Specifying and Verifying a Decimal Representation in Java for Smart Cards*

Cees-Bart Breunesse, Bart Jacobs, Joachim van den Berg

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{ceesb,bart,joachim}@cs.kun.nl

February 5, 2002

Abstract

This article describes a case study concerning a component of a Java Purse applet developed by the smart card manufacturer Gemplus. This component is a representation of decimal numbers in Java. The decimal component is annotated with specifications consisting of invariants and pre- and post-conditions, describing the functional behaviour. These specifications are written in the specification language JML. After translation of the source code and the annotations to the language of the theorem prover PVS, the correctness of these annotations is proved interactively.

1 Introduction

The direct topic of this article is a case study in program specification and verification. As such it gives an impression of the state of affairs in tool-assisted verification of realistic programs—in the world of Java smart cards. Also, it describes some of the crucial ingredients of the actual verification process. Such a case study may be interesting, but is in itself of limited interest. The indirect topic, however, is the impact that such specification and verification efforts may have on program development and (ultimately) on certification of smart card applets. In brief, the message is that actual program verification:

- is becoming feasible for Java smart card applets, enabling higher levels of certification and competitive advantages for the manufacturers that apply such techniques;
- does not only lead to correct but also to optimised code—which is very relevant on a smart card with restricted resources. Typically, when class invariants are made explicit, they can be assumed to hold when methods are called, so that various checks in the code can be dropped safely (and provably so). This is not a new insight, see for instance [11, Section 11.6], but one which is worth re-emphasising with a concrete illustration.

Smart cards receive much attention in Europe—mostly as bank cards or as SIMs in GSM cell phones—but recently also in other parts of the world, for instance as general identity card. The European VerifiCard project aims to develop techniques for establishing the correctness for crucial

*The research presented forms part of the European IST project *VerifiCard*, see `www.verificard.org`.

components of the JavaCard platform and for application programs (applets). Such techniques are needed for the evaluation of smart cards at higher levels (within the Common Criteria framework). Evaluation at a high level is for instance required to give a digital signature with a smart card appropriate legal status.

JavaCard is a "superset of a subset" of Java, designed by SUN for programming smart cards: it is a subset in the sense that for example threads, floats, strings, multi-dimensional arrays are not supported. The JavaCard API is also a subset of Java's API, because packages like AWT with GUI classes are obviously not needed. The JavaCard API does have some additional classes, because it also serves as an interface to the operating system of the smart card. These specifics are of no concern to us here, so we use JavaCard as just a stripped-down version of Java.

Within the VerifiCard project, example applets are provided by industrial parties. This article, concentrates on the JavaCard Purse applet by Gemplus (see [1]). This electronic purse is a real-life example of a banking application. It consists of a global balance, the operations to modify this balance, but also a cryptographic protection mechanism to authenticate and authorize, and a pincode mechanism.

The case study we present in this article considers only a small part of the Gemplus electronic purse, as the complete applet is 65K of source code. The code is meant to run on smart cards and should be robust but most importantly: it should act according to its specification. One of the many classes in the purse is the Decimal class which provides the numeric representation of a decimal number, e.g. the balance. The Decimal class also contains methods to modify the balance, for example adding, subtracting or multiplying. A balance and methods to modify this balance might not seem an interesting verification challenge at first, but one should keep in mind that in JavaCard there are no floating point numbers available. This means that decimal numbers must be represented in a different fashion. Gemplus solved this problem by taking two shorts (the largest primitive type available in JavaCard) that together make up the decimal number. Also interesting about the Decimal class is that it is an essential component of the entire Purse, with some interesting computational content. Its (in)correctness thus affects other operations throughout the entire applet. All this makes it a good choice on which to concentrate one's specification and verification efforts.

The Loop group in Nijmegen works on program specification and verification of Java source code. Verification is done within the theorem prover PVS (see [15]). Getting Java source code and the specifications for Java ready to be verified inside this theorem prover happens via a (shallow) embedding of the Java language and the specification language JML (see [10]) in PVS. For this purpose a compiler called the Loop-tool has been developed which does this translation automatically (see [3]). (A deep embedding of Java into Isabelle has been developed in Munich [14, 13].) The proof obligations in PVS can then be rewritten and simplified by using appropriate, provably sound Hoare rules (see [9]). A related approach [12] uses Hoare logic at the Java syntax level to generate verification conditions for PVS.

Proving correctness of Java methods is proving that methods act according to their specification. The first step in proving correctness of methods is thus writing specifications. The way we specify Java source code is by using the annotation language JML (see [16]). With JML we write pre- and post-conditions for methods, invariants for classes and other clauses to model, for example, exceptional behavior. These annotations are added to the code as a special kind of Java comments. One of the attractive featurs that JML offers is the wide range of tools, for systematic testing, static checking, or formal verification.

In a companion paper to ours [7, 6] the ESC/Java tool (see [17]) is applied to the same Purse case study, using their own specifications[1]. By statically analyzing the source code with ESC/Java, they made many implementation details explicit, such as input checking on parameters, and invariants on the fields. An advantage of this approach is that ESC/Java automatically checks whether

[1]Typically for ESC/Java, all postconditions are simply True. In our case we have non-trival postconditions describing the functional behaviour of the methods at hand.

```
/*@ normal_behavior
  @     requires true;
  @  modifiable intPart, decPart;
  @      ensures intPart == -\old(intPart) &&
  @              decPart == -\old(decPart);
  @*/
public Decimal oppose() {
   intPart = (short) -intPart;
   decPart = (short) -decPart;
   return this;
}
```

Figure 1: Basic JML annotated example: `oppose()`.

the implementation satisfies its specification by iterating the method bodies a number of times. However, this approach is both unsound and incomplete, and, therefore, it will not guarantee that the implementation satisfies its specification in all cases. In other words, it is not a formal proof.

The ideal way to combine these two approaches is to first use ESC/Java to detect the most common programming errors, and then to use the LOOP tool and PVS to actually verify the most critical fragments of the code.

