On the combination of Java Card Remote Method Invocation and JML

Martijn Oostdijk and Martijn Warnier

Dept. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{martijno,warnier}@cs.kun.nl

Abstract This paper explores the possibilities for using the Java Modeling Language (JML) to specify Java Card applets that use Remote Method Invocation (JCRMI). The JCRMI framework makes it possible to call methods directly on a Java Card smart card without the (explicit) use of low level byte sequences, called APDUs. We introduce a new way of designing JCRMI applets, using the Java Modeling Language (JML) to formally specify (part of) its code. It turns out that some advanced JML specification features, such as model variables, are necessary to specify JCRMI applets. Two JML tools, the JML runtime assertion checker and the LOOP tool, are subsequently used to verify that the implementation satisfies the JML specifications. We conclude that the JML specifications are simpler and easier to write, understand and verify when using JCRMI. Ideally this should lead to more trustworthy and error free code.
Keywords: Java, JML, Remote Method Invocation, program specification, verification
Classification: 68N19, 68N30, 68Q60 (AMS'00); D.1.5, D.2.4, F.3.1 (CR'98).

1 Introduction

The Java Card [8] language can be used to write Java applications for Smart Cards. The language forms a super-subset of Java. Java Card is a subset in the sense that only single-threaded programs are possible and some larger integral data types (such as `long`, `float` and `double`) are omitted. It is a superset because there are some Java Card specific constructs like an applet firewall mechanism, which ensures that multiple Java Card applications (usually called applets) can loaded safely on a card without compromising security, and a dedicated cryptographic API. Typical examples of Smart Cards are SIMs in mobile phones and electronic wallets.

The latest version of Java Card [9], version 2.2 (June 2002) introduces *Java Card Remote Method Invocation* (JCRMI). At the moment of writing (October 2003) there are no actual cards available to end users which support Java Card 2.2.[1] Sun does provide a simulator, namely the C-language Java Card Runtime Environment `cref`, included in the Java Card 2.2 Development Kit [18].

[1] However several card manufactures are already at the testing stage.

The main advantage of JCRMI is that it provides a higher abstraction level than previous versions of Java Card. Low-level Application Protocol Data Units (APDUs) are no longer directly used for communication between a client (sometimes called a terminal) and a smart card.

In Sect. 2 we give an analysis of JCRMI and of the differences with earlier versions of Java Card. A simple JCRMI applet is used as a running example. In Sect. 3 the Java specification language JML [1,17] is used to give precise specifications of Java Card fragments of this applet. Two tools, the JML runtime assertion checker [10] and the LOOP tool [4,16], are used to verify these JML specifications. Note that we do not analyze the RMI-framework itself, rather we show how to use JCRMI and how to combine it with JML. Sect. 4 ends the paper with several conclusions and suggestions for future work.

2 Java Card Remote Method Invocation

In previous versions of Java Card (i.e. before 2.2) all communication between a card and a client occurred with Application Protocol Data Units (APDUs). The international ISO7816 standard (part 4) [3] describes the structure of APDUs. In a Java Card applet these were represented as simple byte arrays with little more structure than a header containing instructions as to which method should be called, information about the length of the data and some simple (optional) parameters.

Since APDUs provide a low level way of communication, a lot of the effort in programming Java Card applets and clients is spent on finding the correct translation from the high level Java (Card) method calls to low level APDUs and vice-versa. In previous versions of Java Card an applet programmer had no other choice but to use low level operations to implement this translation. Java Card Remote Method Invocation (JCRMI) addresses this problem.

Java Remote Method Invocation is a framework which allows objects to call methods of objects residing on other Java Virtual Machines (JVM) in the same manner as local methods are called. In the context of Java Card this means that objects residing on the client side can call methods from applets running on a Java Card. Originally developed at Gemplus Research Labs by Jean Jacques Vandewalle and Laurent Lagosento JCRMI was eventually sold to SUN, who put it in the Java Card 2.2. standard.

