SPECIFICATION OF THE JAVACARD API IN JML

Towards formal specification and verification of applets and API implementations

Erik Poll, Joachim van den Berg, Bart Jacobs
Computing Science Institute, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{erikpoll,joachim,bart}@cs.kun.nl

Abstract This paper reports on an effort to increase the reliability of JavaCard-based smart cards by means of formal specification and verification of JavaCard source code. As a first step, lightweight formal interface specifications, written in the specification language JML, have been developed for all the classes in the JavaCard API (version 2.1). They make many of the implicit assumptions underlying the current implementation explicit, and thus facilitate the use of this API and increase the reliability of the code that is based on it. Furthermore, the formal specifications are amenable to tool support, for verification purposes.

1. INTRODUCTION

Program specification and verification has always be one of the central issues in computer science. Despite enormous theoretical progress in this area, the practical impact is still modest. Over the last few years the situation is slightly improving, due to the availability of modern verification tools (like theorem provers and model checkers), supported by fast hardware. Early work in program specification and verification was based on mathematically clean and abstract programming languages, with special logics for correctness formulas. But nowadays, correctness issues are being investigated for real-life programming languages (like Java), and formal logical languages are used enabling tool support for specification and verification.

This paper fits in that modern formal methods tradition. It uses the specification language JML for annotation of the Java classes in the

JavaCard API[1] (version 2.1), see also [5]. Its aim is to increase the reliability of JavaCard-based smart cards by means of formal specification and verification of JavaCard source code. JavaCard is a good target for the application of formal methods, for several reasons: JavaCard applets are distributed in large numbers, and are often used in (safety or security) critical applications, so that programming errors can have serious consequences. But JavaCard applets are usually small programs, designed to run on a processor with modest resources. Also, the language of these applets, JavaCard, is relatively simple, with a relatively small API, in comparison to full Java. This makes the application of formal methods to JavaCard a feasible and useful enterprise, which can have an impact.

This paper reports on the first steps in the use of JML for JavaCard: very basic specifications have been written for all the classes in the JavaCard API. These specifications are 'lightweight' in that they concentrate on conditions for normal and abrupt termination (i.e. the throwing of exceptions), given by preconditions and invariants, and omit the functional specification, which are typically written in postconditions. We call such specifications termination specifications. These specifications are very easy to read and to write, and, despite their simplicity, they provide useful documentation and make many implicit assumptions explicit.

The API specification will be published on the web [14]. The ideal scenario is that it will develop into an actively used 'reference specification', that will form a basis for future versions of the JavaCard API implementation. (This fits in a component-oriented approach, where interface specifications form the basis for software composition.) Therefore, we explicitly solicit feedback from the JavaCard (user and development) community, so that our specifications reflect the common understanding of what should be in the JavaCard API.

The JML project

JML (for Java Modeling Language) [11, 12] is a specification language tailored to Java, primarily developed at Iowa State University. It allows assertions to be included in Java code, specifying for instance pre- and postconditions and invariants in the style of Eiffel and the well-established Design by Contract approach [15]. JML is being integrated with the specification language used for ESC/Java, the extended static checker developed at Compaq System Research Center [13, 4].

[1] developed by Sun Microsystems, see http://java.sun.com/products/javacard/.

At Nijmegen a formal semantics has been developed of a large subset of Java, which includes all of JavaCard. A compiler has been built, the LOOP tool, which translates a Java program into logical theories describing its semantics [9, 2, 6, 14]. These logical theories are in a format that can serve as input for theorem provers, which can then be used to prove properties of the original Java program, thus achieving a high level of reliability for this program. Currently the LOOP tool supports output for the theorem provers PVS [16] and Isabelle [17]. This approach to verification of Java has demonstrated its usefulness for instance with the proof of a non-trivial invariant for the Vector class in the standard Java API [7]. The LOOP tool is currently being extended to JML, so that it can be used to verify JML-annotated Java source code. We should emphasise that this is source code, and not bytecode verification.

