Formal Specification of the JavaCard API in JML: the APDU class

Erik Poll, Joachim van den Berg, Bart Jacobs

{erikpoll,joachim,bart}@cs.kun.nl
Computing Science Institute, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

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, formal interface specifications, written in the specification language JML, have been developed for all the classes that make up the JavaCard API. These specifications are "lightweight" in the sense that they are incomplete and specify only some aspects of the API, but they already provide a useful addition to the existing informal API specifications. Moreover, the fact that these specification are written in a formal language makes them amenable to tool support, for verification purposes. As an illustration, the JML specifications of the `APDU` (Application Protocol Data Unit) class in the JavaCard API are discussed in detail.

Key words: smart card, JavaCard, formal specification

1 Introduction

Program specification and verification has always been 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 has slightly improved, 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 (version 2.1.1) [9]. 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 used in large numbers and in (safety or security) critical applications, where programming errors can have serious consequences. 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: basic specifications have been written for all the classes in the JavaCard API. These specifications are incomplete – sometimes called "lightweight" – specifications that focus on necessary (but not always sufficient) conditions for avoiding unwanted behaviour of methods, e.g. the occurrence of certain run-time exceptions.

The JML specifications will be published on the web [14]. The ideal scenario is that they will develop into an actively used 'reference specification', that will form a basis for future versions of the JavaCard API. Therefore, we explicitly solicit feedback from the JavaCard (user and development) community, so that our specifications reflect the common understanding of the precise behaviour of the JavaCard API.

The JML project

JML (for Java Modeling Language) [10,11] 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 "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 [13,5].

At Nijmegen, a formal semantics has been developed for 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 [8,2,6,14]. 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 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 verification, not byte code verification.

One 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 the generation of run-time checks on preconditions for testing, at Iowa State University, extended static checking, at Compaq System Research Center, and verification using the LOOP tool, at the University of Nijmegen.

JML specifications for JavaCard

JML specifications of the JavaCard API are of interest both for developers of implementations of the API and for developers of applets. 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. Our formal specifications are based on the existing informal (but quite detailed) specifications of the JavaCard API (version 2.1.1), included as appendix in [4] and available as Javadoc-generated HTML from Sun's website [9].

Using a formal specification language such as JML still leaves a choice as to how detailed the specifications we write should be. For any program there is a wide spectrum of possible specifications. At one end of the spectrum are the very complete and detailed specifications. The reference implementation of the JavaCard API [9] 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. More precisely, our formal specifications concentrate on specifying the preconditions of methods that ensure normal behaviour of the method, i.e. preconditions that rule out some – or all – unwanted run-time exceptions. Such specifications are relatively easy to write and easy to check, and can be used to guarantee the absence of most run-time exceptions. This is important, as omitting the proper handling of such exceptions is a common source of failures.

The paper is organised as follows. It starts with an introduction to JML in Section 2. Section 3 gives typical examples of the specifications we have written for methods in the JavaCard API. Section 4 then discusses the specification of the `APDU` class, the largest class in the JavaCard API, and Section 5 discusses the relation between this specification and the reference implementation of the class. The article ends with some conclusions.

2 JML

This section gives a brief introduction to the specification language JML. It describes only the subset of JML used in this paper; for more complete descriptions of JML see [10,11].

JML allows Java code to be annotated with specifications, expressing for example preconditions, postconditions, or invariants, in the style of Eiffel, also known as "Design by Contract", see [15]. However, JML provides many enhancements making it much more expressive. One of these is the possibility to specify when certain exceptions may be thrown, must be thrown, or may not be thrown. Other enhancements include the possibility of introducing specification-only variables, called model variables, and the possibility of specifying (the absence of) side-effect of methods through so-called modifiable clauses.

JML annotations are a special kind of Java comments: JML annotations are preceded by `//@`, or enclosed between `/*@` and `@*/`, so that they are simply ignored by a Java compiler. These comments can be included in a `.java` source file, or written in separate `.jml` files.