Figure 1 describes the JCRMI architecture. Basically JCRMI adds a *middleware* layer (indicated by the dotted box in the Fig. 1) which translates calls to methods of the remote object to APDUs (marshaling). On the card the APDUs are translated back to calls to methods of the remote object (unmarshaling).

The *remote object* resides on the card and is created upon applet installation. The client can get a reference to this remote object (the dashed arrows in Fig. 1). When the client calls a method on the remote object (on the card) the method is actually called on a *stub object* which resides on the client side. This stub 'translates' the method call to an APDU command and sends this message to the card. On the Java Card side this APDU is passed on to a *skeleton* object

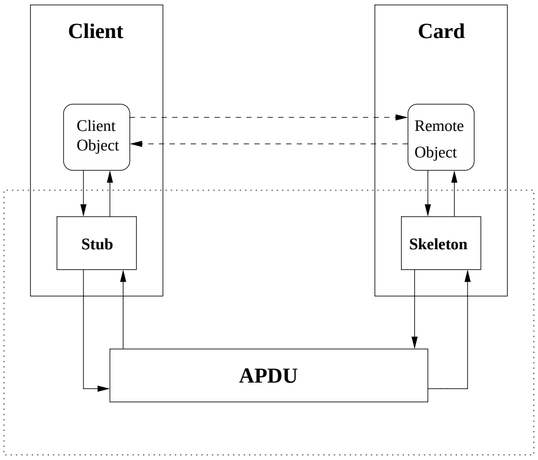

Figure 1. Java Card Remote Method Invocation architecture

which translates the APDU back to a method call. The method call is invoked and the return value is again passed on to the skeleton which translates this back into an APDU which is sent to the client. Finally on the client side the stub translates the APDU back to a return value. From the client's perspective it looks like the method call was made on a local object, removing the need for (manual) translations to and from low-level APDUs.

Three different levels concerning the JCRMI-framework can be distinguished: (i) the applet level, (ii) the API-level and (iii) the OS-level. This paper is concerned with the applet level, i.e., how to use the JCRMI functionality from a developers perspective. At the API level there are some additional classes, most notably a `Dispatcher` class, a `RMIService` class and a `CardRemoteObject` class. The `Dispatcher` provides methods to add services to its registry, dispatch APDU commands to registered services, and remove services from its registry. Remote objects are added to the register via the dispatcher making them available for remote method calls. This happens at applet creation when the constructor of the actual applet is called by the `install` method. The `RMIService` together with the `CardRemoteObject` class contain the minimum required functionality to implement JCRMI. The marshaling and unmarshaling libraries are present at the OS-level. The *skeleton* class that is used for this purpose is created dynamically at applet installation by the OS. In the remainder of this paper we only look at JCRMI at the applet level.

There are some limitations to JCRMI in comparison to standard Java RMI, especially because JCRMI only works one way (from client to applet), the smaller data types and limited resources available in Java Cards. The Java Card 2.2 Runtime Environment (JCRE) Specification [20, Chapter 8] states that only primitive types and single-dimensional arrays of primitive types can be used as parameters of remote method calls. Return types can be primitive types, single-dimensional arrays of primitive types, any remote interface type and a `void` return type.

During remote method invocation then all parameters, including single dimensional arrays, are always transmitted by value [20]. This means that they are encoded, in this case according to the JCRMI message protocol for Java Card 2.2. For example a representation of a byte array consists of a type encoding and the length of the array followed by its actual contents. All these are simply concatenated and form an APDU (a byte sequence). The same encoding is used for all return types except for return values corresponding to remote object references. These are returned to the client as a reference. When a remote method invocation terminates abnormally an exception is returned, which consists of the type of the exception (ISOException, UserException etc.) and the status word. Again the exception object is returned by value.