An advantage of using a formal specification language is that it becomes possible to provide tool support. Current work on tool support for JML focuses on:

- verification using LOOP tool, at the University of Nijmegen,
- extended static checking by ESC/Java, at Compaq System Research Center in Palo Alto, and
- generation of runtime checks on preconditions for testing, at Iowa State University.

JML specifications for JavaCard

JML specifications of the JavaCard API are of interest for parties on both sides of the interface the API provides, i.e. for developers of applets on the one hand, and for developers of API implementations on the other hand. The specifications can be used to specify and verify essential properties of implementations of the JavaCard API, starting with the current reference implementation itself, and as a basis for the specification and verification of properties of individual applets that use the API.

Once a formal specification language has been chosen, there is still a choice of how detailed specifications should be. For any program there is a whole spectrum of possible specifications. At one end of the spectrum are the very complete and detailed specifications. The reference implementation of the JavaCard API is an example of such a specification. At the other end of the spectrum are very incomplete or 'lightweight' specifications. These are the kind of specifications we have given for the JavaCard API, version 2.1 [10]. More precisely, the specifications we have

given only specify when methods are guaranteed not to throw unwanted runtime exceptions. We call such specifications termination specifications. Such specifications are relatively easy to write and easy to check, and can be used to guarantee the absence of most runtime exceptions. This is important, since omitting the proper handling of such exceptions is a common source of failures. Our formal specifications are based on the informal (but quite detailed) specification of the JavaCard API, that is contained as javadoc documentation in the reference implementation of the JavaCard API. Essentially, they are a rediscovery of many of the design ideas and decisions that went into the (current) implementation.

The paper is organised as follows. It starts with a gentle introduction to JML, concentrating on the pre- and post-conditions for methods (including abrupt termination), and invariants. Section 3 discusses the typical features of the kind of specifications we have given, and the subsequent Section 4 describes several typical examples of specifications for methods from the JavaCard API, including a discussion of typical specification issues in the presence of inheritance. Finally, the paper ends with some conclusions.

2. JML

This section introduces the JML notation used in our formal specification. For our relatively simple termination specifications, only a small subset of the full JML syntax is actually used. So what is described here is by no means all of JML, see [11, 12].

JML allows Java code to be annotated with specifications, for example with preconditions, post-conditions, and invariants, in the style of Eiffel, also known as "Design by Contract", see [15, 8]. However, JML provides many enhancements making it much more expressive. One of these, of particular relevance to this paper, is the possibility to specify when certain exceptions may be thrown, must be thrown, or may not be thrown.

JML annotations are a special kind of Java comments: JML annotations are preceded by //@, or enclosed between /*@ and @*/.

Pre- and Postconditions in JML

Methods can be specified in the usual way, by giving pre- and post-conditions. The simplest method specifications are of the form

```
/*@ normal_behavior
     requires : <precondition> ;
      ensures : <postcondition> ;
```

```
@*/
```

Such a specification states that if the precondition holds at the beginning of a method invocation, then the method terminates normally (i.e. without throwing an exception) and the postcondition will hold at the end of the method invocation. This is like a (total) correctness formula in Hoare logic [1].

Pre- and postconditions can simply be standard Java boolean expressions. JML adds several operators, for instance quantifiers \exists and \forall, but for the simple specifications given here none of these additional operators are needed.

In Java methods can terminate abruptly, by throwing exceptions. A more general form of method specification makes it possible to specify in what circumstances which exceptions may be thrown. These method specifications are of the form

```
/*@ behavior
    requires : <precondition> ;
     ensures : <postcondition>;
     signals : (Exception1) <condition1>;
     :
     signals : (Exceptionn) <conditionn>;
 @*/
```

Such a specification states that if the precondition holds at the beginning of a method invocation, then the method either terminates normally or terminates abruptly by throwing one of the listed exceptions. If the method terminates normally, then the postcondition will hold. If the method throws an exception, then the corresponding condition will hold.