Methods can be specified in the usual way, by giving pre- and postconditions. 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 the JML specifications in this paper don't use these. An example of a `normal_behavior` specification is given in Example 1 in Section 3.

Java methods can terminate abruptly, by throwing exceptions. A more general form of method specification makes it possible to specify which exceptions may be thrown, and under which circumstances. 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. For an example, see Example 2 in Section 3. These `behavior` specifications can be translated into an extended Hoare logic dealing with abrupt termination, see [6]. A `normal_behavior` specification is just a special case of a `behavior` specification, namely one with `signals: (java.lang.Exception) false`.

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.

In addition to pre- and post-conditions, JML annotations can specify invariants. An invariant is a property that holds after creation of an object by one of the constructors, and that is preserved by all 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, in the class `APDU`, the following invariant is maintained for the byte array field `buffer`:

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

This invariant is typical of objects with array fields. 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.

In addition to pre- and postconditions, a method specification in JML can include `modifiable` clauses. These clauses specify so-called frame conditions, which say that only certain fields may have their values changed by a method. For example, `modifiable:x` specifies that a method changes only field `x`. We won't discuss these annotations in this paper in detail, but modifiable clauses

will be included in the examples we give.

To specify a class it is sometimes convenient (or necessary) to introduce additional, specification-only, variables. For this purpose JML provides so-called model fields. Model fields, preceded by the keyword `model`, are just like ordinary fields, but are for specification purposes only. They can be mentioned in JML annotations, but not in the Java code. By convention, all our model fields have names that start with an underscore.

Finally, the same keywords that can be used in Java to control the visibility of fields and methods, e.g. public, private, etc. , can be used in JML to control the visibility of method specifications and invariants. E.g. an invariant that is maintained by an implementation of the class but which clients of the class need not know about will not be make `public`, but `private` or `protected`. For a public method we of course want a public specification, but there may be an additional, say protected, specification, which gives additional information for subclasses.

3 Example JML Specifications

As mentioned before, we have developed very basic specifications for all the classes in the JavaCard API, with the concrete goal to specify preconditions for methods and invariants for classes to rule out as many unwanted exceptions as possible. These specifications are relatively easy to write, and easy to check, but still provide useful information. In this section we discuss some typical examples to give the flavour of such specifications.

Whenever possible, methods have been specified by a `normal_behavior`. This requires a precondition that guarantees normal termination, i.e. that rules out that any exceptions are 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`.

Example 1 *The method* `arrayCompare` *compares parts of two arrays, given offsets within those arrays and a length saying how many array elements are to be compared. A formal JML specification is given below:*

```
public static native byte arrayCompare(byte[] src,
                                        short srcOff,
                                       byte[] dest,
                                        short destOff,
                                        short length)
```

```
 throws ArrayIndexOutOfBoundsException,
        NullPointerException;
 /*@ public_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* `ArrayIndexOutOfBoundsException`*s. These requirements immediately follow from the detailed informal specification in the JavaCard API documentation.*
- *The postcondition is simply* `true`*. This is the case with most of the specifications we have developed. This means that nothing is specified about the functionality of the method. Still, the specification is not trivial, because it does specify that the method will not throw an exception if the precondition is met.*
- *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*[1] *. 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.*

Not all methods can be specified by a `normal_behavior`. Some methods can

[1] Also, one has to be careful with such specifications, as it should *not* be specified that a `NullPointerException` *must* be thrown if `src` or `dest` is a null reference. If for example `src` is null and `destOff > dest.length` then the method may throw an `ArrayIndexOutOfBoundsException` instead. The informal API specification in fact warns that programmers should not rely on getting a specific exception if there is the possibility of throwing more than one exception. Of course, by not specifying anything about what happens outside the precondition, as we do here, we avoid this danger altogether.

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`.

Example 2 *The method* `arrayCopy` *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. A JML specification is given below:*