JCRMI applets can be developed fairly generically. The starting point of JCRMI applet development is the remote interface. The applet implements this interface and the client has to know which remote methods can be called, so both card and client know the interface. In our approach the interface is annotated with a JML specification so both parties know what to expect. After the interface and its JML specification are specified, client and applet can be developed in parallel. The `rmic` compiler is used to generate a `Stub` class for the Remote object[2]. The client has to include the `Stub` in order to be able to translate method calls to APDUs and vice-versa. The use of this special compiler marks the only difference between JCRMI and the normal Java Card applet development and installation process. Fig. 2 gives an overview of this process. The arrows are labeled with the tools that are used for the transitions.

As an aside, it is good to note that JCRMI itself does not involve any security in the form of authentication or confidentiality/integrity of parameters and results. There are a number of ways to add this security: the user can write his own security service which implements this functionality, another alternative is to use a Global Platform [13] enabled card and use the security features provided by the Global Platform API, or one can use the JASON framework [6], which extends JCRMI with integrated support for security.

We now introduce our running example which is an adaptation of an applet from [14]. It can be seen as a simple phone card which can be credited once and debited until the balance is zero. The reason we present this trivial example is twofold; We want to explain the essence of JCRMI which can be best explained using a simple example and we want to be able to run the applet on a simulator. The only simulator that is publicly available for JCRMI is the `cref` simulator from SUN which does not support any cryptographic functionality. We do not show a corresponding client since writing one is relatively straightforward. By using the OpenCard Framework API [2] one only has to obtain a reference to the `Remote` object of the applet to start calling methods on the card using JCRMI.

[2] The `rmic` compiler also generates a skeleton object. When using JCRMI the skeleton object is not used since loading a JCRMI applet on a Java Card 2.2 card generates the corresponding skeleton dynamically.

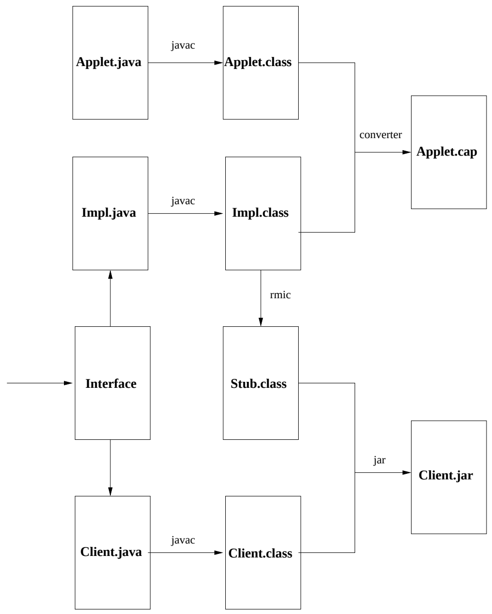

Figure 2. JCRMI applet development process

A JCRMI applet consists of at least one interface and two classes:

1. *A remote interface*, this interface extends the `java.rmi.Remote` interface and gives the signature of the methods that can be called with JCRMI. This interface must also be present on the client side.
2. *An implementation of the remote interface*, this class contains the actual implementation and is also used to generate a stub class for the client.
3. *The applet class*, this class extends `javacard.framework.Applet` and contains the usual `select`, `install` and `process` methods inherited from this class. This class forms the entry point for all method calls and directs these to the actual implementation (2).

In the sections below we look at the remote interface and its implementation in a bit more detail.

2.1 Remote interface

The remote interface is part of the applet. It also needs to be imported by the client so that the client knows which remote methods can be called. The interface contains the signature of the remote methods. We first give a Java listing of the interface and subsequently explain the Java code.

```
package simplermi;

import java.rmi.*;
import javacard.framework.UserException;

public interface SimpleRMIinterface extends Remote {

   public static final short ALREADY_ISSUED  = (short) 0x6000;
   public static final short NO_BALANCE_LEFT  = (short) 0x6001;
   public static final short NEGATIVE_BALANCE = (short) 0x6002;

   public void setBalance(short b) throws UserException, RemoteException;
   public short getValue() throws RemoteException;
   public void decValue() throws UserException, RemoteException;
}
```

The interface describes three remote methods, namely `setBalance(short b)`, `getValue()` and `decValue()`. For now it is not important what these methods exactly do. In Sect. 2.2 we give the actual implementation of the interface and in Sect. 3.1 we give a precise JML specification.

