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

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

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

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 postconditions, describing the functional behavior. These specifications are written in the specification language JML. After translation of the annotated source code to 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 the art 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 optimized 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, Sect. 11.6], but one which is worth re-emphasizing 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,

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

for instance as general identity card. The European VerifiCard project aims to develop techniques for establishing the correctness for crucial 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. JavaCard is extended with a transaction logging mechanism, and a firewall protection mechanism to control inter-applet communication. 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, 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. 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. The Decimal class is an essential component of the Purse. 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 verified 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 and Weakest Precondition 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 establishing the 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 postconditions 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 features that JML offers is the wide range of tools, for systematic testing, static checking, or formal verification.

In a companion paper to ours [6, 7] the ESC/Java tool (see [17]) is applied to the same Purse case study, although using different specifications[1]. By statically analyzing the source code with ESC/Java, many implementation details are made explicit, such as input checking on parameters, and invariants on the fields. An advantage of this approach is that ESC/Java automatically checks whether 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 organized as follows. In the first section we familiarize the reader with JML. In Sect. 2 we describe some members and specify the method for adding Decimals. In Sect. 3 we shortly discuss how the translation from annotated Java source code to PVS is achieved. In Sect. 4 we present the verification of the method for addition. In Sect. 5 we discuss the method for multiplying two Decimals.

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 Fig. 1 the source code and annotations for a small segment of the Decimal class is depicted. Method *oppose* returns the opposite decimal number.

The code with the JML specification in Fig. 1 can be compiled by a normal Java compiler because the annotations are within Java comments. Note that

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

```
  class Decimal {
2  short intPart, decPart;
   final static short PRECISION = 1000;
4
   //@ invariant PRECISION == 1000;
6
   /*@ normal_behavior
8    @     requires d != null;
     @  modifiable intPart, decPart;
10   @      ensures     intPart == -\old(intPart)
     @              && decPart == -\old(decPart);
12   @*/
     public Decimal oppose(Decimal d) {
14     intPart = -intPart;
       decPart = -decPart;
16     return this;
     }
18 }
```

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

we use a "@" to distinguish JML annotations from normal comments. The precondition (*requires* clause) says that the argument to *oppose* may not be *null*. After execution and normal termination of *oppose* it is ensured that *intPart* and *decPart* are negated (*ensures* clause). We are able to refer to the values of *intPart* and *decPart* before execution of *oppose* by writing *old(intPart)* and *old(decPart)* respectively. The purpose of the *modifiable* clause is to maintain a list of all fields that can be modified by *oppose*.

In JML we may also write clauses that specify behavior when exceptional or abnormal termination occurs. The examples in this paper do not exhibit such behavior. Therefore, these clauses are omitted.

The class invariant is expressed in JML by an *invariant* clause at line 5. A class invariant should be maintained by every method in that class, which means that after execution of *oppose* the invariant should hold, assuming it holds before execution.

The predicates used in the *invariant, requires* and *ensures* clauses resemble Java boolean expressions. This is convenient in 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.

2 Decimal class specification

The Decimal class source code was handed to us without any formal specification. For methods like *add* or *mul* it is luckily not very hard to guess the intended

```
public class Decimal extends Object {
    public static final short PRECISION;
    private short intPart;
    private short decPart;
    public Decimal add(Decimal) throws DecimalException;
    public Decimal mul(Decimal) throws DecimalException;
    public Decimal round();
    private void add(short, short);
    private void mul(short, short);
    public Decimal oppose();
}
```

Fig. 2. Discussed fields and methods of the Decimal class

functionality. In Fig. 2 we list the members of the Decimal class relevant to this paper. The whole class entails 40 members.

The sparse comments in 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* which equals 1000. By examining the method bodies of the Decimal class we find that the code contradicts with the informal comments. The value of *decPart* can be less then 0 (think about *oppose* in the previous section). We use this information to extend the invariant from Fig 1 to the one below. Make we make no explicit statements on the bounds of *intPart*.

```
//@ invariant -PRECISION < decPart && decPart < PRECISION
              && PRECISION == 1000;
```

Note that because of the limited precision of the 3 digits decimal part we have to deal with a significant rounding when multiplying, whereas method *add* runs without loss of precision. Rounding highly complicates the specification for multiplication as we shall see in Sect. 5.

In the next section we take a look at the private helper method *add*.

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

In Fig. 3 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.

The arguments e and f of *add* are the integer and the decimal part respectively. At line 9 we first add e to *intPart*. The following if-then-else block (lines 12 to 19) ensures that *intPart* and *decPart* are both positive or both negative. To see how this works, consider for example $intPart = 2$ and $decPart = -400$. The if-then-else block ensures that $intPart = 1$ and $decPart = 600$.