Finally, a third form of method specification that can be used is

```
/*@ exceptional_behavior
    requires : <precondition> ;
     signals : (Exception1) <condition1>;
     :
     signals : (Exceptionn) <conditionn>;
 @*/
```

Such a specification states that if the precondition holds then the method will terminate abruptly by throwing one of listed exceptions, and if one of these exceptions is thrown then the corresponding condition will hold.

Both normal_behavior and exceptional_behavior are just special cases of behavior, and can be regarded as useful syntactic sugar. All these

behaviors can be translated into an extended Hoare logic dealing with abrupt termination, see [6].

For a single method several specifications of the forms above can be given, joined by the keyword also. The method should then meet all these specifications. With pre- and postconditions in Eiffel this is not possible.

In addition to pre- and postconditions, a method specification in JML can also include modifiable clauses. These clauses specify so-called frame conditions, which say that only certain (instance or class) fields may have their values changed by a method. For example, modifiable:x specifies that a method may only change field x. Because we do not want to discuss these clauses in this paper, all examples of behavior specifications will either include a clause modifiable:\not_specified to say that nothing is specified about the fields the method may change, or a clause modifiable:\nothing to say that the method does not change any fields.

Invariants in JML

In addition to pre- and postconditions, JML annotations can also specify invariants. An invariant is a property that should hold after creation of an object by one of the constructors, and that should be preserved by all the methods. So any invariant is implicitly included in pre- and postconditions of all methods. Note that an invariant must also be preserved if a method throws an exception.

For example, for the class AID (Application Identifier), which includes a byte array field theAID, we have an invariant

```
/*@ invariant: theAID != null  &&
             5 <= theAID.length && theAID.length <= 16;
  @*/
```

For the class APDU (Application Protocol Data Unit), which includes two byte array fields, buffer and ramVars, we have an invariant

```
/*@ invariant: buffer != null && ramVars != null &&
             buffer.length == APDU.BUFFERSIZE &&
             ramVars.length == APDU.RAM_VARS_LENGTH;
  @*/
```

Invariants are not mentioned in the informal API specification, nor in the API reference implementation. Still, invariants provide useful documentation, and often play an important role as (implicit) assumptions in considerations about the correctness of code. This will be illustrated later, e.g. in Example 4.1.

3. TERMINATION SPECIFICATIONS FOR THE JAVACARD API

We have developed very basic specifications for all the classes in the JavaCard API. A concrete goal was to specify preconditions for methods to rule out as many unwanted exceptions as possible. Such "termination" specifications are relatively easy to write, and easy to check, but still provide crucial information about the behaviour of the API classes. The specifications expose many of the considerations and the implicit assumptions that have gone into the design of the API reference implementation. In this section we discuss some typical examples to give the flavour of the specifications we have given for all methods in the JavaCard API.

Whenever possible, methods are specified by a normal_behavior. This requires a precondition which guarantees normal termination, i.e. which rules out that any exceptions will be thrown. The precondition usually imposes fairly obvious restrictions on the parameters of the method, e.g. that references are not null, that indices are within array bounds, etc. A typical example is the specification of arrayCompare in the class Util. This method compares parts of two arrays, given offsets within those arrays and a length saying how many array elements are to be compared. Its termination specification is given below:

```
public static native byte arrayCompare(byte[] src,
                                       short srcOff,
                                      byte[] dest,
                                       short destOff,
                                       short length)
throws ArrayIndexOutOfBoundsException,
     NullPointerException;
/*@ normal_behavior
    requires: src != null && dest != null &&
              srcOff >= 0 && destOff >= 0 && length >= 0
              && srcOff + length <= src.length &&
              destOff + length <= dest.length;
  modifiable: \nothing;
     ensures: true;
 @*/
```

Some points to note about this specification:

- The precondition states very obvious requirements on the parameters needed to avoid NullPointer- and ArrayIndexOutOfBounds-Exceptions. These requirements immediately follows from the de-

tailed informal specification given in the JavaCard API documentation.

- The postcondition is simply true. This means that nothing is specified about the functionality of the method. This is the case with most of the specifications we have developed.

