

# Verifying the Mondex Case Study

Peter H. Schmitt  
Universität Karlsruhe  
Institute for Theoretical Computer Science  
76128 Karlsruhe, Germany  
pschmitt@ira.uka.de  
www.key-project.org

Isabel Tonin  
aicas GmbH  
Haid-und-Neu-Str. 18  
76131 Karlsruhe, Germany  
tonin@aicas.com  
www.aicas.com

## Abstract

*The Mondex Case study is still the most substantial contribution to the Grand Challenge repository. It has been the target of a number of formal verification efforts. Those efforts concentrated on correctness proofs for refinement steps of the specification in various specification formalisms using different verification tools. In this paper we report on a Java Card implementation of the Mondex protocol and on proving its correctness using the KeY tool. The security properties to be proved are formalised in the Java Modelling Language and follow as closely as possible the concrete layer of the previous Z specification. This work demonstrates that with an appropriate specification language and verification tool, it is possible to bridge the gap between specification and implementation ensuring a fully verified result.*

## 1. Introduction

As a response to the growing dependence of almost every part of our private and professional life on the correct working of software and convinced that formal verification can contribute to greater safety and reliability, the Verified Software Grand Challenge was launched. First outlined by Tony Hoare in 2001 it has gathered considerable momentum in the mean time. An up-to-date account can be found in [13].

Mondex is an electronic purse hosted on a smart card. Originally, for its verification, formal specification on an abstract level and on one or two concrete levels were written and refinement theorems were proved using the Z specification language and hand-written proofs (details can be found in the original report [11]). The case study, based on this application, is still the most prominent pilot project in the Grand Challenge context. It was already verified by several

groups using different formalisms: the original Z specification verified with Z/Eves theorem prover, the Alloy specification language and model finder [8], RSL specification language and the theorem prover PVS and model checker SAL [4], and ASM (abstract state machines) and the KIV theorem prover [9].

All these efforts only address the specification level. We report in this paper on taking the last step from the concrete specification to Java Card code. The specification was transformed into annotations using the Java Modeling Language (JML) and program correctness was proved with the KeY prover. We are aware of the technical report [5] that also provides, in fact three, Java Card implementations of the Mondex case study. Its emphasis is however on extending the scope of the original case study by implementing various cryptographic protocols. It does neither involve specification nor verification.

The complete material – source code, technical report, proofs and binaries of the verification system – are available at [http://www.key-project.org/case\\_studies/mondex.html](http://www.key-project.org/case_studies/mondex.html).

**Outline of the paper** In Section 2 we explain our methodology in conducting the case study. We quickly review Java Card technology in Section 3, and the Mondex protocol in Section 4. Section 5 conveys a general idea of the implementation giving detailed comments on two methods. Section 6 is the central part of this contribution and presents the various parts of the specification. In Section 7 we report on the verification efforts and we wrap up the paper with conclusions in Section 8.

**Acknowledgments** The authors would like to thank the Formal Methods Europe (FME) for partly funding this research, and the anonymous referees for their helpful comments which improved the quality of this paper.

## 2. Methodology

In this case study JML was used as a specification language, because of its popularity in the Java Card formal methods community. The *Java Modeling Language* (JML) is designed as a DBC (design by contract) style specification language for Java. The development of JML is a cooperative effort of several researchers, coordinated by Gary T. Leaven's group at Iowa State University. The group also distributes a set of tools for using JML, including a syntax checker and a JML-enabled Java compiler [3].

The operations from the original  $Z$  specification are implemented as Java methods, the abstract  $Z$  domains are mapped to Java data types. The original  $Z$  specification is translated into JML contracts. We found the papers [8] and [9] very helpful in understanding the original specification in [11]. Only the part of the protocol that resides on the Mondex card, which is by far the greatest portion, has been implemented. We completely verify the non trivial security properties of the original case study. The assumptions on the host application and the communication network that are needed for the proofs are clearly stated.

In this case study greater emphasis was placed on writing realistic Java Card code than on following literally the original  $Z$  specification.

For program verification the KeY tool[1] was used. It was jointly developed at Karlsruhe University (Germany), Chalmers University (Gothenburg, Sweden) and the University of Koblenz (Germany). The KeY system is not a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY system is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.