```
public static native short arrayCopy(byte[] src,
                                      short srcOff,
                                     byte[] dest,
                                      short destOff,
                                      short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException, TransactionException;
/*@ public_behavior
  @    requires: src != null && dest != null &&
  @              srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @              srcOff + length <= src.length &&
  @              destOff + length <= dest.length ;
  @  modifiable: dest[destOff..destOff+length-1];
  @     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 easy to see that the formal JML specification of* `arrayCopy` *above captures part of its informal specification given in the JavaCard API documentation.*
- *The precondition does not rule out all run-time 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* `TransactionException`*s in applets, assuming a certain minimal size of the commit buffer.*

Specifications similar to those of `arrayCompare` and `arrayCopy` above have been written for all the methods in the JavaCard API. All these specifications express basic preconditions that rule out unwanted run-time exceptions. More examples are discussed in [18]. In some respects these very basic specifications are already more precise than the existing informal specifications. The precise listing of all possible run-time exceptions that may occur often includes exceptions that are not declared in the code or mentioned in the informal specifications. For example, any method that invokes `arrayCopy` may throw a `TransactionException`, something which is typically not mentioned in the existing informal specifications.

4 The APDU class: public interface specification

For most methods the JML specification is a straightforward translation of (parts of) its informal specifications into JML. For the `APDU` class however this is not the case.

The `APDU` class, the largest class in the JavaCard API, handles the communication between applets and the card terminal, or CAD (card acceptance device). The implementation of the `APDU` class communicates with the card terminal using the ISO7816 protocol, but it hides much of the complexity of this protocol. In particular, as many differences as possible between the T=0 and T=1 variants of ISO7816 are hidden.

Applets receive an `APDU` object as parameter of their `process` method. Using methods such as

```
public static short getInBlockSize()
public static short getOutBlockSize()
public static byte getProtocol()
```

the applet can get relevant information about the protocol implemented by the `APDU` class. The applet can communicate with the card terminal by invoking the following methods on an `APDU` object:

```
public short setIncomingAndReceive()
```

```
public short receiveBytes(short bOff)
public short setOutgoing()
public short setOutgoingNoChaining()
public void setOutgoingLength(short len)
public void sendBytes(short bOff, short len)
public void setOutgoingAndSend(short bOff, short len)
```

These methods are meant to be invoked in a particular order. The specification of this invocation order is the only aspect in which our formal JML specifications are quite different in style from the informal specifications.

The informal API specifications give many constraints of the form "method X should only/should not be invoked if method Y has previously been invoked". For example, the informal specification of the method `receiveBytes` states that an `APDUException` will be thrown "if `setIncomingAndReceive()` not called or if `setOutgoing()` or `setOutgoingNoChaining()` previously invoked". Our formal JML specifications use a finite state machine (or state transition diagram) to specify these constraints. Considering all the constraints listed in the informal specifications leads to the finite state machine given in Figure 1. [2] We believe that such a diagram is much clearer and easier to

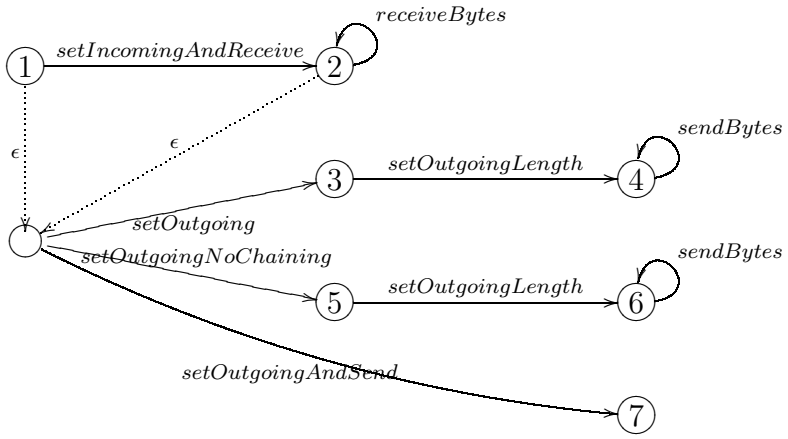

Figure 1. The APDU protocol

understand than a collection of constraints of the form "method X should only/should not be invoked if method Y has previously been invoked". A

