OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

The Mondex case study: from specifications to code

Holger Grandy, Nina Moebius, Markus Bischof, Dominik Haneberg, Gerhard Schellhorn, Kurt Stenzel, Wolfgang Reif

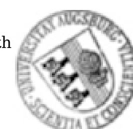
Angaben zur Veröffentlichung / Publication details:

Grandy, Holger, Nina Moebius, Markus Bischof, Dominik Haneberg, Gerhard Schellhorn, Kurt Stenzel, and Wolfgang Reif. 2006. "The Mondex case study: from specifications to code." Augsburg: Universität Augsburg.

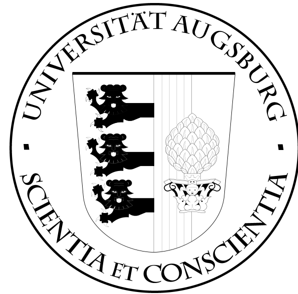
Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under th conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/



Universität Augsburg



The Mondex Case Study: From Specifications to Code

H. Grandy, N. Moebius, M. Bischof,
D. Haneberg, G. Schellhorn, K. Stenzel, W. Reif

Report 31 2006

Institut für Informatik
D-86135 Augsburg

Copyright © H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, W. Reif
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

The Mondex Case Study:
From Specifications to Code

H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, W. Reif
Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany

Abstract

In this paper we introduce three different implementations for the Mondex electronic purse verification challenge [Woo06] [SCW00]. In previous work ([SGHR06] [SGH+07] and [HSGR06]) we verified security and correctness properties of the Mondex money transfer protocol. Here we present a way to translate the formal specifications into running JavaCard code. We introduce three different ways to implement the protocol, one using symmetric cryptography, one using asymmetric cryptography and finally one using special datatypes for cryptographic protocols and symmetric cryptography. All implementations presented in this paper are able to run on a Gemplus GemxpressoRAD ProR3 SmartCard.

Chapter 1

Introduction

Mondex [MCI] cards are smart cards which can be used as electronic purses. They store money and can be used to transfer money from one card to another.

The Mondex case study recently received a lot of attention because its formal verification is seen as a challenge for verification tools [Woo06]. The original specification [SCW00] made in Z [Spi92], the proofs were made by hand with an enourmous effort. In [SGHR06] we have shown that this verification can now be done in a few weeks using the elaborated verification support of the interactive theorem prover KIV [KIVb]. In [SGH$^+$07] (resp. the corresponding technical report [SGH$^+$06]) we describe a weakness of the communication protocol of Mondex which allows a denial of service attack against the cards. There we also show that a slight modification of the protocol corrects this error and verify the modified protocol. Furthermore, the original Mondex specification does not contain any formalization of cryptographic operations. Instead it simply assumes that certain (critical) protocol messages cannot be forged. In [HSGR06] we extend the abstract protocol of Mondex with cryptographic operations and verify that the new protocol is actually a refinement of the original abstract protocol.

All our specifications of the Mondex case study are given as Abstract State Machines [Gur95] [BS03] and all refinements use the ASM Refinement Theory [BR95] [Sch01] [Bör03] [Sch05]. Figure 1.1 shows an overview of our specification levels for the case study.

The logical consequence of this development approach for Mondex is the refinement of the abstract specifications down to running source code (the last level in Fig. 1.1).
We chose the JavaCard programming language and Java SmartCards as a platform. In this paper we present three different ways to implement the abstract protocol of Mondex.

The first version implements the protocol using symmetric cryptography. This approach follows the specifications given in the communication protocol level (the concrete level of the original Mondex work) in Figure 1.1 very closely. It is introduced in Section 2.

Using only symmetric cryptography has some weaknesses. For example, all security properties rely on the fact that the secret key is unknown to an attacker. This means if one card is compromised then all cards would be compromised. Therefore, our second implementation uses asymmetric cryptography to address those weaknesses. This implementation also follows the communication protocol level of Fig. 1.1, but has some little modifications. The reason is that in reality message lengths for smart cards are bounded, so you cannot send very long input messages (such as signatures) together with some other data. This will be explained in Section 3.

Finally, our last implementation for Mondex uses symmetric cryptography and special protocol data types. It is the result of the consequent development using stepwise refinement and therefore follows the specification of the security protocol level very closely. This implementation follows our general approach for security protocol code refinement as described in [GHRS06] [GSR06b]. The corresponding code will be explained in Section 4.

This paper is a follow-up for [SGHR06], [SGH$^+$07] and [HSGR06]. We will not repeat all the details about the specification levels "transactions", "communication protocol" and "security protocol" here. Please refer to these papers for more information about those levels.

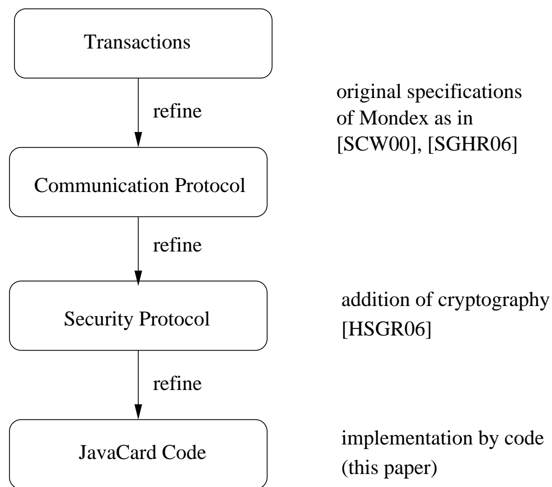


Figure 1.1: Overall structure of our Mondex refinement approach

Chapter 2

A Protocol using symmetric Cryptography

2.1 Implementation

The basis for the following implementation is the specification of the Mondex communication protocol level as shown in Fig. 1.1.

On the communication protocol level transferring money is done using a protocol with 5 steps (for details see also [SGHR06]). To execute the protocol each purse needs a status that indicates how far it has progressed in the execution of the protocol steps. The possible states a purse may be in are given by the enumeration status = idle | epr | epv | epa. Purses not participating in any transfer are in the idle state. To avoid replay attacks each purse stores a sequence number nextSeqNo that is used in the next transaction. This number is incremented at the start of every protocol run. The current amount of money stored on the card is given by the field balance. During the run of the protocol each purse stores the current payment details (paydetails). These are tuples consisting of the names of the from and to purse, the sequence numbers they use and the amount of money that is transferred. The state of a purse also contains a log exLog of failed transfers represented by their payment details.

This state is now implemented in an applet class for a Java Smart Card. Our class implementing the communication protocol is called *Purse*. Its definition and its fields (representing the ASM state functions of [SGHR06]) is given as:

```
public class Purse extends Applet {
  ...
  // protocol fields
  private short          balance;
  private byte[]         name;
  private short          seqNo;
  private byte           state;
  // pay details and exception log
  private byte[]         currentPD;
  private byte[]         exlog;
  private short          currentExLogIndex;
  // key
  private DESKey         sharedkeyobj;
  ...
}
```

The actual money *balance* of the card is represented as a *short* value (our JavaCards do not support integers). The name is given as a byte array of 8 bytes. The sequence number (*seqNo*) is also a *short* value. Payment details (*currentPD)* are given as a byte array containing the same information as the abstract data type paydetails above. The *currentPD* array is 22 bytes long: 2 * 8

bytes are used to store the two card names (from and to), 2 * 2 bytes are used for the two sequence numbers (fromno and tono) and 2 bytes are used for the amount of money to be transferred (value). The exception log (exlog) which is a list of paydetails in the specification is implemented as a byte array of length 5 * 22. This is sufficient to store 5 PayDetails one after another. The next free index in this exception log array is given by the field *currentExLogIndex*. Additionally, we have to consider encryption. The ASM communication protocol level does not contain any cryptographic operations. Messages are simply considered to be unforgeable. Of course, when giving an implementation, cryptography has to be used. The cryptographic protocol for the symmetric version of Mondex written in a commonly used standard notation for cryptographic protocols [Car94] is:

1. terminal → from : STARTFROM, seqNo(to), name(to), amount
2. from → to : $\{$STARTTO, seqNo(from), name(from), amount$\}_{K_S}$
3. to → from : $\{$REQ,paydetails$\}_{K_S}$
4. from → to : $\{$VAL,paydetails$\}_{K_S}$
5. to → from : $\{$ACK,paydetails$\}_{K_S}$

The secret key K_S is shared between all Mondex cards. In our implementation, this key is given by the field *sharedkeyobj* as an object of type DESKey (representing a 3DES symmetric key). An overview of the protocol on the concrete level is shown in Fig. 2.1.

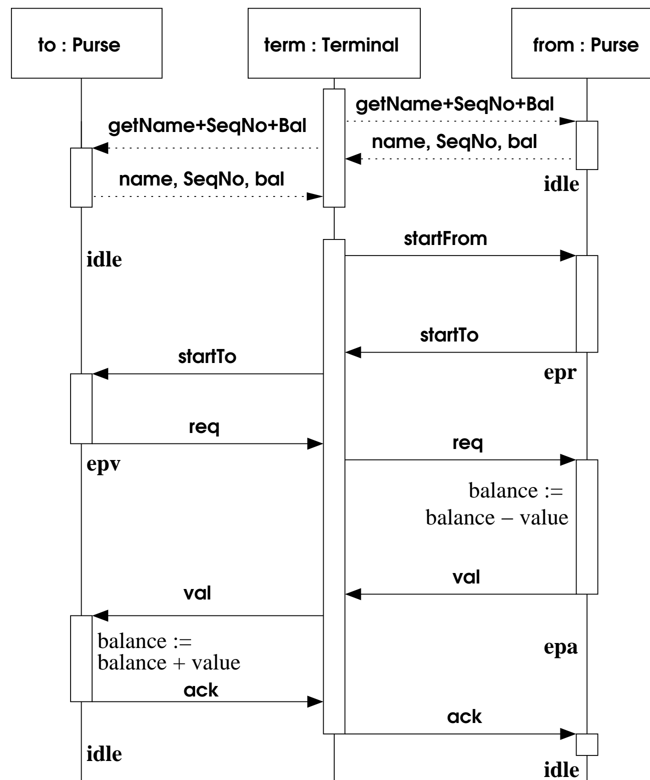


Figure 2.1: An overview of the Mondex protocol

The first message getName+SeqNo+Bal (shown by dashed lines in Fig. 2.1) is needed in a real implementation of the Mondex scenario, since the terminal must be able to get the information about card names, their sequence numbers and balances. This information is used in the next protocol step startFrom.

When using Java SmartCards, communication is based on byte arrays. Smart cards communicate using Application Protocol Data Units (APDU). APDUs are byte arrays which consist of

a header and a data part. The header is 5 bytes long and contains an instruction byte and length information. Each message sent to a smart card will be transformed by the JCRE (the operating system of the smart card) in a parameter for the *process(APDU)* method defined in the applet class implementing the protocol. The message format for our symmetric implementation is given in the following table (header information is omitted. Only the instruction byte is given as the first part of each message):

Protocol Step	**Input**	**Output**
GetNameSeqNoBal	GNS	name + seqNo + balance
StartFrom	SF + name(to) + seqNo(to) + amount	ST + { ST + name(from) + seqNo(from) + amount}$_{\mathsf{K_S}}$
StartTo	ST + { ST + name(from) + seqNo(from) + amount}$_{\mathsf{K_S}}$	REQ + { REQ + paydetails}$_{\mathsf{K_S}}$
Req	REQ + { REQ + paydetails }$_{\mathsf{K_S}}$	VAL + {VAL + paydetails}$_{\mathsf{K_S}}$
Val	VAL + {VAL + paydetails}$_{\mathsf{K_S}}$	ACK + {ACK + paydetails}$_{\mathsf{K_S}}$
Ack	ACK + { ACK + paydetails }$_{\mathsf{K_S}}$	-
GetExLog	GEL	{name + exlog}$_{\mathsf{K_B}}$
DeleteExlog	DEL + {DEL + exlog}$_{\mathsf{K_B}}$	-

Additionally to the original money transfer protocol we implemented the archiving functionality for the exception log. The exception log can be sent to the bank (using a key $\mathsf{K_B}$ shared between the banking server and all Mondex cards). When receiving the encrypted exception log of a card, the server decrypts it, stores the log in its internal database, encrypts it again using the same key and sends it back to the card. When the card receives this encrypted message it decrypts it and checks whether the transmitted exception log and its own exception log match. If they do the card can be sure that the server got the exception log and can safely delete it.

