
234       @                 toUnsigned(m[i]) * twopower(m.length-i-1));
235       @ }
236       @*/
237 
238     /**
239      * Subtracts the contents of the second int arrays (little) from the
240      * first (big).  The first int array (big) must represent a larger number
241      * than the second.  This method allocates the space necessary to hold the
242      * answer.
243      */
244     /*@ private normal_behavior
245       @ requires calcMagSum(big) > calcMagSum(little);
246       @ requires (big.length > 0 ==> big[0] != 0) && (little.length > 0 ==> little[0] != 0);
247       @ ensures simpleSum(\result) == simpleSum(big) - simpleSum(little);   // may have leading zeros
248       @ assignable \nothing;
249       @*/
250     private static /*@ helper @*/ int[] subtract(int[] big, int[] little) {
251         int bigIndex = big.length;
252         int result[] = new int[bigIndex];
253         int littleIndex = little.length;
254         long difference = 0;
255 
```

```
256         // Subtract common parts of both numbers
257         /*@ loop_invariant 0 <= littleIndex && littleIndex <= little.length
258           @             && 0 <= bigIndex && bigIndex <= big.length
259           @             && big.length - bigIndex == little.length - littleIndex
260           @             && partialSum(result, bigIndex)
261           @                + (difference>>32)*twopower(big.length-bigIndex) // borrow from next digit
262           @               == partialSum(big, bigIndex)
263           @                - partialSum(little, littleIndex);
264           @ assignable result[(result.length-little.length)..(result.length-1)];
265           @ decreases littleIndex;
266           @*/
267         while (littleIndex > 0) {
268             difference = (big[--bigIndex] & LONG_MASK) -
269                          (little[--littleIndex] & LONG_MASK) +
270                          (difference >> 32);
271             result[bigIndex] = (int)difference;
272         }
273 
274         // Subtract remainder of longer number while borrow propagates
275         boolean borrow = (difference >> 32 != 0);
276         /*@ loop_invariant 0 <= bigIndex && bigIndex <= big.length
277           @                && partialSum(result, bigIndex)
278           @                 == partialSum(big, bigIndex) - partialSum(little, 0)
279           @                    + (borrow ? twopower(big.length-bigIndex) : (\bigint)0);
280           @ assignable result[*];
281           @ decreases bigIndex;
282           @*/
283         while (bigIndex > 0 && borrow)
284             borrow = ((result[--bigIndex] = big[bigIndex] - 1) == -1);
285             /* --bigIndex;
286              * result[bigIndex] = big[bigIndex] - 1;
287              * borrow = (big[bigIndex] == 0);
288              */
289 
290         // Copy remainder of longer number
291         /*@ loop_invariant bigIndex >= 0
292           @             && partialSum(result, bigIndex) == partialSum(big, bigIndex)
293           @                                              - partialSum(little, 0);
294           @ assignable result[*];
295           @ decreases bigIndex;
296           @*/
297         while (bigIndex > 0)
298             result[--bigIndex] = big[bigIndex];
299 
300         return result;
301     }
302 
303     /**
304      * Throws an {@code ArithmeticException} if the {@code BigInteger} would be
305      * out of the supported range.
306      *
307      * @throws ArithmeticException if {@code this} exceeds the supported range.
308      */
309     /*@ private normal_behavior
310       @ requires mag.length <= MAX_MAG_LENGTH
311       @      && (mag.length != MAX_MAG_LENGTH || mag[0] >= 0);
312       @ assignable \nothing;
313       @
314       @ also
315       @
316       @ private exceptional_behavior
317       @ requires mag.length > MAX_MAG_LENGTH
318       @       || mag.length == MAX_MAG_LENGTH && mag[0] < 0;
319       @ signals (ArithmeticException e) mag.length > MAX_MAG_LENGTH
```