This paper is organised as follows. In the first section below we familiarize the reader with JML. In Section 2 we dive into the specifications of the Decimal class, and in particular the specification of the method for adding two Decimals in Section 2.1. After specification comes verification, so in Section 3 we shortly discuss how the translation from Java annotated with JML to PVS is achieved. In Section 4 we present the verification of the method for addition. In Section 5 we discuss two other interesting methods in the Decimal class: the method for multiplying two Decimals and the method for rounding.

1.1 Specifying Java source code

This section explains some ins and outs of JML by taking simple parts of the Decimal class and annotating them with JML. First a short note on the Decimal class itself. A decimal number is represented by an object of the Decimal class. A Decimal object has two short fields that together form the actual representation of this decimal number. These shorts are named *intPart* and *decPart*. The first one, *intPart*, represents the integer part. The second one, *decPart* represents the decimal part. Thus, the decimal number 9.715 is represented by an object of class Decimal where *intPart = 9* and *decPart = 715*.

In Figure 1 the source code and annotations for method *oppose* from class Decimal is depicted. This method changes the sign of the Decimal by changing the sign of the two fields in the Decimal object that make up the decimal.

The code with the JML specification in Figure 1 can be compiled by a normal Java compiler because the annotations are within Java comments. Note that we use a "@" to distinguish JML annotations from normal comments. The specification for method *oppose* reads as follows: we make no specific assumptions on the pre-condition of method *oppose*, so we write *requires true;*. After execution and guaranteed normal termination of *oppose* it is ensured that *intPart* and *decPart* are negated. This is shown in the post-condition, labeled *ensures*. We are able to refer to the "old" values of *intPart* and *decPart* before execution of *oppose* by writing *old(intPart)* and *old(decPart)* respectively. The purpose of the clause called *modifiable* is to maintain a list of all fields that can be modified by *oppose*.

The annotation language JML has far more features than we use in the case study, so let us

```
    private DecimalException decimal_exception
2               = new DecimalException();

4 //@ invariant decimal_exception != null;

6
  /*@ behavior
8   @    requires d != null;
    @  modifiable intPart, decPart;
10  @     ensures    intPart == d.intPart
    @             && decPart == d.decPart;
12  @     signals  (DecimalException e)
    @                 d.intPart < 0;
14  @*/
    public Decimal setValue(Decimal d)
16    throws DecimalException{
        return setValue(d.getIntPart(),d.getDecPart());
18  }
```

Figure 2: Example showing an invariant and exceptional termination: `setValue(Decimal).`

consider another part of the Decimal class in which all JML concepts relevant for the case study are used.

Method *setValue* in Figure 2 copies the decimal value of its argument to the current Decimal object by using a private helper method, also called *setValue* (the code of this helper method is not shown). We use this example to introduce the JML clauses *invariant* and *signals*. This helper method of *setValue* may throw an exception for reasons not discussed here. What follows from the code of the helper function is that this exception is always the same *decimal_exception* object. It is impossible to throw a *null* object, so it is a good idea to create an invariant in which we enforce the exception object to be a reference. This is expressed by the JML clause labeled *invariant* at line 4. A class invariant should be maintained by every method in that class, which means that after execution of *setValue* the invariant should hold, assuming it holds before execution.

Now consider the specification for *setValue*. Note that the first keyword of the specification is *behavior* instead of *normal_behavior* as we saw in the specification for *oppose* in Figure 1. This is because methods in Java do not always terminate normally, but can also hang or throw an exception. When throwing an exception, the *signals* clause is used to specify the state in which we end up after throwing the exception. The first argument of the *signals* clause is the exception object itself (e), the second argument is the predicate that must be satisfied ($d.intPart < 0$). So, when proving the correctness of method *setValue* we must distinguish between normal and exceptional behavior. If we know that some method does not throw an exception and will terminate normally, for example the method *oppose* in Figure 1, we can skip the *signals* clause by starting the specification with *normal_behavior*. Note that in both normal and exceptional termination, we must establish the invariant. In JML we can also specify what conditions cause a method to hang, but as hanging methods do not occur in the case study, we ignore this.

The predicates used in the *invariant, requires, ensures* and *signals* clauses resemble Java boolean expressions. This is convenient in the way that one's specification is written using basically the same syntax as the source code itself.

Now that we have enough knowledge of JML, let us take a look the Decimal class in a broader perspective.

```
   public class Decimal extends Object {
2      public static final short PRECISION;
       private DecimalException decimal_exception;
4      private short intPart;
       private short decPart;
6      public Decimal add(Decimal) throws DecimalException;
       public Decimal sub(Decimal) throws DecimalException;
8      public Decimal mul(Decimal) throws DecimalException;
       public Decimal round();
10     private void add(short, short);
       private void mul(short, short);
12     public Decimal setValue(Decimal);
       public Decimal oppose();
14 }
```

Figure 3: Discussed fields and methods of the Decimal class

2 Decimal class specification

The Decimal class source code was handed to us without any formal specification. For method names like "add" or "mul" it is luckily not very problematic to guess the intended functionality. In Figure 3 we list the members of the Decimal class which we discuss or discussed, including the ones in Figures 1 and 2. The whole class entails 40 members. In lines 4 and 5 in Figure 3 we see the private fields that actually make up the decimal number.

Lines 6, 7 and 8 contain three basic operations: adding, subtracting and multiplying respectively. These methods work on Decimal objects and are public, whilst the actual work is done by the private helper functions at lines 10 and 11. There is no helper function for subtracting because subtracting is implemented by adding the negated argument. The last method in the listing in Figure 3 not yet mentioned is the round method at line 9, which, as we see in Section 5.2, has an incorrect implementation.

The sparse comments that come with the source code tell us that the decimal part should be between 0 and 999. The precision of decimal numbers is represented by the value of field *PRECISION* at line 2 (which equals 1000). By examining the method bodies of the Decimal class we find that the code contradicts with the informal comments in the sense that *decPart* should be between plus or minus the precision (recall the method *oppose* in the previous section). We use this information to extend our invariant from Figure 2 to the one below. The value of *intPart* is unbounded, that is, it is only bounded by its type.

```
//@ invariant -PRECISION < decPart && decPart < PRECISION
              && decimal_exception != null;
```

Note that because of the limited precision of the 3 digits decimal part we have to deal with a significant rounding when multiplying, whereas *add* and *sub* are without loss of precision. This highly complicates the specification for multiplication as we shall see in Section 5.1.