The full implementation of the protocol can be found in appendix A. In the following we will describe the implementation of the protocol step *val* as an example. The implementations of the other protocol steps are similar to this.

The input message for *val* is $VAL + \{VAL + currentPD\}_{sharedkey}$. The card is supposed to be in the *EPV* ("expecting value") state. The relevant part of the main protocol method *process(APDU)* for our implementation is:

```
public void process(APDU arg0) throws ISOException {
  JCSystem.beginTransaction();
  byte[] buffer = arg0.getBuffer();
  byte ins = buffer[ISO7816.OFFSET_INS];
  if (ins == ISO7816.INS_SELECT)
    return;
  else if (...)
    ...
  else if (state == EPV && ins == VAL
      && checkStep(arg0, VAL))
    val(arg0);
  else if (...)
    ...
```

```
    else {
      abort();
      JCSystem.commitTransaction();
      ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);
    }
    JCSystem.commitTransaction();
  }
```

Every protocol step starts with a call of *JCSystem.beginTransaction()* to ensure that the protocol step will be finished as a whole or completely rolled back in case of card removal or any other error. Afterwards, we do a case distinction over the instruction byte and the internal state of the card to determine which step should be performed. If the applet is in state *EPV* and the instruction byte is *VAL* (as it should be) we additionally have to check whether the (encrypted) content of the message is wellformed. This is done by the method *checkStep(APDU, byte)* (which is reused in the steps *REQ* and *ACK* because the checks in those steps are the same):

```
  private boolean checkStep(APDU arg0, byte ins) {
    short l = arg0.setIncomingAndReceive();
    if (l != (short) 24)
      return false;

    byte[] buffer = arg0.getBuffer();

    decrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
        (short) 24);

    // encrypted instruction has to be ok, pay details must
    // match
    if (buffer[ISO7816.OFFSET_CDATA] != ins
        || Util.arrayCompare(buffer,
            (short) (ISO7816.OFFSET_CDATA + 1), currentPD,
            (short) 0, (short) currentPD.length) != 0)
      return false;
    return true;
  }
```

At first, *checkStep* tests whether the input data block has the correct length of 24 bytes (3 encrypted 3DES blocks of length 8 bytes). Then, it decrypts the input buffer in place and checks whether the contained instruction byte matches the protocol step (it should be *VAL* here). Additionally, we have to check whether the pay details contained in the encrypted block match the local copy of the pay details. This ensures that the current message was sent by the right from purse. If everything is ok the method returns *true*. Now, the method *val(APDU)* is called.

```
  private void val(APDU arg0) {
    byte[] buffer = arg0.getBuffer();

    // increment balance
    balance = (short) (balance + Util.getShort(currentPD,
        index_amount));

    // build apdu for other card
    // we can use current input buffer, values should be
    // right already
    // except encryption/instruction byte
    buffer[1] = ACK;
    buffer[5] = ACK;
    encrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
        (short) (currentPD.length + 1));
    state = IDLE;
```

```
    arg0.setOutgoingAndSend((short) 0, (short) 29);
}
```

The *val* protocol step increments the stored balance by the amount currently in transit and constructs an output message. This is the *ACK* message for the other card. The current input buffer is reused, since it already contains the information that is needed (the current pay details). Only the instruction byte has to be changed (once in the header and once in the encrypted block). The state is reset to *IDLE* and the message is sent out. This completes the *val* protocol step.

If something has gone wrong during the *checkStep* method or the input instruction or the internal state do not match, the purse has to abort the current transaction, since obviously something bad happened (either an attack or a transmission error). This is done by the *abort* method:

```
private void abort() {
    logIfNeeded();
    state = IDLE;
}

private void logIfNeeded() {
    if (state == EPV || state == EPA) {
        Util.arrayCopy(currentPD, (short) 0, exlog,
            currentExLogIndex, (short) currentPD.length);
        currentExLogIndex += (short) currentPD.length;
    }
}
```

If the purse is in state *EPV* or *EPA* (those two are the only two critical protocol states where money could be lost), *abort* has to store the current pay details into the *exlog*. This is done in the method *logIfNeeded*. Logging is performed by copying the current pay details *currentPD* into the *exlog* array at the next free position *currentExLogIndex*. Afterwards *currentExLogIndex* has to be incremented accordingly.

Since we are talking about a real implementation and not (as in the specification of Mondex) about a list of pay details in the exception log, we have to care about exception log overflows. This is handled by the checks done when receiving the initial *startTo* and *startFrom* messages which start a new protocol run. Those checks now have to test whether the exlog is currently full (*currentExlogIndex* is equal to *exlog.length*). If this happens, no further protocol run can be started and the exlog has to be deleted.

2.2 Properties

The desired property for an implementation of Mondex is full functional correctness. When proving functional correctness additional properties of the program are needed, for example that no runtime exceptions occur. This is relevant for the logging when the current *paydetails* are copied to the *exlog* array (an *ArrayIndexOutOfBoundsException* may occur here).

Since the abstract specification talks about unbounded data types such as lists or natural numbers, the presented implementation of Mondex differs a little from its abstract specification. When implementing the specification the unbounded data types are replaced by finite arrays or *short* values. This leads to the necessity of additional checks for overflows (which may occur if positive values are loaded onto the card) and underflows (which may occur if negative values are loaded).

That implies that for verification purposes an invariant on the class is needed. E.g. the following properties for the *currentExLogIndex* should always hold before and after *process*[1] is called and should prevent an *ArrayIndexOutOfBoundsException* when logging (using the notation of the Java calculus in KIV [Ste04] [Ste05] where *st.applet* is a reference to the *Purse* applet object):

[1]Please note that this property is no class invariant, it does not hold before and after *logIfNeeded* for example.

$$
\begin{array}{l}
\quad (\mathrm{st[st.applet.state]} \neq \mathrm{IDLE} \rightarrow \\
\qquad \mathrm{st[st.applet.currentExLogIndex]} + 22 < \mathrm{st[st.applet.exlog.length]}) \\
\wedge\ (\exists\ \mathrm{i}.\ 0 \leq \mathrm{i} \wedge \mathrm{i} \leq 5 \wedge \mathrm{st[st.applet.currentExLogIndex]} = 22 * \mathrm{i})
\end{array}
$$

If the applet is not in the *IDLE* state (line 1), the *currentExLogIndex* is small enough to allow at least one additional log entry (line 2). Additionally, the *currentExLogIndex* is always a multiple of 22 (the *currentPD.length*) and is at most equal to 5 * 22 bytes (line 3) because this is the maximum exlog length (supporting five log entries).

The invariant for the balance and the current pay details would at least be:

$$
\begin{array}{l}
\quad 0 \leq \mathrm{st[st.applet.balance]} \\
\wedge\ 0 \leq \mathrm{getShort(st[st.applet.currentPD],\ 20,\ st)}\ \textit{//amount of money to transfer} \\
\wedge\ (\mathrm{st[st.applet.state]} = \mathrm{EPV} \\
\qquad \rightarrow \mathrm{getShort(st[st.applet.currentPD],\ 20,\ st)} \\
\qquad\qquad + \mathrm{st[st.applet.balance]} \leq 32767) \\
\wedge\ (\mathrm{st[st.applet.state]} = \mathrm{EPR} \\
\qquad \rightarrow 0 \leq \mathrm{st[st.applet.balance]} \\
\qquad\qquad - \mathrm{getShort(st[st.applet.currentPD],\ 20,\ st))}
\end{array}
$$

The balance and the amount currently in transit (stored in the *currentPD* array at index 20, encoded as two bytes) are always positive. Additionally, if there is any value currently in transit and the purse is in *EPV* state (meaning that a transaction to this purse is currently going on), the amount in transit plus the current balance is lower or equal than 32767 (the maximum positive *short* value). This will guarantee that no overflow will occur in the *val* protocol step. The counterpart to this property is that if the purse is in *epr* state the amount in transit will not lead to an underflow of the balance.

Certainly, for verification of functional correctness, much more properties would be needed. The ones presented in this section should be seen as an example.

Chapter 3

A Protocol using asymmetric Cryptography

3.1 Implementation

The implementation of the Mondex protocol introduced in Section 2 using symmetric encryption has the disadvantage that the same symmetric key is used by all purses. That means, if an attacker is able to find out this key he is able to decrypt all messages that are exchanged between two arbitrary purses. Thus, to avoid this weakness we implemented a second version of the Mondex protocol using asymmetric cryptography which is introduced here. Instead of using symmetric encryption each purse has an asymmetric key pair that is used to sign the data to be sent, i.e. to guarantee data integrity.
As the implementation of the symmetric version this implementation is based on the specification of the Mondex communication protocol level. In the specification a predicate *authentic : name* $\rightarrow$ *bool* is defined which specifies if a purse with name *name* is authentic. In the asymmetric version of the protocol the check of authenticity is implemented by verifying that a signature issued by the bank. Each purse stores a signature $Sig_{privkey(bank)}$*(pubkey(purse), name(purse))*, i.e. the public key and name of a purse signed by the bank. This implementation fulfils the same security properties as the symmetric variant but is closer to the specification because of the implemented check of authenticity.
Another difference between the specification and implementation of the asymmetric protocol is caused by the fact that the length of messages sent to resp. from Java SmartCards is restricted to 255 bytes. Since the *startFrom* and *startTo* messages of the asymmetric protocol are longer than 255 bytes, these messages are divided into submessages. Thus, the protocol hast two *startFrom* and three *startTo* messages.
The cryptographic protocol for the asymmetric version of Mondex is:

1.	term → from	`startFrom1`	:	pubkey(to), name(to), seqNo(to), amount
2.	term → from	`startFrom2`	:	$\text{Sig}_{\text{privkey(bank)}}$(pubkey(to), name(to))
3.	from → to	`startTo1`	:	pubkey(from), name(from), seqNo(from), amount
4.	from → to	`startTo2`	:	$\text{Sig}_{\text{privkey(bank)}}$(pubkey(from), name(from))
5.	from → to	`startTo3`	:	$\text{Sig}_{\text{privkey(from)}}$(message 3, message 4)
6.	to → from	`REQ`	:	paydetails, $\text{Sig}_{\text{privkey(to)}}$(REQ, paydetails)
7.	from → to	`VAL`	:	paydetails, $\text{Sig}_{\text{privkey(from)}}$(VAL, paydetails)
8.	to → from	`ACK`	:	paydetails, $\text{Sig}_{\text{privkey(to)}}$(ACK, paydetails)

to and *from* denote the from resp. to purse, *term* denotes the terminal. *pubkey(to)*, *name(to)* and *seqNo(to)* indicate the public key, name and sequence number of the to purse, amount is the amount of money that is going to be transfered. *privkey(bank)* denotes the private key of

the bank that issues the signatures for all purses. In this implementation the public key of the bank is known to all purses and stored by each purse in a field *bankkey*. Sig_{key}*(data)* denotes the signature of *data* with the *key*. Paydetails are the details of the current transaction, i.e. *name(to)*, *name(from)*, *seqNo(to)*, *seqNo(from)* and *amount*.
Figure 3.1 shows the communication between the purses and terminal during a protocol run.