- The specification of arrayCompare could easily be made stronger. For instance, the informal specification of the JavaCard API states that a NullPointerException may be thrown if src or dest is a null reference, as one would expect. We could easily specify this in JML as well. We have chosen not to do so to keep the formal specifications as short and simple as possible[2]. And, as one would expect (or hope), it turns out that no part of the JavaCard reference implementation in fact relies on the property that arrayCompare may throw a NullPointerException if src or dest is a null reference.

- The method arrayCompare is declared as native, which means that it is to be implemented by platform-dependent code. Indeed, the reference implementation does not provide an implementation of this method. For such methods precise specifications are of course of crucial importance.

We cannot specify all methods by giving a normal_behavior. Some methods can throw exceptions that are very hard – if not impossible – to rule out with a simple precondition. Such methods are specified using behavior instead of normal_behavior. A typical example is the specification for arrayCopy in the class Util. This method copies part of one array into another array. Like arrayCompare it can throw a NullPointer- or ArrayIndexOutOfBoundsException. But it can also throw a TransactionException, namely when the commit capacity (the maximum number of bytes of persistent data which can be modified during a card transaction) is exceeded. Its specification is given below:

```
public static native short arrayCopy(byte[] src,
                                      short srcOff,
                                     byte[] dest,
                                      short destOff,
```

[2]Also, one has to be careful with such specifications, as it should not be specified which exception gets thrown if there is the possibility of throwing more than one exception (e.g. when src is null and destOff > dest.length). The informal API specification in fact warns that programmers should not rely on getting a specific exception in such cases. Of course, by not specifying anything about when certain exceptions are thrown, as we do here, we avoid this danger altogether.

```
                                short length)
throws ArrayIndexOutOfBoundsException,
     NullPointerException, TransactionException;
/*@ behavior
     requires: src != null && dest != null &&
               srcOff >= 0 && destOff >= 0 && length >= 0
               && srcOff + length <= src.length &&
               destOff + length <= dest.length ;
   modifiable: \not_specified;
     ensures: true;
     signals: (TransactionException) true;
 @*/
```

Some points to note about this specification:

- Again, the postcondition is true, so the specification does not describe any functionality. And again, it is trivial to see that the specification of arrayCopy above captures part of its informal specification given in the JavaCard API documentation.

- The precondition does not rule out all runtime exceptions, as it leaves open the possibility that a TransactionException is thrown. One could try to strengthen the precondition to exclude this possibility, but that would be much harder. Unlike a NullPointer- or ArrayIndexOutOfBoundsException, a TransactionException is not due to an obvious mistake by the client invoking this method.

  A TransactionException is thrown when the space in the commit buffer is exhausted. In this buffer the JCRE (JavaCard Runtime Environment) retains the original contents of updated values until a transaction is committed, to support the rollback of a transaction in case of power loss. One could consider giving a second specification of arrayCopy, in addition to the one above, that states that no TransactionException is thrown if some (stronger) precondition, guaranteeing the availability of sufficient space in the commit buffer, is met. Such a specification would make it possible to prove the absence of TransactionExceptions in applets, assuming a certain minimal size of the commit buffer.

We have written specifications similar to those of arrayCompare and arrayCopy above for all the methods in the JavaCard API, using either behavior or normal_behavior. A few methods have been specified using exceptional_behavior rather than behavior or normal_behavior, namely those which are specifically meant to throw exceptions (i.e. the throwIt methods in all the exception classes).

4. EXAMPLES OF DEVELOPING AND CHECKING JML SPECIFICATIONS

Obviously we cannot discuss the JML specifications for the whole JavaCard API here. We will present several typical examples to give an impression of the kind of verifications required to check that specifications are met, the difficulties involved in developing specifications, and the relation between our formal JML specifications and the informal ones given in the JavaCard documentation.

The first example illustrates an informal verification of a specification, and the crucial role of invariants in this.