In the next section we take a look at the private helper method *add* at line 10 in Figure 3.

2.1 Specifying adding decimals: `void add(short,short)`

In Figure 4 we present the specification and code for the *add* method. Note that only the specification is written by us, the code is used as is, complete with French variable names.

First, we look at the code in Figure 4. The arguments e and f of the add method are the integer and the decimal part respectively. At line 9 we first add e to *intPart*. The following if-then-else block (lines 11 to 18) makes sure that *intPart* and *decPart* are both positive or negative. To see how this works, consider for example $intPart = 2$ and $decPart = -400$. This is converted to the decimal value 1.600.

At line 20, the value of field f is added to *decPart*, and again we encounter a similar if-then-else block (lines 22 to 43) that does a conversion to assure *intPart* and *decPart* to be both positive or negative. The final block also guarantees that the decimal part is within range specified by our invariant: between -*PRECISION* and *PRECISION*. The method body seems needlessly complicated and some pieces even are redundant. Namely, we can remove the first if-then-else block (lines 10 to 18) and the first part of the second if-then-else block (lines 22 to 30) and still prove the same specification to be correct. On the other hand, division and modulo operations (at lines 38 and 39) are expensive in terms of CPU cycles. Therefore, the implementation of *add* with the if-then-else blocks in place might be more efficient in terms of speed. By explicitly using the information in our invariant, we can even write an *add* without the expensive division and modulo operators (Section 4.2).

As we are not Gemplus programmers, we continue with the code at hand. Let us consider the specification for the addition method. The *requires* clause is set to $f < PRECISION$ && $f > -PRECISION$ at line 2, which says that the decimal part of the value we add satisfies the invariant. This is crucial as we see later in the proof. Normally, one keeps the *requires* clause as weak as possible, so that the specification can be re-used without much effort, because re-using a specification forces us to establish the *requires* clause.

The *modifiable* clause (in which we state what variables may be changed by the method) is set to *intPart* and *decPart* which makes sense, because even though we modify other variables like *retenue* or *signe*, these variables are local to the method, and therefore do not have to be listed in the *modifiable* clause.

The *ensures* clause might need some more explanation. A first intuitive post-condition would be just adding e to *intPart* and f to *decPart*. But because we need *decPart* to stay within the bounds set by the invariant, we would have to correct this by modulo and remainder operations similar to those in the code itself. We want the specification to be simple and short, so we need another way to get rid of the precision correction. The solution is simple and elegant: we only look at the values of the integer and decimal part combined. We have already seen that an object with $intPart = 2, decPart = -400$ is equal to an object with $intPart = 1, decPart = 600$. This is because of the simple fact that $intPart * PRECISION + decPart$ has the same value for both objects. We use this value to compare the Decimal object before and after addition of *e, f*.

Now that we have established a specification that seems correct and informative enough, we try to prove it correct. Before we can plunge into the specifics of a machine-assisted proof we first must know some more of the way Java and JML are represented within the theorem prover PVS. We do this in the next section.

3 Representing JML-annotated Java in PVS

The LOOP group in Nijmegen has developed a tool that translates Java source code and corresponding JML annotations to a higher order logic like the one used in PVS. In PVS we can then prove the properties of this Java code by stepping through the method using special Hoare-rules. In this section we see the Hoare rule for composition as used in proofs. First we need to know how Java and JML are being represented in PVS.

Converting Java to a logic is a non-trivial business. Also, it is not within the scope of this article to explain exactly how this translation takes place (see [3]), but for short we say that a (shallow)

```
/*@ normal_behavior
  @     requires f < PRECISION && f > -PRECISION;
  @  modifiable intPart, decPart;
  @      ensures
  @      (\old(intPart) + e)*PRECISION + \old(decPart) + f
  @      == intPart*PRECISION + decPart;
  @*/
private void add(short e, short f){
  intPart += e;

  if ( intPart > 0 && decPart < 0 ) {
     intPart--;
     decPart = (short) (decPart + PRECISION);
  }
  else if ( intPart < 0 && decPart > 0 ){
     intPart++;
     decPart =(short) (decPart - PRECISION);
  }

  decPart += f;

  if ( intPart > 0 && decPart < 0 ) {
     intPart--;
     decPart = (short) (decPart + PRECISION);
  }
  else if ( intPart < 0 && decPart > 0 ){
     intPart++;
     decPart = (short) (decPart - PRECISION);
  }
  else {

     short retenue = (short) 0;
     short signe = 1;
     if ( decPart < 0 ) {
         signe = (short) -1;
         decPart = (short) -decPart;
     }
     retenue = (short) (decPart / PRECISION);
     decPart = (short) (decPart % PRECISION);
     retenue *= signe;
     decPart *= signe;
     intPart += retenue;
  }
}
```

Figure 4: `add(short,short)` source code and JML specification

semantics of Java is formalised in the higher order logic of PVS. All primitive types, reference types, and constructs such as statements and expressions have their semantic equivalent in the higher order logic. The logic files generated by the LOOP-tool depend on a hand-written Prelude in which these constructs are defined (see [2, 8]). We restrict ourselves to explaining the transformed combination of Java and JML that is of direct concern to the verifier.

For every method a labeled product is generated that contains the method body and all specification clauses for that method. We call this labeled product a "behavior specification". In Figure 5 we show the (simplified) behavior specification in PVS notation for method *setValue* from Figure 2. A labeled product is represented in PVS by a record. For example the PVS construct `(# l := 5 #)` is a record where field `l` has value `5`.

```
StatBehavior
 ((#
  requires := LAMBDA (pre: OM?) :
                DecimalJML_invariant?(pre) AND
                setValue?short?behavior_requires(par?v)(pre),
  statement := setValue?short(par?v)(c?) ,
  ensures := LAMBDA (post:OM?) :  LAMBDA (ret?setValue: DecimalJML?Type) :
                DecimalJML_invariant?(post),
                setValue?short?behavior_ensures(par?v)(ret?setValue)(post) AND
                setValue?short?behavior_modifiable(post)
  signals := LAMBDA (epost: OM?) : LAMBDA (e?: DecimalExceptionJML?Type) :
                DecimalJML_invariant?(post)
                setValue?short?behavior_modifiable(post) AND
                setValue?short?behavior_signals(par?v)(e?)(post)
 #))
```

Figure 5: Behavior specification for method `setValue(short)`.