## 3. Java Card Technology

To help the reader understanding the specification and implementation in the rest of this paper we need to say a few words on Java Card technology. The typical architecture is shown in Fig. 1 consisting of a back-end application, an off-card host application, the Card Acceptance Device (CAD), e.g. a card reader, and the on-card applets. The CAD first of all provides power to the card and forwards Application Protocol Data Unit (APDU) commands from the host application to the card and APDU responses in the reverse direction. The Java Card Runtime Environment (JCRC) receives APDUs and forwards them to the identified applet. A Java Card applet extends the Applet class from the API and must implement the `install()`, `select()`, `process()`, and `deselect()` methods. An applet's

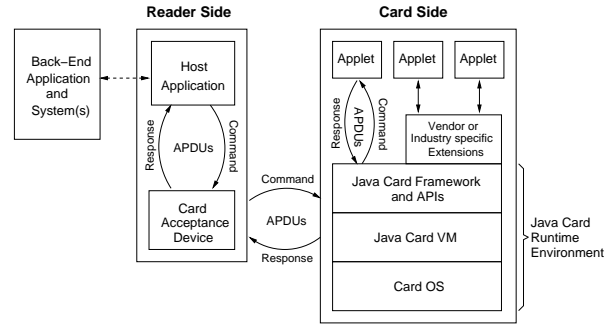


Figure 1. Architecture of a Java Card Application

life cycle begins when it is downloaded to the card and the JCRC invokes its (static) `Applet.install()` method. The applet registers itself with the JCRC by calling the `Applet.register()` method. The applet remains inactive in its unselected state until it gets selected by the host application. To notify the applet that this has happened the JCRC calls its `select()` method, which typically performs appropriate initializations. Once the selection is done, the JCRC passes on any incoming APDU by calling the applet's `process()` method with APDU as a parameter. Java Card objects may be persistent or transient. In this paper only persistent objects occur. The JCRC handles the exceptions not caught by the applet itself and supports atomic transactions for persistent objects. When after power loss, caused e.g., by card tear or some mechanical or electrical failure, power is re-applied to the card and the JCRC ensures that the transactions in progress when power was lost are aborted and persistent objects are rolled back to their previous state. In this paper we will be concerned only with the applet code and on a few occasions with actions of the JCRC.

## 4. The Mondex Protocol

The Mondex protocol allows to transfer electronic cash securely from one purse, called the fromPurse to another, called the toPurse, with as little centralised control as possible. In this paper we start from what is called the concrete model in the  $Z$  specification. The details of the Mondex protocol have been described in the literature quoted above, notably in [11, 9]. To make this paper more self-contained we summarise the gist of it. This will also give us the opportunity to introduce our perspective on it.

In the concrete model a card may be in one of the states `Idle`, `Epr`, `Epv`, `Epa`, `Endf`, `Endt`, may execute one of the operations `StartFrom`, `StartTo`, `Req`, `Val`, `Ack` and the messages `req`, `val`, and `ack` are exchanged as summarized in Fig.2. The states `eaFrom` and `eaTo` from

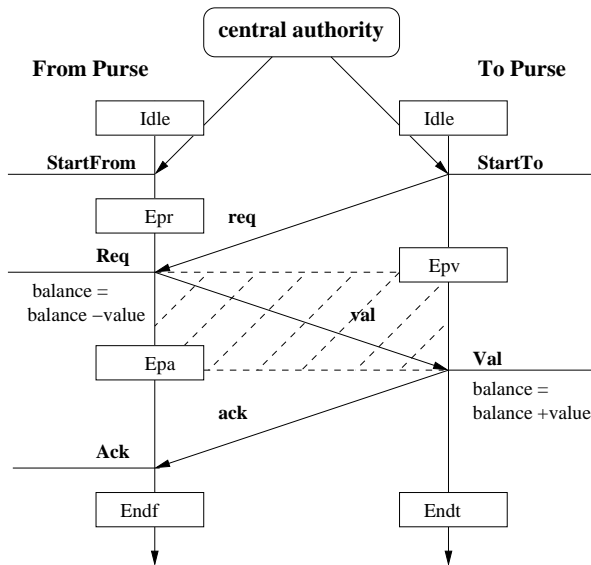


Figure 2. Modified Protocol Definition

the **Z** specification have been merged into one state `Idle` as has already been done in [9]. In addition we introduce the new states `Endf` and `Endt` instead of reusing the `Idle`. We found this more realistic, since the state `Idle` will only be established in the initialization phase for the next transaction by the JCRE calling the `select()` method.

Each purse holds information on its balance, its status, details of the current transaction, the next sequence number `nextSeq` to be used, and a log of aborted transactions `exLog`.

The operation to be performed by a purse depends on the message received in the form of an APDU via the JCRE and on the status of the purse. Messages not processed (erroneous or not valid for the status) are simply ignored by the purse.

**StartFrom** A purse receiving this message in state `Idle` will be the `fromPurse` in the beginning transaction. This message contains the transaction counter part details, i.e., the data identifying the `toPurse`, its message sequence number and the value to be transferred. The `fromPurse` executes its `start_from_operation` which checks whether it has sufficient funds to pay the amount. If that is the case it stores the transaction details, increases its next sequence number and moves to the state `Epr`.