Example 4.1 (AID) The method equals of the class AID compares the AID bytes in two AID instances. Our termination specification of equals is

```
public boolean equals( Object anObject )
 /*@ normal_behavior
       requires: true;
     modifiable: \nothing;
       ensures: true;
  @*/
```

This specification states that equals always terminates normally, i.e. never throws an exception. This very weak specification is already more precise than the informal specification: the informal specification explicitly states that equals does not throw a NullPointerException, but does not say anything about whether or not it may throw other exceptions.

The reference implementation of the API gives the following implementation of equals:

```
{if ( !(anObject instanceof AID)
    || ((AID)anObject).theAID.length != theAID.length)
      return false;
 return (Util.arrayCompare(((AID)anObject).theAID,
                           (short)0, theAID, (short)0,
                           (short)theAID.length)
      == 0);
}
```

We will give an informal argument that this implementation of equals meets its JML specification, i.e. that it terminates without throwing exceptions. This comes down to showing that the invocation of the method Util.arrayCompare terminates normally, as this is the only possible source of exceptions in the code fragment above. Normal termination

of Util.arrayCompare requires that its precondition given earlier is met; substituting the actual values for the formal parameters yields:

```
((AID) anObject).theAID != null && theAID != null
&& 0 >= 0 && 0 >= 0 && theAID.length >= 0
&& 0 + theAID.length <= ((AID)anObject).theAID.length
&& 0 + theAID.length <= theAID.length
```

Recall the invariant of class AID:

```
theAID != null && 5 <= theAID.length && theAID.length <=16
```

This leaves only the following properties to be established:

(i) ((AID)anObject).theAID != null
(ii) theAID.length <= ((AID)anObject).theAID.length

It follows from the if-statement that Util.arrayCompare will only be invoked if:

(iii) (anObject instanceof AID)
(iv) ((AID)anObject).theAID.length == theAID.length

It follows from (iv) that (ii) holds. It follows from (iii) that the parameter (AID)anObject has runtime type AID. We may therefore assume that it satisfies the invariant for this class, and hence (i) holds. So all conditions needed to ensure normal termination of Util.arrayCompare are met, and hence the reference implementation of equals in the class AID meets its JML specification. □

Note that to understand that the reference implementation is correct, the invariant of the class AID is really needed. Also, it should be clear from the example above that once we have the class invariant of AID and the specification of Util.arrayCompare, then verifying that the method equals of AID meets its JML specification is not that hard. The reasoning involved is well within the capabilities of modern theorem provers.

The example also illustrates that even these very basic JML specifications can be more precise than the existing informal specifications because they explicitly rule out more runtime exceptions.

The example below illustrates a more complicated argument about correctness of code from the API reference implementation.

Example 4.2 (PackedBoolean) The class PackedBoolean provides efficient management of volatile storage space. Instances of this class contain an

array of bytes container that is used to store boolean values. The point of this is that only one bit rather than one byte is used for each boolean. The class provides methods put and get to access the bits in the byte array container. For example, get(n) will return the (n % 8)-th bit of the byte container[n / 8], where / and % are the integer division and remainder operations.

In the reference implementation an instance of this class is created in which the length of the byte array is 2 (in the class Dispatcher, via the class PrivAccess), thus providing space for 16 booleans. Trying to use it for more than 16 booleans will – not surprisingly – result in an ArrayIndexOutOfBoundsException. The fact that no more than 16 booleans will be allocated in this instance of PackedBoolean is a 'global' property, and cannot be checked by looking at an individual class. Developing termination specifications for all methods will bring the hidden assumption that no more that 16 booleans may be allocated to the surface, as shown below.

First we consider the specification of the class PackedBoolean. For this it is convenient to a feature of JML not mentioned so far, namely a specification-only field. JML provides specification-only variables, which are just like ordinary variables but are for specification purposes only, i.e. they can be used in JML annotations but not in the Java code. For the class PackedBoolean we introduce a specification-only field for the number of booleans that can be fitted in the container array:

```
//@ public model byte _NUMBER_OF_PACKED_BOOLS;
```

This specification variable will simply be equal to 8*container.length. The advantage of using a specification variable rather than the expression 8*container.length is of course that it abstracts away from the implementation of PackedBoolean.