It would make sense for the predicate labeled `requires` in Figure 5 to be the direct representation of the JML *requires* clause from Figure 2. But the PVS `requires` clause is a representation of the pre-condition of method *setValue*, and the pre-condition must also contain the invariant of the class from where our method originated. Therefore, the `requires` for method *setValue* consists of a conjunction of the invariant `DecimalJML_invariant?` and the translation of the JML *requires* clause `setValue?short?behavior_requires`. The `requires` clause is a lambda abstraction of type `OM?` where `OM?` represents the memory state, so that the predicate can be evaluated in the pre- and post-states.

The `ensures` clause has an extra lambda abstraction of type `DecimalJML?Type` which is the PVS representation of the type of the object that is returned by *setValue*. The `signals` clause has a lambda abstraction of type `DecimalExceptionJML?Type` which is the PVS representation of the DecimalException object which can be thrown by *setValue*. Note that the post-condition `ensures` and the post-condition `signals` for exceptional behavior also contain a predicate `setValue?short?behavior_modifiable`. This predicate represents the fact that the fields other than the ones mentioned in the JML clause *modifiable* may not be changed during execution of *setValue*. The actual method body is labeled by the `statement` tag, it is not yet unwrapped.

The meaning of the behavior specification is expressed by the PVS statement `StatBehavior` in Figure 5. The definition of `StatBehavior` (not given here) is that for any initial state `pre` which satisfies the pre-condition `requires`, the post-conditions `ensures` and `signals` hold in the states `post` and `epost` that result from normal and exceptional execution respectively[2].

[2]The labeled tuples used during the actual verification have three more labels (for return, break and continue), but these need not concern us here. See [9] for further details.

```
composition : LEMMA
  (EXISTS(P1 : [OM? -> bool]) :
      StatBehavior((# requires := P,
                      statement := s1,
                      ensures := P1,
                      signals := r #))
      AND
      StatBehavior((# requires := P1,
                      statement := s2,
                      ensures := Q,
                      signals := r #))
  IMPLIES
  StatBehavior((# requires := P,
                  statement := s1 ; s2,
                  ensures := Q,
                  signals := r #))
```

Figure 6: Hoare rule for composition

In general, proving a post-condition in the state that results from executing a method cannot be performed in one step, because a method and therefore its resulting state can be complex. Proving a behavior specification can thus best be done by splitting up the method body in smaller parts. This is achieved by making use of Hoare logic. We have implemented and proved correct Hoare rules for all Java language constructs like composition, if-then-else-clauses, while- and for-loops and try-catch-statements (see [9]). These Hoare rules are written in terms of behavior specifications as shown above. We discuss one such a Hoare rule which is the rule for composition. The standard rule for composition, written here as ";", is given in natural deduction style below.

$$\frac{[P]\ s1\ [P1]\ ,\ [P1]\ s2\ [Q]}{[P]\ s1; s2\ [Q]}\,composition \tag{1}$$

Our rule for composition is shown in PVS style in Figure 6. Apart from the style, rule (1) is comparable to the one in Figure 6. Using a lemma like composition in a proof is simple: when we end up with a goal that matches the sequent of the implication in Figure 6 we may apply the implicative rule on it, gaining two new proof obligations. As one can see, this happens at many nodes labeled "(composition-...)" in the proof-tree in Figure 7.

Now that we know some more about Java annotated with JML and its meaning in the higher order logic of PVS, let us do some actual verification.

4 Decimal class verification

In this section we finally present a complete proof for the private helper method *add*. The only question remaining is *how* to present a proof. As we have just seen, the LOOP-tool converts easily understandable Java code and readable specifications to obscure PVS code, which is unpleasant for most people. Luckily, most of the proofs of imperative programming languages just consist of applying the Hoare rule for composition a number of times. The informative part of the rule for composition is its intermediate predicate (`P1` in Figure 6) which reflects the effects of the first statement (`s1`). We can write down intermediate predicates in Java source code by using a yet unmentioned feature of JML: the *assert* clause (its semantics is similar to its ancient C predecessor, and also available in JDK 1.4). The *assert* clause is not yet interpreted by the LOOP-tool such that it can instantiate the intermediate predicates for composition automatically. This will be possible in the near future, but for now we use it merely as a convenient way to present the proof.

Although PVS code is not quite suited to present here, the PVS proof-tree is. A proof-tree is generated during proving and graphically presents the verifier with an overview of the choosen steps (applications of Hoare rules for example) and amount of subgoals still left to prove. The proof-tree for method *add* is shown in Figure 7. Every node in the proof-tree represents a proof-step given by the verfier. Whenever a proof is completed, the PVS proof-tree turns blue.

4.1 Verifying `void add(short,short)`

First, recall that the specification and source code of the *add* method in Figure 4 is converted to a behavior specification like the one in Figure 5. We can immediately apply the Hoare rule for composition as we have a PVS `statement` field which is of the form `intPart?becomes(intPart(x) + par?e)(x) # ...` corresponding to Java source code of the form *intPart*+ = *e* ; ... where the assignment is the left hand part (statement `s1` in Figure 6) of the composition.