[2] The transitions labelled with ϵ instead of a method name can be taken without invoking a method. E.g. `setOutgoing` can be invoked in state 1 or 2. These ϵ transitions only serve to make the diagram easier to understand.
To keep the specification simple enough to treat in detail here, we make the minor simplification of ignoring the possibility that `sendBytes` throws an `APDUException` with reason `NO_TO_GETRESPONSE`; including this would require an additional error state, reachable by `sendBytes`. We also ignore the method `sendBytesLong`, whose specification is almost identical to that of `sendBytes`.

further advantage is that it is easy to formalise in JML. To express the constraints on the invocation order given by Figure 1 in JML we introduce a model variable `_APDU_state`

```
//@  public model int _APDU_state;
//@  public invariant: 1 <= _APDU_state && _APDU_state <= 7;
```

and include propositions about the value `_APDU_state` in pre- and postconditions, e.g.

```
public short setIncomingAndReceive()
 /*@ public_behavior
   @    requires: _APDU_state == 1 && ... ;
   @     ensures: _APDU_state == 2 && ... ;
   @*/
```

When an applet receives an APDU as parameter of its `process` method this APDU should be in state 1, so the precondition of `process` will include

```
public abstract void process(APDU apdu)
/*@ public_behavior
  @  requires: apdu != null && apdu._APDU_state == 1 && ...
  @  ...
  @*/
```

This "forces" the applet to invoke methods on `apdu` in a correct order. Note that there are no public methods to restore an APDU to its initial state 1. This is the task of the JCRE and of no concern to clients of the API. Indeed, it is important that clients should not be able to do this.

Figure 1 does not tell the whole story. An additional constraint is that the number of response bytes sent using the method `sendBytes` should be equal to the length passed as argument to `setOutgoingLength`. Specifying this requires another model variable:

```
//@  public model int _Lr;
```

The variable `_Lr` will record the remaining response length. Not surprisingly, the method `setOutgoingLength(len)` will set `_Lr` to `len`:

```
public void setOutgoingLength(short len) throws APDUException;
 /*@ public_behavior
   @    requires: ...
   @     ensures: _Lr == len && ...
   @*/
```

and the method sendBytes will decrease the value of _Lr with len; sendBytes should not be called with len greater than the remaining response length _Lr:

```
  public void sendBytes(short bOff, short len)
  /*@ public_behavior
    @    requires: 0 <= len && len <= _Lr && ...
    @     ensures: _Lr == \old(_Lr)-len && ...
    @*/
```

The postcondition above uses the JML syntax \old(_Lr) to refer to the "old" value _Lr, i.e. the value _Lr at the beginning of the method invocation. The informal specification does not say whether or not invocations of sendBytes with len equal to 0 are allowed. The reference implementation does allow it, so we have chosen the specification above to allow it.

A final aspect of the specifications of the APDU methods is that an APDU object contains an array buffer that is used for storing the bytes that have been received or that are to be sent, and methods for receiving and sending bytes should take care to respect the buffer bounds. For example, the method sendBytes(short bOff, short len) sends len bytes starting at offset bOff in buffer to the card terminal; its precondition should include 0 <= bOff && bOff+len <= BUFFERSIZE.