We have the following invariant for the class PackedBoolean:

```
/*@ invariant:
    container != null  &&
    _NUMBER_OF_PACKED_BOOLS == container.length * 8;
 @*/
```

The methods for accessing the booleans in the byte array can now be specified as below. In combination with the invariant, the preconditions guarantee that no NullPointer- or ArrayIndexOutOfBoundsException can occur.

```
public boolean get( byte identifier )
```

```
/*@ normal_behavior
      requires: 0 <= identifier  &&
                identifier < _NUMBER_OF_PACKED_BOOLS;
    modifiable: \not_specified;
       ensures: true;
 @*/

public void put ( byte identifier, boolean value )
/*@ normal_behavior
      requires: 0 <= identifier  &&
                identifier < _NUMBER_OF_PACKED_BOOLS;
    modifiable: \not_specified;
       ensures: true;
 @*/
```

To allocate a boolean in an instance of the class PackedBoolean, clients call the method allocate, which returns the identifier that is to be used in subsequent calls of the methods get and set to address a particular boolean. Instances of the class PackedBoolean have a field nextId, which is used to keep track of how many booleans have already been allocated. The method allocate simply returns the field nextId and increments it by 1. An obvious invariant for this field is:

```
/*@ invariant:
     0 <= nextId && nextId < _NUMBER_OF_PACKED_BOOLS;
 @*/
```

and the specification of allocate is

```
public byte allocate()
/*@ normal_behavior
      requires: nextId+1 < _NUMBER_OF_PACKED_BOOLS;
    modifiable: \not_specified;
       ensures: \result < _NUMBER_OF_PACKED_BOOLS;
 @*/
```

The precondition ensures that we never allocate more booleans than for which there is space. The JML keyword \result in the postcondition refers to the value returned by the method.

The specification above forces all classes using a PackedBoolean to ensure that they do not exceed its capacity. For example, the constructor of the APDU class allocates eight booleans:

```
APDU()
```

```
{ ...
  thePackedBoolean   = PrivAccess.getPackedBoolean();
  incomingFlag       = thePackedBoolean.allocate();
  sendInProgressFlag = thePackedBoolean.allocate();
  outgoingFlag       = thePackedBoolean.allocate();
  outgoingLenSetFlag = thePackedBoolean.allocate();
  lrIs256Flag        = thePackedBoolean.allocate();
  noChainingFlag     = thePackedBoolean.allocate();
  noGetResponseFlag  = thePackedBoolean.allocate();
  ...
}
```

Its precondition will have to include

```
requires: PrivAccess.getPackedBoolean().nextId + 8
        < thePackedBoolean._NUMBER_OF_PACKED_BOOLS;
```

By the specification of allocate it then follows that the constructor above establishes invariants like

```
 0 <= outgoingFlag &&
outgoingFlag < thePackedBoolean._NUMBER_OF_PACKED_BOOLS;
```

for the class APDU. This in turn guarantees that methods like

```
 private boolean getSendInProgressFlag()
 { return thePackedBoolean.get( sendInProgressFlag ); }
```

in the class APDU will not throw any runtime exceptions, because the precondition of get is met. □

As the example above shows, the development of even very basic JML specifications forces many implicit assumptions out into the open. Writing JML annotations while developing the code would require less effort than writing them afterwards as we have done. The post-hoc writing of JML specifications essentially forces us to (re)discover many of the considerations that were part of the original design, but which cannot be found back anywhere in the code or in the informal documentation.

Specification Inheritance

Inheritance is a key feature of object-oriented (OO) programming. It provides extensibility: subclasses can extend existing classes, and code written for those original classes can be reused for any new subclasses. However, this extensibility comes at a price. It is no longer possible to decide statically which code will be executed when a method is invoked,

because, due to late binding, this will depend on the runtime type of an object. This makes it hard to reason about object-oriented programs: it is dangerous to rely on certain properties of a method, because these might not hold for implementations of this method in future subclasses.