To keep track of where we are in the verification phase, we print the JML assertions with the method body as we go. The part up to and including the first intermediate predicate is shown in *assert* (lines 10 and 11) style below.

```
   /*@ normal_behavior
 2   @     requires f < PRECISION && f > -PRECISION;
     @  modifiable intPart, decPart;
 4   @      ensures
     @      (\old(intPart) + e)*PRECISION + \old(decPart) + f
 6   @      == intPart*PRECISION + decPart;
     @*/
 8 private void add(short e, short f){
    intPart += e;
10 //@ assert intPart * PRECISION + decPart
   //@          == (\old(intPart) + e) * PRECISION + decPart;
```

This intermediate predicate already resembles the ensures clause very much. The PVS code that the verifier has to enter in the theorem prover is shown below. This is the same intermediate predicate as the assert above, but now in the PVS representation of Java. Note that the application of the rule for composition is also shown at node 1 in Figure 7.

```
LAMBDA(x:OM?) :
 intPart(c?!1(p?!1))(x) * PRECISION(Decimal?clg("Decimal")(-1))(x)
 + decPart(c?!1(p?!1))(x)
=
 (intPart(c?!1(p?!1))(Z?!1) + par?e!1) * PRECISION(Decimal?clg("Decimal")(-1))(x)
 + decPart(c?!1(p?!1))(x)
```

The next composition in the code has the if-then-else as the left hand part. Annotating the body of this if-then-else clause is not interesting, because the clause will not effect our intermediate predicate. Below is the code up to the next interesting assert: right after the second assignment. The point in the proof corresponds with node 2 in Figure 7.

```
12  if ( intPart > 0 && decPart < 0 ) {
       intPart--;
14     decPart = (short) (decPart + PRECISION);
    }
16  else if ( intPart < 0 && decPart > 0 ){
```

```
      intPart++;
18    decPart =(short) (decPart - PRECISION);
   }
20 decPart += f;
   //@ assert intPart * PRECISION + decPart
22 //@          == (\old(intPart) + e) * PRECISION + decPart + f
   //@          && -2 * PRECISION < decPart
24 //@          && decPart < 2 * PRECISION;
```

By adding *f* to *decPart* we could break the invariant. Fortunately, we know that both *f* and *decPart* are between *-PRECISION* and *PRECISION*. We use this information to enrich the intermediate predicate so that we can later restore the class invariant. The last piece of code is again an if-then-else block. The first two cases are not interesting, as they establish the *ensures* clause in a trivial way. The last case is more interesting, because there we see a modulo, division and some sign operations. This case is only executed when *intPart* and *decPart* are both greater or less than zero.

```
   if ( intPart > 0 && decPart < 0 ) {
26    intPart--;
      decPart = (short) (decPart + PRECISION);
28 }
   else if ( intPart < 0 && decPart > 0 ){
30    intPart++;
      decPart = (short) (decPart - PRECISION);
32 }
   else {
34    short retenue = (short) 0;
      short signe = 1;
36 //@ assert decPart
   //@          == (\old(intPart) + e) * PRECISION + decPart + f
38 //@              - intPart * PRECISION
   //@          && -2 * PRECISION < decPart
40 //@          && decPart < 2 * PRECISION
   //@          && retenue == 0
42 //@          && signe == 1;
```

We are now at node 3 in the proof-tree. The assert shown above is not magic. We just take the values of *retenue* and *signe* along, because we need them later. Also, we move *decPart* to the left hand of the equation in line 37. This is because we might change the sign of *decPart* in the next if-then block.

```
      if ( decPart < 0 ) {
44        signe = (short) -1;
          decPart = (short) -decPart;
46    }
   //@ assert decPart
48 //@          == signe *
   //@              ((\old(intPart) + e) * PRECISION + decPart + f
50 //@               - intPart * PRECISION)
   //@          && decPart >= 0
52 //@          && decPart < 2 * PRECISION;
   //@          && retenue == 0
54 //@          && (signe == 1 || signe == -1);
```

Even if the sign of *decPart* is not changed, we know that the value of *decPart* equals the value of *decPart* in the previous assert multiplied by *signe* (line 48). Furthermore, we know that *decPart* is positive (line 51). The assert above corresponds to node 4 in the proof-tree.

The assignment below is more interesting. The value in *retenue* will represent the division of *decPart* by *PRECISION*. This value is of course rounded because of its type. Because we know that *decPart* has an upper bound of $2 * PRECISION$ the value of *retenue* is either 0 or 1. The property that the Gemplus programmers are using in the following assignments is that for two shorts a and b the following holds $a == (a/b) * b + a \ \% \ b$. We may thus replace *decPart* in the equation at line 47 by $retenue * PRECISION + (decPart \ \% \ PRECISION)$ leading to the new intermediate predicate below (node 5 in the proof-tree).

```
     retenue = (short) (decPart / PRECISION);
56 //@ assert retenue * PRECISION + (decPart % PRECISION)
   //@          == signe *
58 //@               ((\old(intPart) + e) * PRECISION + decPart + f
   //@                 - intPart * PRECISION)
60 //@          && -2 * PRECISION < decPart
   //@          &&  decPart < 2 * PRECISION;
62 //@          && (retenue == 1 || retenue == 0)
   //@          && (signe == 1 || signe == -1)
64 //@          && decPart >= 0;
```

By the construction of our intermediate predicate so far, we can step through the next assignment by a simple substitution. The assert below corresponds to node 6 in Figure 7.

```
     decPart = (short) (decPart % PRECISION);
66 //@ assert retenue * PRECISION + decPart
   //@          == signe *
68 //@               ((\old(intPart) + e) * PRECISION + decPart + f
   //@                 - intPart * PRECISION)
70 //@          && (retenue == 1 || retenue == 0)
   //@          && (signe == 1 || signe == -1)
72 //@          && decPart >= 0;
```

Note that by taking the modulo of *decPart* we have restored the class invariant. The following assignment multiplies *retenue* by *signe*. Because *signe* is either 1 or -1, *retenue* becomes either -1, 1 or 0. We are at node 7 in the proof-tree.

```
     retenue *= signe;
74 //@ assert retenue * PRECISION + decPart * signe
   //@          == (\old(intPart) + e) * PRECISION + decPart + f
76 //@                 - intPart * PRECISION
   //@          && (retenue == 1 || retenue == 0 || retenue == -1)
78 //@          && (signe == 1 || signe == -1)
   //@          && decPart >= 0;
```

We also use some elimination rules to remove the *signe* at line 67. Now we end up with $decPart * signe$ at line 74 from which the *signe* will disappear when the next assignment is executed (node 8).

```
80   decPart *= signe;
   //@ assert retenue * PRECISION + decPart
82 //@          == (\old(intPart) + e) * PRECISION + decPart + f
   //@                 - intPart * PRECISION
84 //@          && (retenue == 1 || retenue == 0 || retenue == -1)
```

The intermediate predicate starts resembling the *ensures* clause more and more, the one assignment left finishes the job. This is represented by node 9 in the proof-tree which is a leaf.

```
     intPart += retenue;
```

```
86 //@ assert decPart + intPart * PRECISION
   //@        == (\old(intPart) + e) * PRECISION + decPart + f;
88  }
   }
```

We may write the assertion at line 66 because we can eliminate $retenue * PRECISION$ on both sides. This concludes the proof.