**StartTo** A purse receiving this message in state `Idle` will be the `toPurse` in the beginning transaction. This message contains the transaction counter part details, i.e., the data identifying the `fromPurse`, its message sequence number and the value to be transferred. The `toPurse` executes its `start_to_operation` which stores the

transaction details, increases its next sequence number, sends a `req` message, and moves to the state `Epv`.

**Req** If the `fromPurse` receives a correct `req` message in state `Epr` it executes its `req_operation` which decreases its own balance by the transaction value, send a `val` message and moves to state `Epa`.

**Val** If the `toPurse` receives a correct `val` message in state `Epv` it executes its `val_operation` which increases its own balance by the transaction value, sends an `ack` message, and moves to state `Endt`.

**Ack** If the `fromPurse` receives a correct `ack` message in state `Epa` it executes its `ack_operation` which changes its state to `Endf`.

The Mondex case study uses the concept of an *ether* to abstract away from all details on how messages are send, received or encrypted. A purse places outgoing messages into the ether, and takes incoming messages from it. The ether may lose messages and adversaries may inject messages into it. It is however assumed that the messages cannot be forged. That is to say if e.g., the `fromPurse` receives a request messages that matches its current transaction this message must have been send by the `toPurse`. Furthermore there is no explanation in the original specification how the initial messages that trigger the `StartFrom` and `StartTo` operations are coordinated. Some centralized entity that knows the sequence numbers of both the `fromPurse` and the `toPurse` is needed for this.

It is assumed that a transaction can be interrupted at any time. Money therefore can be lost in case the `fromPurse` has already decreased its balance and the `toPurse` has not yet increased its balance. The region in the protocol where money might be lost is shown highlighted with dashed lines in Figure 2. To account for lost money the payment details are logged if necessary, i.e., if interruption occurs in states `Epa` or `Epv`. However money is considered lost only when both purses log the same transaction. The decision on this is performed by an external entity. The logged transactions of each purse are transferred to an archive where they can then be compared. Three extra operations are necessary for this. They do not refer to any counter part purse and do not cause any change in the purse's status or balance

**Abort** for bringing an interrupted purse back to a valid initial state. It checks whether a transaction has to be logged and sets the purse's status to the initial `Idle`.

**readExLog** for copying the logged transactions (the *Exceptional Log*) to the archive.

**clearExLog** for deleting the logged transactions from the purse's exceptional log. To enable this operation, the purse has to receive a valid *clear code* (a numerical

value generated injectively from a set of logs stored in the archive provided by an external source).

## 5. Implementation

In the implementation, the abstract domains from the **Z** specification had to be replaced:

1. natural numbers to Java Card data types,
2. sets of payment details to arrays `PayDetails[]` plus a pointer `logIdx` to the position of the next insertion,
3. messages to APDUs.

A purse is implemented through the Java class `ConPurseJC` which extends the class `Applet` from the Java Card API with the following fields (the same defined in the **Z** specification)

```
public class ConPurseJC extends Applet
{
    private short balance;
    private PayDetails [] exLog;
    private short name;
    private short nextSeq;
    private byte status;
    private byte logIdx;
    private PayDetails transaction;
    ...
}
```

We only need one more class `PayDetails` with the fields in one-to-one correspondence with those from the original **Z** specification:

```
public class PayDetails
{
    short fromName;
    short toName;
    short value;
    short fromSeq;
    short toSeq;
    ...
}
```

In this paper we will only convey the general flavor of the implementation by reproducing two methods. The full source code is contained in the case study report [12, Appendix A].

We first look at the abstract method `process()` from the `Applet` class which every applet must implement, Fig. 3. For type setting reasons we wrote **I** instead of **ISO7816**.

Line 2: After the applet is successfully selected, the JCRE dispatches incoming APDUs to the `process` method. At this point, only the first 5 bytes [CLA, INS, P1, P2, LC] are available in the APDU buffer.

```
public void process(APDU apdu) 1
{byte [] buffer = apdu.getBuffer(); 2
  if ((buffer[I.OFFSET_CLA] == 0) && 3
      (buffer[I.OFFSET_INS] == 4
        (short)(0xA4))) 5
    ISOException.throwIt(SW_IGNORED); 6
  if (buffer[I.OFFSET_CLA] != Mondex_CLA) 7
    ISOException.throwIt 8
      (I.SW_CLA_NOT_SUPPORTED); 9
  if (Util.getShort(buffer, I.OFFSET_P1) 10
      != name) 11
    ISOException.throwIt(SW_IGNORED); 12
  switch (buffer[I.OFFSET_INS]) 13
  {case StartFrom: 14
    start_from_operation(apdu); break; 15
   case StartTo: 16
    start_to_operation(apdu); break; 17
   case Req: req_operation(apdu); break; 18
   case Val: val_operation(apdu); break; 19
   case Ack: ack_operation(apdu); break; 20
   case ReadExLog: 21
    read_ex_log_operation(apdu); break; 22
   case ClearExLog: 23
    clear_ex_log_operation(apdu); break; 24
   default: ISOException.throwIt 25
             (I.SW_INS_NOT_SUPPORTED); 26
  } } 27
```

Figure 3. The `process` method code

Lines 7–9: Check whether the APDU addresses the Mondex card applet.