```
320       @                              || mag.length == MAX_MAG_LENGTH && mag[0] < 0;
321       @ signals_only ArithmeticException;
322       @ assignable \nothing;
323       @*/
324     private void checkRange() {
325         if (mag.length > MAX_MAG_LENGTH || mag.length == MAX_MAG_LENGTH && mag[0] < 0) {
326             reportOverflow();
327         }
328     }
329
330     /*@ private exceptional_behavior
331       @ signals (ArithmeticException e) true;
332       @ signals_only ArithmeticException;
333       @ assignable \nothing;
334       @*/
335     private static /*@ helper @*/ void reportOverflow() {
336         throw new ArithmeticException("BigInteger would overflow supported range");
337     }
338
339     /*@ private model_behavior
340       @ requires m.length > 0 ==> m[0] != 0;
341       @ ensures \result
342       @         == (\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
343       @ ensures m.length > 0 ==> \result > 0;
344       @ ensures \result >= 0;
345       @ ensures (\forall int[] m2;                  // A: a.length < b.length -> a.result < b.result
346       @             0 <= m2.length && m2.length < m.length && (m2.length > 0 ==> m2[0] != 0);
347       @             calcMagSum(m2) < \result);
348       @ ensures (\forall int[] m3;                  // B: equal lengths and elements -> equal results
349       @             m3.length == m.length && (\forall int j; 0 <= j && j < m.length; m3[j] == m[j]);
350       @             \result == calcMagSum(m3));
351       @ accessible m[*];
352       @ private static helper model \bigint calcMagSum(int[] m) {
353       @     return (\sum int i; 0 <= i && i < m.length; toUnsigned(m[i]) * twopower(m.length-i-1));
354       @ }
355       @*/
356
357     /**
358      * Compares the magnitude array of this BigInteger with the specified
359      * BigInteger's. This is the version of compareTo ignoring sign.
360      *
361      * @param val BigInteger whose magnitude array to be compared.
362      * @return -1, 0 or 1 as this magnitude array is less than, equal to or
363      *         greater than the magnitude aray for the specified BigInteger's.
364      */
365     /*@ normal_behavior
366       @ requires val.<inv>;
367       @ ensures calcMagSum(val.mag) == calcMagSum(mag) ==> \result == 0;
368       @ ensures calcMagSum(val.mag) > calcMagSum(mag) ==> \result == -1;
369       @ ensures calcMagSum(val.mag) < calcMagSum(mag) ==> \result == 1;
370       @ assignable \nothing;
371       @*/
372     final int compareMagnitude(BigInteger val) {
373         int[] m1 = mag;
374         int len1 = m1.length;
375         int[] m2 = val.mag;
376         int len2 = m2.length;
377         if (len1 < len2)
378             return -1;
379         if (len1 > len2)
380             return 1;
381
382         /*@ loop_invariant 0 <= i && i <= len1 && (\forall int j; 0 <= j && j < i; m1[j] == m2[j]);
383           @ assignable \nothing;
```