4.2 Optimised addition

Having done the verification of addition based on the Decimal class invariant, we realised that the real issue is (possibly) restoring the invariant after the assignment $decPart += f$. As we have seen, once this assignment has taken place, we still know that $-2 * PRECISION < decPart \,\&\&\, decPart < 2 * PRECISION$. Hence the invariant can be restored by adding or subtracting $PRECISION$ at most once. Therefore, addition can also be performed in the following way.

```
private void addOpt(short e, short f){
    intPart += e;
    decPart += f;
    if ( decPart <= -PRECISION ) {
        decPart += PRECISION;
        intPart--;
    }
    else if ( decPart >= PRECISION ) {
            decPart -= PRECISION;
            intPart++;
    }
}
```

This short implementation satisfies the same specification as the longer variant in Figure 4[3]. It avoids the costly division and remainder operations of the original implementation, and is thus much better suited for implementation on a smart card. This optimised implementation can only be written after the class invariant has been made explicit. Hence specification can have an impact on code development.

5 Two more interesting members of the Decimal class

In the next section we specify the helper method for multiplying Decimals. This is interesting by its complexity. Finally, in Section 5.2 we show the method for rounding Decimals which has an incorrect implementation.

5.1 Decimal multiplication: `void mul(short,short)`

As we saw in section 2 the *Decimal* class contains two methods *mul* for multiplication: one private and one public. The public one takes a *Decimal* object *d* as argument, and calls the private multiplication method with the *intPart* and *decPart* of the *d* as arguments. Hence the private *mul* does the real work, and is thus most interesting. It has two arguments of type short representing the decimal that is multiplied with the decimal represented by the current object. This method is probably the most difficult one of the whole *Decimal* class, both in terms of specification and of implementation.

[3] To our embarrassment, verification showed an implementation error in our first verion of *addOpt*, in which we used < and > instead of <= and >=.