Lines 10–12: Check whether the APDU addresses this purse.

Lines 13–26: Calls the method indicated by the instruction byte INS.

As a second example we consider the code for the `val_operation()`, Fig. 4. We skip the details of the method `checkSameTransaction(apdu)` called in line 4. It basically checks whether the data of the transaction counter part contained in the APDU coincide with that stored in the purse's `transaction` field. In lines 5 – 9 updating balance and state is grouped into one atomic step using Java Cards transaction. If `!status == Epv` (line 3) `...throwIt(SW_IGNORED)` (line 15) is executed. This does not throw an exception as Java's `throw` command does. Rather a response APDU with the indicated process interrupt code will be sent. The same mechanism is used in lines 12 + 13 to catch exceptions thrown in the `try` block in lines 5 – 10. The process interrupt codes `SW_IGNORED`, `SW_TRANSACTION_ERROR` etc., are defined in the applet. They are passed on by the JCRE to the

```

1 private void val_operation(APDU apdu)
2     throws IOException
3 {if (status == Epv) {
4     checkSameTransaction(apdu);
5     try{JCSystem.beginTransaction();
6         balance =
7         (short)(balance+transaction.value);
8         status = Endt;
9         JCSystem.commitTransaction();
10        }
11    catch (TransactionException e)
12        {IOException.throwIt
13            (SW_TRANSACTION_ERROR);
14        } }
15    else IOException.throwIt(SW_IGNORED);
16 }

```

**Figure 4. The val\_operation method code**

host application. In this case study we do not write code for the host application. It is enough for our purposes to assume that it reacts in a sensible manner, e.g., if SW\_IGNORED is received it is ignored. Thus, if the fromPurse receives a val message when it is not in state Epv we assume this message is ignored and the applet is ready to react to the next APDU. All these assumptions have no influence on the correctness proofs.

Sending of messages is also not modeled in our implementation since this can only be done by the host application and not by the applet. We assume e.g., that after receiving a successful response APDU from val\_operation the host application will send the ack message to the transaction counter part.

## 6. Specification

The project involves three rather different kinds of specifications which we will address separately in the next three subsections.

### 6.1. Protocol Specification

Let us call the specifications addressed in this subsection the protocol specification, just to have a name for them. If we can prove that the Java Card code satisfies the protocol specifications we are sure that it implements correctly the Z protocol from [11]. We will only go through the specifications for the two methods already encountered in Sect.5. For a complete account we refer to the case study report.

Let us first look at the JML contract for the val\_operation, Fig. 5. JML contracts are placed as special

```

/*@ public behavior
   @ requires apdu != null;
   @ assignable balance, status;
   @ ensures
   @ ((balance == \old(balance)
   @     +transaction.value) &&
   @ (\old(status)==Epv)
   @ && (status==Endt));
   @ signals_only IOException;
   @ signals (IOException e)
   @ (balance==\old(balance)
   @ && status == \old(status));
   @*/
private void val_operation(APDU apdu)

```

**Figure 5. The val\_operation method spec**

comments in front of the method they apply to. The precondition (line 2), identified by the key word *requires*, simply states that the argument should be a non-null object. The *assignable* clause in line 3 gives the expressions that may at most be changed by the method. In our example *balance* and *status* may change. According to the JML semantics this entails that for any object *o* with *!o==self* the fields *o.balance* and *o.status* will **not** change. The postcondition, lines 4 – 8, identified by the key word *ensures* states the conditions that must be true after normal termination of the method, while lines 10 – 12 state what must be true in the case of abrupt termination. Furthermore line 9 gives a restriction on the types of exceptions that are at most allowed to be thrown.

Line 5 – 6 and line 8 are postconditions that correspond to the definition of the val\_operation given informally in Sect.4. In JML the construct *\old(exp)* in a postcondition refers to the value of *exp* in the pre-state. Line 7 as part of the *ensures* clause for normal termination has the effect, that the val\_operation is required to terminate abruptly if called in a state different from Epv which is the behavior required by the original specification.