Combining all the aspects discussed above leads to the formal specifications given below. In these specifications the JML keyword \result is used to refer to the result of a method, and some informal comments are given in JML specifications by including them between (* and *). Despite the preconditions all methods below can still throw APDUExceptions, e.g. if communication with the card terminal fails.

```
public short setIncomingAndReceive() throws APDUException
/*@ public_behavior
  @    requires: _APDU_state == 1;
  @  modifiable: _APDU_state, buffer[5..5+\result-1];
  @     ensures: _APDU_state == 2 &&
  @              (* data received in buffer[5..5+\result-1] *);
  @     signals: (APDUException) true;
  @*/

public short receiveBytes(short bOff) throws APDUException
/*@ public_behavior
  @    requires: _APDU_state == 2 && 0 <= bOff &&
  @              bOff+getInBlockSize() <= BUFFERSIZE;
  @  modifiable: _APDU_state, buffer[bOff..bOff+\result-1];
  @     ensures: _APDU_state == 2 && 0 <= \result &&
```

```
  @              bOff+\result <= BUFFERSIZE &&
  @              (* data received in
  @                 buffer[bOff..bOff+\result-1] *);
  @     signals: (APDUException) true;
  @*/

public short setOutgoing() throws APDUException
/*@ public_behavior
  @    requires: _APDU_state == 1 || _APDU_state == 2
  @  modifiable: _APDU_state;
  @     ensures: _APDU_state == 3;
  @     signals: (APDUException) true;
  @*/

public void setOutgoingLength(short len) throws APDUException
/*@ public_behavior
  @    requires:   (_APDU_state == 3 && 0 <= len && len <= 256)
  @             || (_APDU_state == 5 &&
  @                 0 <= len && len <= getOutBlockSize()-2);
  @  modifiable: _APDU_state, _Lr;
  @     ensures: _APDU_state == \old(_APDU_state)+1 &&
  @              _Lr == len;
  @     signals: (APDUException) true;
  @*/

public void sendBytes(short bOff, short len)
                    throws APDUException
/*@ public_behavior
  @    requires: (_APDU_state == 4 || _APDU_state == 6) &&
  @               0 <= len && len <= _Lr &&
  @               bOff + len <= BUFFERSIZE;
  @  modifiable: _Lr;
  @     ensures: _APDU_state == \old(_APDU_state) &&
  @              _Lr == \old(_Lr)-len &&
  @              (* buffer[bOff..bOff+length-1] sent *);
  @     signals: (APDUException) true;
  @*/

public void setOutgoingAndSend(short bOff, short len)
                              throws APDUException
/*@ public_behavior
  @    requires: (_APDU_state == 1 || _APDU_state == 2) &&
  @               0 <= bOff && bOff + len <= BUFFERSIZE;
  @  modifiable: _APDU_state,_Lr;
  @     ensures: _APDU_state == 7 && _Lr == 0 &&
```

```
  @              (* buffer[bOff..bOff+length-1] sent *);
  @     signals: (APDUException) true;
  @*/
```

The specifications above can be more precise by stating the possible reasons of any `APDUException`s that are thrown. For example, if `receiveBytes` throws an `APDUException`, then this is because of an `IO_ERROR` or a `T1_IFD_ABORT`, so we can specify:

```
public short receiveBytes(short bOff) throws APDUException;
/*@ public_behavior
  @   ...
  @   signals: (APDUException e)
  @          e.getReason() == APDUException.IO_ERROR
  @       || e.getReason() == APDUException.T1_IFD_ABORT;
  @*/
```

One could be more precise still and specify that an `APDUException` with reason `T1_IFD_ABORT` can only be thrown if the `APDU` implements the T=1 variant of ISO7816, i.e. if `getProtocol() == PROTOCOL_T1`.

The specifications above do not say anything about the behaviour of the methods outside the given preconditions. The specifications could be made more precise by including specifications of the behaviour of the methods outside the preconditions. For example, the informal specification of `receiveBytes` states that and `APDUException` may be thrown with the reason codes `ILLEGAL_USE`, `BUFFER_BOUNDS`, `IO_ERROR` or `T1_IFD_ABORT`. So, the specification of `receiveBytes` could be extended with

```
/*@ also
  @   public_behavior
  @    requires: true;
  @     ensures: true;
  @     signals: (APDUException e)
  @          (_APDU_state != 2
  @           && e.getReason() == APDUException.ILLEGAL_USE )
  @       || ( (bOff < 0 || bOff+getInBlockSize() > BUFFERSIZE)
  @           && e.getReason() == APDUException.BUFFER_BOUNDS)
  @       || e.getReason() == APDUException.IO_ERROR
  @       || e.getReason() == APDUException.T1_IFD_ABORT;
  @*/
```