```
384           @ decreases len1 - i;
385           @*/
386         for (int i = 0; i < len1; i++) {
387             int a = m1[i];
388             int b = m2[i];
389             if (a != b)
390                 return ((a & LONG_MASK) < (b & LONG_MASK)) ? -1 : 1;
391         }
392         return 0;
393     }
394
395     /*@ private model_behavior
396       @ requires arr.length > 0;                                        // case 1: arr = {}
397       @ requires (\exists int i; 0 <= i && i < arr.length; arr[i] != 0); // case 2: arr = {0 .. 0}
398       @ ensures \result >= 0;
399       @ ensures \result < arr.length && arr[\result] != 0
400       @      && (\forall int i; 0 <= i && i < \result; arr[i] == 0);     // case 3: arr = {x1 .. xn}
401       @ private static helper model int firstNonZeroIndex(int[] arr);
402       @*/
403
404     /**
405      * Returns a copy of the input array stripped of any leading zero bytes.
406      */
407     /*@ private normal_behavior
408       @ ensures \result != val;                          // difference to trustedStripLeadingZeroInts
409       @ ensures \result.length <= val.length;
410       @ ensures val.length == 0 ==> result.length == 0;             // case 1: arr = {}
411       @ ensures (\forall int i; 0 <= i && i < val.length; val[i] == 0)
412       @                 ==> \result.length == 0;                    // case 2: arr = {0 .. 0}
413       @ ensures (\exists int i; 0 <= i && i < val.length; val[i] != 0)
414       @                 ==> \result[0] != 0;                        // case 3: arr = {x1 .. xn}
415       @ ensures (\forall int i; 0 <= i && i < \result.length;
416       @             \result[i] == val[firstNonZeroIndex(val) + i]); // copy of old values, same order
417       @ assignable \nothing;
418       @*/
419     private static /*@ helper @*/ int[] stripLeadingZeroInts(int val[]) {
420         int vlen = val.length;
421         int keep;
422
423         // Find first nonzero int
424         /*@ loop_invariant 0 <= keep && keep <= vlen
425           @             && (\forall int i; 0 <= i && i < keep; val[i] == 0);
426           @ assignable \strictly_nothing;
427           @ decreases vlen - keep;
428           @*/
429         for (keep = 0; keep < vlen && val[keep] == 0; keep++)
430             ;
431         return java.util.Arrays.copyOfRange(val, keep, vlen);
432     }
433
434     /**
435      * Returns the input array stripped of any leading zero bytes.
436      * Since the source is trusted the copying may be skipped.
437      */
438     /*@ private normal_behavior
439       @
440       @ ensures \result.length <= val.length;
441       @ ensures val.length == 0 ==> result.length == 0;             // case1: arr = {}
442       @ ensures (\forall int i; 0 <= i && i < val.length; val[i] == 0)
443       @                 ==> \result.length == 0;                    // case2: arr = {0 .. 0}
444       @ ensures (\exists int i; 0 <= i && i < val.length; val[i] != 0)
445       @                 ==> \result[0] != 0;                        // case3: arr = {x1 .. xn}
446       @ ensures (\forall int i; 0 <= i && i < \result.length;
447       @             \result[i] == val[firstNonZeroIndex(val) + i]); // copy of old values, same order
```

```
448       @ assignable \nothing;
449       @*/
450     private static /*@ helper @*/ int[] trustedStripLeadingZeroInts(int val[]) {
451         int vlen = val.length;
452         int keep;
453 
454         // Find first nonzero int
455         /*@ loop_invariant 0 <= keep && keep <= vlen
456           @             && (\forall int i; 0 <= i && i < keep; val[i] == 0);
457           @ assignable \strictly_nothing;
458           @ decreases vlen - keep;
459           @*/
460         for (keep = 0; keep < vlen && val[keep] == 0; keep++)
461             ;
462         return keep == 0 ? val : java.util.Arrays.copyOfRange(val, keep, vlen);
463     }
464 }
```

A.3. Comparable.java

```
1  package java.lang;
2  import java.util.*;
3 
4  public interface Comparable/*<T>*/ {
5      public int compareTo(/*T*/ Object o);
6  }
```

A.4. System.java

```
1  package java.lang;
2 
3  public final class System {
4      // remarks: assume src and dest to be int[], src != dest, no exceptional
5      // -> should be enough for our use case
6      /*@ public normal_behavior
7        @ requires src instanceof int[] && dest instanceof int[];
8        @ requires src != dest;
9        @ requires srcPos >= 0 && destPos >= 0;
10       @ requires length >= 0;
11       @ requires srcPos + length <= ((int[])src).length && destPos + length <= ((int[])dest).length;
12       @ ensures (\forall int i; 0 <= i && i < length;
13       @             ((int[])dest)[destPos + i] == ((int[])src)[srcPos + i]);
14       @ assignable \strictly_nothing;
15       @*/
16     public static native void arraycopy(Object src,  int  srcPos,
17                                          Object dest, int destPos,
18                                          int length);
19 }
```

B. Settings of KeY

```
1  [Strategy]Timeout=-1
2  [Strategy]MaximumNumberOfAutomaticApplications=7000
3  [Strategy]ActiveStrategy=JavaCardDLStrategy
4 
5  [StrategyProperty]STOPMODE_OPTIONS_KEY=STOPMODE_DEFAULT
6  [StrategyProperty]SPLITTING_OPTIONS_KEY=SPLITTING_DELAYED
7  [StrategyProperty]LOOP_OPTIONS_KEY=LOOP_INVARIANT
8  [StrategyProperty]BLOCK_OPTIONS_KEY=BLOCK_CONTRACT
9  [StrategyProperty]METHOD_OPTIONS_KEY=METHOD_CONTRACT
```