The contract for the process method is the longest in the case study and one of the most interesting. For a complete account we again refer to the case study report. Here we will show and comment on significant parts of it in Figures 6 to 8. Its only precondition (Fig.6, line 2) will be satisfied since the JCRE will not pass a null object. It is in general a good idea, even more so for the top level method, to state as little preconditions as possible. JML is tailored towards expressing contracts for individual methods. It was not meant to be used to express constraints on the temporal order of method execution, as can e.g., be done easily with finite state automata or state charts. Yet, there is already one paper [6] that put JML to use to specify a bi-

```

1  /*@ public behavior
2  @ requires apdu != null;
3  @ assignable ...
4  @ ensures
5  @ ((\old(logIdx) != logIdx) ==>
6  @   ((logIdx==0) &&
7  @   (status==Idle) &&
8  @   (\old(status)==Idle)))
9  @ &&
10 @ ((\old(status)==status) ==>
11 @   (\old(balance)==balance) &&
12 @   (\old(nextSeq)==nextSeq))
13 @ &&

```

**Figure 6.** process method spec, part 1

lateral key exchange protocol. The method of choice was to express the protocol through explicit postconditions of the main method. We do the same in lines 4 – 82: if the process method satisfies its contract, then we know that it faithfully implements the protocol. It can be observed that the postconditions do not simply describe the transitions from Fig.2 they also relate these transitions to the change of variables. The exact form of the contract is arrived at by iterating through failed attempts to derive the safety properties from the postcondition and strengthening the contract. Lines 10 – 12 address the operations that do not cause a change in status while the bulk of the contract, line 14 – 82, deals with operations that change the status field.

```

13 @ &&
14 @ ((\old(status) != status) ==>
15 @
16 @ \old(apdu._buffer[I.OFFSET_INS])
17 @ == apdu._buffer[I.OFFSET_INS]
38 @ && (\old(status)==Epa ==>
39 @   (status==Endf &&
40 @   apdu._buffer[I.OFFSET_INS]==Ack
41 @   && balance==\old(balance)))

```

**Figure 7.** process method spec, part 2

In Fig. 7 lines 38 – 41 specify the postulated action of the Mondex protocol if a purse is in state Epa and receives an ack message. The message type is given by the INS field of the APDU and since the specification occurs as a postcondition we need to add lines 16 – 17 requiring that the process method does not change this field. The specification contains similar parts, not shown here, for all the other states. But, this is only half of the story. The post-

conditions should be strong enough to make sure that the code does not implement more actions than specified by the protocol. This is done by the fragment of the specification shown in Fig. 8.

```

@ (status != Idle) 43
@ && (status==Epr ==> 44
@   \old(status) == Idle) 45
@ && (status==Epv ==> 46
@   \old(status) == Idle) 47
@ && (status==Epa ==> 48
@   \old(status) == Epr) 49
@ && (status==Endf ==> 50
@   \old(status) == Epa) 51
@ && (status==Endt ==> 52
@   \old(status) == Epv) 53

```

**Figure 8.** process method spec, part 3

The specifications so far only concerned postconditions required to be true after normal termination of the process method. In Fig. 9 in line 82 the JML keyword `signals_only` is used to express that only ISOException may occur and lines 83 – 87 states the conditions that have to be satisfied should an exception be thrown.

```

@ signals_only ISOException; 82
@ signals (ISOException e) ( 83
@   \old(balance)==balance && 84
@   \old(status)==status && 85
@   \old(logIdx)==logIdx && 86
@   \old(nextSeq) == nextSeq); 87
@*/ 88
public void process(APDU apdu) 89

```

**Figure 9.** process method spec, part 4

It remains to comment on the invariants for the class ConPurseJC, see Fig. 10. These mainly state that program variable stay within their legal ranges. But, it also contains the *Sufficient Funds Property* from [11], lines 10 – 11. On the implementation level there is a parallel property, that states that adding the transferred value to the balance of the toPurse does not cause overflow, lines 12 – 14. Finally lines 15 – 17 show an example of a JML expression involving quantification.

## 6.2. Security Specification

The original specification states the following top level security properties:

```

1 public class ConPurseJC extends Applet
2 {
3   /*@ public invariant
4     @ (exLog != null) &&
5     @ (exLog.length > 0) &&
6     @ ...
7     @ (balance >= 0) &&
8     @ (balance <= ShortMaxValue) &&
9     @ ...
10    @ ((status == Epr) ==>
11      @ (transaction.value <= balance)) &&
12    @ ((status == Epv) ==>
13      @ (transaction.value <=
14        @ (ShortMaxValue - balance))) &&
15    @ (\forall byte i;
16      @ i >= 0 && i < exLog.length;
17      @ exLog[i] != null);
18    @*/
19 ... }

```

**Figure 10. Invariants for class ConPurseJC**

**Security Property 1 No value creation:** no value may be created in the system. The sum of all purses' balance does not increase.