Notice that the interface extends the interface `Remote`. This is necessary if we want to use JCRMI. All methods can throw a `RemoteException` which is thrown to indicate that a communication-related exception has occurred during the execution of a remote method call. In essence this is the same as the `APDUException` in earlier versions of Java Card.

The other exception that can be thrown is the `UserException`. These are the same as in earlier versions of Java Card. The three constants are user defined status words used by these exceptions.

2.2 Implementation of the remote interface

The implementation of the remote interface from the previous section is pretty straightforward. We again first give the listing of this class and continue with a short explanation of the code.

```
package simplermi;

import javacard.framework.UserException;
import javacard.framework.service.CardRemoteObject;
import java.rmi.RemoteException;

public class SimpleRMIimpl extends CardRemoteObject
                           implements SimpleRMIinterface{

  private short balance;
  private boolean issued;
```

```
  public SimpleRMIimpl(){
    super();
    balance = 0;
    issued = false;
  }


  public void setBalance(short b) throws UserException, RemoteException{
    if (issued)
      UserException.throwIt(ALREADY_ISSUED);
    if (b > 0) {
      issued = true;
      balance = b;
    } else
      UserException.throwIt(NEGATIVE_BALANCE);
  }

  public short getValue() throws RemoteException{
    return balance;
  }

  public void decValue() throws UserException, RemoteException{
    if (balance > 0)
      balance--;
    else
      UserException.throwIt(NO_BALANCE_LEFT);
  }
}
```

The implementation of the three methods is not discussed here. The main idea is that the `balance` is set in a secure environment at the beginning of the card's life-cycle. Hence no cryptographic protection is required. Notice that the only JCRMI specifics for this class are that it `extends CardRemoteObject`. In all other aspects it is a normal Java class.

The 'real' applet (i.e. the class that `extends javacard.framework.Applet`) is basically a wrapper class that forms the entry point of the applet and passes the remote method calls on to the implementation in the class `SimpleRMIimpl`. Since the applet class can be completely generic and is of itself not that interesting we do not show it here. For completeness sake we have included this implementation in Appendix A.

There are two main differences between 'standard' Java Card and Java Card RMI. First of all, there is the obvious abstracting away from APDUs in JCRMI. Although it is still possible to handle APDUs manually (either by writing a specific `service` or by just handling them in the `process` method as before) it is no longer *necessary* when using JCRMI. This extra abstraction layer clearly makes programming easier and hopefully prevents faults with byte operations such as right shift (`>>`) or bitwise-and (`&`) which are no longer necessary (or at least not as much).

Second, standard Java Card has a single entry point, the `process` method. Often the body of this method contains a nested `switch` which matches the instruction byte `INS` and some dedicated state variable (see e.g. [15]). In this fashion the `process` method can handle the complete control flow of the applet. When using JCRMI there are multiple entry points in the applet (i.e. each method which belongs to the `Remote` object forms an entry point) making it harder to deal with the control flow. Each method has to check itself if it can be called in a particular state.

To illustrate these points consider for example the `processGetValue()` method below. In Java Card 2.1 this method typically looks something like this:

```
private void processGetValue(APDU apdu) {
  byte[] buffer = apdu.getBuffer();
  apdu.setOutgoing();
  apdu.setOutgoingLength((short)2);
  buffer[0] = (byte)((balance >> 8) & 0x00FF);
  buffer[1] = (byte)(balance & 0x00FF);
  apdu.sendBytes((short)0, (short)2);
}
```

Notice that this code is clearly more complicated then the JCRMI example of the same method (i.e., `getValue()` from the previous Section) which method's body consists of one return statement.

3 Specifying the JCRMI applet with JML

The (by now standard) Java Modeling Language, JML [17], is a behavioral interface specification language designed to specify Java modules. It can be used for classes as a whole, via class invariants and constraints, and for the individual methods of a class, via method specifications consisting of pre-, post- and frame-conditions (assignable clauses). In particular, it is also possible within a method specification to indicate if a particular exception may occur and which post-condition results in that case.

