Formal methods for smart cards: an experience report[☆]

C.-B. Breunesse[b,][*], N. Cataño[a], M. Huisman[a], B. Jacobs[b]

[a]*INRIA Sophia Antipolis, France*
[b]*University of Nijmegen, The Netherlands*

Abstract

This paper presents a case study in the formal specification and verification of a smart card application. The application is an electronic purse implementation, developed by the smart card producer Gemplus as a test case for formal methods for smart cards. It has been annotated (by the authors) with specifications using the Java Modeling Language (JML), a language designed to specify the functional behavior of Java classes. The reason for using JML as a specification language is that several tools are available to check (parts of) the specification w.r.t. an implementation. These tools vary in their level of automation and in the level of correctness they ensure. Several of these tools have been used for the Gemplus case study. We discuss how the usage of these different tools is complementary: large parts of the specification can be checked automatically, while more precise verification methods can be used for the more intricate parts of the specification and implementation. We believe that having such a range of tools available for a single specification language is an important step towards the acceptance of formal methods in industry.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Formal specification languages; Smart cards; Java Card; Formal verification; JML; ESC/Java; Annotations

1. Introduction

With the emergence of smart cards, industry has become more interested in techniques to establish the correctness and security of the applications developed. Typical smart card applications like electronic purses and health care information holders contain privacy-sensitive information. For the acceptance of the use of smart cards in such domains, it is necessary that the users trust that details of their private life are not passed on to third parties. Industry has realized that the only way to ensure this is by the rigorous use of formal techniques for the specification and verification of smart card applications. Moreover, this is enforced in higher levels of evaluation schemes like Common Criteria. This paper does not deal with security aspects like data leakage, but investigates functional behavior and possible abnormalities such as null pointer exceptions. Proper functional behavior and safety properties are a first step towards secure applications. Other researchers investigate how JML can be used to specify actual security features [26].

However, as always, the problem is that formal methods are considered difficult to use. There is a wide range of tools available that can be used to establish different properties of an implementation. In general, each tool uses its own specification language. Thus, with every new tool one has to understand the techniques, underlying theory and the specification language used. And if one wishes to use different validation techniques on the same application, one has to adapt the specification accordingly each time. This large overhead to apply new techniques makes industry reluctant to apply formal methods; if they use formal methods at all, then preferably with a single tool using a specification language that they master well. Therefore, a first step to make formal methods more accessible would be to have a single specification language and different tools that can match (different aspects of) this specification with an implementation. These tools can then vary in their level of precision, but also in their ease of use. In general, tools that are very precise and allow one to check arbitrarily complicated specifications will need more user interaction than tools that check a limited subset of the specifications.

An interesting development in this direction is the JML project. JML — short for Java Modeling Language [18,19] — is an annotation language for Java. It allows one to write functional behavior specifications for Java programs, using a Java-like syntax. JML is designed to be relatively easy to understand for an experienced Java programmer. In fact, by now JML has become the de facto standard source code level specification language for the functional behavior of Java programs in the academic community. As a result, more and more tools that aim at the verification of Java programs are adopting JML as the property specification language (see [3] for an overview).

This paper reports on work done in this context, using different tools on (parts of) a single JML specification. For a single applet (consisting of 42 classes and 432 kB in total of code and documentation) a specification has been written in JML. The specification for this applet has been checked using ESC/Java [21,12], a tool for automatic static checking of Java programs, aiming at finding efficiently the most common program errors. Parts of the specification and implementation also have been verified within the LOOP project [29], a project that aims at full verification of Java programs using interactive theorem proving. The LOOP approach has been applied to some algorithms that manipulate data in an

intricate way, and whose verification falls completely out of the scope of ESC/Java (and other automatic tools).

Elsewhere the authors have reported on the individual case studies in two separate papers [2,5] with two separate tools, but here we want to show how the different techniques complement and contribute to each other in a natural way. We do not want to argue that a certain approach is better; in fact we think they should both be used. Given an implementation, for large parts of a specification, it might be sufficient to use static checking techniques to gain confidence in its correctness, but for the crucial algorithms full verification is necessary. However, the effort needed for verification is actually reduced by using static checking beforehand, because this can already identify the errors that are relatively easy to find.

The electronic purse case study that forms the basis for this work has been provided by Gemplus. It has been developed by several trainees, and later been extended by some members of the Gemplus research lab. The case study is publicly available.[1] It is intended to be an example of a Java Card[2] applet on which different formal methods could be tested. Gemplus provides the applet without a formal specification: the JML specifications are ours. It was known beforehand that the code contained bugs, but it gives a reasonable impression of how Java Card applets are structured. The work done on this case study, both with ESC/Java and LOOP, convinced smart card manufacturers to adopt JML and (at least) static checking in their development process.

This paper is structured as follows. In the next section the language JML is introduced, together with several tools using JML as the input language. In particular, ESC/Java and the LOOP tool are introduced. Then, Section 3 gives more details about the electronic purse case study. Next, Section 4 discusses several aspects of specifying Java Card applications and Section 5 gives an overview of how the purse case study is annotated and checked using ESC/Java, while Section 6 focuses on a single class and discusses its verification — using LOOP — in full detail. Finally, Section 7 draws conclusions and sketches a view on the further use of formal methods in industry.

2. JML

2.1. The language

The development of JML — short for Java Modeling Language [18] — was started by Gary Leavens and his team at Iowa State University, but is now a community process with many people involved [19]. There is a group of active users and tool developers, where new language proposals are discussed before they actually become part of the "official" language standard.

---

[1] Via http://www.gemplus.com/smart/r_d/publications/case-study/.

[2] Java Card is a dialect of Java that is used to program smart card applications. The Java Card language is an extended subset of Java; in particular it does not contain multi-threading, floats, doubles and multi-dimensional arrays, but it does contain some additional constructs, such as shareable interfaces, which are used to enable communication between different applets.

The JML language is designed to be easy and accessible for an average Java programmer. Therefore, the specifications use Java syntax and are written in the source code as specially marked comments. Markers `/*@ .. @*/` and `//@` enable the various JML tools to recognize the comments as JML annotations.

A simple JML specification consists of pre- and post-conditions for methods (denoted by the keywords `requires` and `ensures`) and class invariants, restricting the reachable state space of an object. However, if wanted and needed, much more complicated properties can be expressed. Here we present only a few language constructs that are necessary for understanding this paper; we refer to the standard language documents for a full description of the language [18,19].

First of all, a method specification can contain exceptional postconditions, so-called `exsures` or `signals` clauses. An exceptional postcondition specifies which conditions should hold, if a method terminates abruptly, because of an exception. A typical usage of exceptional postconditions is to specify that the exception is thrown before any instance variables have been changed (thus implicitly preserving the class invariants; see the example in Fig. 1 below).

Method specifications also can contain `modifies` clauses (also known as `assignable` or `modifiable` clauses, or frame conditions) that restrict which variables may be changed by a method call. Modifies clauses are crucial when doing modular program verification [20].

It is also possible to restrict the reachable state space of an object by specifying a `constraint`. In JML, this denotes a relation between the pre- and post-state of a method, restricting how a variable might change. One can for example specify that a variable is constant or that it can only increase. Notice that invariants and constraints could be expressed as pre-post-condition specifications for each method, but by specifying them explicitly, one gets a higher level of abstraction. Moreover, in this way they immediately carry over to subclasses, i.e. all methods in subclasses have to respect the invariants and constraints of superclasses.

Sometimes one also likes to specify explicitly that a condition holds at a particular point in a method body. For this, JML provides the `assert` annotation. If a method body contains such an `assert` annotation, this means that whenever control reaches this point in the method body, the associated condition should hold. This can be used for example to add outlines of correctness proofs to the implementation of a method body — as is used by the LOOP compiler; see below — but also to state that a particular control point never can be reached (using `assert false;`).

As mentioned before, the JML specifications use Java syntax. More precisely, the various conditions are written as side-effect-free boolean-typed Java expressions. To make the language more expressive and usable, several additional specification constructs are available. Again, we mention only the few that are relevant for this paper. For further information we refer to the JML documentation [19].