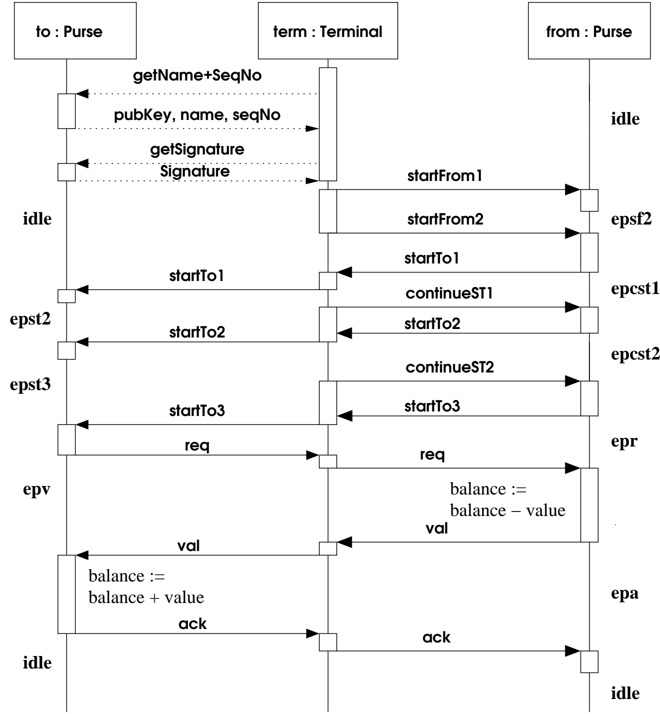


Figure 3.1: An overview of the asymmetric Mondex protocol

After requesting the public key, name, sequence number and signature of the to purse (*getName-SeqNo* and *getSignature*), these values as well as the amount to be transfered are forwarded to the from purse (*startFrom1* and *startFrom2*). The from purse verifies the signature of the to purse and creates the paydetails of the current transaction. Afterwards it sends its public key, name, sequence number, amount to be transfered as well as its signature to the to purse (*startTo1* and *startTo2*). Finally, the signature of these two messages is sent to the to purse (*startTo3*). The to purse verifies both signatures and creates the paydetails of the current transaction. The following messages, i.e. *req*, *val* and *ack*, are the same as in the specification of the Mondex communication protocol level with the modification that each message is signed by the sending purse. The following table gives detailed information about the messages that are sent. The format is the same as the one used in Section 2.

Protocol Step	**Input**	**Output**
GetNameSeqNo	GNS	pubkey + name + seqNo
StartFrom1	SF1 + pubkey(to) + name(to) + seqNo(to) + amount	-

StartFrom2	SF2 + $Sig_{privkey(bank)}$(pubkey(to), name(to))	ST1 + pubkey(from) + name(from) + seqNo(from) + amount
StartTo1	ST1 + pubkey(from) + name(from) + seqNo(from) + amount	-
ContinueST1	-	ST2 + $Sig_{privkey(bank)}$(pubkey(from), name(from))
StartTo2	ST2 + $Sig_{privkey(bank)}$(pubkey(from), name(from))	-
ContinueST2	-	ST3 + $Sig_{privkey(from)}$(ST3, pubkey(from), name(from), seqNo(from), amount, $Sig_{privkey(bank)}$(pubkey(from), name(from)))
StartTo3	ST3 + $Sig_{privkey(from)}$(ST3, pubkey(from), name(from), seqNo(from), amount, $Sig_{privkey(bank)}$(pubkey(from), name(from)))	REQ + paydetails + $Sig_{privkey(to)}$(REQ, paydetails)
Req	REQ + paydetails + $Sig_{privkey(to)}$(REQ, paydetails)	VAL + paydetails + $Sig_{privkey(from)}$(VAL, paydetails)
Val	VAL + paydetails + $Sig_{privkey(from)}$(VAL, paydetails)	ACK + paydetails + $Sig_{privkey(to)}$(ACK, paydetails)
Ack	ACK + paydetails + $Sig_{privkey(to)}$(ACK, paydetails)	-
GetExLog	ARCHIVE	exlog + name + $Sig_{privkey}$(exlog, name)
DeleteExlog	DELLOG + $Sig_{privkey(bank)}$(exlog)	-
GetBalance	BAL	balance

The implementation has the same fields as the symmetric version (i.e. balance, name, seqNo, state, currentPD, exlog, exlogCounter), only the keys differ. A purse stores its own private and public key as well as the public key of the bank. Moreover, the public key of the second purse participating in the money transfer is stored during the transaction process. All keys are of type RSAPublicKey resp. RSAPrivateKey. To sign data we use RSA-SHA1 signatures with PKCS1 padding.

The method *process* that is called when sending a command APDU to the smart card is the same as the one in the implementation of the symmetric protocol. As in Section 2 we will describe the protocol step VAL as an example step. The remaining steps are implemented in a similar way.
If a purse receives an APDU with instruction byte VAL, is in state EPV and *checkStep(APDU arg0)* returns *true*, the message *val(APDU arg0)* is called. In *checkStep* it is checked if the message has a length of 150 bytes, i.e. 22 bytes containing the paydetails and 128 bytes for the signature. Furthermore, the method checks whether the received paydetails are equal to the paydetails stored by the purse and the signature is verified.

```
private boolean checkStep(APDU arg0, byte ins) {
  if (!(arg0.setIncomingAndReceive() == 150)) {
          return false;}

  byte[] buffer = arg0.getBuffer();

  //compare received paydetails with current paydetails
  if(!(Util.arrayCompare(buffer, ISO7816.OFFSET_CDATA, currentPD,
                       (short) 0, (short)22) == 0)) {return false; }

  // copy INS, currentPD to temp
  temp[0] = ins;
  Util.arrayCopy(currentPD, (short) 0, temp, (short) 1, (short) 22);

  //verify signature Sig[otherpurse](INS,currentPD)
  sign.init(otherrsapub,Signature.MODE_VERIFY);
  if(!sign.verify(temp, (short)0, (short)23, buffer, (short)
     ((short)ISO7816.OFFSET_CDATA + (short)22), (short)128)) {return false;}

  return true;
}
```

The method *val(APDU arg0)* increases the balance of the to purse by the amount of the current transfer, generates the ACK message and sends it back to the terminal. Therefore, the purse signs the byte array (ACK, currentPD) and creates the command APDU that is sent to the terminal and then forwarded to the from purse.

```
private void val(APDU arg0) {
  byte[] buffer = arg0.getBuffer();

  // increase balance of to purse
  balance = (short) ((short) balance + Util.getShort(
              currentPD, index_amount));

  // send ack message
  // copy currentPD to buffer
  Util.arrayCopy(currentPD, (short) 0, buffer, ISO7816.OFFSET_CDATA,
  (short) currentPD.length);

  // create signature and copy signature to buffer
  temp[0] = ACK;
  //temp[1]..temp[22] stores the current paydetails (see method checkStep)
  sign.init(myrsapriv, Signature.MODE_SIGN);
  sign.sign(temp, (short) 0, (short)((short)currentPD.length + (short)1),
   buffer, (short)((short)ISO7816.OFFSET_CDATA + (short)currentPD.length));

  buffer[0] = (byte) 0x90; // CLA
  buffer[1] = ACK; // INS
  buffer[2] = (byte) 0; // P1
  buffer[3] = (byte) 0; // P2
  buffer[4] = (byte) 150; // length of data:
                        // currentPD.length + signature.length

  arg0.setOutgoingAndSend((short) ISO7816.OFFSET_CLA, (short)155);
  state = IDLE;
}
```

The archiving of exception logs of a purse at the bank and deleting it from the card is implemented by sending the exception log, the name of the purse as well as the signature $\mathrm{Sig}_{privkey}$(exlog, name) to the bank. It is assumed that the bank knows the public key of all purses, verifies the signature

and archives the received exception log. Afterwards, the bank sends the message DELLOG + $\mathrm{Sig}_{privkey(bank)}$(exlog) back to the purse that compares the signed exlog with its own exlog. If the signature check is successful the purse knows that the bank has stored its log and deletes it. Otherwise it ignores the message.
The full implementation of the protocol can be found in Appendix B.

3.2 Properties

For the asymmetric version of the protocol the same properties concerning runtime exceptions as for the symmetric version have to hold. Additionally, one has to consider the *temp* array that is used to temporarily store values in the asymmetric implementation. Since some protocol steps as *startFrom* and *startTo* are divided into multiple messages, some values are copied to the *temp* array in one method and read from there in another method for signature verification that is called afterwards. Thus, for functional correctness one has to prove that data that is expected to be stored in the *temp* array was set correctly by a previous method and the array fields have not been modified afterwards.

Chapter 4

A Protocol using special Data Types

4.1 Implementation

The last implementation we show is different from the two previous ones. It is based on our generic refinement approach for security protocols called Prosecco and described in [GHRS06] [HGRS05] [GSR06a]. [HSGR06] gives an overview of a security protocol specification for the Mondex case study. It uses a generic security protocol data type called *document* for the messages and for the internal state of the cards. The use of such a generic message data type is common in the literature for security protocol analysis, see e.g. [Pau98]. The following implementation transforms the *document* data type to Java classes. Since the communication interface of smart cards is based on APDUs (which are plain byte arrays), we have to add a transformation layer to the smart card implementation. This transformation layer is responsible for encoding and decoding Java objects to byte arrays and sending them using the normal APDU I/O interface. Some details regarding this transformation and the corresponding layer can be found in [GSR06b].

With the use of such Java classes a proof that the Java source code is a refinement of the security protocol specification becomes possible. Applying the refinement approach to the source code of Mondex presented below is current work in progress.

A part of the specification of the *document* data type is given as an example:

$$\begin{aligned}\textbf{Document} = \ldots \mid\ & \textbf{IntDoc}(\text{value} : \text{int}) \mid \textbf{HashDoc}(\text{hash} : \text{Document}) \mid \\ & \textbf{EncDoc}(\text{key} : \text{Key}, \text{doc} : \text{Document}) \mid \\ & \textbf{Doclist}(\text{docs} : \text{Documentlist}) \mid \ldots\end{aligned}$$

A document is either an *IntDoc* that represents an integer, a *HashDoc* that represents the hash value of a document, or an *EncDoc* that represents a document which is encrypted with a key. A *Doclist* contains a list of other documents.

Since it must be proven that the implementation produces the same output as the abstract specification claims (including the same type), a similar distinction of the types of the messages is crucial for the refinement. When using the Java language it is a natural approach to use objects for the representation of messages. For example, the implementation for IntDocs is:

```
public class IntDoc extends Document {
  private byte[] value;

  public IntDoc(byte[] val) {
    value = val;
  }

  public byte[] getValue(){
```

```
    return value;
  }
  ...
}
```

The integer value of the abstract type *IntDoc* is implemented by a byte array which represents the arbitrary large abstract integer. The other types of documents are described later on. When using such an implementation for security protocols there must be a method for sending and receiving the Java document instances.