**Security Property 2.1 All value accounted:** all values must be accounted in the system. The sum of all purses' balance and lost components does not change.

**Security Property 2.2 Exception Logging:** if a purse aborts a transfer at a point where value could be lost, then the purse logs the details.

**Security Property 3 Authentic purses:** a transfer can only occur between authentic purses.

**Security Property 4 Sufficient Funds:** a transfer can occur only if there are sufficient funds in the from purse.

Property 4 has already been taken care of. We will comment here on Properties 2.1 and 2.2. Details on the proof of Property 1, which can be viewed as a special case of Property 2, and Property 3 are again contained in the case study report.

As a preparatory step we define the helper function `bookedValue` for every instance `o` of the class `ConPurseJC`:

$$o.\text{bookedValue}() = \begin{cases} -o.\text{transaction.value} & \text{if } (o.\text{status} == \text{Epa}) \text{ or } (o.\text{status} == \text{Endf}) \\ +o.\text{transaction.value} & \text{if } o.\text{status} == \text{Endt} \\ 0 & \text{otherwise} \end{cases}$$

and prove that the following JML constraint for the class `ConPurseJC`:

```

/*@ public constraint
  @ ((\old(balance) != balance) ==>
  @ ((balance - \old(balance))
  @ ==bookedValue()));
  @*/

```

This means it has to be proved that the above implication is true as a postcondition for every methods in `ConPurseJC`.

The remaining machine checked arguments in this subsection are completely mathematical, and that is the way we will present it here. In the case study report, a different approach is taken: the security properties were specified in JML as postconditions of a virtual method `showProperties()` with empty body.

In the original specification the *world* of all purses and the total sum of all balances and losses are considered. Since by the `assignable` clause of method `process()` the assumptions of the argument in [11, Sect.2.4] apply we may restrict attention to one purse `o` and its counterpart `x`.

If a transaction is interrupted, by card tear, physical failure or initiated by the host application receiving an `ISOException` first nothing happens and the state of the card will remain as it was at the time of failure. Only when a new transaction with the card is started by calling the `select` method logging will happen, see Fig. 11. If on the other

```

public boolean select() {
  abort_if_necessary();
  return true;
}
private void abort_if_necessary()
  throws ISOException{
  if (!(status == Epv) || (status == Epa))
    status = Idle;
  else if (logIdx >= exLog.length)
    ISOException.throwIt(SW_LOG_FULL);
  else { try {
    exLog[logIdx] = transaction;
    JCSystem.beginTransaction();
    logIdx++;
    status = Idle;
    JCSystem.commitTransaction();}
  catch (TransactionException e)
    {ISOException.throwIt
      (SW_TRANSACTION_ERROR);
    }}}

```

**Figure 11. Implementation of the `select` method**

hand  $o$  reaches normally `Endf`, we know, since the postconditions on Figures 7 and 8 are true that this can only happen if it received an `ack` message. By the assumptions of the  $Z$  specification on the ether `ack` messages cannot be forged. So an `ack` message must have been send by  $x$ , or more precisely by the host application handling  $x$ . The host application is not implemented in our case study. But, the case study report clearly states the properties of the host application that are used in the proofs. Here we need, that the host application only sends an `ack` message if it receives a response APDU from  $x$  that signals successful completion of the method `val_operation` leaving  $x$  in state `Endt`. If we instantiate `o.status == Endf` and `x.status == Endt` in the above constraint we obtain:

```
o.balance - o.\old(balance) ==
-o.transaction.value
x.balance - x.\old(balance) ==
+x.transaction.value
```

Since  $x$  is the transaction counter part of  $o$  we furthermore have `o.transaction.value == x.transaction.value` and

```
o.balance + x.balance ==
o.\old(balance) + x.\old(balance)
```

follows. This reasoning was not supported by an automatic prover tool.

In the case study report a much more general implication is shown, see Fig. 12. Here,  $(A \text{ ? } t : s)$  is a conditional term that evaluates to  $t$  if  $A$  is true and to  $s$  otherwise. Let us say that a purse is in its *critical section* if it is in state `Epa` or `Epv` (the shaded area in Fig. 2). Then the conditional term, call it `loss`, in the last 3 lines of Fig. 12 evaluates to

```
transaction.value   if both purses, o and x
                    are in their critical region
0                   otherwise
```

Using the invariant on `bookedValue()` the last four lines of Fig. 12 may thus be summarised as

```
o.balance - o.\old(balance) +
x.balance - x.\old(balance)
+ loss = 0
```