First of all, there is a special keyword `\result` that refers to the return value. This keyword can only be used in `ensures` clauses. One can refer to the value of an expression E in the pre-state (before method body execution) by writing `\old(`E`)`. This keyword can be used both in the `ensures` and `signals` clauses. Keyword `\old` is used to see how an expression differs from its original value.

```
public class Decimal extends Object{

    public static final short MAX_DECIMAL_NUMBER = (short) 32767;
    public static final short PRECISION = (short) 1000;

    /*@ spec_public @*/ private short intPart = (short) 0;
    /*@ spec_public @*/ private short decPart = (short) 0;

    /*@ invariant 0 <= intPart && intPart <= MAX_DECIMAL_NUMBER &&
      @            0 <= decPart && decPart < PRECISION;
      @
      @ model int decimal;
      @ represents decimal <- intPart * PRECISION + decPart;
      @ depends decimal <- intPart, decPart;
      @*/

    /*@ behavior
      @    requires true;
      @    modifies decimal;
      @    ensures decimal == v * PRECISION;
      @    signals (DecimalException e)
      @            v < 0 &&
      @            decimal == \old(decimal);
      @*/
    public Decimal setValue(short v) throws DecimalException{
      if(v < 0)
        DecimalException.throwIt(DecimalException.DECIMAL_OVERFLOW);
      intPart = v;
      decPart = (short) 0;
      return this;
    }
    .
    .
    .
}
```

Fig. 1. Fragment of the annotated class `Decimal`.

Finally, to have a higher level of abstraction in specifications, JML allows one to declare so-called model variables. These are variables that exist only at the level of the specification. Declarations of model variables have the same format as declarations of normal variables, but are preceded by the keyword `model`. Model variables can be related to concrete variables (or other model variables) by `represents` and `depends` clauses. A represents clause specifies how the value of a model variable can be calculated from the values of the concrete variables. A depends clause only specifies on which concrete variables the value of the model variable depends. Hence if all the concrete variables in the depends clause are unchanged, one can assume that the value of the model variable is also unchanged, and if the model variable may be modified, the concrete variables implicitly also may be modified. If a represents clause is given, the information in the depends clause is redundant. However, since often it is not possible to give the represents clause — for example when specifying an abstract class — it is useful to also have the

depends clause. For more information on depends and represents clauses and their use in modular verification we refer to the work of Leino [20]. ESC/Java provides a lightweight variation of model variables, called `ghost` variables. However, since it is not possible to specify represents and depends clauses, their use is limited.

To conclude this section, Fig. 1 shows part of the JML specification of the class `Decimal` from the case study at hand. The class `Decimal` represents decimal numbers as composed of an integer and decimal part (`intPart` and `decPart`, respectively). As JML does not allow us to include private fields in public specifications, we add `/*@ spec_public @*/` immediately before the declarations of these two fields. This keyword causes the fields to be included in the scope of every specification. The class invariant restricts the possible values of these variables: `intPart` is a positive number, less than the constant `MAX_DECIMAL_NUMBER` — the maximal value of a `short`, while `decPart` ranges between 0 and the constant `PRECISION`. The value of `PRECISION` is 1000, thus the class `Decimal` will represent decimal numbers up to three decimal places.

To denote the value of the decimal number represented, a model variable `decimal` has been declared. The value of this variable depends on `intPart` and `decPart`, and the `represents` clause shows how.

As an example, we show the specification of the method `setValue`, which takes a single argument `v` and as a result sets the decimal number to represent `v.000`. We do not explicitly specify any precondition for this method[3] (it is a defensive specification, as discussed in Section 4). The method may modify `decimal`, thus implicitly it may modify the variables `intPart` and `decPart`. If the method terminates normally, the value of `decimal` is set appropriately (and because of the represents clause and the class invariant this implies that `decPart` and `intPart` are set appropriately). If the method terminates abruptly with a `DecimalException`, then this is because the argument `v` is less than 0. In this case, the value of `decimal` (and implicitly of `intPart` and `decPart`) is unchanged.

2.2. The tools

As mentioned before, there is a wide range of tools available using JML as specification language. This is one of the reasons why JML is becoming the de facto standard specification language for source code level specification of Java programs. For our case study, we have used the Extended Static Checker for Java (ESC/Java) [21,12] and the LOOP compiler [29]. We will describe these tools in some detail, followed by a brief description of other tools that are available for JML. Again, for a more extensive overview of the various tools, see [3].

2.2.1. ESC/Java

ESC/Java has been developed at Compaq Research, in the group led by Rustan Leino [21,12]. Currently, it is no longer maintained by Compaq, but it is available as open source software. An improvement making it compatible with official JML (ESC/Java 2) is done by David Cok (Kodak) and Joseph Kiniry (Nijmegen) [10]. At the time of writing

---

[3] Thus, by default, this method has precondition `requires true;`.

```
//@ requires o != null;                  void p(Object o) {
void m(Object o) {                         o.j();
  o.n();                                 }
}                                        ESC/Java checks whether o is not
ESC/Java finds no problem.               null
//@ requires true;
```

Fig. 2. ESC/Java example.

and verifying the ESC/Java specifications in this paper, version 2 was not yet available. Therefore, the ESC/Java version used here is Compaq release 1.2.4.

The initial design goal of ESC/Java was to develop a tool which could efficiently find common programming errors, such as indexing an array out of bounds or null-pointer dereferencing. The user can guide the search for errors by putting appropriate annotations in the code. For example, a user might specify that some method argument should always be a non-null-reference (the `requires` clause of method m in Fig. 2). When checking the method body, the tool will then assume that this is actually the case (thus the call `o.n()` in Fig. 2 will not raise a null pointer exception), but for every call to this method, it will check whether the assumption actually holds (thus ESC/Java will warn for a potential problem in the call `m(x)` in method p in Fig. 2, because it cannot ensure that x is non-null).

ESC/Java proceeds by generating verification conditions, based on the annotations and the code. The verification conditions are sent to a dedicated theorem prover, called Simplify.[4] ESC/Java issues a warning if Simplify cannot establish the proof obligation. However, such a warning does not necessarily mean that the program is incorrect, as both Simplify and the modeling of the Java semantics underlying ESC/Java are unsound and incomplete. The designers chose to accept this, in order to keep the simplicity and good performance of the tool. Also, if the tool does not issue any warning, this does not necessarily mean that the program is correct. Again, to keep the performance and simplicity of the tool, and to avoid many spurious warnings, the designers of the tool do not always generate all the necessary verification conditions. For example, a loop statement by default is approximated by a single loop iteration. The ESC/Java manual [21] contains a detailed list of known sources of unsoundness and incompleteness of the original ESC/Java.

The specification language that is used by ESC/Java is not exactly a subset of JML. Already some work has been done on integrating the two languages [8] and currently work on this is continuing. However, for this paper the exact differences are not relevant. It is sufficient to know that ESC/Java supports the specification constructs described above, except for the `constraint` and `modifies` construct and the support for model variables.

2.2.2. LOOP tool

The LOOP tool has been developed at the University of Nijmegen. Its purpose is to offer a sound environment within the theorem prover PVS [25] in which formal verification of JML specifications written for Java source code can be performed. What is usually called

---

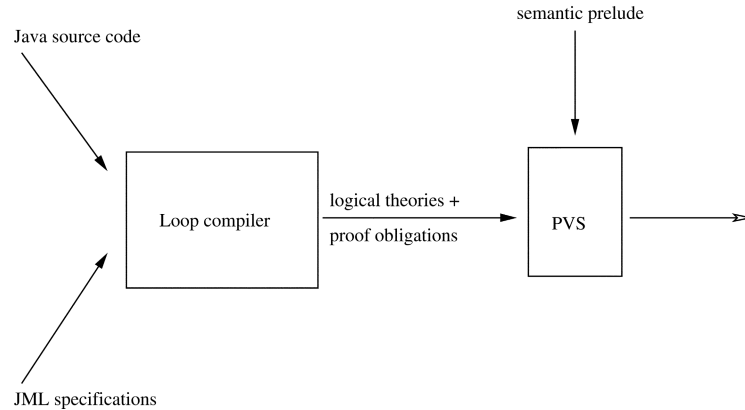[4] See http://research.compaq.com/SRC/esc/Simplify.html.