The full implementation for all document classes as well as the Mondex implementation using documents can be found in Appendix C. Some explanations can be found in [GSR06b] and [GSR06a].

Again, we use the *val* protocol step as an example. At first, we take a look at the implementation of the state of the purses:

```
public class Purse{
  // state fields
  private byte[]      name;
  private short       sequenceNo;
  private short       balance;
  private byte        state;
  private Doclist     pd;
  private short       exLogCounter;
  private Doclist[]   exLog;
  // KEY
  private SessionKey  key;
  // COMMUNICATION
  private SimpleComm  comm;

  ...
}
```

The state of the purse is given by a byte array for the name, two shorts for the current sequence number and the card balance and a byte for the current state (*idle, epr, epv, epa*) as in the other two implementations. The current payment details are now implemented by an instance of class *Doclist* (which is a list of Documents). The structure of the payment details is illustrated by the following initialization statement taken from the *Purse* constructor:

```
pd = new Doclist(
       new Document[]{
         new IntDoc(new byte[8]),  // from-purse card name
         new IntDoc(new byte[2]),  // from-purse sequence number
         new IntDoc(new byte[8]),  // to-purse card name
         new IntDoc(new byte[2]),  // to-purse sequence number
         new IntDoc(new byte[2])   // amount
       });
```

The payment details are given as a list of documents (*Doclist*) consisting of five *IntDocs*, which store the relevant values.

The exception log *exlog* is stored as an array of *Doclist* instances. The elements of this array have the same structure as the payment details. The next free index in this *exlog* array is given by an *exLogCounter*, similar as in the previous two implementations.

The implementation uses the same security protocol as the one described in Section 2. For this protocol [HSGR06] gives a formal specification which uses cryptography. The main idea behind the implementation here is to adhere to the security protocol specification as good as possible. Communication in this specification is done by sending documents between the cards. The implementation now does the same using an implementation of the communication interface shown below:

```
public interface SimpleComm {
        public void send(Document d);
        public boolean available();
        public Document receive();}
```

This interface provides methods to send a document (which means encoding it into an APDU and sending the APDU), to receive a document (which means decoding of a received APDU into a document class instance and returning it) and to check, whether some input is currently available for the card. The implementation of this communication interface uses a TLV encoding style. It is omitted here but can be downloaded from our website at [KIVa].

We now have a look at the implementation of the main method for executing protocol steps in class *Purse* which uses the communication interface above. This is the method *step* in class *Purse*:

```
public void step() {
  Document outdoc = null;
  Document indoc = null;
  // check if there is a document in the inbox
  if(comm.available())
        indoc = comm.receive();
  else return;
  // nothing received
  if(indoc==null) return;
  // no memory available
  if(exLogCounter == exLog.length) return;
  indoc = checkIndoc(indoc);
  switch(getInsByte(indoc)) {
    case INS_START_FROM:
      ...; break;
    case ... :
      ...; break;
    case INS_VAL:
      outdoc = val(indoc.getPart(2));
      break;
    ...
    default:
      abort();           break;
  }
  // send outdoc
  if(outdoc!=null)
        comm.send(outdoc);
}
```

The method first checks whether some input for the card is available (*comm.available()*). If there is some, we receive it as a Document pointer structure. If the input is non-null and the card's exception log is not full, we first check if the input document is wellformed. This is done by the method *checkIndoc* which checks whether the input document has the correct structure expected for the current protocol step. E.g. a wellformed *val* message is an encrypted message (*EncDoc*) which contains a *Doclist* with an *IntDoc* (with the instruction *VAL*) and *Doclist* instance containing the current payment details (see above). This means *checkIndoc* first decrypts the received message with the shared key and then checks the described structure if the instruction inside the encrypted part was *VAL*.

After successfully checking the input document, the actual *val* protocol step is performed by the *val(Document)* method:

```
private Document val(Document paydetailsother) {
  if(!pd.equals(paydetailsother))
        return null;
  balance =
```

```
    (short)(balance +
            Util.getShort(
             pd.getPart((short)5).getValue(),(short)0));
  state = STATE_IDLE;
  return generate_ReqValAck_msg(INS_ACK);
}
```

To perform a *val* protocol step, we first check whether the transmitted payment details are equal to the local copy (*pd.equals(...)*). If they are, we increment the balance by the amount given by the payment details and set the state to *IDLE*. In the end, we generate an acknowledge message (*generate_ReqValAck_msg(INS_ACK)*), which constructs a corresponding *document* pointer structure). Finally, we return the *ACK* message which will be sent to the other card at the end of the *step* method described above.

4.2 Properties

The implementation described in this Section makes extensive use of pointer structures to represent the messages sent between the cards and to implement the state of the cards. The idea here is to implement the *document* data type of the security protocol very closely. When verifying such an implementation (the refinement proof for this implementation is current work in progress), we discovered that it is crucial to have good verification support for pointer structures. Some results for the verification technique for programs using complex pointer structues can be found in [SGR06].

For the Mondex example we use an invariant for the *Purse* class, which describes that the corresponding pointer structure is wellformed. Since memory allocation on smart cards cannot be done at runtime (there is no garbage collection and the memory is very limited), we have to allocate every piece of memory that is needed already inside the constructor of the purse. This also means that all *document* messages that will be sent by the card have to be pre-allocated at instance creation time. The same is true for the payment details field, the exception log and every other field of the purse.

This leads to an invariant that describes for every field which pointer structure is stored within the field. An example is given below for the payment details field *pd* of class *Purse*. This example uses the notation and predicates of the Java Calculus in KIV described in [Ste04] [Ste05] [SGR06].

is-pd-ref(r,st) $\leftrightarrow$
validrefnotnull(r, Doclist,st)
$\wedge$ validrefnotnull(st[r − .docs].refval, mkarraytype(Document),st)
$\wedge$ st[r.docs.length] = 5
$\wedge$ st[r.docs.type] = mkarraytype(Document)
$\wedge$ validrefnotnull(st[r.docs[0]], IntDoc, st)
$\wedge$ validrefnotnull(st[r.docs[1]], IntDoc, st)
$\wedge$ validrefnotnull(st[r.docs[2]], IntDoc, st)
$\wedge$ validrefnotnull(st[r.docs[3]], IntDoc, st)
$\wedge$ validrefnotnull(st[r.docs[4]], IntDoc, st)
$\wedge$ okarray(st[r.docs[0].value],byte_type,8,st)
$\wedge$ okarray(st[r.docs[1].value],byte_type,2,st)
$\wedge$ okarray(st[r.docs[2].value],byte_type,8,st)
$\wedge$ okarray(st[r.docs[3].value],byte_type,2,st)
$\wedge$ okarray(st[r.docs[4].value],byte_type,2,st)

Predicate *is-pd-ref(r,st)* describes when a pointer structure starting at reference r is a valid representation of payment details in Java store *st*. The reference itself has to have type *Doclist* and must not be *null* (*validrefnotnull(r, Doclist,st)*). The *docs* field of this *Doclist* has to be of the right type (array of Document) and its length must be five (since payment details have five slots). Every payment detail entry must be of type *IntDoc* (given by *validrefnotnull(st[r.docs[i]],*

IntDoc, st)) and the values of the entrys must be arrays of type byte of the correct length (*okarray(st[r.docs[i].value],byte_type,j,st)*).

Chapter 5

Conclusion

We presented three implementations for the Mondex case study.

Two of them are based on APDU communication with byte arrays. Both implementations try to implement the communication protocol specification level (see Fig. 1.1) as close as possible. The communication protocol level (the concrete level of [SCW00] or [SGHR06]) does not contain any cryptography in the specification yet. Both implementations have to add cryptography to ensure security of the protocol. One of them is using symmetric cryptography, the other one is using asymmetric cryptography. The use of asymmetric cryptography has some security advantages (if one card is hacked physically, this does not lead to the possibiliy of an attack against all other cards), but also requires a more complicated protocol because of limited APDU lengths.

We also show a way to implement the Mondex case study using specialized security protocol data types and symmetric cryptography. This implementation is based on the security protocol level which already introduces cryptography and is shown to be a refinement of the communication protocol level in [HSGR06]. The direct transformation of the datatypes of this specification level into Java classes makes a refinement proof for the Java code possible. This is current work in progress based on the approach described in [GSR06a].