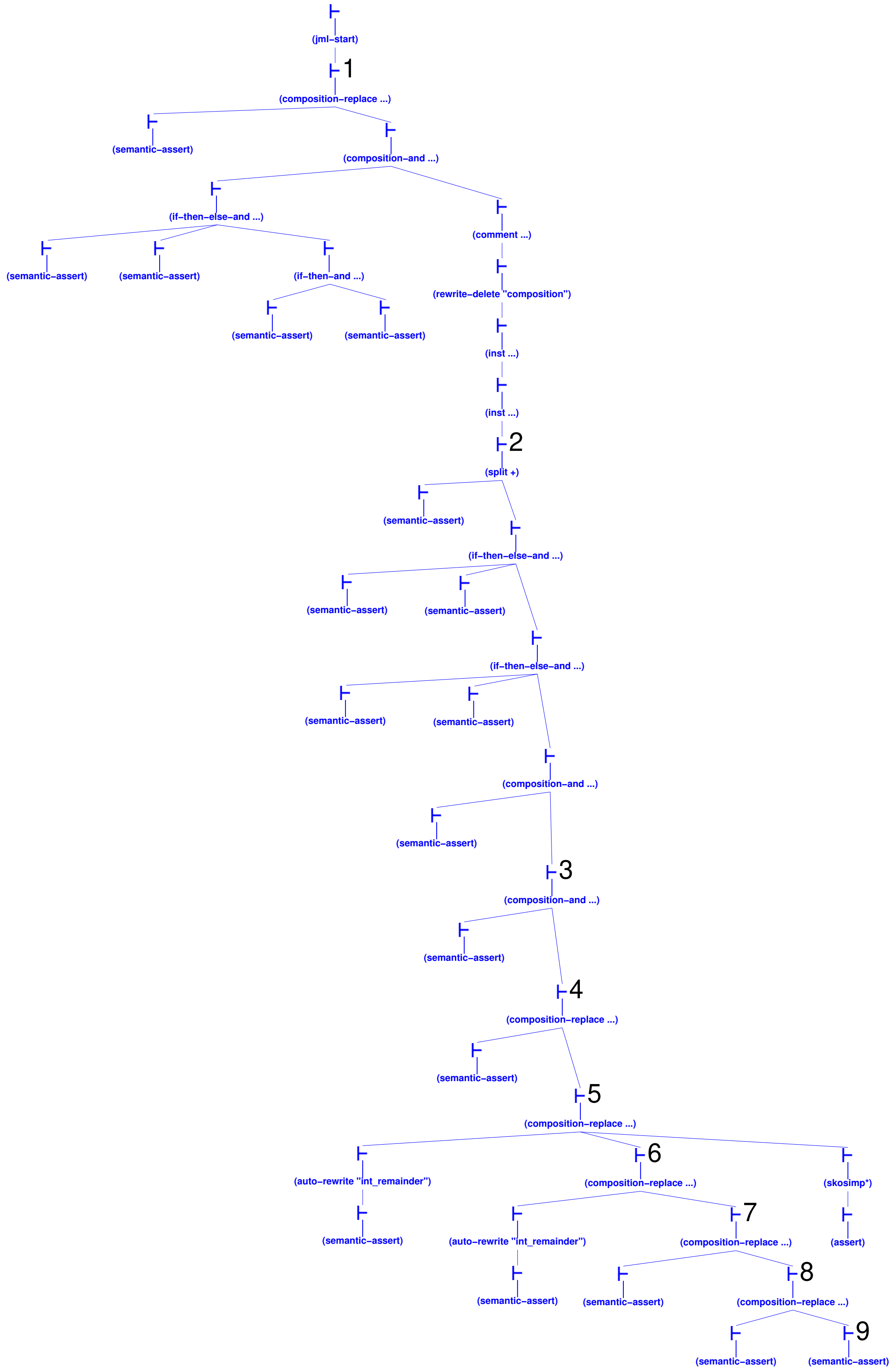

Figure 7: Proof-tree for `add(short,short)`

```
/*@ normal_behavior
  @    requires -PRECISION < f && f < PRECISION;
  @  modifiable intPart, decPart;
  @     ensures intPart * PRECISION + decPart ==
  @         \old(intPart) * e * PRECISION
  @         +
  @         \old(intPart) * f
  @         +
  @         \old(decPart) * e
  @         +
  @         /// difficult rest-part, consisting of:
  @         /// sign of product of decimal parts
  @         (( (f >= 0 && \old(decPart) >= 0) ||
  @            (f < 0 && \old(decPart) < 0) )
  @               ? 1
  @               : -1)
  @         *
  @         /// thousand-part of product of rounded decimal parts
  @         (( ((-100 <= f && f <= 100)
  @                 /// absolute value
  @                 ? java.lang.Math.abs(f)
  @                 /// round last digit to 0 of absolute value
  @                 : 10 * (java.lang.Math.abs(f) / 10))
  @            *
  @            ((-100 <= \old(decPart) && \old(decPart) <= 100)
  @                 /// absolute value
  @                 ? java.lang.Math.abs(\old(decPart))
  @                 /// round last digit to 0 of absolute value
  @                 : 10 * (java.lang.Math.abs(\old(decPart))/10))
  @           ) / 1000);
  @*/
private void mul(short e, short f){ ... // see Figure 9 }
```

Figure 8: JML specification of the private multiplication method.

We first consider its specification, in Figure 8. Like in the specification of addition in Figure 4, we use a combined formulation for *intPart* and *decPart*. The arguments are e and f for the *intPart* and *decPart* of the multiplier. An easy calculation shows that we can write the combined multiplication as follows.

$$\begin{aligned}
&\frac{\Big(\texttt{\textbackslash old(intPart) * PRECISION + \textbackslash old(decPart)}\Big) * \Big(\texttt{e * PRECISION + f}\Big)}{\texttt{PRECISION}} \\
&\quad = \texttt{\textbackslash old(intPart) * e * PRECISION} \\
&\qquad \texttt{+ \textbackslash old(intPart) * f} \\
&\qquad \texttt{+ \textbackslash old(decPart) * e} \\
&\qquad \texttt{+ ...}
\end{aligned} \tag{2}$$

The difficulty lies in the part `...` that we left out. In the Java implementation it involves a certain truncation, which is rather complicated to describe in JML: it covers the greater part of Figure 8, from line 11 onwards. We shall try to explain what is going on, using the standard mathematical operations abs and sgn, where sgn is currently not available in JML

$$\mathsf{abs}(x) \;=\; \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases} \qquad\qquad \mathsf{sgn}(x) \;=\; \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Next we should keep in mind that division in Java (and JML) involves truncation. For instance, $10 * (347/10) = 340$ and $10 * (-347/10) = -340$. So we can define an auxilary function:

$$\mathsf{trunc}(x) \;=\; \begin{cases} 10 * (x/10) & \text{if } x \geq 100 \\ x & \text{otherwise} \end{cases}$$

The rest part `...` in (2), as described in JML in Figure 8, then amounts to:

$$\texttt{...} \;=\; \mathsf{sgn}\big(\texttt{f * \textbackslash old(decPart)}\big) * \frac{\mathsf{trunc}(\mathsf{abs}(\texttt{f}))\texttt{ * }\mathsf{trunc}(\mathsf{abs}(\texttt{\textbackslash old(decPart)}))}{1000} \tag{3}$$

The combined specification (2) and (3) can then be proved for the *mul* implementation in Figure 9. We shall briefly walk through this implementation, explaining some of the more special points, because discussing all intermediate predicates like in Section 4 would go too far.

The method body in Figure 9 can be split into two parts:

(a) lines 2–22, establishing the equation (2) without the dots `...`;

(b) lines 23–51, establishing the "rest" equation (3) for `...`.

Part (a) uses two `for` loops, firstly for abs(`e`)-times adding `\old(intPart)` and `\old(decPart)` in lines 8 and 9, and secondly for `\old(intPart)` adding only `f` in lines 18 and 19. These two steps are combined in line 22, via the intermediate values `intPart_` and `decPart_`.

Part (b) starts by setting in two `if` statements `signe` = sgn(`f * \old(decPart)`) which is the first part of (3). At the same time, `arrondis1` = abs(`\old(decPart)`) and `arrondis2` = abs(`f`). The two subsequent `while` loops handle the truncations. By our invariant and precondition, both these `while` loops are executed at most one time. The explicit use of the numbers `100` and `1000` is a bit clumsy here. It would be more abstract to use `PRECISION/10` and `PRECISION` instead[4]. The third `while` loop determines the appropriate compensation factor after truncation. This loop is executed zero, one or two times. Finally, `temp` becomes our "rest" equation (3), which is added to the final result.