JML annotations are to be understood as predicates, which should hold for the associated Java code. These annotations are included in the Java source files as special comments indicated by `//@`, or enclosed between `/*@` and `*/`. They are ignored by the Java compiler and recognized by special tools like the JML runtime checker [10], ESC/Java [12], the LOOP tool [4,16], the Krakatoa verification condition generator [11] and the JACK tool [7] which was specially created for verifying JML annotated Java programs by the Smart Card manufacturer Gemplus. We give an example JML method specification of some method `m()`:

```
/*@ behavior
      requires precondition;
    assignable items that can be modified;
       ensures normal postcondition;
    signals (E) exceptional postcondition;
```

```
  */
 public void m()
```

Such method specifications may be understood as an extension of correctness triples $\{P\}m\{Q\}$ used in Hoare logic, because they allow both normal and exceptional termination. We shall see more examples of method specifications below.

JML is intended to be usable by Java programmers. Its syntax is therefore very much like Java. However, it has a few additional keywords, such as `==>` (for implication), `\old` (for evaluation in the pre-state), `\result` (for the return value of a method, if any), and `\forall` and `\exists` (for quantification). Other JML language constructs are explained as we encounter them.

Below we give behavioral JML specifications for the remote interface and the class that implements this interface. Most of the JML specifications will be presented in the remote interface. Since it is part of the applet, but also available for the client it makes sense to concentrate the specifications here.

3.1 Remote interface

We first present the code of the interface including JML specifications and will then discuss the specification in more detail.

```
package simplermi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javacard.framework.UserException;

public interface SimpleRMIinterface extends Remote {

  /*@ model instance short _balance; */
  /*@ model instance int _state; */
  /*@ model final static int _STATE_INIT; */
  /*@ model final static int _STATE_ISSUED; */
  /*@ model final static int _STATE_LOCKED; */

  /*@ invariant
        _STATE_INIT != _STATE_ISSUED &&
        _STATE_INIT != _STATE_LOCKED &&
        _STATE_ISSUED != _STATE_LOCKED; */

  /*@ invariant
        _state == _STATE_INIT ||
        _state == _STATE_ISSUED ||
        _state == _STATE_LOCKED; */

  /*@ invariant
        _balance >= 0
        && ( _state == _STATE_INIT ==> _balance == 0 )
```

```
       && ( _state == _STATE_ISSUED ==> _balance > 0 )
       && ( _state == _STATE_LOCKED ==> _balance == 0 ); */

  /*@ constraint
       ( _state == _STATE_ISSUED && \old(_state) == _STATE_ISSUED ) ==>
       0 < _balance  && _balance <= \old(_balance); */

  public static final short ALREADY_ISSUED  = (short) 0x6000;
  public static final short NO_BALANCE_LEFT  = (short) 0x6001;
  public static final short NEGATIVE_BALANCE = (short) 0x6002;

  /*@ behavior
       requires true;
     assignable _balance, _state;
        ensures (\old(_state) == _STATE_INIT && b > 0)
                ==> _balance == b;
        signals (UserException)
                ( b <= 0 || \old(_state) != _STATE_INIT ) &&
                _balance == \old(_balance);
        signals (RemoteException) true;
   */
  public void setBalance(short b) throws UserException, RemoteException;

  /*@ behavior
       requires true;
     assignable \nothing;
        ensures \result == _balance;
        signals (RemoteException) true;
   */
  public short getValue() throws RemoteException;

  /*@ behavior
       requires true;
     assignable _balance, _state;
        ensures (\old(_state) == _STATE_ISSUED && \old(_balance) > 1)
          ==> (_state == _STATE_ISSUED &&
              _balance == \old(_balance) - 1)
              && (\old(_state) == _STATE_ISSUED && \old(_balance) == 1)
          ==> (_state == _STATE_LOCKED && _balance == 0);
        signals (UserException)
              ( \old(_state) != _STATE_ISSUED ) &&
              _balance == \old(_balance) && _state == \old(_state);
        signals (RemoteException) true;
   */
  public void decValue() throws UserException, RemoteException;
}
```