Appendix A

Source Code: Symmetric cryptography using byte arrays

```
/*
 * Copyright (C) 2006 Holger Grandy, Department of Software
 * Engineering, University of Augsburg, Germany This program
 * is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details. You should have received a copy of the
 * GNU General Public License along with this program; if
 * not, write to the Free Software Foundation, Inc., 51
 * Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
 */
/*
 * Mondex Electronic Purse Implementation using symmetric
 * cryptography and pre shared keys Author: Holger Grandy,
 * Department of Software Engineering, University of
 * Augsburg, Germany This is an example implemenation for
 * the Mondex electronic purse scenario. This implementation
 * is a case study for research and for academic purposes
 * only. It has nothing to do with the real implementation
 * of Mondex Electronic Purses owned by Mastercard
 * International. There is absolutely no relationship
 * between the authors of this source code and National
 * Westminster Bank, Mastercard International or any other
 * institution involved in the Mondex Smart Cards. The
 * underlying specification for this implementation can be
 * found at
 * http://www.informatik.uni-augsburg.de/swt/projects/mondex.html
 * in Project "Mondex ASM Refinement - improved version"
 */
package mondex;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
```

```java
import javacard.framework.ISOException;
import javacard.framework.JCSystem;
import javacard.framework.Util;
import javacard.security.DESKey;
import javacard.security.KeyBuilder;
import javacardx.crypto.Cipher;

public class Purse extends Applet {
  // instructions
  private static final byte SF              = (byte) 2;
  private static final byte ST              = (byte) 6;
  private static final byte REQ             = (byte) 12;
  private static final byte VAL             = (byte) 14;
  private static final byte ACK             = (byte) 16;
  private static final byte GNS             = (byte) 18;
  private static final byte GEL             = (byte) 20;
  private static final byte DEL             = (byte) 22;
  private static final byte INIT            = (byte) 24;
  // states
  private static final byte IDLE            = (byte) 1;
  private static final byte EPR             = (byte) 5;
  private static final byte EPV             = (byte) 6;
  private static final byte EPA             = (byte) 7;
  private static final byte EPI             = (byte) 8;
  // pd indizes
  private static final byte index_fromname  = (byte) 4;
  private static final byte index_toname    = (byte) 12;
  private static final byte index_fromseqno = (byte) 0;
  private static final byte index_toseqno   = (byte) 2;
  private static final byte index_amount    = (byte) 20;
  private Cipher             cipher;
  // my protocol fields
  private short              balance;
  private byte[]             name;
  private short              seqNo;
  private byte               state;
  // pay details and exception log
  private byte[]             currentPD;
  private byte[]             exlog;
  private short              currentExLogIndex;
  // keys and signatures
  private byte[]             sharedkey;
  private byte[]             bankkey;
  private DESKey             sharedkeyobj;
  private DESKey             bankkeyobject;

  /**
   * Constructor, initialize fields and cryptographic keys
   */
  public Purse() {
        currentPD = new byte[8 /* name */+ 8 /* other name */
            + 2 /* seqno */+ 2 /* other seqno */
            + 2 /* amount */
        ];
        // support 5 exception log entries
        exlog = new byte[(short) (5 * currentPD.length)];
        currentExLogIndex = 0;
        // symmetric 3DES session key for communication with
```

```
        // other cards
        // value will be set in init phase
        sharedkey = new byte[24];
        // symmetric 3DES banking key, used only for archiving
        // and deleting
        // exception logs, value will be set in init phase
        bankkey = new byte[24];
        // session key object, value will be set in init phase
        sharedkeyobj = (DESKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES3_3KEY,
            true);
        // banking key object, value will be set in init phase
        bankkeyobject = (DESKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES3_3KEY,
            true);
        // current state
        state = EPI;
        // name will be set in INIT phase
        name = new byte[8];
        // initial balance will be set in INIT phase
        balance = 0;
        cipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NOPAD,
            true);
        register();
}

/**
 * create a new instance of this purse
 */
public static void install(byte[] b, short i, byte b0) {
        new Purse();
}

/**
 * noting to do when selecting this applet
 */
public boolean select() {
        return true;
}

/**
 * main case distinction, checks input / state and
 * corresponding protocol step function
 */
public void process(APDU arg0) throws ISOException {
        JCSystem.beginTransaction();
        byte[] buffer = arg0.getBuffer();
        byte ins = buffer[ISO7816.OFFSET_INS];
        if (ins == ISO7816.INS_SELECT)
          return;
        else if (state == EPI && ins == INIT)
          setFields(arg0);
        else if (state == IDLE && ins == GNS)
          getNameSeqNoBal(arg0);
        else if (state == IDLE && ins == GEL)
          getExlog(arg0);
        else if (state == IDLE && ins == DEL
            && checkExlog(arg0))
```

```java
        delExlog(arg0);
      else if (state == IDLE && ins == SF
          && checkStartFrom(arg0))
        startFrom(arg0);
      else if (state == IDLE && ins == ST
          && checkStartTo(arg0))
        startTo(arg0);
      else if (state == EPR && ins == REQ
          && checkStep(arg0, REQ))
        req(arg0);
      else if (state == EPV && ins == VAL
          && checkStep(arg0, VAL))
        val(arg0);
      else if (state == EPA && ins == ACK
          && checkStep(arg0, ACK))
        ack(arg0);
      else {
        abort();
        JCSystem.commitTransaction();
        ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);
      }
      JCSystem.commitTransaction();
}

// ************************************************
// personalize card
// ************************************************
/**
 * sets internal fields (both keys, name, balance), only
 * called once after initial installation of applet this
 * must happen in a secured environment at the bank
 * before the customer gets the card
 */
private void setFields(APDU arg0) {
      short l = arg0.setIncomingAndReceive();
      if (l != 58)
        return;
      byte[] buffer = arg0.getBuffer();
      // set banking key
      Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, bankkey,
          (short) 0, (short) 24);
      bankkeyobject.setKey(bankkey, (short) 0);
      // set shared card key
      Util.arrayCopy(buffer,
          (short) (ISO7816.OFFSET_CDATA + 24), sharedkey,
          (short) 0, (short) 24);
      sharedkeyobj.setKey(sharedkey, (short) 0);
      // set own name
      Util.arrayCopy(buffer,
          (short) (ISO7816.OFFSET_CDATA + 24 + 24), name,
          (short) 0, (short) 8);
      // set initial balance
      balance = Util.getShort(buffer,
          (short) (ISO7816.OFFSET_CDATA + 24 + 24 + 8));
      // init state
      state = IDLE;
}
```

```
// ************************************************
// check methods for input / state
// ************************************************
/**
 * check method for input / state when receiving
 * req/val/ack
 */
private boolean checkStep(APDU arg0, byte ins) {
      short l = arg0.setIncomingAndReceive();
      if (l != (short) 24)
        return false;
      byte[] buffer = arg0.getBuffer();
      decrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
          (short) 24);
      // encrypted instruction has to be ok, pay details must
      // match
      if (buffer[ISO7816.OFFSET_CDATA] != ins
          || Util.arrayCompare(buffer,
              (short) (ISO7816.OFFSET_CDATA + 1), currentPD,
              (short) 0, (short) currentPD.length) != 0)
        return false;
      return true;
}

/**
 * check method, called when receiving startTo
 */
private boolean checkStartTo(APDU arg0) {
      short l = arg0.setIncomingAndReceive();
      if (l != (short) 16)
        return false;
      byte[] buffer = arg0.getBuffer();
      // check seq no
      if (seqNo == 32767)
        return false;
      decrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
          (short) 16);
      // check instruction
      if (buffer[ISO7816.OFFSET_CDATA] != ST)
        return false;
      // check other name != own name
      if (Util.arrayCompare(buffer,
          (short) (ISO7816.OFFSET_CDATA + 1 + 2), name,
          (short) 0, (short) name.length) == (short) 0)
        return false;
      // check exlog not full
      if (currentExLogIndex == exlog.length)
        return false;
      // check positive amount
      if ((short) (Util.getShort(buffer,
          (short) (ISO7816.OFFSET_CDATA + 1 + 2 + 8))) <= (short) 0)
        return false;
      // check overflow
      if ((short) (Util.getShort(buffer,
          (short) (ISO7816.OFFSET_CDATA + 1 + 2 + 8)) + balance) <= (short) 0)
        return false;
      return true;
}
```

```
/**
 * check method, called when receiving startFrom
 */
private boolean checkStartFrom(APDU arg0) {
        short l = arg0.setIncomingAndReceive();
        byte[] buffer = arg0.getBuffer();
        if (l != 12)
          return false;
        // check seq no
        if (seqNo == 32767)
          return false;
        // check other name != own name
        if (Util.arrayCompare(buffer,
            (short) (ISO7816.OFFSET_CDATA + 2), name,
            (short) 0, (short) name.length) == (short) 0)
          return false;
        // check exlog not full
        if (currentExLogIndex == exlog.length)
          return false;
        // check positive amount
        if ((short) (Util.getShort(buffer,
            (short) (ISO7816.OFFSET_CDATA + 1 + 2 + 8))) <= (short) 0)
          return false;
        // check underflow
        if ((short) (Util
            .getShort(buffer, (short) (ISO7816.OFFSET_CDATA
                + (short) 2 + (short) 8))) > balance)
          return false;
        return true;
}

/**
 * check method, called when receiving a "delete exlog"
 * message
 * input: DEL + Enc[Bank](DEL + exlog)
 * output: ok
 */
private boolean checkExlog(APDU arg0) {
        arg0.setIncomingAndReceive();
        byte[] buffer = arg0.getBuffer();
        decrypt(bankkeyobject, buffer, ISO7816.OFFSET_CDATA,
            (short) 112);
        // check instruction
        if (buffer[(short) (ISO7816.OFFSET_CDATA)] != DEL)
          return false;
        // decrypted exlog must match
        if (Util.arrayCompare(buffer,
            (short) (ISO7816.OFFSET_CDATA + 1), exlog,
            (short) 0, (short) exlog.length) != 0)
          return false;
        return true;
}

// ***********************************************
// aborting and logging
// ***********************************************
/**
```

```
 * resets state and logs current pay details if in EPA
 * or EPV
 */
private void abort() {
      logIfNeeded();
      state = IDLE;
}

/**
 * logs current pay details if in EPA or EPV
 */
private void logIfNeeded() {
      if (state == EPV || state == EPA) {
        Util.arrayCopy(currentPD, (short) 0, exlog,
                currentExLogIndex, (short) currentPD.length);
        currentExLogIndex += (short) currentPD.length;
      }
}

// ************************************************
// archiving stuff
// ************************************************
/**
 * getExLog protocol step
 * input: GEL
 * output: Enc[Bank](name + exlog)
 */
private void getExlog(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      Util.arrayCopy(name, (short) 0, buffer, (short) 0,
          (short) name.length);
      Util.arrayCopy(exlog, (short) 0, buffer, (short) 8,
          (short) exlog.length);
      short len = encrypt(bankkeyobject, buffer, (short) 0,
          (short) (exlog.length + name.length));
      arg0.setOutgoingAndSend((short) 0, len);
}

/**
 * delete exlog entries and reset counter
 */
private void delExlog(APDU arg0) {
      for (short i = 0; i < (short) exlog.length; i++)
        exlog[i] = 0;
      currentExLogIndex = 0;
}

// ************************************************
// protocol steps from here
// ************************************************
/**
 * ack protocol step
 * input: ACK + Enc[SharedKey](ACK + Paydetails)
 * output: ok
 */
private void ack(APDU arg0) {
      state = IDLE;
}
```

```
/**
 * val protocol step
 * input: VAL + Enc[SharedKey](VAL + Paydetails)
 * output: ACK + Enc[SharedKey](ACK + Paydetails)
 */
private void val(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      // increment balance
      balance = (short) (balance + Util.getShort(currentPD,
          index_amount));
      // build apdu for other card
      // we can use current input buffer, values should be
      // right already
      // except encryption/instruction byte
      buffer[1] = ACK;
      buffer[5] = ACK;
      encrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
          (short) (currentPD.length + 1));
      state = IDLE;
      arg0.setOutgoingAndSend((short) 0, (short) 29);
}

/**
 * req protocol step
 * input: REQ + Enc[SharedKey](REQ + Paydetails)
 * output: VAL + Enc[SharedKey](VAL + Paydetails)
 */
private void req(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      balance = (short) (balance - Util.getShort(currentPD,
          index_amount));
      // build apdu for other card
      // we can use current input buffer, values should be
      // right already
      // except encryption/instruction byte
      buffer[1] = VAL;
      buffer[5] = VAL;
      encrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
          (short) (currentPD.length + 1));
      buffer[(short) (ISO7816.OFFSET_CDATA + 24)] = (byte) 30;
      state = EPA;
      arg0.setOutgoingAndSend((short) 0, (short) 30);
}

/**
 * startTo protocol step
 * input:
 * STARTTO + Enc[SharedKey](STARTTO
 * + seqno_other + name_other + amount)
 * output:
 * REQ + Enc[SharedKey](REQ + Paydetails)
 */
private void startTo(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      // copy other seq no
      Util.arrayCopy(buffer,
          (short) (ISO7816.OFFSET_CDATA + 1), currentPD,
```

```
            index_fromseqno, (short) 2);
        // copy other name
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + 1 + (short) 2),
            currentPD, index_fromname, (short) 8);
        // copy amount
        Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA
            + 1 + (short) 2 + (short) 8), currentPD,
            index_amount, (short) 2);
        // set own seqno
        Util.setShort(currentPD, index_toseqno, seqNo);
        // set own name
        Util.arrayCopy(name, (short) 0, currentPD,
            index_toname, (short) 8);
        // build apdu for other card
        buffer[0] = (byte) 0x90;
        buffer[1] = REQ;
        buffer[4] = (byte) 24;
        buffer[ISO7816.OFFSET_CDATA] = REQ;
        // copy paydetails
        Util.arrayCopy(currentPD, (short) 0, buffer,
            (short) (ISO7816.OFFSET_CDATA + 1),
            (short) currentPD.length);
        encrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
            (short) (currentPD.length + 1));
        buffer[(short) (ISO7816.OFFSET_CDATA + 24)] = (byte) 30;
        state = EPV;
        seqNo += (short) 1;
        arg0.setOutgoingAndSend((short) 0, (short) 30);
}

/**
 * startFrom protocol step
 * input:
 * STARTFROM + seqno_other + name_other + amount
 * output:
 * STARTTO + Enc[SharedKey](STARTFRom +
 * seqno + name + amount)
 */
private void startFrom(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy other seq no
        Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA ),
            currentPD, index_toseqno, (short) 2);
        // copy other name
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 2),
            currentPD, index_toname, (short) 8);
        // copy amount
        Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA
            + (short) 2 + (short) 8), currentPD, index_amount,
            (short) 2);
        // set own seqno
        Util.setShort(currentPD, index_fromseqno, seqNo);
        // set own name
        Util.arrayCopy(name, (short) 0, currentPD,
            index_fromname, (short) 8);
        // build apdu for other card
```

```
        buffer[0] = (byte) 0x90;
        buffer[1] = ST;
        buffer[4] = (byte) 16;
        buffer[5] = ST;
        Util.setShort(buffer,
            (short) (ISO7816.OFFSET_CDATA + 1), seqNo);
        Util.arrayCopy(name, (short) 0, buffer,
            (short) (ISO7816.OFFSET_CDATA + 1 + 2),
            (short) name.length);
        Util.arrayCopy(currentPD, index_amount, buffer,
            (short) (ISO7816.OFFSET_CDATA + 1 + 2 + 8),
            (short) 2);
        encrypt(sharedkeyobj, buffer, ISO7816.OFFSET_CDATA,
            (short) 13);
        buffer[(short) (ISO7816.OFFSET_CDATA + 16)] = (byte) 30;
        // increment seq no
        seqNo += (short) 1;
        state = EPR;
        arg0.setOutgoingAndSend((short) 0, (short) 22);
}

// ************************************************
// query messages from here
// ************************************************
/**
 * getNameSeqNoBalance protocol step
 * input: GNS
 * output: seqno + name + balance
 */
private void getNameSeqNoBal(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy my seq no
        Util.setShort(buffer, (short) (ISO7816.OFFSET_CDATA),
            seqNo);
        // copy my name
        Util.arrayCopy(name, (short) 0, buffer,
            (short) (ISO7816.OFFSET_CDATA + 2),
            (short) name.length);
        // copy my balance
        Util.setShort(buffer,
            (short) (ISO7816.OFFSET_CDATA + 2 + name.length),
            balance);
        state = IDLE;
        arg0.setOutgoingAndSend(ISO7816.OFFSET_CDATA,
            (short) (name.length + (short) 2 + (short) 2));
}

// ************************************************
// cryptography
// ************************************************
/**
 * encrypt buffer beginning at index encryptionbegin
 * with length len with Key k, store result in place
 */
private short encrypt(DESKey k, byte[] buffer,
          short encryptionbegin, short datalen) {
        short i = datalen;
        if (datalen % 8 != 0)
```

```
            i = (short) (datalen - (datalen % 8) + 8);
        for (short j = datalen; j < i; j++) {
            buffer[(short) (ISO7816.OFFSET_CDATA + j)] = (byte) 0;
        }
        cipher.init(k, Cipher.MODE_ENCRYPT);
        cipher.doFinal(buffer, encryptionbegin, i, buffer,
            encryptionbegin);
        return i;
    }

    /**
     * decrypt buffer beginning at index begin with length
     * len with Key k, store result in place
     */
    private void decrypt(DESKey k, byte[] buffer,
            short begin, short len) {
        cipher.init(k, Cipher.MODE_DECRYPT);
        cipher.doFinal(buffer, begin, len, buffer, begin);
    }
}
```