[4]We have decided to follow this clumsiness in our JML specification in Figure 8.

```
private void mul(short e, short f){
  short intBackup = intPart;
  short decBackup = decPart;
  short nbIter = e;
  if ( nbIter < 0 ) { nbIter = (short) -nbIter;}
  intPart = (short) 0;
  decPart = (short) 0;
  for ( short i = (short) 0; i < nbIter; i++){
    add(intBackup, decBackup);
  }
  if ( e < 0 ) { oppose(); }
  short intPart_ = intPart;
  short decPart_ = decPart;
  intPart = (short) 0;
  decPart = (short) 0;
  nbIter = intBackup;
  if ( nbIter < 0 ) { nbIter = (short) -nbIter; }
  for ( short i = (short) 0; i < nbIter; i++ ){
    add((short) 0, f);
  }
  if (intBackup < 0 ) { oppose(); }
  add(intPart_, decPart_);
  short signe = (short)1;
  short arrondis1 = decBackup;
  if ( arrondis1 < 0 ) {
    arrondis1 = (short)  -arrondis1;
    signe = (short) -signe;
  }
  short arrondis2 = f;
  if ( arrondis2 < 0 ) {
    arrondis2 = (short) -arrondis2;
    signe = (short) -signe;
  }
  short decal = (short) 0;
  while ( arrondis1 > 100 ) {
    arrondis1 /= (short) 10;
    decal++;
  }
  while ( arrondis2 > 100 ) {
    arrondis2 /= (short) 10;
    decal++;
  }
  short temp = (short)  (arrondis1 * arrondis2);
  short aux = (short) 1000;
  while ( decal > 0 ) {
    aux /= (short) 10;
    decal--;
  }
  temp /= aux;
  temp *= signe;
  add((short) 0, temp); }
```

Figure 9: Implementation of the private multiplication method.

5.2 Rounding decimals: `Decimal round()`

The *round* method of the Decimal class shows that programming arithmetic is tricky business. The semantics of rounding decimals is clear by intuition: return the closest integer, where an integer is represented by a Decimal object whose *decPart* is zero.

The implementation of the *round* method is shown in Figure 10. Despite its complexity (why use a while-loop), the code seems to work. For example, suppose we have a Decimal object representing 8.990. After the while-loop at line 4 the value of *aux* equals 9, so we increase *intPart* by one, and return a Decimal representing 9.0.

But the code in Figure 10 only takes the most significant digit of *decPart* into account. This means that for absolute values of *decPart* ranging from 6–10, 60–100 and 501–599 the rounding is always incorrect. To see this, suppose a Decimal object representing 8.009. After the while-loop the value of *aux* equals 9, so we end up with a Decimal object representing 9.0. Or, suppose a decimal value 8.550 which will be rounded to 8.0. A rounding like this could be bad for the bank, or worse, bad for the clients of the bank.

Such an implementation error is not necessarily found via a machine-assisted correctness proof of a specification. But it makes clear that by thinking about and developing a specification, errors like these can be noticed fairly easy.

```
   public Decimal round(){
2      short aux = decPart;
       if ( aux < 0 ) aux = (short) -aux;
4      while ( aux > 10 ) aux /= (short) 10;
       if ( aux > 5 ) {
6          if ( decPart > 0 ) intPart++;
           else intPart--;
8      }
       decPart = (short) 0;
10     return this;
   }
```

Figure 10: Implementation of the rounding method

6 Conclusions

This paper describes a case study in JML specification and PVS verification for JavaCard, in the line of [4]. It forms an incremental improvement, going beyond [4], by tackling a Java class with complicated computational content and non-trivial functional specifications. It demonstrates that the “LOOP” verification technology extends to such examples.

What else can be learned from this verification? We have several points.

- It cannot be over-emphasised that formalizing class invariants is very important, because it makes implicit assumptions explicit. In the *Decimal* class the Gemplus developers have (informally) described an invariant *0 <= decPart < PRECISION*, but this one is broken by the (public) method *oppose*. Hence we come to a weaker invariant *-PRECISION < decPart < PRECISION*. More generally, formal specifications provide unambiguous documentation (as long as the semantics is clear, of course).
- Functional specifications can both be compact and readable (like for the *add* method in Figure 4) and verbose and unreadable (like for *mul* in Figure 8). Still, the specification adds

useful information which is not readily available from inspecting the code. Actual verification may then be justified in critical applications.

- Code specification and verification are iterative processes. Most often (in our experience) one stops a particular verification at some stage to alter the specification—and restart the verification. But also, one may wish to alter the code itself. We have chosen not to do so in this case study, since we consider the Java code as given, fixed input; several others, like [5], also use the *Decimal* class as case study, so changing the code leads to confusion. But there were several occasions where we would have liked to alter the code. Above we have already mentioned several of such infelicities, sloppinesses, or bugs.

In the end there is always the question: how much time did this take? Unfortunately, we cannot give an exact answer, because: this case study has been used by one of the authors (CBB) to learn the verification technology. Also, it has been used to improve our proof technology, especially be writing many dedicated proof strategies in PVS (especially by JvdB). And finally, halfway the verification we decided to switch to the next beta release of PVS, in order to maximize our influence on the development of this new version. All these factors caused considerable delays, which would not be there if this case study would have been done with all parameters optimally tuned. But probably this never happens in practice.

References

[1] Gemplus Purse applet.
`http://www.gemplus.com/smart/r_d/publications/case-study/index.html`.

[2] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.

[3] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.

[4] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of *LNCS*, pages 137–150. Springer Verlag, 2001.

[5] Néstor Cataño Collazo. La clause modifiable de JML: Sémantique, vérification et application. Master's thesis, INRIA de Sophia-Antipolis, 2001.

[6] N. Cataño Collazos and M. Huisman. Comments on the gemplus purse applet.
`http://www-sop.inria.fr/lemme/verificard/electronic_purse/`.

[7] N. Cataño Collazos and M. Huisman. Formal specification of Gemplus' electronic purse case study.
`http://www-sop.inria.fr/lemme/verificard/electronic_purse/`, 2002.

[8] M. Huisman. Reasoning about JAVA programs in higher order logic with PVS and Isabelle, 2001.

[9] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2001.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.cs.iastate.edu/~leavens/JML.html`.

[11] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} rev. edition, 1997.

[12] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes Comp. Sci.*, pages 63–77. Springer, Berlin, 2000.

[13] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Techn. Univ. München, 2000.

[14] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lect. Notes Comp. Sci., pages 119–156. Springer, Berlin, 1998.

[15] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.

[16] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.

[17] Extended static checker ESC/Java. Compaq System Research Center.
See `www.research.compaq.com/SRC/esc/Esc.html`.