```
[StrategyProperty]DEP_OPTIONS_KEY=DEP_ON
[StrategyProperty]QUERY_NEW_OPTIONS_KEY=QUERY_ON
[StrategyProperty]QUERYAXIOM_OPTIONS_KEY=QUERYAXIOM_ON
[StrategyProperty]NON_LIN_ARITH_OPTIONS_KEY=NON_LIN_ARITH_DEF_OPS
[StrategyProperty]QUANTIFIERS_OPTIONS_KEY=QUANTIFIERS_NON_SPLITTING_WITH_PROGS
[StrategyProperty]CLASS_AXIOM_OPTIONS_KEY=CLASS_AXIOM_DELAYED
[StrategyProperty]AUTO_INDUCTION_OPTIONS_KEY=AUTO_INDUCTION_OFF
[StrategyProperty]USER_TACLETS_OPTIONS_KEY1=USER_TACLETS_OFF
[StrategyProperty]USER_TACLETS_OPTIONS_KEY2=USER_TACLETS_OFF
[StrategyProperty]USER_TACLETS_OPTIONS_KEY3=USER_TACLETS_OFF
[StrategyProperty]INF_FLOW_CHECK_PROPERTY=INF_FLOW_CHECK_FALSE

[Choice]DefaultChoices=assertions-assertions\:on,
initialisation-initialisation\:disableStaticInitialisation,
intRules-intRules\:arithmeticSemanticsIgnoringOF,
programRules-programRules\:Java,
runtimeExceptions-runtimeExceptions\:ban,
JavaCard-JavaCard\:off,
Strings-Strings\:on,
modelFields-modelFields\:treatAsAxiom,
bigint-bigint\:on,
sequences-sequences\:on,
moreSeqRules-moreSeqRules\:off,
reach-reach\:on,
integerSimplificationRules-integerSimplificationRules\:full,
wdOperator-wdOperator\:L,
wdChecks-wdChecks\:off,
permissions-permissions\:off,
joinGenerateIsWeakeningGoal-joinGenerateIsWeakeningGoal\:off
```

Listing 4.1: Settings of KeY as displayed by `"Proof"` → `"Show All Active Settings"` (reordered and shortened for better readability).

C. User-Defined Taclets

```
// user-defined Taclets for the BigInteger verification case study

\schemaVariables {
  \term int i0,i1,t,a,b;
  \term Heap h;
  \variables int uSub;
}

\rules(integerSimplificationRules:full){

\lemma
bsum_all_summands_gez {
    \find(bsum{uSub;}(i0,i1,t))
    \varcond(\notFreeIn(uSub, i0, i1))
    "Precondition": \add(==> \forall uSub; (i0 <= uSub & uSub < i1 -> t >= 0));
    "Use Case": \add(bsum{uSub;}(i0,i1,t) >= 0 ==>)
};

bsum_estimation {
    \find(bsum{uSub;}(a+1,b,(java.math.BigInteger::toUnsigned(h,t)
                          * java.math.BigInteger::twopower(h,b-uSub-1))))
    \varcond(\notFreeIn(uSub, a,b,h))
    "Valid Indices": \add(==> b >= a+1 & a+1 >= 0);
    "Use Case": \add( bsum{uSub;}(a+1,b,java.math.BigInteger::toUnsigned(h,t)
                                      * java.math.BigInteger::twopower(h,b-uSub-1))
                    < java.math.BigInteger::twopower(h,b-a-1) ==>)
};

```

```
29 bsum_estimation_2 {
30     \find(bsum{uSub;}(a,b,(java.math.BigInteger::toUnsigned(h,t)
31                           * java.math.BigInteger::twopower(h,b-uSub-1))))
32     \varcond(\notFreeIn(uSub, a,b,h))
33     "Valid Indices": \add(==> b >= a & a >= 0);
34     "Use Case": \add( bsum{uSub;}(a,b,java.math.BigInteger::toUnsigned(h,t)
35                                       * java.math.BigInteger::twopower(h,b-uSub-1))
36                     < java.math.BigInteger::twopower(h,b-a) ==>)
37 };
38
39 }
```