At line 22, the value of field f is added to *decPart*, and again we encounter a similar if-then-else block (lines 23 to 44) that does a conversion to ensure *intPart* and *decPart* are both positive or both negative. Lines 23 to 44 also guarantee that the decimal part is within range specified by our invariant: between -*PRECISION* and *PRECISION*. The method body is needlessly complicated and some pieces are redundant. Namely, we can remove the first if-then-else block (lines 12 to 19) and the first part of the second if-then-else block (lines 23 to 32) and still prove the same specification to be correct. On the other hand, division and modulo operations (at lines 39 and 40) 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, see Sect. 4.2.

Consider the specification for *add*. The *requires* $f < PRECISION$ && $f > -PRECISION$ says that the decimal part of the value we add satisfies the invariant. 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. Note that the *ensures* clause uses $intPart * PRECISION + decPart$ as a convenient representation of the value of a Decimal object.

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 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) semantics of Java is formalized 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 (i.e. a record) is generated that contains the method body and all specification clauses for that method. We call this labeled product a "behavior specification". In Fig. 4 we show the (simplified) behavior specification in PVS notation for method *oppose* from Fig. 1. The PVS code `(# l := 5 #)` is a record where field `l` has value `5`.

```
/*@ 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;
 //@ assert intPart * PRECISION + decPart
 //@          == (\old(intPart) + e)*PRECISION + \old(decPart);
 if ( intPart > 0 && decPart < 0 ) {
    intPart--;
    decPart = decPart + PRECISION;
 }
 else if ( intPart < 0 && decPart > 0 ){
    intPart++;
    decPart = decPart - PRECISION;
 }
 //@ assert intPart * PRECISION + decPart
 //@          == (\old(intPart) + e)*PRECISION + \old(decPart);
 decPart += f;
 if ( intPart > 0 && decPart < 0 ) {
    intPart--;
    decPart = decPart + PRECISION;
 }
 else if ( intPart < 0 && decPart > 0 ){
    intPart++;
    decPart = decPart - PRECISION;
 }
 else {

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

Fig. 3. `add(short,short)` source code and JML specification

```
StatBehavior
 ((#
  requires := LAMBDA (pre: OM?) :
                 DecimalJML_invariant?(pre) AND
                 oppose?behavior_requires(pre),
  statement := oppose?body ,
  ensures := LAMBDA (post:OM?) :  LAMBDA (ret?oppose: DecimalJML?Type) :
                 DecimalJML_invariant?(post),
                 oppose?behavior_ensures(ret?oppose)(post) AND
                 oppose?behavior_modifiable(post)
 #))
```

Fig. 4. Behavior specification for method `oppose()`.

The `requires` predicate in Fig. 4 includes not just the precondition given in the code, but also explicitly includes the invariant, which is implicit in all JML preconditions. 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 a lambda abstraction of type `DecimalJML?Type` which is the PVS representation of the type of the object returned by *oppose*. The postcondition also contains a predicate `oppose?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 *oppose*. 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 Fig. 4. The definition of `StatBehavior` (not given here) is that for any initial state `pre` which satisfies the pre-condition `requires`, the postcondition `ensures` holds in state `post` that results from normal execution[2].

In general, proving a postcondition 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 into 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.

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

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

Our rule for composition is shown in PVS style in Fig. 5. Apart from the style, rule (1) is comparable to the one in Fig. 5.

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

Fig. 5. Hoare rule for composition

Apart from Hoare rules, we also have Weakest Precondition (WP) rules at our disposal. Using these we can verify code fragments that contain no repetition automatically. Details of this WP calculus will appear in a separate publication.

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 present a proof for the private *add*. The typical way in which we prove the correctness of methods in PVS is that we use the composition rule to chop up our proof obligation in several pieces. We then apply the WP-tactic to prove these pieces automatically. 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. The intermediate predicate for the composition rule needs to be entered in PVS by the verifier. We can also write intermediate predicates in Java source code by using a yet unmentioned feature of JML: the *assert* clause. 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 the *assert*s to indicate where we cut up the method body.

Although PVS code is not quite suited to be presented here, the PVS proof-tree is. A proof-tree is generated during the process of proving and graphically presents the verifier with an overview of the chosen steps (applications of Hoare rules for example) and amount of subgoals still left to prove. The proof-tree for method *add* is shown in Fig. 6. Every node in the proof-tree represents a tactic applied by the verifier.

4.1 Verifying `void add(short,short)`

Because *add* contains no repetitions, we should be able to apply the WP-tactic to finish the proof. Unfortunately, if do not chop up the method body into smaller pieces, the proof obligation is too large for PVS to handle.

Therefore, the proof consists of two applications of the Hoare rule for composition. In Fig. 6 these applications are labeled “composition-replace” and “composition-and”. Applying composition twice results in three chunks of code, namely line 9, lines 12 to 19 and lines 19 to 44. These chunks are verified using WP-tactic. In the proof-tree in Fig. 6 this is denoted by a label “wp-assert”.

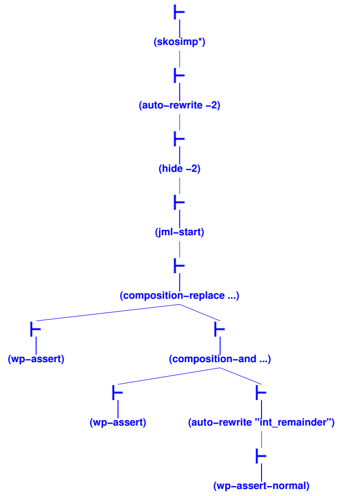

Fig. 6. Proof-tree for `add(short,short)`

The two intermediate predicates used to apply the rules for composition are shown at lines 10 and 20 in Fig. 3 marking the position where the code is cut. Note that these assertions are almost identical to the ensures clause.

4.2 Optimized addition

After taking a closer look at the *add* code we realized that the real issue is (possibly) restoring the invariant after the assignment $decPart += f$. Once this assignment has occurred, 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[3] the same specification as the longer variant in Fig. 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 optimized implementation can only be written after the class invariant has been made explicit. Hence specification can have an impact on code development.

5 Decimal multiplication

In this section we specify the private method *mul* for multiplication. It has two arguments of type short, similar to the private *add* method. The *mul* method is probably the most difficult one of the whole *Decimal* class, both in terms of specification and of implementation.

We first consider its specification, in Fig. 7. Like in the specification of addition in Fig. 3, 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.

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

where

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

The difficulty lies in part X. In the Java implementation it involves a certain truncation, which is rather complicated to describe in JML: it covers the greater part of Fig. 7, from line 8 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

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

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