The JML specifications use so-called *model-fields* [17,5]. We use model fields here because we want to say something about the `_balance` and the `_state` of the applet. Note that these two model fields are marked as *instance* model fields, normally Java interfaces can only contain `public static final` fields. The JML keyword `instance` indicates that these fields have to be linked to actual fields in the implementation of this interface (see Sect. 3.2). The other three models are used as constants representing the different states: `_STATE_INIT`, `_STATE_ISSUED` and `_STATE_LOCKED`. We will use the convention that model field names start with an underscore (_) to easily distinguish them from proper fields.

The specification has three *invariants*. These are predicates which have to be maintained by all methods. We use three separate invariants only for readability; semantically they are conjuncted into one big invariant. The first two invariants are not that interesting, since they say that all three states are different and no other states exist. The third invariant gives a more high level specification: the `_balance` is always positive or zero, if the `_state` is `_STATE_INIT` then the `_balance` is zero, if the `_state` is `_STATE_ISSUED` the `_balance` is positive, and if the `_state` is `_STATE_LOCKED` the `_balance` is again zero. Summarizing we want to say that only in state `_STATE_ISSUED` the `_balance` is positive and in the other (non-operational) states the `_balance` is zero.

The specification also uses a *constraint* which describes "class-wide" relations between pre- and post- states. The JML keyword `\old` is used here to refer to a pre-state of a method, i.e. the state before a particular method is called. The constraint says that in normal use the `_balance` can only be decremented. This expresses an important safety property. Note that there is some overlap in the invariants and constraints, this is done deliberately to make the specification as transparent as possible. We use constraints here mostly for the specification of control flow (like in [15]).

The three method specifications are pretty similar, so we only look at the specification of method `public void decValue()`. The specification of the other two methods should be self explainable. The `requires` clause of the method is `true` meaning that this method can always be called. The only fields that can be altered by the method are the `_balance` and `_state` fields, as specified in the `assignable` clause. The `ensures` clause is more interesting, it states that under normal circumstances the `_balance` will be decremented by one and if the `_balance` field reaches zero the applet will go to the locked state `_STATE_LOCKED`. There are also two `signals` clauses: a `UserException` will be thrown if someone tries to call this method from another state than state `_STATE_ISSUED` and the `_balance` and `_state` will then stay the same; it is also possible that a `RemoteException` will be thrown in which case we have no additional information about the value of fields or states.

3.2 Implementation of the remote interface

Since the implementation of the remote interface has to respect the specification of the interface it implements we only have to specify how the model fields used in the interface are related to real fields. We use two `represents` clauses in JML to give the relation between the model fields from the remote interface and actual fields in the implementation class:

```
/*@ private represents _balance <- balance; */

/*@ private represents _state <- (!issued ? _STATE_INIT :
                                issued && balance > 0 ? _STATE_ISSUED :
                                _STATE_LOCKED); */
```

For the model field `_balance` this is trivial: we make it equal to the private field `balance` (which stores the 'real' balance). The model field `_state` can be described in terms of a boolean value `issued` and the `balance`. The boolean `issued` is set to value `false` during applet creation (in the constructor) and set to `true` if the method `public void setBalance(short b)` is called. So if `issued` has value `false` we are in state `_STATE_INIT`. If `issued` is `true` there remain two possibilities: (i) `balance > 0` in which case the state is `_STATE_ISSUED` or (ii) `balance == 0`, when we are in state `_STATE_LOCKED`.

Note also that the above representation in this class is `private`: the class has to implement the remote interface and its specification, but there is no reason why the specification of the interface should know *how* the class implements the interface and its specification. In this way a clear separation of concerns can be established.