Appendix B

Source Code: Asymmetric cryptography using byte arrays

```
/*
 * Copyright (C) 2006 Nina Moebius, Department of Software
 * Engineering, University of Augsburg, Germany This program
 * is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details. You should have received a copy of the
 * GNU General Public License along with this program; if
 * not, write to the Free Software Foundation, Inc., 51
 * Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
 */
/*
 * Mondex Electronic Purse Implementation using asymmetric
 * cryptography Author: Nina Moebius, Department of Software
 * Engineering, University of Augsburg, Germany This is an
 * example implemenation for the Mondex electronic purse
 * scenario. This implementation is a case study for
 * research and for academic purposes only. It has nothing
 * to do with the real implementation of Mondex Electronic
 * Purses owned by Mastercard International. There is
 * absolutely no relationship between the authors of this
 * source code any National Westminster Bank, Mastercard
 * International or any other institution involved in the
 * Mondex Smart Cards. The underlying specification for this
 * implementation can be found at
 * http://www.informatik.uni-augsburg.de/swt/projects/mondex.html
 * in Project "Mondex ASM Refinement - improved version"
 */
package mondex;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
```

```
import javacard.framework.Util;
import javacard.framework.JCSystem;
import javacard.security.KeyBuilder;
import javacard.security.RSAPublicKey;
import javacard.security.RSAPrivateKey;
import javacard.security.Signature;

// Mondex Implementation using asymmetric cryptography
public class Purse extends Applet {
  // instructions
  private static final byte   SF1              = (byte) 2;
  private static final byte   SF2              = (byte) 4;
  private static final byte   ST1              = (byte) 6;
  private static final byte   CONTINUE_ST1     = (byte) 22;
  private static final byte   CONTINUE_ST2     = (byte) 24;
  private static final byte   ST2              = (byte) 8;
  private static final byte   ST3              = (byte) 10;
  private static final byte   REQ              = (byte) 12;
  private static final byte   VAL              = (byte) 14;
  private static final byte   ACK              = (byte) 16;
  private static final byte   GNS              = (byte) 18;
  private static final byte   GSI              = (byte) 20;
  private static final byte   ARCHIVE          = (byte) 26;
  private static final byte   DELLOG           = (byte) 30;
  private static final byte   BAL              = (byte) 28;
  // states
  private static final byte   IDLE             = (byte) 1;
  private static final byte   EPSF2            = (byte) 2;
  private static final byte   EPST2            = (byte) 3;
  private static final byte   EPST3            = (byte) 4;
  private static final byte   EPR              = (byte) 5;
  private static final byte   EPV              = (byte) 6;
  private static final byte   EPA              = (byte) 7;
  private static final byte   EPCST1           = (byte) 8;
  private static final byte   EPCST2           = (byte) 9;
  // state + instruction overloaded
  private static final byte   INIT1            = (byte) 91;
  private static final byte   INIT2            = (byte) 92;
  private static final byte   INIT3            = (byte) 93;
  private static final byte   INIT4            = (byte) 94;
  // pd indizes
  private static final byte   index_fromname   = (byte) 8;
  private static final byte   index_toname     = (byte) 0;
  private static final byte   index_fromseqno  = (byte) 18;
  private static final byte   index_toseqno    = (byte) 16;
  private static final byte   index_amount     = (byte) 20;
  // my protocol fields
  private short               balance;
  private byte[]              name;
  private short               seqNo;
  private byte                state;
  // pay details and exception log
  private byte[]              currentPD;
  private byte[]              exlog;
  private short               exlogCounter;
  // my keys and signatures
  private byte[]              mypubkey;
  private byte[]              myprivkey;
```

```
private byte[]           mypubexp;
private RSAPublicKey     myrsapub;
private RSAPrivateKey    myrsapriv;
private RSAPublicKey     otherrsapub;
private byte[]           othermodulus;
private byte[]           mysignature;
private RSAPublicKey     bankkey;
private Signature        sign;
// constant public bank key
private static final byte[] bankmodulus     = new byte[] {
        (byte) 0x83, (byte) 0xD5, (byte) 0xA8, (byte) 0xE7,
        (byte) 0x87, (byte) 0x28, (byte) 0xF6, (byte) 0xFD,
        (byte) 0x56, (byte) 0x09, (byte) 0x22, (byte) 0x43,
        (byte) 0x15, (byte) 0xA1, (byte) 0xC0, (byte) 0x40,
        (byte) 0x54, (byte) 0xBD, (byte) 0x9A, (byte) 0x3B,
        (byte) 0xD7, (byte) 0x9A, (byte) 0xAA, (byte) 0xF7,
        (byte) 0x9B, (byte) 0x8B, (byte) 0xF4, (byte) 0xA1,
        (byte) 0xFC, (byte) 0x96, (byte) 0xEF, (byte) 0xC8,
        (byte) 0xB7, (byte) 0xFF, (byte) 0x0C, (byte) 0xE7,
        (byte) 0x98, (byte) 0xF5, (byte) 0xDA, (byte) 0x7C,
        (byte) 0x19, (byte) 0x21, (byte) 0xD9, (byte) 0x13,
        (byte) 0x4A, (byte) 0x6D, (byte) 0x03, (byte) 0x05,
        (byte) 0x35, (byte) 0xDF, (byte) 0x86, (byte) 0x41,
        (byte) 0x8E, (byte) 0x50, (byte) 0x14, (byte) 0xFF,
        (byte) 0x15, (byte) 0xE2, (byte) 0xFC, (byte) 0xDD,
        (byte) 0x25, (byte) 0x2D, (byte) 0xFA, (byte) 0x75,
        (byte) 0x64, (byte) 0xA0, (byte) 0x44, (byte) 0x5A,
        (byte) 0xBE, (byte) 0x12, (byte) 0x58, (byte) 0x5D,
        (byte) 0xC3, (byte) 0xD3, (byte) 0x5E, (byte) 0x0E,
        (byte) 0xD5, (byte) 0x88, (byte) 0x08, (byte) 0x28,
        (byte) 0xA1, (byte) 0x45, (byte) 0x59, (byte) 0x93,
        (byte) 0xC8, (byte) 0x50, (byte) 0x3F, (byte) 0x0C,
        (byte) 0xE8, (byte) 0x86, (byte) 0x97, (byte) 0x52,
        (byte) 0x76, (byte) 0x9F, (byte) 0x34, (byte) 0xDF,
        (byte) 0x71, (byte) 0xEB, (byte) 0x8B, (byte) 0x5C,
        (byte) 0x64, (byte) 0x1E, (byte) 0x00, (byte) 0xAD,
        (byte) 0xA4, (byte) 0xFF, (byte) 0x67, (byte) 0xBA,
        (byte) 0x31, (byte) 0x31, (byte) 0xEC, (byte) 0x46,
        (byte) 0xE7, (byte) 0xC5, (byte) 0xB7, (byte) 0x22,
        (byte) 0xC4, (byte) 0x10, (byte) 0x9D, (byte) 0xB9,
        (byte) 0xFC, (byte) 0x5D, (byte) 0x6F, (byte) 0x56,
        (byte) 0x1F, (byte) 0xD7, (byte) 0x26, (byte) 0x9F, };
// one temporary array for signature checks and
// temporary data storage
private byte[]           temp;

public Purse() {
    currentPD = new byte[(short) 8 /* nameto */
        + (short) 8 /* namefrom */+ (short) 2 /* seqnoto */
        + (short) 2 /* seqnofrom */+ (short) 2 /* amount */
    ];
    temp = new byte[1024];
    // support 5 exception loggings
    exlog = new byte[(short) (5 * currentPD.length)];
    exlogCounter = 0;
    // init my fixed fields
    balance = (short) 200;
    seqNo = (short) 1;
```

```
        name = new byte[(short) 8];
        mypubexp = new byte[] { 1, 0, 1 };
        mypubkey = new byte[(short) 128];
        myprivkey = new byte[(short) 128];
        mysignature = new byte[(short) 128];
        othermodulus = new byte[128];
        state = IDLE;
        sign = Signature.getInstance(
            Signature.ALG_RSA_SHA_PKCS1, true);
        myrsapub = (RSAPublicKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_RSA_PUBLIC,
            KeyBuilder.LENGTH_RSA_1024, true);
        myrsapriv = (RSAPrivateKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_RSA_PRIVATE,
            KeyBuilder.LENGTH_RSA_1024, true);
        otherrsapub = (RSAPublicKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_RSA_PUBLIC,
            KeyBuilder.LENGTH_RSA_1024, true);
        otherrsapub.setExponent(new byte[] { (byte) 0x01,
            (byte) 0x00, (byte) 0x01 }, (short) 0, (short) 3);
        // initialize bank public key
        bankkey = (RSAPublicKey) KeyBuilder.buildKey(
            KeyBuilder.TYPE_RSA_PUBLIC,
            KeyBuilder.LENGTH_RSA_1024, true);
        bankkey.setExponent(new byte[] { (byte) 0x01,
            (byte) 0x00, (byte) 0x01 }, (short) 0, (short) 3);
        bankkey.setModulus(bankmodulus, (short) 0,
            (short) bankmodulus.length);
        register();
}

public static void install(byte[] b, short i, byte b0) {
        new Purse();
}

public boolean select() {
        return true;
}

// *********************************************
// main case distinction, basically calls checkstate and
// corresponding protocol step function
// *********************************************
public void process(APDU arg0) throws ISOException {
        JCSystem.beginTransaction();
        byte[] buffer = arg0.getBuffer();
        if (buffer[ISO7816.OFFSET_INS] == ISO7816.INS_SELECT)
          return;
        else if (buffer[ISO7816.OFFSET_INS] == BAL
           && state == IDLE)
          getBalance(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == INIT1
           && state == IDLE && checkLengthName(arg0))
          setName(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == INIT2
           && state == INIT2 && checkLength(arg0))
          setPubKey(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == INIT3
```

```
            && state == INIT3 && checkLength(arg0))
          setPrivKey(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == INIT4
            && state == INIT4 && checkLength(arg0))
          setSignature(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == GNS
            && state == IDLE)
          getNameSeqNo(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == GSI
            && state == IDLE)
          getSig(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == SF1
            && state == IDLE && checkStartFT(arg0))
          startFrom1(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == SF2
            && state == EPSF2 && checkSF2(arg0))
          startFrom2(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == ST1
            && state == IDLE && checkStartFT(arg0))
          startTo1(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == CONTINUE_ST1
            && state == EPCST1)
          continueST1(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == ST2
            && state == EPST2 && checkST2(arg0))
          startTo2(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == CONTINUE_ST2
            && state == EPCST2)
          continueST2(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == ST3
            && state == EPST3 && checkST3(arg0))
          startTo3(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == REQ