Specification inheritance [15, 3] is the principle that a class inherits the specification of its superclass and the specifications of any of the interfaces it implements. This principle addresses exactly the difficulty in reasoning about object-oriented code mentioned above: It guarantees that it is safe to assume some properties of a method because these properties will not be violated in future subclasses. It means that in subclasses we are only allowed to weaken preconditions and strengthen postconditions. This constrains the use of inheritance: one can no longer make methods behave completely differently by overriding (but this is not good programming practice anyway).

Specification inheritance exposes the fundamental complexity of specification and verification in an OO setting. One has to be careful not to make specifications too strong, because this may rule out interesting subclasses in the future. This means specification requires some foresight. What often happens in practice is that one wants to add a subclass but finds that it does not meet the specification of its superclass. One can then weaken the superclass specification to allow the subclass, but that signals that existing client code of the superclass may be affected. This is illustrated in Example 4.3 below.

Java enforces some form of specification inheritance for throws clauses: a method in a subclass cannot throw exceptions that are not declared in the supertype. Of course, this does not apply to runtime exceptions, as they do not have to be declared. (In Java these are called unchecked exceptions.) As the earlier examples illustrate, our JML specifications effectively extend this policy to runtime exceptions.

There are not many places where specification inheritance is an issue in the JavaCard API. In fact, quite a few classes that make up the JavaCard API are final. These cannot be extended, so for these classes specification inheritance can never become a problem. Two places where specification inheritance is an issue are

- the abstract class Applet, and
- the interface PIN.

The class Applet is obviously meant to be extended; after all, it is an abstract class. The specification of the class Applet should give properties that we want all possible JavaCard applets to have. Similarly, the

specification of the interface PIN should give properties that we want all possible implementations of this interface to have.

Even for the very basic specifications we consider one can argue about what the specifications of Applet and PIN should be. For instance, any applet has to provide an implementation of the method

```
public abstract void process(APDU apdu)
```

that will be called by the JCRE to process incoming APDU's. Several specifications for this methods are possible: one could not specify anything about this method at all, or one could specify that it may only throw a limited set of exceptions. For instance, when developing an applet, one might want to ensure that its process method can only throw an ISOException or an APDUException.

The example below shows that, even with our very basic specifications, specification inheritance already brings to light some subtleties in the reference implementation of the JavaCard API.

Example 4.3 The interface PIN contains a method isValidated(), that returns true if a valid PIN value has been presented since the last card reset or last invocation of reset(). A first guess for its specification would simply be:

```
public boolean isValidated()
  /*@ normal_behavior
        requires: true;
      modifiable: \not_specified;
        ensures: true;
   @*/
```

It states that this method never throws an exception. But, surprisingly, this specification is already too strong. This is because the class OwnerPIN, which implements the interface PIN and therefore inherits its specification, provides an implementation of isValidated that may throw a SystemException. (This in itself is already far from obvious! But giving termination specifications in JML for all methods in the class OwnerPIN will bring this fact to light.) So the implementation of isValidated in OwnerPIN does not meet the specification above. It does meet the weaker specification

```
  /*@ behavior
        requires: true;
      modifiable: \not_specified;
        ensures: true;
```

```
        signals: (SystemException) true;
@*/
```

So specification inheritance means that for the method of isValidated in the interface PIN we should also give this weaker specification, or something weaker still. The advantage of this is that, by looking at its JML specification, any user of the interface PIN will be aware that implementations of this interface may throw a SystemException. □

We already saw that JML specifications can be more precise than the informal specifications, because they rule out more runtime exceptions (e.g. in Example 4.1). The example above show that JML specifications can also be more precise about runtime exceptions for the opposite reason, i.e. because they explicitly state that an exception may be thrown even though this is not mentioned anywhere in the informal specification or in the code of the reference implementation.

5. CONCLUSION

The termination specifications of the JavaCard API provide useful documentation, as the examples we have given illustrate. Many properties expressed by the JML annotations can directly be found in the informal specification, but some cannot. In these cases the JML specification of the JavaCard API is more informative than both the source code of the reference implementation and the informal specification, and makes explicit many considerations and assumptions that are implicit in the design. In particular, the specifications we have developed are useful in preventing uncaught exceptions, a common source of failures.