Fig. 3. LOOP tool schema.

the LOOP tool actually consists of a collection of tools. The relation between the different parts is shown in Fig. 3.

Java source code and associated JML specifications are fed into the LOOP compiler which generates PVS code. The Java source code is translated to PVS logical theories which are based on the handwritten "semantic prelude". This prelude defines sequential Java in all its details. The JML specifications are translated into PVS predicates. The aim is to show that the predicates generated from the JML annotation hold for the translated Java source code. The actual interactive verification work thus takes place *inside* PVS.

In the LOOP translation of Java methods to PVS theories, the structure of the methods remains intact. While proving, we can therefore step through the method body using different techniques. The following list gives an overview of available techniques for proving the correctness of a method given its specification.

(1) All Java language constructs have appropriate *Hoare rules* associated with them [17]. All these rules are proved correct in terms of the underlying semantics. The Hoare rules are used to split up the proof obligation in several smaller proof obligations. For example, the Hoare rule for composition splits up the proof obligation in two parts. The downside of reasoning with Hoare rules is that they require user interaction. The Hoare rule for composition needs an intermediate predicate which is the post-condition for the first statement, and the pre-condition for the second one.

(2) To avoid excessive user interaction, several *Weakest Precondition* (WP) techniques [16] can be applied to enable automatic proofs of non-recursive programs. Like the Hoare rules, the Weakest Precondition rules are also proved correct in terms of the underlying Java semantics.

(3) When a proof is completely split up and decomposed after applying Hoare or Weakest Precondition rules, the remaining proof obligations, if any, must be verified by "semantic rules" and logical deduction. These semantic rules are either generated by the LOOP compiler, or present in the semantic prelude [14]. The semantic rules

describe the actual Java semantics. Applying these rules eventually brings a Q.E.D., or an unprovable formula.

The three proof methods above are used in combination. Sometimes a method body is too long to be handled by WP (PVS might fail due to lack of resources). The body is then split up by Hoare rules. When a proof obligation is on the granular level (i.e. assignments) we revert to using semantic rules.

Because the whole system is sound (modulo the soundness of PVS, and the handwritten semantic prelude of course), specifications verified by LOOP can be trusted. The biggest downside of doing such heavyweight verification work is the cost of user interaction.

For example, when applying the Hoare rule for composition on statement `s1;s2`, an intermediate predicate has to be constructed to serve as a post-condition for `s1` and a pre-condition for `s2`. Constructing intermediate predicates in PVS is painstaking because one has to write these in terms of the semantic prelude. Fortunately, we can avoid constructing such a predicate in PVS by writing intermediate predicates *in* the Java code using in-line JML assertions. The LOOP tool converts these assertions to intermediate predicates in terms of the semantic prelude. Implementing support for in-line assertions is part of the plan to reduce user interaction. Apart from in-line assertions, there is also support for loop variants (JML keyword `decreasing`) and invariants (JML keyword `maintaining`). The JML used by the LOOP compiler is a subset of JML. We do not, for example, cover quantifications outside behavior specifications. The core of JML is supported though. We also added some modifiers for JML assert statements (`assert`) saying whether the previous assert still holds.

2.2.3. Other tools

To illustrate the wide range of formal techniques that are available when one decides to use JML as specification language, we briefly survey several other tools that use JML as specification language.

When the development of JML started, it was intended to be used with a runtime assertion checker (as advocated in the Design by Contract approach in Eiffel [23]). The JML tool, which is developed at Iowa State University, does exactly this: it translates annotations into runtime checks. When running the translated code on example input, every time an assertion is violated, an exception is thrown.

Further, at MIT one uses JML as the specification language for Daikon [9], which is a tool for invariant detection. Based on several test runs, this tool tries to establish possible class invariants.

There are also several tools which are inspired by ESC/Java. Chase [6], developed at INRIA Sophia Antipolis, is a tool for checking the modifies clauses, an aspect of JML which is not treated by ESC/Java. At Compaq, a tool called Calvin [13] has been developed. Calvin can be used to check properties about multi-threaded programs. It uses a technique to reduce the proof obligations to proof obligations on single threads (using appropriate annotations), which then can be checked using ESC/Java.

Further, at Gemplus, a tool called Jack has been developed [4]. Jack works similar to ESC/Java, in that it generates proof obligations based on the annotations and then sends those off to a prover, but it aims at being sound and complete. Jack has been designed in
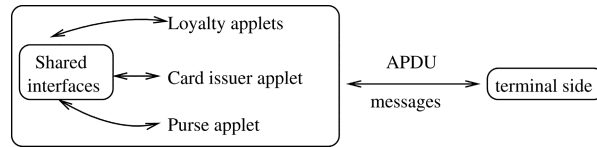
Fig. 4. The electronic purse.

such a way that it can interface different proof tools (currently AtelierB, Simplify and Coq). Further, Jack has been integrated with a standard IDE, which makes it easy to use for a Java developer. Other tools for full program verification using interactive theorem provers are Jive and Krakatoa. The Jive tool [24], which is developed in Kaiserslautern, implements a Hoare logic for Java [27] and generates proof obligations for PVS or Isabelle.[5] The Krakatoa tool [22] is developed at INRIA Rocquencourt. It uses the Why tool [11] to generate verification conditions as Coq proof obligations.

3. The Gemplus electronic purse

The Gemplus electronic purse is a smart card application that has been developed to serve as a realistic example for researchers working on formal methods for the Java Card platform [1]. However, it is too big to fit on most of the cards that are currently available.

Any Java Card application consists of two parts: the terminal side, implementing the configuration and communication functionalities, and the card side, implementing the Java Card application itself (see Fig. 4). These two sides communicate with each other by sending APDU (Application Protocol Data Unit) messages, which is an ISO standard defining the way in which commands and data are structured. In Java Card, the APDU is a class with a member of type byte array containing the raw data.

On the card side, the electronic purse provides the card holder with banking functionalities such as *credit*, *debit* and *currency change*. The card side of the purse application contains three kinds of applets: *loyalty applets*, the *card issuer applet* and the *purse applet*. These applets communicate with each other by means of shared interfaces, the standard mechanism of communication between applets. When the purse applet wishes to call a method of a loyalty applet, it requests the loyalty applet for an object implementing the loyalty shareable interface. The loyalty applet then decides whether to give the purse a reference to such a shareable interface object.

The card issuer applet keeps information of the card holder such as name and identifier and PIN code, which is necessary to initialize the card.

The purse applet implements the basic operations of credit, debit and currency change, and also implements mechanisms for installing, selection and de-selection of the applet. The purse applet interacts with loyalty applets in such a way that whenever the card holder has made a purchase, the loyalty applet can use this information to increase an internal counter of loyalty points. These loyalty points can be used later to make purchases.

---

[5] In fact, Jive does not use JML, but it uses a JML-like language.

```
/*@
  @ requires aid != null;
  @ modifies data[*];
  @ ensures (\forall int i; 0 <= i & i < nbLoyalties ==>
  @             (\forall int k; 0 <= k & k < data[i].aid.length ==>
  @                  data[i].aid[k] == aid.theAID[k]) ==>
  @                  !data[i].logfullInformation);
  @*/
void removeNotification(AID aid) {
    byte i = 0;
    while(i < nbLoyalties) {
        AllowedLoyalty al = data[i];
        if(al.getAID().equals(aid)) {
            al.dontKeepInformed();
        }
        else i++;
    }
}
```

Fig. 5. Example heavyweight specification.

The purse applet keeps track of the balance of the purse, the transactions done by the purse, the different currency changes that have taken place and the different loyalty programs that the card holder is subscribed to. Certain operations can be done only for a restricted set of users, which can be recognized for instance by requiring a PIN code. The purse applet defines the different access conditions and also binds these access conditions to operations. So, when the card holder wishes to do some operation, the purse application will check whether the card holder has the appropriate permissions.

Finally, the electronic purse contains several classes implementing cryptographic concepts. In this case study we did not study this.