            && state == EPR && checkStep(arg0, REQ))
          req(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == VAL
            && state == EPV && checkStep(arg0, VAL))
          val(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == ACK
            && state == EPA && checkStep(arg0, ACK))
          ack(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == ARCHIVE
            && state == IDLE)
          archive(arg0);
        else if (buffer[ISO7816.OFFSET_INS] == DELLOG
            && state == IDLE && checkDelLog(arg0))
          delLog(arg0);
        else {
          abort();
          JCSystem.commitTransaction();
          ISOException.throwIt(ISO7816.SW_DATA_INVALID);
        }
        JCSystem.commitTransaction();
}

// ***************************************
// protocol steps from here
// ***************************************
```

```
// input: Ack((name_To, name_from, SeqNo_From, SeqNo_To,
// Amount), Sig[To](Ack,name_To, name_from, SeqNo_From,
// SeqNo_To, Amount))
// output: ok
private void ack(APDU arg0) {
      state = IDLE;
}

// input: Val((name_To, name_from, SeqNo_From, SeqNo_To,
// Amount), Sig[From](Val,name_To, name_from,
// SeqNo_From, SeqNo_To, Amount))
// output: Ack((name_To, name_from, SeqNo_From,
// SeqNo_To, Amount), Sig[To](Ack,name_To, name_from,
// SeqNo_From, SeqNo_To, Amount))
private void val(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      // increase balance of to purse
      balance = (short) ((short) balance + Util.getShort(
          currentPD, index_amount));
      // send ack message
      // copy currentPD to buffer
      Util.arrayCopy(currentPD, (short) 0, buffer,
          ISO7816.OFFSET_CDATA, (short) currentPD.length);
      // create signature and copy signature to buffer
      temp[0] = ACK;
      sign.init(myrsapriv, Signature.MODE_SIGN);
      sign
          .sign(
              temp,
              (short) 0,
              (short) ((short) currentPD.length + (short) 1),
              buffer,
              (short) ((short) ISO7816.OFFSET_CDATA + (short) currentPD.length));
      buffer[0] = (byte) 0x90; // CLA
      buffer[1] = ACK; // INS
      buffer[2] = (byte) 0; // P1
      buffer[3] = (byte) 0; // P2
      buffer[4] = (byte) 150; // length of data:
      // currentPD.length +
      // signature.length
      arg0.setOutgoingAndSend((short) ISO7816.OFFSET_CLA,
          (short) 155);
      state = IDLE;
}

// input: Req((name_To, name_from, SeqNo_From, SeqNo_To,
// Amount), Sig[To](Req,name_To, name_from, SeqNo_To,
// SeqNo_From, Amount))
// output: Val((name_To, name_from, SeqNo_From,
// SeqNo_To, Amount), Sig[From](Val,name_To, name_from,
// SeqNo_To, SeqNo_From, Amount))
private void req(APDU arg0) {
      byte[] buffer = arg0.getBuffer();
      // decrement balance of from purse
      balance = (short) ((short) balance - Util.getShort(
          currentPD, index_amount));
      // send val message
      // copy currentPD to buffer
```

```
        temp[0] = VAL;
        Util.arrayCopy(currentPD, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) currentPD.length);
        sign.init(myrsapriv, Signature.MODE_SIGN);
        sign
            .sign(
                temp,
                (short) 0,
                (short) ((short) currentPD.length + (short) 1),
                buffer,
                (short) ((short) ISO7816.OFFSET_CDATA + (short) currentPD.length));
        buffer[0] = (byte) 0x90; // CLA
        buffer[1] = VAL; // INS
        buffer[2] = (byte) 0; // P1
        buffer[3] = (byte) 0; // P2
        buffer[4] = (byte) 150; // length of data:
        // currentPD.length +
        // signature.length
        buffer[155] = (byte) 155; // le (= length of ack)
        arg0.setOutgoingAndSend((short) ISO7816.OFFSET_CLA,
            (short) (156));
        state = EPA;
}

// input: ST3(Sig[From]( Pubkey_From, Name_From,
// SeqNo_From, Amount, Sig[Bank](Pubkey_From,
// Name_From)))
// output: Req((name_To, name_from, SeqNo_From,
// SeqNo_To, Amount), Sig[To](Req,name_To, name_from,
// SeqNo_From, SeqNo_To, Amount))
private void startTo3(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy paydetails of to purse to output buffer
        Util.arrayCopy(currentPD, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) currentPD.length);
        // copy REQ, paydetails to temp
        temp[0] = REQ;
        Util.arrayCopy(currentPD, (short) 0, temp, (short) 1,
            (short) currentPD.length);
        // sign REQ, paydetails and copy signature to output
        // buffer
        sign.init(myrsapriv, Signature.MODE_SIGN);
        sign
            .sign(
                temp,
                (short) 0,
                (short) 23,
                buffer,
                (short) ((short) ISO7816.OFFSET_CDATA + (short) 22));
        buffer[0] = (byte) 0x90; // CLA
        buffer[1] = REQ; // INS
        buffer[2] = (byte) 0; // P1
        buffer[3] = (byte) 0; // P2
        buffer[4] = (byte) 150; // length of data:
        // currentPD.length +
        // signature.length
        buffer[155] = (byte) 156; // le: val + apdu buffer
        arg0
```

```
            .setOutgoingAndSend(ISO7816.OFFSET_CLA, (short) 156);
        state = EPV;
}

// input: ST2(Sig[Bank](Pubkey_From, Name_From))
// output: ok
private void startTo2(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy signature to temp array (-> temp =
        // [pubkeyFrom, nameFrom, seqNo_From, Amount,
        // Sig[bank](pubkeyFrom, nameFrom)])
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, temp,
            (short) ((short) 128 + (short) 8 + (short) 5),
            (short) 128);
        state = EPST3;
}

// input: ST1(Pubkey_From, Name_From, SeqNo_From,
// Amount)
// output: ok
private void startTo1(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        temp[0] = ST3;
        // copy other public key
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
            othermodulus, (short) 0, (short) 128);
        otherrsapub.setModulus(othermodulus, (short) 0,
            (short) 128);
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, temp,
            (short) 1, (short) 128);
        // copy other name to currentPD
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 128),
            currentPD, index_fromname, (short) 8);
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 128), temp,
            (short) 129, (short) 8);
        // copy other seqno
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 136),
            currentPD, index_fromseqno, (short) 2);
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 136), temp,
            (short) ((short) 129 + (short) 8), (short) 2);
        // copy amount
        Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA
            + (short) 128 + (short) 8 + (short) 2), currentPD,
            index_amount, (short) 2);
        Util.arrayCopy(buffer, (short) (ISO7816.OFFSET_CDATA
            + (short) 128 + (short) 8 + (short) 2), temp,
            (short) ((short) 129 + (short) 8 + (short) 2),
            (short) 2);
        // set own seqno
        Util.setShort(currentPD, index_toseqno, seqNo);
        // set own name
        Util.arrayCopy(name, (short) 0, currentPD,
            index_toname, (short) 8);
        seqNo += 1;
```

```java
        state = EPST2;
}

// input: CONTINUE_ST1
// output: ST2(Sig[Bank](Pubkey_From, Name_From))
private void continueST1(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        buffer[0] = (byte) 0x90; // CLA
        buffer[1] = ST2; // INS
        buffer[2] = (byte) 0; // P1
        buffer[3] = (byte) 0; // P2
        buffer[4] = (byte) mysignature.length; // length of
        // data
        Util.arrayCopy(mysignature, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) mysignature.length);
        arg0.setOutgoingAndSend(ISO7816.OFFSET_CLA,
            (short) ((short) 5 + (short) mysignature.length));
        state = EPCST2;
}

// input: CONTINUE_ST2
// output: ST3(Sig[From]( Pubkey_From, Name_From,
// SeqNo_From, Amount, Sig[Bank](Pubkey_From,
// Name_From)))
private void continueST2(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        buffer[0] = (byte) 0x90; // CLA
        buffer[1] = ST3; // INS
        buffer[2] = (byte) 0; // P1
        buffer[3] = (byte) 0; // P2
        buffer[4] = (byte) 128; // length of data
        buffer[5 + 128] = (byte) 156; // le: req + apdu
        // header
        // copy ST3, my pubkey, name, SeqNo, Amount to temp
        temp[0] = ST3;
        Util.arrayCopy(mypubkey, (short) 0, temp, (short) 1,
            (short) 128);
        Util.arrayCopy(name, (short) 0, temp,
            (short) ((short) 128 + (short) 1), (short) 8);
        Util.arrayCopy(currentPD, index_fromseqno, temp,
            (short) 137, (short) 2);
        Util.arrayCopy(currentPD, index_amount, temp,
            (short) 139, (short) 2);
        // copy Signature of From Purse to temp
        Util.arrayCopy(mysignature, (short) 0, temp,
            (short) 141, (short) 128);
        // sign data of temp array
        sign.init(myrsapriv, Signature.MODE_SIGN);
        sign.sign(temp, (short) 0, (short) 269, buffer,
            ISO7816.OFFSET_CDATA);
        // send signature
        arg0.setOutgoingAndSend(ISO7816.OFFSET_CLA,
            (short) ((short) 6 + (short) 128));
        Util.arrayCopy(mypubexp, (short) 0, temp, (short) 0,
            (short) 3);
        state = EPR;
}
```

```
// input: SF2(Sig[Bank](Pubkey_To + name_To)
// output: ST1(Pubkey_From, Name_From, SeqNo_From,
// Amount)
private void startFrom2(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // build apdu for TO purse
        buffer[0] = (byte) 0x90;
        buffer[1] = ST1;
        buffer[2] = (byte) 0x00;
        buffer[3] = (byte) 0x00;
        buffer[4] = (byte) ((short) 128 + (short) 8 + (short) 2 + (short) 2);
        // copy own public key
        Util.arrayCopy(mypubkey, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) 128);
        // copy own name
        Util.arrayCopy(name, (short) 0, buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 128),
            (short) 8);
        // copy own seq no
        Util.arrayCopy(currentPD, index_fromseqno, buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 136),
            (short) 2);
        // copy amount
        Util.arrayCopy(currentPD, index_amount, buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 128
                + (short) 8 + (short) 2), (short) 2);
        arg0.setOutgoingAndSend((short) ISO7816.OFFSET_CLA,
            (short) 145);
        state = EPCST1;
}

// input: SF1(Pubkey_To, SeqNo_To, Name_To, Amount)