This additional specification constrains the possible behaviour of `receiveBytes` outside the precondition given earlier. Note that this additional specification is not really of interest to the programmer of well-behaved applets. In well-coded applets invocations of `receiveBytes` should never cause an `APDUException`

with reason ILLEGAL_USE or BUFFER_BOUNDS. Still, for someone implementing the API it is relevant to know how to react to ill-behaved applets. Since JML specifications for a class can be distributed over several files, it would make sense to collect specifications that are not of interest to an applet developer but that are only of interest to an API implementor, such as the one above, in a separate file.

5 The APDU class: implementation

To prove that a particular API implementation meets the specification above, the model variables used in the specification need to be related to some fields or methods that are present in the implementation.

The reference implementation of the APDU class uses five booleans flags to record whether or not certain methods have been invoked:

```
private boolean getIncomingFlag()
private boolean getSendInProgressFlag()
private boolean getOutgoingFlag()
private boolean getOutgoingLenSetFlag()
private boolean getNoChainingFlag()
```

Together, these flags record the _APDU_state of an APDU object. The relation between these flags and the _APDU_state can be expressed by invariants, such as

```
/*@ private invariant:
  @     _APDU_state == 3
  @ <==>
  @     getOutgoingFlag() && !getNoChainingFlag()
  @     && !getOutgoingLenSetFlag();
  @*/
```

Note that this invariant is private, as it is of no concern to clients of the APDU class.

Trying to establish the relation between the flags and the _APDU_state it turns out that the reference implementation does not distinguish between the states 4 and 7. This is hardly surprising, given that setOutgoingAndSend(bOff,len) is implemented as

setOutgoing(); setOutgoingLength(len); sendBytes(bOff, len).

in the reference implementation. The best invariant one can find is

```
/*@ private invariant:
  @     _APDU_state == 4 || _APDU_state == 7
  @ <==>
  @     getOutgoingFlag() && !getNoChainingFlag()
  @     && getOutgoingLenSetFlag();
  @*/
```

All this means that the statement in the informal specification that "`sendBytes` throws an `APDUException` if `setOutgoingAndSend` has been previously invoked" is not quite true for the reference implementation: following an invocation of `setOutgoingAndSend` with invocation of `sendBytes` will not always throw an `APDUException`. However, after invoking of `setOutgoingAndSend` the value of `_Lr` will be zero, so, by the precondition of `sendBytes`, we can only invoke `sendBytes` without throwing an `APDUException` if the second argument is equal to zero, and such invocations are completely harmless.

Relating the model variable `_Lr` to the implementation is much simpler than `_ADPU_state`, as the implementation includes a method `getLr()` which returns exactly `_Lr`:

```
//@ private invariant: getLr() == _Lr ;
```

We could have chosen to specify the public interface of `APDU` in terms of boolean flags as actually used in the reference implementation, which would have made it easier to relate the reference implementation and the specification. We have chosen not to do this because we believe that a specification of the correct invocation sequences with a diagram like Figure 1 is much easier to understand than a specification involving several boolean flags, and that in such a diagram one is much less likely to make mistakes.

We should stress that so far we have not attempted formal verification (i.e. using theorem provers) of the `APDU` implementation, but just trying to convince ourselves that the reference implementation meets our JML specifications already raised some interesting questions. For example, the reference implementation of the method `setOutgoingAndSend` invokes `sendBytes`, so, like `sendBytes`, it can throw an `APDUException` with reason `NO_TO_GETRESPONSE` or `T1_IFD_ABORT`. This is something the informal specification fails to mention. Another example is that the reference implementation of the method `receiveBytes` throws an `APDUException` with reason `BUFFER_BOUNDS` when `bOff + getInBlockSize() >= BUFFERSIZE`, whereas one would expect this exception to be thrown only when `bOff + getInBlockSize() > BUFFERSIZE`.