4. Specifications for smart card applets

When writing behavioral specifications, several issues concerning the style of specifications have to be considered. Notice that many of these issues are relatively independent of the typicalities of smart card applications. However, the choices that we make are often influenced by the application domain. The first subsection discusses several of these issues, and explains our choices for this case study. The second subsection discusses the existing specifications for the Java Card API.

4.1. Specification style

4.1.1. Lightweight vs. heavyweight specifications
The first point one has to decide upon is how "heavy" a specification should be. That is, does one just want to specify under which conditions (no) exceptions will occur, or does one also want to specify exactly which postconditions are established, i.e. the complete behavior of a method or class.

Naturally, the more information that is given by a specification, the better it reflects the behavior of a program. However, in some cases the postcondition of a method might be too complicated to formulate or would basically require repeating the method implementation. One also has to remember that giving specifications only makes sense when one has reasonable confidence that the specification is correct. Thus, when specifying methods with complicated control structures or data manipulations and checking these specifications with an automatic tool such as ESC/Java, it might not make sense to specify complicated postconditions, because one can never rely on their correctness, given the limitations of such tools. If one wishes to actually specify and verify such complex postconditions, one should use e.g. LOOP and do full verification.

Additionally, it is also important to consider what is the return of the investment put into writing specifications, i.e. how does the number of bugs found relate to the amount of time spent on writing the specifications. If most bugs can be found by writing lightweight specifications only, one seriously has to consider whether it is worth the extra effort of making heavier specifications. This issue is very important in industry, where every investment has to be economically justified. Thus, it is important to find the right balance between completeness and reliability of a specification.

To illustrate this, Fig. 5 contains a heavyweight specification of `removeNotification` in class `LoyaltiesTable`. The precise behavior of this method is not so important, but what is important to see is that this method contains two loops (one explicitly in the `while` statement, and one implicitly in the call `aid.equals`), and the postcondition aims at describing precisely the intended behavior of these loops. However, in this case ESC/Java is not able to establish whether the postcondition is correct; one would need full verification — using e.g. LOOP — for this.

With respect to this, we also would like to emphasize that even so-called lightweight specifications are useful, because they describe exactly under which conditions (no) exceptions will occur. For the correct functioning of a program, unexpected exceptions are often a bigger threat than the risk of incorrect calculations. Also, programmers tend to pay more attention to the correctness of a computation than to whether it will handle all possible cases and will not throw an unexpected exception.

The work that is done in this case study illustrates how the tools that one has at hand influence the level of detail in the specifications. Most of the specifications have been checked with ESC/Java, so that one cannot rely on the correctness of postconditions for methods with complicated control structures or data manipulations, such as `removeNotification` in Fig. 5. Similarly, for the addition and multiplication methods in the class `Decimal`, ESC/Java cannot establish any postcondition other than `true`. In contrast, using the LOOP tool, specifications describing the complete behavior of these two methods have been verified, as discussed in Section 6.

4.1.2. Defensive vs. offensive specifications

Independently of the question how complete one's specifications should be, one also has to decide whether to write defensive or offensive specifications.

An offensive method implementation requires that its input parameters are valid and correct. It will not test whether this is the case; it is up to the caller of the method to ensure it. If the method is called with inappropriate parameters, nothing can be guaranteed

```
Offensive specification for m()        Defensive specification for m()
/*@ requires a != null;               /*@ requires true;
  @ ensures Q;                          @ ensures Q;
  @ signals (E) false;                  @ signals (E) \old(a) == null;
  @*/                                   @*/
void m(int[] a) {                      void m (int[] a) {
...                                    ...
}                                      }
```

Fig. 6. Offensive vs. defensive specification.

about its behavior. In contrast, a defensive method implementation does not make any requirements on its input parameters: before manipulating them it will check for their validity. Typically, it will throw an exception if the input is invalid (or do some sort of error recovery, e.g. replacing it by some default value).

When writing specifications, this aspect of the method's behavior will be made explicit, An offensive specification will state in its precondition under which conditions the method will function correctly (i.e. as specified). Nothing is guaranteed on the behavior of the method, if the precondition is not respected. When the program is verified, for each method call to the method, one will be obliged to show that the precondition is respected. A defensive specification, however, will not make any requirements on the caller of the method (i.e. its precondition will be `requires true;`), but it will typically specify which exceptions are thrown when the input parameters are invalid. Fig. 6 sketches an offensive and a defensive specification for the same method. When verifying the program, in the offensive case, one has to verify that the precondition `a != null` is respected for each call to the method `m()`. In the defensive case, no proof obligation exists for the precondition, but one has to take into account that the method might finish abruptly, because of an exception, if it is called with `a == null`.

Often, offensive specifications are considered as the better way to write specifications (see e.g. [23]). However, it might be the case that it is prescribed explicitly that no assumptions may be made on the state of the caller. In such cases it is necessary to write defensive specifications. This choice is ultimately governed by an object's interface to the outside world. For example, for a private method it might still be possible to write an offensive specification, if all calls to this method guarantee the precondition.

In this case study we started writing specifications for existing code. At first we decided to leave the code unchanged.[6] Therefore we write defensive specifications for methods that make explicit tests on the values of their arguments, thus following the implementation. However, as discussed in Section 5, we also found several places where unnecessary tests were made and the exceptional cases never could occur.

---

[6] However, when writing formal specifications one gets a good idea about possible improvements. Therefore, Section 6 presents the verification of an improved version of the code (for the Decimal class only).

4.1.3. Level of abstraction of specifications

When writing readable specifications, it is important to have a reasonable level of abstraction. Two techniques for achieving a higher level of abstraction are the use of pure methods, i.e. side-effect-free methods, and the use of model variables in specifications. Both are provided by JML and supported by the LOOP compiler, but ESC/Java does not support them (except for the use of ghost variables, which is a more limited variation of model variables).

Since within this case study most of the specifications are checked using ESC/Java, the specifications often are more verbose than we would have liked. In particular, most of the specifications are written in terms of the concrete variables and without introducing any abstractions. However, for the class `Decimal` specifications with and without model variables exist. Fig. 1 (in Section 2.1) contains part of the specification of `Decimal` with model variables. As an example, the same method `setValue` is specified as follows for checking with ESC/Java.

```
/*@
  @ requires true;
  @ modifies intPart, decPart;
  @ ensures intPart == v;
  @ ensures decPart == (short) 0;
  @ ensures \result == this;
  @ signals (DecimalException) v < 0;
  @*/
public Decimal setValue(short v) throws DecimalException{ ... }
```

An example where abstraction could improve the readability of the specification is in the communication between card and terminal. As explained above, this communication takes place by sending so-called APDUs, which are commands encoded as arrays. However, when writing specifications one would prefer to do this in terms of the abstract notion of commands, instead of in terms of the contents of the APDU buffer.

4.2. Specifications for the Java Card API

We used the specifications for the Java Card API written by Erik Poll and other members of the LOOP project [28] as the basis for this work. In short, these are offensive specifications that describe when methods are guaranteed not to throw unwanted runtime exceptions. We cannot present here the JML specifications for the whole Java Card API, but as an example we discuss the method `arrayCopy` of the Java Card API.

```
/*@ requires src!=null & srcOff>=0 & srcOff+length<=src.length
  @    & dest!=null & destOff>=0 & destOff+length<=dest.length
  @    & length>=0;
  @ modifies dest[*];
  @ ensures (\forall int i; (i<=0 & i<dest.length) ==>
  @          (destOff<=i & i<destOff+length) ?
  @              dest[i] == src[srcOff + (i - destOff)] :
  @              dest[i] == \old(dest[i]));
  @ ensures \result == destOff+length;
  @ signals (TransactionException e)
  @              e._reason == TransactionException.BUFFER_FULL &
```

```
@               JCSystem._transactionDepth == 1;
@  signals (NullPointerException) false;
@  signals (ArrayIndexOutOfBoundsException) false;
@*/
public static final native short arrayCopy(
        byte[] src, short srcOff, byte[] dest, short destOff,
        short length)
throws ArrayIndexOutOfBoundsException, NullPointerException,
       TransactionException;
```

The method `arrayCopy` copies `length` bytes located at position `srcOff` in array `src` to position `destOff` in destination array `dest`. The precondition for this method requires that `src` and `dest` have the right size and that the number of bytes for copying is a positive number. The postcondition specifies that if the method terminates without raising any exception, then the method will have copied `length` bytes from `src` starting in the position `srcOff`, to `dest` starting in the position `destOff`. Otherwise, the method terminates abruptly raising an exception of type `TransactionException` but the method will never raise a `NullPointerException` or `ArrayIndexOutOfBoundsException` exception.