Writing JML specifications is easier when using JCRMI. Looking ones more at the `private void processGetValue(Apdu apdu)` method from Sect. 2.2 this point should be obvious. Its JML specification can be seen below:

```
/*@ behavior
      requires true;
    assignable apdu.buffer[*];
       ensures balance == (short)((apdu.buffer[0] & 0x00FF) << 8 |
                                  (apdu.buffer[1] & 0x00FF));
       signals (APDUException) true;
 */
 public void processGetValue(APDU apdu)
```

Compare this to the specification of the corresponding JCRMI `getValue()` method from Section 3.1. The low level APDUs in standard Java Card often cause significant specification problems. The readability of the JML specifications also improves quite dramatically when JCRMI is used. We think that these advantages easily overcome the lack of a single entry point in JCRMI style applets.

3.3 Verifying the JML specification

We use two of the available JML tools to verify that the JML specifications are respected by the implementation i.e., that the implementation described in Sect. 2.2 satisfies the specification of the remote interface from Sect. 3.1: The JML runtime assertion checker and the LOOP tool.

The JML runtime assertion checker The JML runtime assertion checker (`jmlrac`) [10] basically checks if any of the JML specifications are violated during a particular execution of a Java (Card) program. Runtime assertion checking is a form of testing and testing can typically find errors but can not confirm the *absence* of errors. If the user wants to have complete confidence that the program does what the JML specification says then he or she has to create a complete test set consisting of all possible inputs for the program. Normally this is not feasible for any significant program. It is of course a lot easier to test all possible `short` input values for a JCRMI method like `setBalance(short b)` then to send all possible APDUs for a similar method in standard Java Card.

However using `jmlrac` is easy, which makes it ideal to quickly find simple errors in both code and specification. We used `jmlrac` with a simple test set that checked things like the impossibility to decrement below zero or to set the balance more then once. This resulted in no errors at all.

Normally when one wants to use some form of testing for a Java Card applet it is necessary to either use a Java Card simulator or use a real Java Card smart card. In the case of JCRMI it is not necessary to use a real card or simulator. The interface and implementation class can be stripped of their JCRMI features (specifically extending `Remote` for the interface and extending `CardRemoteObject` for the implementation) and compiled with a standard java compiler. The user only has to write a wrapper class containing a `main` method that calls the methods from the implementation class. Note that this approach has some limitations. Java Card specific API calls and other constructs like the firewall can not be tested in this manner.

JML runtime assertion checking is only possible in this way since assertion checking of Java Card applets requires a Java Card simulator written in Java. At the moment of writing the C-language Java Card Runtime Environment `cref` is the only publicly available runtime environment which supports JCRMI .

The LOOP tool If a user wants to have more certainty that the given specification is respected by the code he or she should use a verification method which proves that the implementation is correct. In contrast with the JML runtime assertion checker, the LOOP tool [4,16] is used for program verification. It can be used to prove that a program satisfies a JML specification.

The LOOP tool translates Java code and JML annotations into theories for the theorem prover PVS [19]. Within PVS there is a semantics of the sequential part of Java and a semantics for a substantial subset of JML which includes model fields (see [5]). Proving the correctness of the specifications requires some user interaction which makes this a verification method that should only be used for certain core parts of programs. In this case the user interaction was limited because the JCRMI program and specification are relatively simple. All proofs required only the most basic user interaction possible and the actual verification for all three programs only used the automatic weakest precondition strategy discussed in [14].

4 Conclusions and Future Work

Traditional Java Card programs have a single entry point with messy and error-prone low-level (byte) operations. Precise JML specifications of such code are hard to write, read and understand. Java Card Remote Method Invocation provides a level of abstraction that simplifies this process. It allows one to write more readable code with proper JML specifications. Precise JML specifications are especially important when different vendors have to implement terminals which communicate with the same smart card. Moreover the JML specified remote interface can also be used by different smart card application developers (e.g. different banks) to develop their own applets.

Verification of JML specifications is also easier in the JCRMI setting: testing is easier since no simulator or actual cards are necessary, which probably also speeds up the initial development phase; advanced program verification techniques are also more feasible because both code and specifications are simpler.