```
/*@ normal_behavior
  @     requires -PRECISION < f && f < PRECISION;
  @   modifiable intPart, decPart;
  @     ensures intPart * PRECISION + decPart ==
  @          \old(intPart) * e * PRECISION
  @          + \old(intPart) * f
  @          + \old(decPart) * e
  @          + ( ((f >= 0 && \old(decPart) >= 0) ||
  @               (f <  0 && \old(decPart) < 0))
  @              ? 1
  @              : -1)
  @          * (( ((-100 <= f && f <= 100)
  @                   ? abs(f)
  @                   : 10 * (abs(f) / 10))
  @               *
  @               ( (-100 <= \old(decPart) &&
                      100 >= \old(decPart))
  @                    ? abs(\old(decPart))
  @                    : 10 * (abs(\old(decPart))/10))
  @             ) / 1000);
  @*/
private void mul(short e, short f){ ... // see Fig. 5 }
```

Fig. 7. JML specification of the private multiplication method.

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 auxiliary function:

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

The combined specification (2) and (3) can then be proved for the *mul* implementation in Fig. 5. We shall briefly sketch the proof, explaining some of the more special points. The method body in Fig. 5 can be split into two parts:

(a) lines 2–24, establishing the equation (2) without `X`;
(b) lines 25–51, establishing the equation (2) including `X`;

Part (a) uses two `for` loops, firstly for $\mathsf{abs}(\mathtt{e})$-times adding `\old(intPart)` and `\old(decPart)` in lines 9 and 10, and secondly for `\old(intPart)` adding only `f` in lines 20 and 21. These two steps are combined in line 24, via the intermediate values `intPart_` and `decPart_`. Part (b) starts by setting `signe` $=$ $\mathsf{sgn}\big(\mathtt{f * \backslash old(decPart)}\big)$ which is the first part of (3), in two `if` statements. At the same time, `arrondis1` $=$ $\mathsf{abs}(\mathtt{\backslash old(decPart)})$ and `arrondis2` $=$ $\mathsf{abs}(\mathtt{f})$. The two subsequent `while` loops handle the truncations. It follows from our invariant and precondition that both these `while` loops are executed at most once. 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 X equation (3), which is added to the final result.

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

Fig. 8. Implementation of the private multiplication method.

6 Conclusions

This paper describes a case study in JML specification and PVS verification for JavaCard, in the style of [4]. It forms an incremental improvement on [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.

[4] We have decided to follow this clumsiness in our JML specification in Fig. 7.

- It cannot be over-emphasized 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 Fig. 3) and verbose and unreadable (like for *mul* in Fig. 7). 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; 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.

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 had been done with all parameters optimally tuned. But probably this never happens in practice. But if we have to make an estimate: a method like *mul* (with a specification) would take a day to verify.

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

[6] N. Cataño and Marieke Huisman. Formal specification of Gemplus's electronic purse case study. In *Formal Methods Europe (FME 2002)*, Lect. Notes Comp. Sci. Springer, 2002. To appear. An earlier version was presented at Verificard'02, first annual meeting of the VerifiCard project, Marseille, January 2002.

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

[8] M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle*. PhD thesis, Univ. Nijmegen, 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] G. T. Leavens, Albert L. Baker, and C. 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. `www.research.compaq.com/SRC/esc/Esc.html`.