This is the kind of specifications that can be checked efficiently automatically: a strong precondition about array variables, ranges of values in which arrays can be accessed and conditions preventing the array variables of being null. This strong precondition allows one to specify that certain exceptions never can be raised; in this case `ArrayIndexOutOfBoundsException` and `NullPointerException`. However, notice that this is not the case for the exception `TransactionException`: this exception can occur if for example the card holder decides to remove his card before the session is finished.

5. Checking the specifications for the electronic purse

After having discussed the general issues involved in writing specifications for smart card applications, this section focuses on the actual specifications for the electronic purse. As said before, we wrote a formal specification — using JML — for most classes of the Gemplus electronic purse case study. The majority of these specifications have been checked only with ESC/Java, but the specifications for the class `Decimal` have been verified completely, using LOOP technology. The next section discusses the LOOP verification in more detail; this section presents the general ideas behind the specification, and in particular it shows how relatively lightweight use of formal methods — using automatic tools only — can already help to improve an application. We show in particular how the formal specification helped to identify ambiguities and inconsistencies in the informal documentation and unnecessary checks in the implementation. However, this does not mean that full verification is not necessary: the verifications as described in the next section fall completely out of the scope of what can be checked with ESC/Java. On the other hand, the verification with ESC/Java on the entire purse falls out of the scope of the LOOP tool because the purse is simply too large. Manageable parts of source code that can be verified with LOOP are in the order of hundreds of lines of code.

Elsewhere [5], we have already reported on the different kinds of errors and possibilities for improvement that we found in the case study with ESC/Java. Here we will not discuss these in full detail, but we will give a more general overview of the typical kind of problems that can easily be found by automatic tools. However, to give an indication: we found roughly 20 errors and possibilities for improvement. Further, we found approximately 25 unreachable code fragments. Notice that most of these errors are straightforward to find, requiring only simple specifications.

When writing the specifications, we decided to take the code *as it is* as a starting point. Only when we found real mistakes in the code, would we change it, but we did not change the design or general structure. However, writing formal specifications forces one to think carefully about the code, and often it makes one aware of possibilities for improvement. Therefore, the verifications in the next section are actually done on an improved version of the class `Decimal`.

Since most of the specifications are checked only with ESC/Java, we cannot have unlimited trust in their correctness. Therefore, we aimed at writing specifications that were as "heavy" as possible, but where we still could feel confident in the outcome of the tool. The full annotations — with an overview of remaining warnings — are available on a website.[7] Typically, if one wishes to do full verification on fragments of the code, the remaining warnings and the methods containing loops or recursion are the most interesting, i.e. they are the most likely to still contain errors.

When checking the specifications, one is often forced to work in a bottom-up manner. First one has to annotate the classes that are at the bottom of the class hierarchy and that are used by many of the other classes, e.g. in this case study the utility classes. Afterwards, using these specifications, one can go up higher in the class hierarchy.

We found that writing and checking the formal specifications in particular was important for the following two reasons:

- formal specifications help to maintain the consistency between documentation and implementation and to avoid ambiguities in the documentation; and
- formally specifying class invariants — which exist often implicitly in the programmer's mind — allows one to show the redundancy of tests on the wellformedness of object states.

We will illustrate these aspects with some examples.

5.1. Informal documentation vs. formal specification

Over time, the electronic purse case study has been developed by several people (trainees and employees of Gemplus). Looking at the documentation and the implementation, it is evident that these different developers have not always worked with the same ideas in mind. For example, in the class `Decimal`, the informal documentation reads:

---

[7] See `http://www-sop.inria.fr/lemme/verificard/electronic_purse`.

Two important notes about a decimal number: it is limited to 32767 (short representation) and the decimal part must be done in the interval [000, 999].

This informal documentation immediately suggests that an appropriate class invariant would be the following (where `MAX_DECIMAL_NUMBER` is 32767 and `PRECISION` is 1000).

```
invariant 0 <= intPart && intPart <= MAX_DECIMAL_NUMBER &&
          0 <= decPart && decPart < PRECISION;
```

Notice that this class invariant is straightforward to write given the informal class documentation; no difficult formalizations are needed. However, having written this formal class invariant enables one to use the different JML tools to check whether this invariant is maintained everywhere. This is useful, because from the implementation of the class, it becomes clear that not all developers have adhered to this design decision about decimal numbers. For example, having this formal invariant allows one to detect immediately that there is a problem with the method `oppose`.

```
public Decimal oppose(){
  intPart = (short) -intPart;
  decPart = (short) -decPart;
  return this;
}
```

This method should not be declared `public`, because it can break the class invariant, by making `intPart` and `decPart` negative.

Having a formal class specification, instead of informal documentation only, cannot avoid all inconsistencies. For example, the class `Decimal` also contains methods `isPositif` and `isNegatif`, which denote whether the value is positive or negative, respectively. Since these methods do not break the class invariant, one does not immediately detect that they are useless, given the informal documentation. However, using the different JML tools it is easy to show that the results of these methods are actually constant values, and thus that these methods do not give any new information. For example, the method `isNegatif` can be specified as follows.

```
/*@
  @ ensures \result == false;
  @*/
public boolean isNegatif(){
   return (compareTo((short) 0) < (short) 0 );
}
```

Formal specifications do not only help to maintain an implementation over time, but because of their precise semantics they can also help to avoid the ambiguities, which typically occur in natural language documentation. To illustrate this, we take an example from the class `Transaction`. In this class, an instance variable `type` is declared as follows.

```
/* the transaction type: debit or credit */
/*@ spec_public @*/ private byte type;
```

Since the class also contains the following constant declarations:

```
/* the transaction status */
public static final byte TYPE_CREDIT = (byte)50;
/* the transaction status */
public static final byte TYPE_DEBIT  = (byte)51;
```

it seems natural to specify the following class invariant.[8]

```
/*@ invariant type == TYPE_CREDIT ||
  @              type == TYPE_DEBIT;
  @*/
```

However, when checking whether this invariant is respected by all methods in the class, it turns out that it is not: the class contains a method `reset()` which contains the assignment `type = INDETERMINE;`, where `INDETERMINE` is another constant declared in the class. One could consider this as an error — given the documentation — but it also seems reasonable that this is done on purpose. After all, when initializing or resetting a transaction, one cannot say whether it will be a credit or debit operation, and also neither of them is a typical default value. Having a formal specification would clarify the initial intention of the developer, and would allow one to signal this problem earlier.

To summarize, we would like to stress that writing a — relatively easy — formal specification, instead of informal documentation helps to maintain the consistency of an implementation over time and to avoid ambiguities. When design decisions, such as restricting the possible range of a variable, are specified formally, each time the implementation is modified the various JML tools can be used to check whether the design decisions have not been violated. And if they are violated, one is forced to decide whether the design decision or the implementation has to be changed. Notice that many of these violations can be detected by automatic tools, thus they can be found with quick checks — over and over again — during the development process.

5.2. Class invariants

In fact, class invariants do not only serve to formally specify certain design decisions; they can also be used as a formal motivation that certain tests are unnecessary. To illustrate this, we consider the method `setTaux`,[9] also from the class `Transaction`.

```
void setTaux(Decimal tx) throws ISOException  {
  try{
    taux.setValue(tx);
  }
  catch(DecimalException e){
    //@ assert false;
    ISOException.throwIt(PurseApplet.DECIMAL_OVERFLOW);
  }
}
```

---

[8] Notice that there exist several programming languages in which this class invariant could be described precisely using enumeration types. Violations are then found immediately by type checking.

[9] *Taux* is the French word for rate.