6 Conclusion

Despite the fact that our formal specifications of the JavaCard API are incomplete, we believe they provide a useful documentation to complement the existing informal specifications, included as appendix in [4] and available as Javadoc-generated HTML from Sun's website [9].

Some properties expressed by the JML annotations cannot be found in the informal specification. In these cases the JML specification of the JavaCard API is more informative than the informal specification and even the source code of the reference implementation. For instance, our JML specifications state precisely for every method which exceptions it may throw at run-time. This information is useful in preventing uncaught exceptions, a common source of failures. Furthermore, the JML invariants make explicit many considerations and assumptions that are implicit in the design, and which are relied upon for the correctness of the code.

For properties expressed by the JML annotations that can be found in the informal specifications, using a formal specification language has the advantage of leaving no room for ambiguity. For example, the requirement `bOff + getInBlockSize() <= BUFFERSIZE` in the JML specification of `receiveBytes` corresponds exactly to the expression "enough buffer space for incoming block size" in the informal specification, but is of course more precise.

Writing our formal JML specifications for the JavaCard API has not been very difficult. Writing JML annotations while developing the code, instead of afterwards as we have done, would require less effort still. In particular, the discovery of invariants that are maintained is often a rediscovery of design ideas and decisions that have gone into the existing implementation. Here the JML specifications expose some of the thoughts and considerations that have gone into the design of the API. All JML annotations should be easy to understand for any Java programmer, assuming some basic knowledge of formal methods. Indeed, one of the design goals of JML is 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. Several tools are being developed for JML (see also [12]):

- At Iowa State University, tools are being developed for generating Javadoc-like HTML from JML specifications, and for generating code that includes run-time checks of preconditions and invariants. As in Eiffel [15], code with such run-time checks helps in debugging. While useful in the development and testing phase, leaving run-time checks in the final JavaCard source code

of an applet or API implementation is probably undesirable, for reasons of efficiency and size. [3]

- The extended static checker ESC/Java [5], developed at Compaq System Research Center, does not check JML assertions at run-time, but tries to check them at compile-time. This tool can automatically check for certain kinds of common errors in Java(Card) code, such as dereferencing `null` or indexing an array outside its bounds. The extended static checker should be a useful tool in the development of both applets and API implementations, pointing out some violations of preconditions and invariants fully automatically, at the push of a button.
- Of course one cannot expect arbitrarily complex properties to be checked fully automatically by ESC/Java, but one can try proving these interactively using the LOOP tool being developed at the University of Nijmegen. The LOOP tool translates JML-annotated code into proof obligations for theorem provers such as PVS or Isabelle. This approach is more labour-intensive, but for vital properties of JavaCard API implementations and applets the effort may well be justified. The first – very modest – steps to verify JML-annotated JavaCard code formally using the LOOP tool and the theorem prover PVS are reported in [3].

More examples of JML specifications for the JavaCard API are discussed in [18]. 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, providing a useful addition to the existing documentation of the JavaCard API, and bringing to light ambiguities in the existing informal specifications. Future work will focus on developing more detailed JML specifications of the JavaCard API, and using these as the basis for formal verification of source code – both of applets and API implementations – 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 (WADT'99)*, number 1827 in LNCS. Springer, 2000.

[3] J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification

[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.

of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *JavaCard Workshop (JCW'2000)*. INRIA Sophia Antipolis, 2000.

[4] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

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

[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, 2000.

[7] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. Conditionally accepted for publication in *Software Tools for Technology Transfer*, 2000.

[8] 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.

[9] *The Java Card 2.1.1 Application Programming Interface (API)*. Sun Microsystems, 2000.

[10] 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.

[11] 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.

[12] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'2000 Companion*. ACM, 2000. Also as Tech. Rep., Dept of Computer Science, Iowa State University, TR00-15, August 2000.

[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] B. 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, 1996.

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

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