Writing termination JML specifications in JML for the JavaCard API is not very difficult, assuming some basic knowledge of formal methods. Writing JML annotations while developing the code, instead of afterwards as we have done, would require less effort still. All annotations are easy to understand for any Java programmer. (It is in fact one of the goals of JML that it should be readily understandable for Java programmers.)

Using a formal specification language rather than informal English makes it possible to provide tool support. The conventional tool support for Design by Contract is the automatic inserting of runtime tests in code to check no preconditions are violated. There are other efforts underway to provide such support for Java, e.g. [8]. While useful in the development and testing phase, leaving such tests in the final JavaCard source code of

applets or of API implementations is probably undesirable, for reasons of efficiency and size[3].

Our goal is to go further than runtime testing of specifications, and give compile-time proofs that specifications are met. Relatively simple properties, like those given in the specifications discussed here, should be proved fully automatically. Experiments are underway to see how strong specifications can be made while still being automatically enforceable by the extended static checker ESC/Java [4]. Once the extended static checker ESC/Java will be released, it can then provide useful tool support for the development of both applets and API implementations, allow automatic verification of simple specifications at the push of a button. One cannot expect arbitrarily complex properties to be proved fully automatically, but these can still be proved interactively using the LOOP tool as a front-end to theorem provers such as PVS or Isabelle. This approach is of course more labour intensive, but especially for vital properties of JavaCard API implementations and applets the effort may well be justified.

All JML specifications for the JavaCard API will be made available on our webpages [14]. We hope this will be a useful service to the JavaCard community, in providing a proper addition to the existing documentation of the JavaCard API. We also plan to develop more detailed (functional) JML specifications of the API for the verification of JavaCard source code using the LOOP tool.

References

[1] K. R. Apt. Ten years of Hoare's logic: a survey – Part I. ACM Trans. on Prog. Lang. and Syst., 3(4):431–483, 1981.

[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 and C. Choppy, editors, Recent Trends in Algebraic Development Techniques, number 1827 in Lecture Notes in Computer Science. Springer, Berlin, 2000.

[3] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In Proceedings 18th International Conference on Software Engineering, Berlin, Germany, pages 258–267. IEEE, 1996.

[3]Indeed, the informal JavaCard API specification explicitly states that implementations of the API should not do any parameter checking, but leave it up to the virtual machine to throw appropriate exceptions.

[4] Extended static checker ESC/Java. Compaq System Reserch Center, http://www.research.digital.com/SRC/esc/Esc.html.

[5] U. Hansmann, M.S. Nicklous, T. Schäck, and F. Seliger. Smart Card Application Development Using Java. Springer, 2000.

[6] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, Fundamental Approaches to Software Engineering, number 1783 in Lecture Notes in Computer Science, pages 284–303. Springer, Berlin, 2000.

[7] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Report CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen, 2000. An earlier version appeared in: B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter (eds.), Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop. Technical Report 251, Fernuniversität Hagen, 1999, p.37-44.

[8] iContract. R. Kramer, http://www.reliable-systems.com/tools/-iContract, 1999.

[9] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pages 329–340. ACM Press, 1998.

[10] The Java Card 2.1 Application Programming Interface (API). Sun Microsystems, 1999.

[11] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer, 1999.

[12] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, Dept. of Comp. Sci., Iowa State University, 1999. Tech. Rep. 98-06.

[13] K.R.M. Leino, J.B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop, pages 37–44. Techn. Rep. 251, Fernuniversität Hagen, 1999. Also as Technical Note 1999-002, Compaq Systems Research Center, Palo Alto.

[14] The LOOP project. http://www.cs.kun.nl/~bart/LOOP/index.html.

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

[16] 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 Lecture Notes in Computer Science, pages 411–414. Springer, Berlin, 1996.

[17] L.C. Paulson. Isabelle: A Generic Theorem Prover. Number 828 in Lecture Notes in Computer Science. Springer, Berlin, 1994.