The implication in Fig. 12 is proved using the KeY prover. The justifications for the premises of the implication are done as above by referring to the postconditions of `process`, to the assumptions on the ether, and to the assumptions on the host application. In a real application the determination that money has been lost is performed by an external entity examining the archive of the exception logs of all purses. This is beyond the implementation considered here. What we can guarantee is that logging has been done if necessary, which happens in the call to the `select()` method.

```
(status != Idle) ==>
(\exists ConPurseJC x;
 x!= null
 && x.transaction==transaction
 && x.name != name
 && (status==Endf==>
 x.status==Endt)
 && (status==Endt ==>
 (x.status==Epa
 || x.status==Endf))
 && (status==Epa ==>
 (x.status==Epv
 || x.status==Endt))
 && (status==Epv ==>
 (x.status==Idle || x.status==Epr
 || x.status==Epa))
 && (status==Epr ==>
 (x.status==Idle
 || x.status==Epv)))
==>
((status == Idle) ||
(\exists ConPurseJC x;
 x!= null
 && x.transaction==transaction
 && x.name != name
 && ((bookedValue()>0) ==>
 (x.bookedValue()<0))
 && ((x.bookedValue()>0) ==>
 (bookedValue()<0))
 && (bookedValue()+ x.bookedValue()+
 (((status==Epa || status==Epv)
 && (x.status==Epa || x.status==Epv))
 ? transaction.value : 0) ==0)))
```

**Figure 12. Proof obligation for Property 2**

### 6.3. Other Requirements

Apart from those in the  $Z$  specification there are other security requirements to be considered that apply to Java Card programs in general. A comprehensive set has been developed within the *SecSafe* project and published in [7], see also [1, Chapter 14].

- ([7, Sect.3.4]) To avoid leaking the information about error conditions inside the applet a well written Java Card applet should only allow exceptions of the type `ISOException` to reach the top level of the applet. This property was formally proved to hold.
- ([7, Sect.3.4]) Transactions should be well formed: no transaction is started before aborting or committing a previous one; no transactions is aborted or committed

without have being started; and no open transaction is left for the JCRE to close. This property was not proven but due to the code size, one can easily be convinced that it holds.

- ([7, Sect.3.5]) All updates relevant for preserving the invariants are performed within a transaction mechanism, resulting in a *tear safe* implementation. The strong preservation of the invariants was proved using the *throughout* configuration option of the KeY system.
- ([7, Sect.3.6]) Integer operations should not overflow.

The last item touches on the general issue of data refinement in program development methodologies. The usual procedure is to refine an abstract data type  $A$  by a concrete data type  $C$  in such a way that all behavior of  $A$  is preserved by  $C$ . In the particular case at hand the mathematical, infinite, integers from  $\mathbf{Z}$  are replaced by Java integer types. This is not a data refinement step and we cannot hope to prove a general theorem on how statements true in the  $\mathbf{Z}$  semantics remain true in the Java semantics. Instead we verify locally that the statements that actually occur in the specification are also true when Java semantics is used. The KeY system offers three configuration options for integer semantics (1) the mathematical semantics, which is of course unsound, (2) mathematical semantics plus showing the absence of overflow, which leads to correct verification results, but will miss to proof programs correct that deliberately use overflow, (3) the Java semantics. We used option (2). For an in-depth discussion of this issue we refer to [2].

## 7. Verification

There are two types of proof obligations that are generated automatically by the KeY system from the JML annotated Java code. Behaviour verification for every method  $m$  is the task to show that starting in any state where the preconditions and the invariants are true after symbolic execution of  $m$  the postconditions are true. There are two types of proof obligations that are generated automatically by the KeY system from the JML annotated Java code. Behavior verification of a method  $m$ , the first type, is the task to show that starting in any state where the preconditions and the invariants are true, after symbolic execution of  $m$  the postconditions are true. Proof statistics are given in Table 1. The verification of the contract for `showProperties` does not directly depend on any code, as explained above. The behavioral verification of `process` does depend on its code. It has been put in the compartment of implementation independent proofs in Table 1 because it does not symbolically execute the code of the methods it calls, but uses for better modularity the contracts of these methods.

The second type of proof obligations is the verification of invariants and constraints. For a method  $m$  it has to be

Method	Nodes	Branches	Time (min)
USING CONTRACTS			
process	4,731	54	10
showProperties	6,565	50	10
USING IMPLEMENTATION			
startFrom	3,818	102	5
startTo	3,975	105	5
req	3,482	95	5
val	3,525	91	5
ack	2,370	69	5
clear_ex_log	1,352	37	5
read_ex_log	28,292	490	35
abort_if_necessary	2,427	57	5

**Table 1. Methods' Behavior Verification**

Method	Nodes	Branches	Time (min)
STRONG INVARIANT			
startFrom	19,084	44	10
startTo	19,015	40	10
req	23,165	64	15
val	18,689	51	15
ack	14,199	32	10
clear_ex_log	7,588	18	5
abort_if_necessary	8,761	33	5

**Table 2. Invariant Preservation Verification**

proved that, in any state satisfying all invariants and the method's precondition, after symbolic execution of  $m$  again all invariants and all constraints are true.