This method has a parameter `tx`, which is an instance of class `Decimal`. Thus, we know that `tx` respects the class invariant for `Decimal`, as specified. The method then assigns `tx` to the instance variable `taux`, using the `setValue` method of the class `Decimal`. Since this method can throw an exception, the call to `setValue` is put in a `try-catch` statement. However, given the specification of `setValue` and since the argument `tx` is a legal instance of class `Decimal`, respecting its invariant, we can actually prove that this exception will never be thrown. This is emphasized by the `//@ assert false;` annotation, which means that this part of the program text will never be reached. In fact, because of the class invariant of `Decimal`, we can give the following method specification for `setTaux`.

```
/*@ requires tx != null;
  @ modifies taux.intPart, taux.decPart;
  @ ensures taux.intPart == tx.intPart;
  @ ensures taux.decPart == tx.decPart;
  @ signals (ISOException) false;
  @*/
```

Notice in particular that we do not have to specify any conditions on `tx.intPart` or `tx.decPart`; it is sufficient that we know that `tx` respects its class invariant.

As another example on the use of class invariants, we consider the class `Access-Condition`. The electronic purse is implemented in such a way that certain operations can be performed only when the card is in a particular state. The class `AccessCondition` implements operations to check whether the card is in the appropriate state. Following the documentation of the electronic purse, there is a variable `condition`, which can have only one of the following constant values: FREE, LOCKED, SECRET_CODE, SECURE_MESSAGING or SECRET_CODE and SECURE_MESSAGING. This last combination is represented by the bitwise or of the two constants. This restriction on the variable `condition` is easily expressed as a class invariant.

```
public static final byte FREE             = (byte)1;
public static final byte LOCKED           = (byte)2;
public static final byte SECRET_CODE      = (byte)4;
public static final byte SECURE_MESSAGING = (byte)8;
/*@ spec_public @*/ private byte condition = FREE;

/*@ invariant condition == FREE ||
  @           condition == LOCKED ||
  @           condition == SECRET_CODE ||
  @           condition == SECURE_MESSAGING ||
  @           condition == (SECRET_CODE | SECURE_MESSAGING)
  @*/
```

When deciding whether the card is in the appropriate state to perform a certain operation, this is basically implemented by a case distinction (a Java `switch` statement), as in this method `verify`.

```
/*@ signals (AccessConditionException) false;
  @*/
private final boolean verify(byte c)
    throws AccessConditionException {
  byte t = (byte)0;
```

```
  switch(condition) {
    case FREE: return true;
    case SECRET_CODE: return [...]
    case SECURE_MESSAGING: return [...]
    case SECRET_CODE | SECURE_MESSAGING: return [...]
    case LOCKED: return false;
    default:
      //@ assert false;
      t = AccessConditionException.CONDITION_COURANTE_INVALIDE;
      AccessConditionException.throwIt(t);
      return false;
  }
}
```

For our exposition, it is not important what this method actually computes, but we are interested in its structure. The case distinction distinguishes all possible values of the variable `condition`, as specified by the class invariant. Thus, the default clause will never be reached. Again, this is emphasized by our `//@ assert false;` annotation. Notice that since the default clause never will be reached, the method will never throw an exception, as expressed by the exceptional postcondition. In fact, because of the class invariant it is possible to remove completely the `throws` clause from this method declaration.

Thus to summarize, we would like to emphasize that many class invariants can be specified fairly easily: they directly reflect the informal documentation and do not contain any complicated specification constructs. Moreover, they also can be checked efficiently. Automatic checking techniques, such as ESC/Java, can easily find places where invariants are violated. Finally, an additional advantage of properly specifying class invariants is that it allows one to avoid unnecessary tests, because objects are known to respect their class invariants.

6. Focus on a single class: the verification of Decimal

Next, we switch our attention to the specification and full verification of a single class: `Decimal`. The specification and verification of the *original* `Decimal` class was done earlier [2]. This section is about the specification and verification of a *modified* `Decimal` class: one that fulfills our wishes in terms of clearer, shorter Java code and specifications that are supplemental to the ones constructed and checked using ESC/Java [5]. When we refer to the `Decimal` class, we mean our *modified* variant. Note that the specifications for the `Decimal` class in this section are "out of scope" for fully automatic tools as ESC/Java.

The `Decimal` class receives special attention because:

- it is an independent, self-contained class. Hence, the verification is independent of other specifications of the purse as well. This simplifies the verification process;
- the `Decimal` class is a backbone of the purse applet. Therefore verification is important, and its correctness is critical to the proper functioning of the purse applet. Many classes in the purse use `Decimal` as a library class;
- it is not only a backbone of the applet (so is the class that handles the APDU communications, for example) but it is also a functional class that does the actual

arithmetic needed to increase the balance on the card. Functional specifications can be expressed well using JML;

- its size is limited. Large Java classes with elaborate methods are very costly to specify and verify. The `Decimal` class consists of 40 methods, roughly 1.5 kB, whereas the whole purse is 432 kB of source code.

The `Decimal` class, the header of which is shown in Fig. 1, is used in the purse to store real numbers with a precision of three digits. Because floats do not exists in Java Card, the `Decimal` class is the place where this kind of arithmetic is defined. Two fields of type `short` are used for storage. When storing the value 3.493, the floored value 3 is put in field `intPart` and the rest value as whole number 493 is stored in field `decPart`. Negative numbers are *not* allowed.[10] This information is reflected in the class invariant.

```
/*@ invariant 0 <= intPart && intPart <= MAX_DECIMAL_NUMBER &&
  @            0 <= decPart && decPart < PRECISION;
  @*/
```

To ease reasoning, an abstract field is added to the `Decimal` class specification to denote the stored value. The value of the abstract field has a one-on-one relationship with `intPart` and `decPart`, denoted by its `represents` clause, repeated here for convenience.

```
/*@ model int decimal;
  @ represents decimal <- intPart * PRECISION + decPart;
  @*/
```

This so-called "representation function" is a function over the instance variables `intPart` and `decPart`. The abstract value of 3.493 is equal to 3493.

Out of the 40 members that make up `Decimal`, in the next subsection we give an outline of the specification and verification of only one method, the one that `mul`tiplies decimals. This method is interesting because it is the most complicated one and because its original implementation is "incorrect". Furthermore, it uses Java rounding tricks which give us the possibility to show our LOOP techniques w.r.t. reasoning with bounded arithmetic. Last but not least it shows the need of unbounded integral types in specifications, which is further discussed in Section 6.2.

Method `mul` uses method `add` to produce its result. The specification of `add` is shown below. Details about its verification can be found elsewhere [2].

```
/*@    requires 0 <= f && f < PRECISION &&
  @             0 <= e && e <= MAX_DECIMAL_NUMBER &&
  @             e + intPart + 1 <= MAX_DECIMAL_NUMBER;
  @    modifies decimal;
  @     ensures decimal == \old(decimal) + e * PRECISION + f;
  @    signals(Exception e) false;
  @*/
private void add(short e, short f) { .. }
```

---

[10] They are not allowed in *our* version. Whether or not they could occur in the original version of Gemplus remains unclear; see Section 5.1.

Arguments `e` and `f` contain respectively the integer and decimal part of the number to add. Because the argument to `add` is not a `Decimal` object, the pre-condition explicitly describes to which bounds `e` and `f` must adhere. For `Decimal` objects, these same bounds are described in the invariant.

6.1. Implementation and specification of `mul`

Multiplication of decimals is difficult because of rounding. Suppose we have two `Decimal` objects which represent decimal numbers $a.b$ and $e.f$ where a and e are the corresponding values of `intPart` and b and f are the corresponding values of `decPart`. Mathematically speaking, multiplication of $a.b$ and $e.f$ can be written as the sum of four simpler multiplications.

$$a.b * e.f = a * e + a * 0.f + 0.b * e + 0.b * 0.f. \tag{1}$$

The first three of these multiplications can be computed using the `add` method, because a and e are natural numbers. For example $a * e$ is computed by:

```
for ( short i = (short) 0; i < e; i++) {
   add(a, 0);
}
```

which means in English: do e times add a.