As for future work, we want to look at a reference implementation of the JCRMI functionality for the API and Java Card OS (briefly discussed in Sect. 2) and see if we can formally specify and prove the correctness of this implementation. Once the correctness of the API and OS level of JCRMI is proved formally, the approach discussed in this paper is complete in the sense that correct specifications only depend on the API and OS level of JCRMI.

References

1. JML web site. `http://www.jmlspecs.org`.
2. OpenCard Framework web site. `http://www.opencard.org`.
3. International Standard ISO 7816. available at: `http://www.iso.org/iso/en/isoonline.frontpage`.
4. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in LNCS, pages 299–312. Springer, Berlin, 2001.
5. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs. Proceedings of the ECOOP'2003 Workshop*, 2003.
6. Richard Brinkman and Jaap-Henk Hoepman. Secure method invocation in JASON. In *5th USENIX Smart Card Research and Advanced Application Conference*, pages 29–40, San Jose, CA, USA, November 2002.
7. Lilian Burdy, Jean-Louis Lanet, and Antoine Requet. JACK (Java Applet Correctness Kit), 2002. `www.gemplus.com/smart/r_d/trends/jack.html`.
8. Z. Chen. *Java Card Technology for Smart Cards.* The Java Series. Addison-Wesley, 2000.
9. Zhiqun Chen. *Java Card technology for smart cards: architecture and programmer's guide.* Addison-Wesley, June 2000.
10. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H.R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, Las Vegas, 2002.

11. E. Contejean, J. Duprat, J.-C. Filliâtre, C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. *Journ. of Logic and Algebraic Programming*, to appear. Available via the Krakatoa home page at `www.lri.fr/~marche/krakatoa/`.
12. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.
13. Global Platform. Open platform card specification version 2.1, June 2001. Available at `http://www.globalplatform.org/`.
14. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journ. of Logic and Algebraic Programming*, To appear.
15. Bart Jacobs, Martijn Oostdijk, and Martijn Warnier. Source Code Verification of a Secure Payment Applet. *Journ. of Logic and Algebraic Programming*, To appear.
16. Bart Jacobs and Erik Poll. Java Program Verification at Nijmegen: Developments and Perspective. Technical Report NIII-R0318, Nijmegen Institute for Computer and Information Sciences, 2003.
17. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specification for Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.
18. Sun Microsytems. Java Card 2.2 Development Kit. available at: `http://java.sun.com/products/javacard/`.
19. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
20. Sun Microsytems. *Java Card 2.2 Runtime Environment (JCRE) Specification*, June 2002. available at: `http://java.sun.com/products/javacard/specs.html`.

Appendix

A The JCRMI applet class

The example applet discussed in Section 2 consists of an interface (see Section 2.1), an implementation of this interface (see Section 2.2) and the actual applet class. This last class extends the `javacard.framework.applet` class from the Java Card API. Every applet has to extend this class, in the context of JCRMI the class can be completely generic and is therefore not that interesting. We include its code here for completeness sake:

```
package simplermi;

import java.rmi.*;
import javacard.framework.*;
import javacard.framework.service.*;

public class SimpleRMI extends Applet {
```

```
    private Dispatcher disp;
    private RemoteService serv;
    private Remote sri;

    public SimpleRMI() {
        sri = new SimpleRMIimpl();


        disp = new Dispatcher( (short) 1);
        serv = new RMIService(sri);
        disp.addService(serv, Dispatcher.PROCESS_COMMAND);

        register();
    }


    public static void install(byte[] aid, short s, byte b) {
        new SimpleRMI();
    }

    public void process(APDU apdu) throws ISOException{
        disp.process(apdu);
    }
}
```

One can see that the implementation (remote object `sri`) is added to `Dispatcher` together with the `RMIService` which gives the minimal functionality needed to preform JCRMI. Incoming APDUs are then simply passed to the `Dispatcher` using the `process` method.