However, in this case study *strong invariant preservation* proof obligations were used: rather than just showing that invariants hold before and after method calls, it was shown that they hold after each computation step. This is a much stronger property to prove but leads to tear safe applications. Proof statistics are shown in Table 2.

To give an example of the effect of strong invariants we consider the following alternative implementation of the `abort_if_necessary` method:

```
private void abort_if_necessary ()
    throws IOException
{ if (!(status == Epv) || (status == Epa))
    status = Idle;
  else if (logIdx >= exLog.length)
    IOException.throwIt(SW_LOG_FULL);
  else {
    exLog[logIdx] = transaction;
    logIdx++;
  }
  - - - - -
```

```

    status = Idle ;
  }}

```

It satisfies its contract and preserves all invariants, but it fails to strongly preserve the following invariant if abortion occurs at the dashed line

```
((\ old ( logIdx ) != logIdx ) ==> ( status == Idle ))
```

This means this implementation is not tear safe.

The KeY system could perform the proof steps almost automatically. However, since the prover's strategies may lead to non-optimal proofs a *mixed* approach was taken. Some initial step where done interactively to put the prover on the right track, then automatic proof search was activated and user interaction only resumed to help with quantifier instantiations.

## 8. Conclusions

Verifying an implementation of the Mondex protocol is well within the reach of present verification technology. The special approach used in the case study reported in this paper using JML as a specification language and an interactive theorem prover based on dynamic logic, the KeY system, were adequate for the task at hand. This complements the positive results that have so far been obtained by other researchers with formal correctness proofs of specification refinements. In addition, fully specifying the case study using JML facilitates its understanding by non-formal methods users.

It should be observed that for the proof of the security properties only an implementation of the main method, `process()`, is required. At this stage only the contracts of the other methods are required. This set up allows to find flaws and inconsistencies in the specification early on before implementing everything.

We could also identify weak points in the overall verification process of this case study. This applies in particular to the question how the operations in the specification model will be deployed on the different parts of an application. One operation on the model level could in fact be realized as the combined effect of several operations on the code level. In the Mondex case study e.g., correct logging is achieved by the interplay of the Java Card applet, the JCRE and the host application. Another issue is the switch from abstract data types to the data types in a programming language. This is not a refinement step. Solution to the arising problems are known in many cases. As an example we mention that the restriction to finitely many sequence numbers reduces an absolute guarantee to a probabilistic one.

In future research we will concentrate on a systematic and seamless integration of specification models, be it in **Z**, Alloy or ASM, and specifications on the code level.

## References

- [1] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [2] B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, LNCS 2999, pages 207–226. Springer, 2004.
- [3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proc. Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.
- [4] C. George and A. E. Haxthausen. Specification, proof, and model checking of the Mondex electronic purse using RAISE. Technical Report 352, United Nations University, IIST, Macau., February 2007.
- [5] H. Grundy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, and W. Reif. The Mondex challenge: From specifications to code. Technical Report 2006-31, Universität Augsburg, Institut für Informatik, D-86135 Augsburg, 2006.
- [6] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Proceedings of the 1st International Conference on Security in Pervasive Computing*, volume 2802 of LNCS, pages 213–226. Springer-Verlag, 2004.
- [7] R. Marlet and D. L. Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
- [8] T. Ramananandro. Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Technical report, École Normale Supérieure, Paris, France, September 2006.
- [9] G. Schellhorn, H. Grundy, D. Haneberg, and W. Reif. The Mondex challenge: Machine checked proofs for an electronic purse. TR 2006-02, Universität Augsburg, Institut für Informatik, D-86135 Augsburg, Feb 2006.
- [10] P. Schmitt, I. Tonin, C. Wonnemann, E. Jenn, S. Leriche, and J. J. Hunt. A case study of specification and verification using JML in an avionics application. In *JTRES '06: Proc. 4th international workshop on Java technologies for real-time and embedded systems*, pages 107–116, New York, NY, USA, 2006. ACM Press.
- [11] S. Stepney, D. Cooper, and J. Woodcock. *An Electronic Purse — Specification, Refinement, and Proof*, volume RPG-126 of *Technical Monograph*. Oxford University Computing Laboratory, July 2000.
- [12] I. Tonin. Verifying the Mondex case study. The KeY approach. Interner Bericht 2007-4, Department of Computer Science, Universität Karlsruhe, Karlsruhe, Germany, April 2007.
- [13] J. Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, October 2006.