The tricky part is the last multiplication in (1), namely $0.b * 0.f$, which is called the "rest-part". Because b and f are less than 1000, we know that $0.b * 0.f = (b * f)/10^6$. The result of $(b * f)/10^6$ is always less than 1, and because we are only interested in the three most significant digits (out of six), we want to calculate the truncated value of the rest-part: $\lfloor (b * f)/10^3 \rfloor$, which thus lies within $[0, 1000)$. This truncated value must then be stored in `decPart`. A very reasonable solution is thus to write `add(0, ((b * f)/1000))`. Unfortunately this is not possible in Java Card. Arguments `b` and `f` are shorts, and the result of their multiplication possibly does not fit in a short, which is the biggest type available. We also cannot write `add(0,(b/1000 * f/1000))` because in Java `(b * f)/1000` is not always equal to `(b/1000) * (f/1000)` due to premature rounding of `b/1000` and `f/1000`. Remember, in Java `(347/10) * 10 == 340`.

Gemplus circumvented the above problem in their `mul` method for which we wrote a specification [2]. Unfortunately, their calculation of the rest-part is wrong. The specification for Gemplus's private `mul` is given below. The original code [2] is not very relevant here. The arguments of `mul` are two shorts, an integer and a decimal part. These values are multiplied with the `intPart` and `decPart` of the `this` object. The `ensures` clause defines the four multiplications of (1), where the rest-part is constructed by some complicated rounding. The Gemplus multiplication code satisfies the specification below, but mathematically speaking it is not defining a multiplication.

```
/*@    requires 0 <= f && f < PRECISION &&
  @            0 <= e && e <= MAX_DECIMAL_NUMBER &&
  @            (e + 1) * (intPart + 1) < MAX_DECIMAL_NUMBER;
  @    modifies decimal;
  @     ensures decimal ==
```

```
@                   e * \old(intPart) * PRECISION
@                   +
@                   \old(intPart) * f
@                   +
@                   e * \old(decPart)
@                   +
@                   /// difficult rest-part, given by four options
@                   ( (f > 100 && \old(decPart) > 100)
@                     ?
@                     ( ((\old(decPart)/10) * (f/10)) / 10 )
@                     :
@                     ( (f > 100 && \old(decPart) <= 100)
@                       ?
@                       ( (\old(decPart) * (f/10)) / 100 )
@                       :
@                       ( (f <= 100 && \old(decPart) > 100)
@                         ?
@                         ( ((\old(decPart)/10) * f) / 100 )
@                         :
@                         ( (\old(decPart) * f) / 1000 ))));
@    signals(Exception e) false;
@*/
private void mul(short e, short f) { ... }
```

The rest-part above is composed of case distinctions. Suppose we multiply 0.999 with itself; the answer given according to the above spec is 0.980. In precision arithmetic the answer to this multiplication is 0.998001. Rounded to three digits, this makes 0.998.

In the Nijmegen implementation we chose precision arithmetic. An advantage is that the specification of the rest-part becomes trivial. The private specification for the new `mul` is shown below.

```
/*@    requires 0 <= f && f < PRECISION &&
  @             0 <= e && e <= MAX_DECIMAL_NUMBER &&
  @             (e + 1) * (intPart + 1) < MAX_DECIMAL_NUMBER;
  @    modifies decimal;
  @     ensures decimal =
  @             e * \old(intPart) * PRECISION
  @             +
  @             \old(intPart) * f
  @             +
  @             e * \old(decPart)
  @             +
  @             (f * \old(decPart)) / PRECISION;
  @   signals(Exception e) false;
  @*/
```

The only problem left is to write an implementation that conforms to this specification without overflow. Note again that an implementation would be trivial (same as the spec) if Java Card would have integers. The new code for `mul` is shown below.

```
void mul(short e, short f){
  // intPart.decPart * e.f ==
  //    intPart * e +
```

```
//   0.decPart * e +
//   intPart * f +
//   0.decPart * 0.f
short a = intPart;
short b = decPart;

intPart = (short)0;
decPart = (short)0;

// a * e + 0.b * e
for (short i = (short)0; i < e; i++) {
  add(a, b);
}

// a * 0.f
for (short i = (short)0; i < a; i++){
  add((short) 0, f);
}

// 0.b * 0.f
short a1 = (short)(b / 100);
short a2 = (short)((b - a1 * 100) / 10);
short a3 = (short)(b - a1 * 100 - a2 * 10);

short d1 = (short)((a1 * f) / 10);
short d2 = (short)((a2 * f) / 100);
short d3 = (short)((a3 * f) / 1000);
short gross = (short)(d1 + d2 + d3);

short e1 = (short)((a1 * f - d1 * 10) * 100);
short e2 = (short)((a2 * f - d2 * 100) * 10);
short e3 = (short) (a3 * f - d3 * 1000);
short rest = (short)(e1 + e2 + e3);

add((short)0, (short)(gross + (rest / 1000)));
}
```

The rest-part of the new `mul` is computed by two shorts `gross` and `rest`, which are both additions of three shorts: `d1, d2, d3` and `e1, e2, e3`, respectively. The construction of these shorts is explained by unfolding the rest-part `(decPart * f)/ 1000` below. In the following equations `decPart` and `f` are called a and b respectively. When we write a_i, we mean the ith digit of a (counting from left to right).

$$(a * b)/1000 = ((a_1 \ a_2 \ a_3) * b) /1000.$$

Number a consists of 3 digits.

$$= \left(\begin{array}{ccccc} 100 & * & a_1 & * & b \\ + & 10 & * & a_2 & * & b \\ + & & & a_3 & * & b \end{array} \right) /1000.$$

Multiplication $a * b$ can therefore be split in 3.

$$= \begin{pmatrix} (c_1 & c_2 & c_3 & c_4 & 0 & 0) \\ + & & (c_1' & c_2' & c_3' & c_4' & 0) \\ + & & & (c_1'' & c_2'' & c_3'' & c_4'') \end{pmatrix} / 1000.$$

The results of each of the 3 multiplications are named c, c' and c''.

$$= \left(\begin{pmatrix} (c_1 & c_2 & c_3 & 0 & 0 & 0) \\ + & & (c_1' & c_2' & 0 & 0 & 0) \\ + & & & (c_1'' & 0 & 0 & 0) \end{pmatrix} + \begin{pmatrix} (c_4 & 0 & 0) \\ + & (c_3' & c_4' & 0) \\ + & (c_2'' & c_3'' & c_4'') \end{pmatrix} \right) / 1000.$$

Numbers c, c' and c'' are split in 2.

$$= \begin{pmatrix} (c_1 & c_2 & c_3) \\ + & (& c_1' & c_2') \\ + & (& & c_1'') \end{pmatrix} + \begin{pmatrix} (c_4 & 0 & 0) \\ + & (c_3' & c_4' & 0) \\ + & (c_2'' & c_3'' & c_4'') \end{pmatrix} / 1000.$$

The division by 1000 is distributed over these two components.

$$= \begin{pmatrix} \mathtt{d1} \\ + \ \mathtt{d2} \\ + \ \mathtt{d3} \end{pmatrix} + \begin{pmatrix} \mathtt{e1} \\ + \ \mathtt{e2} \\ + \ \mathtt{e3} \end{pmatrix} / 1000.$$

The names in `tt` font correspond to fields in the `mul` method. All the parts of this multiplication can be computed without the risk of overflow.

The specification of the `mul` method has been verified by a combination of Hoare logic and Weakest Precondition reasoning. Suitable intermediate predicates are inserted as JML assertions in the method body (not shown), and proved. The two for-loops require appropriate (in)variants (not shown). A 32-bit bounded representation for Java's numeric types has been used, see [15], in the translations of both Java and JML expressions. The semantics of 16-bit Java Card is preserved, because in the code no integer types are declared, and all casts are explicit.

6.2. Integral semantics in specifications

By evaluating all JML expressions in a bounded integral representation, overflow can occur in specifications. It is an issue under debate in the JML community what the right semantics of integral types in specifications should be; see [7]. Ideally, one would want to use the unbounded (mathematical) integers in specifications. This has the advantage that one does not have to worry about overflow in specifications. The disadvantage is that by using two different kinds of integral semantics, expressions can have different values in Java and JML; for example `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE` is true in Java, but false in unbounded JML.

The `mul` method specified and verified in the previous subsection is invoked by a public `mul` method which has a `Decimal` object `d` as argument, instead of two shorts. Its post-condition can be written completely in terms of model fields `decimal` and `d.decimal`.

```
/*@     requires d != null &&
  @         (d.intPart + 1) * (intPart + 1) < MAX_DECIMAL_NUMBER;
  @     modifies decimal;
  @      ensures decimal == \old(decimal * d.decimal) / 1000;
  @     signals(Exception e) false;
  @*/
public Decimal mul(Decimal d) {
  mul(d.getIntPart(), d.getDecPart());
  return this;
}
```

Unfortunately, this specification cannot be proved using bounded 32-bit integral semantics. The result of `\old(decimal * d.decimal)` possibly does not fit in an integer. When the JML community agrees on how to deal with bounded and unbounded specifications, the LOOP tool will be adjusted appropriately. Until then, the integral semantics of JML and Java remain coupled in LOOP technology, so it is either all bounded or all unbounded semantics.

7. Conclusions

This paper discusses experiences with the specification and verification of an industrial smart card case study, using JML. One of the advantages of the JML specification language is that it comes with a spectrum of validation tools, ranging from runtime assertion checking through static checking to interactive verification. The case study in this experience report was handled by the two latter verification techniques, using Compaq's ESC/Java tool and the LOOP tool from the University of Nijmegen. It turned out that ESC/Java works best for relatively lightweight specifications, for which it can give immediate feedback. It filtered out many common programming errors from the entire purse applet. The LOOP tool can also be used on such lightweight specifications, but it is typically applied to more detailed functional specifications in selected smaller code fragments. In this case study a precise high-level specification for the `Decimal` class underlying the purse was developed, implemented (in a different way than the purse developers originally did), and proven correct. The verification was done interactively, via a combination of Hoare logic and Weakest Precondition reasoning. This has demonstrated that using different levels of verification, with corresponding tools, gives an attractive (and feasible) combination of global checking and selected local verification.

Part of this verification effort was developing the specifications. One would hope that, in the future, advanced programmers will write such specifications themselves. This would considerably reduce the effort spent on such verification projects. The whole verification underwent several updates and adaptations (w.r.t. [2,5]), so it is hard to estimate the investment. But if we would have to start from scratch, the application of ESC/Java to the whole purse involves a couple of weeks' work. Applying the LOOP tool to the `Decimal` class only involves a similar investment.

As a result of this (and other) work, JML has developed into the standard specification language for Java Card (*e.g.* within the European project VerifiCard [30]).

Future work involves a further development of the JML language, integration of tools around JML, and addressing scalability issues (especially for interactive tools such as LOOP).

References

[1] E. Bretagne, A. El Marouani, P. Girard, J.-L. Lanet, Pacap purse and loyalty specification, Technical Report V 0.4, Gemplus, 2000.

[2] C. Breunesse, B. Jacobs, J. van den Berg, Specifying and verifying a decimal representation in Java for smart cards, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology, AMAST'02, LNCS, vol. 2422, Springer, 2002, pp. 304–318.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, E. Poll, An overview of JML tools and applications, in: Th. Arts, W. Fokkink (Eds.), Formal Methods for Industrial Critical Systems, FMICS'03, ENTCS, vol. 80, Elsevier, Amsterdam, 2003, www.elsevier.com/locate/entcs/volume80.html.

[4] L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: a developer-oriented approach, in: Formal Methods, FME'03, LNCS, vol. 2805, Springer, 2003, pp. 422–439.

[5] N. Cataño, M. Huisman, Formal specification and static checking of Gemplus's electronic purse using ESC/Java, in: L.-H. Eriksson, P.A. Lindsay (Eds.), Formal Methods Europe, FME'02, LNCS, vol. 2391, Springer, 2002, pp. 272–289.

[6] N. Cataño, M. Huisman, Chase: a static checker for JML's assignable clause, in: L.D. Zuck, P.C. Attie, A. Cortesi, S. Mukhopadhyay (Eds.), Verification, Model Checking and Abstract Interpretation, VMCAI '03, LNCS, vol. 2575, Springer, 2003, pp. 26–40.

[7] P. Chalin, JML support for primitive arbitrary precision numeric types: Definition and semantics, in: Workshop on Formal Techniques for Java Programs, FTfJP 2003, 2003; Also available as Technical Report Concordia University 002.2a.

[8] Differences between Esc/Java and JML, 2000. Comes with JML distribution, in file esc-jml-diffs.txt.

[9] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Transactions on Software Engineering 27 (2) (2001) 1–25.

[10] ESC/Java 2, http://www.cs.kun.nl/sos/research/escjava.

[11] J.-C. Filliâtre, Verification of non-functional programs using interpretations in type theory, Journal of Functional Programming 13 (4) (2003) 709–745.

[12] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, SIGPLAN Notices, vol. 37(5), ACM, 2002, pp. 234–245.

[13] C. Flanagan, S. Qadeer, S.A. Seshia, A modular checker for multithreaded programs, in: E. Brinksma, K.G. Larsen (Eds.), Computer-Aided Verification, CAV'02, LNCS, vol. 2404, Springer, 2002, pp. 180–194.

[14] M. Huisman, Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle, Ph.D. Thesis, University of Nijmegen, 2001.

[15] B. Jacobs, Java's integral types in PVS, in: E. Najim, U. Nestmann, P. Stevens (Eds.), Formal Methods for Open Object-Based Distributed Systems, FMOODS 2003, LNCS, vol. 2884, Springer, 2003, pp. 1–15.

[16] B. Jacobs, Weakest precondition reasoning for Java programs with JML annotations, Journal of Logic and Algebraic Programming 58 (1–2) (2004) 61–88.

[17] B. Jacobs, E. Poll, A logic for the Java modeling language JML, in: H. Hussmann (Ed.), Fundamental Approaches to Software Engineering, FASE 2001, LNCS, vol. 2029, Springer, 2001, pp. 284–299.

[18] G.T. Leavens, A.L. Baker, C. Ruby, Preliminary Design of JML: a Behavioral Interface Specification Language for Java, http://www.jmlspecs.org/prelimdesign/.

[19] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, JML Reference Manual, http://www.jmlspecs.org/jmlrefman/.

[20] K.R.M. Leino, Toward Reliable Modular Programs, Ph.D. Thesis, California Institute of Technology, 1995.

[21] K.R.M. Leino, G. Nelson, J.B. Saxe, ESC/Java user's manual, Technical Report SRC 2000-002, Compaq System Research Center, 2000.

[22] C. Marché, C. Paulin-Mohring, X. Urbain, The Krakatoa tool for JML/Java program certification, Journal of Logic and Algebraic Programming 58 (1–2) (2004) 89–106.

[23] B. Meyer, Object-Oriented Software Construction, 2nd revised edition, Prentice Hall, 1997.

[24] J. Meyer, A. Poetzsch-Heffter, An architecture of interactive program provers, in: S. Graf, M. Schwartzbach (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, LNCS, vol. 1785, Springer, 2000, pp. 63–77.

[25] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M.K. Srivas, PVS: Combining specification, proof checking, and model checking, in: R. Alur, T.A. Henzinger (Eds.), Computer-Aided Verification, CAV'96, LNCS, vol. 1102, Springer, 1996, pp. 411–414.

[26] M. Pavlova, L. Burdy, G. Barthe, M. Huisman, J.-L. Lanet, Enforcing high-level security properties for applets, in: CARDIS 2004, 2004 (in press).

[27] A. Poetzsch-Heffter, P. Müller, A programming logic for sequential Java, in: S.D. Swierstra (Ed.), Programming Languages and Systems, ESOP '99, LNCS, vol. 1576, Springer, 1999, pp. 162–176.

[28] E. Poll, Formal interface Java specifications for the Java Card API 2.1.1, http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html with contributions by other members of the LOOP project.

[29] The LOOP Project, http://www.cs.kun.nl/sos/research/loop.

[30] VerifiCard Project homepage, http://www.verificard.org.