// output: ok
private void startFrom1(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy other public key
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
            othermodulus, (short) 0, (short) 128);
        otherrsapub.setModulus(othermodulus, (short) 0,
            (short) 128);
        // copy other name to currentPD
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 128),
            currentPD, index_toname, (short) 8);
        // copy other seq no to currentPD
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 136),
            currentPD, index_toseqno, (short) 2);
        // copy amount to currentPD
        Util.arrayCopy(buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 138),
            currentPD, index_amount, (short) 2);
        // set own seqno
        Util.setShort(currentPD, index_fromseqno, seqNo);
        // set own name
        Util.arrayCopy(name, (short) 0, currentPD,
            index_fromname, (short) 8);
        seqNo += (short) 1;
```

```
        state = EPSF2;
}

// archive logfile
// output: exlog, name, Sig[privkeyPurse](exlog, name)
private void archive(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy exlog, name to temp
        Util.arrayCopy(exlog, (short) 0, temp, (short) 0,
            (short) 110);
        Util.arrayCopy(name, (short) 0, temp, (short) 110,
            (short) 8);
        Util.arrayCopy(temp, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) 118);
        sign.init(myrsapriv, Signature.MODE_SIGN);
        sign.sign(temp, (short) 0, (short) 118, buffer,
            (short) (ISO7816.OFFSET_CDATA + (short) 118));
        arg0.setOutgoingAndSend(ISO7816.OFFSET_CDATA,
            (short) 246);
}

// delete logfile
private void delLog(APDU arg0) {
        for (short i = 0; i < (short) exlog.length; i++) {
          exlog[i] = 0;
        }
        exlogCounter = 0;
}

// abort, i.e. set state to IDLE and write logfile if
// necessary
private void abort() {
        logifneeded();
        state = IDLE;
}

// log if purse is in state EPA or EPV
private void logifneeded() {
        if (state == EPA || state == EPV) {
          // add current paydetails to exLog
          Util.arrayCopy(currentPD, (short) 0, exlog,
                (short) exlogCounter, (short) currentPD.length);
          exlogCounter += (short) currentPD.length;
        }
}

// check if input has length of 128 bytes
private boolean checkLength(APDU arg0) {
        return (arg0.setIncomingAndReceive() == (short) 128);
}

// check if input has length of 8 bytes
private boolean checkLengthName(APDU arg0) {
        return (arg0.setIncomingAndReceive() == (short) 0x08);
}

// check if input, i.e startFrom2 message, has length of
// 128 bytes and signature ok (purse is authentic)
```

```
private boolean checkSF2(APDU arg0) {
       if (!(arg0.setIncomingAndReceive() == 128)) {
         return false;
       }
       byte[] buffer = arg0.getBuffer();
       // copy pubkeyTo, nameTo to temp
       Util.arrayCopy(othermodulus, (short) 0, temp,
           (short) 0, (short) 128);
       Util.arrayCopy(currentPD, index_toname, temp,
           (short) 128, (short) 8);
       // verify Sig[bank](pubkeyTo, nameTo)
       sign.init(bankkey, Signature.MODE_VERIFY);
       if (!sign.verify(temp, (short) 0, (short) 136, buffer,
           ISO7816.OFFSET_CDATA, (short) 128)) {
         return false;
       }
       return true;
}

// check if input, i.e startTo2 message, has length of
// 128 bytes and signature ok (purse is authentic)
private boolean checkST2(APDU arg0) {
       if (!(arg0.setIncomingAndReceive() == 128)) {
         return false;
       }
       byte[] buffer = arg0.getBuffer();
       sign.init(bankkey, Signature.MODE_VERIFY);
       if (!sign.verify(temp, (short) 1, (short) 136, buffer,
           ISO7816.OFFSET_CDATA, (short) 128)) {
         return false;
       }
       return true;
}

// check if input, i.e startTo3 message, has length of
// 128 bytes and signature ok
// temp = [pubkeyFrom, nameFrom, seqNo_From, Amount,
// Sig[bank](pubkeyFrom, nameFrom)])
private boolean checkST3(APDU arg0) {
       if (arg0.setIncomingAndReceive() != 128) {
         return false;
       }
       byte[] buffer = arg0.getBuffer();
       sign.init(otherrsapub, Signature.MODE_VERIFY);
       if (!sign.verify(temp, (short) 0, (short) 269, buffer,
           ISO7816.OFFSET_CDATA, (short) 128)) {
         return false;
       }
       return true;
}

// check if startTo / startFrom input is ok: nameTo !=
// nameFrom, amount > 0?, amount < balance?, seqNo <
// 32767? and exLogCounter < exlog.length
private boolean checkStartFT(APDU arg0) {
       // exLog has max. number of entries
       if (exlogCounter == exlog.length) {
         return false;
```

```
        }
        // check input length
        if (arg0.setIncomingAndReceive() != (short) 140) {
          return false;
        }
        byte[] buffer = arg0.getBuffer();
        // check nameTo!=nameFrom
        if (Util
            .arrayCompare(
                buffer,
                (short) ((short) ISO7816.OFFSET_CDATA + (short) 128),
                name, (short) 0, (short) 8) == 0) {
          return false;
        }
        // check amount
        if (!(Util
            .getShort(
                buffer,
                (short) ((short) ISO7816.OFFSET_CDATA + (short) 138)) > (short) 0)) {
          return false;
        }
        // if ToPurse: amount + balance < 32767 ?
        if (buffer[ISO7816.OFFSET_INS] == ST1
            && (short) ((short) Util
                .getShort(
                    buffer,
                    (short) ((short) ISO7816.OFFSET_CDATA + (short) 138))
                                 + (short) balance) <= (short) 0) {
          return false;
        }
        // if FromPurse: amount < balance ?
        if (buffer[ISO7816.OFFSET_INS] == SF1
            && Util
                .getShort(
                    buffer,
                    (short) ((short) ISO7816.OFFSET_CDATA + (short) 138)) > balance) {
          return false;
        }
        // check seqNo
        if (seqNo == (short) 32767) {
          return false;
        }
        return true;
}

// check if input data has correct length, paydetails ok
// and Signature ok
private boolean checkStep(APDU arg0, byte ins) {
        if (!(arg0.setIncomingAndReceive() == 150)) {
          return false;
        }
        byte[] buffer = arg0.getBuffer();
        // compare received paydetails with current
        // paydetails
        if (!(Util.arrayCompare(buffer, ISO7816.OFFSET_CDATA,
            currentPD, (short) 0, (short) 22) == 0)) {
          return false;
        }
```

```
        // copy INS, currentPD to temp
        temp[0] = ins;
        Util.arrayCopy(currentPD, (short) 0, temp, (short) 1,
            (short) 22);
        // verify signature Sig[otherpurse](INS,currentPD)
        sign.init(otherrsapub, Signature.MODE_VERIFY);
        if (!sign
            .verify(
                temp,
                (short) 0,
                (short) 23,
                buffer,
                (short) ((short) ISO7816.OFFSET_CDATA + (short) 22),
                (short) 128)) {
          return false;
        }
        return true;
}

// check if input has length of 128 bytes and signature
// ok
private boolean checkDelLog(APDU arg0) {
        if (!(arg0.setIncomingAndReceive() == (short) 128)) {
          return false;
        }
        byte[] buffer = arg0.getBuffer();
        sign.init(bankkey, Signature.MODE_VERIFY);
        if (sign.verify(exlog, (short) 0, (short) 110, buffer,
            ISO7816.OFFSET_CDATA, (short) 128)) {
          return false;
        }
        return true;
}

// ************************************
// query messages from here
// ************************************
// input: getsig
// output: Sig[Bank](PubKey_To, Name_To)
private void getSig(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        Util.arrayCopy(mysignature, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) mysignature.length);
        arg0.setOutgoingAndSend(ISO7816.OFFSET_CDATA,
            (short) mysignature.length);
}

// input: getNameSeqNo
// output: (Pubkey_To, Name_To, SeqNo_To)
private void getNameSeqNo(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        // copy my pub key
        Util.arrayCopy(mypubkey, (short) 0, buffer,
            ISO7816.OFFSET_CDATA, (short) mypubkey.length);
        // copy my name
        Util.arrayCopy(name, (short) 0, buffer,
            (short) (ISO7816.OFFSET_CDATA + mypubkey.length),
            (short) name.length);
```

```java
        // copy my seq no
        Util.setShort(buffer, (short) (ISO7816.OFFSET_CDATA
            + mypubkey.length + name.length), seqNo);
        arg0
            .setOutgoingAndSend(
                ISO7816.OFFSET_CDATA,
                (short) (mypubkey.length + name.length + (short) 2));
}

// input: getBalance
// output: Balance
private void getBalance(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        Util.setShort(buffer, (short) (ISO7816.OFFSET_CDATA),
            balance);
        arg0
            .setOutgoingAndSend(ISO7816.OFFSET_CDATA, (short) 2);
}

// ******************************
// initialization steps from here
// ******************************
// input: setName(name)
// output: ok
private void setName(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, name,
            (short) 0, (short) name.length);
        state = INIT2;
}

// input: setPubKey(PubKey_Purse)
// output: ok
private void setPubKey(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, mypubkey,
            (short) 0, (short) mypubkey.length);
        myrsapub.setExponent(mypubexp, (short) 0, (short) 3);
        myrsapub.setModulus(mypubkey, (short) 0, (short) 128);
        state = INIT3;
}

// input: setPrivKey(PrivKey_Purse)
// output: ok
private void setPrivKey(APDU arg0) {
        byte[] buffer = arg0.getBuffer();
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, myprivkey,
            (short) 0, (short) myprivkey.length);
        myrsapriv
            .setExponent(myprivkey, (short) 0, (short) 128);
        myrsapriv.setModulus(mypubkey, (short) 0, (short) 128);
        state = INIT4;
}

// input: setSignature(Sig[Bank](Pubkey_Purse,
// Name_Purse))
// output: ok
private void setSignature(APDU arg0) {
```

```
        byte[] buffer = arg0.getBuffer();
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
            mysignature, (short) 0, (short) mysignature.length);
        state = IDLE;
  }
}
```

Appendix C

Source Code: Symmetric cryptography using Documents

Please note: This is only the implementation of the Purse itself. The transformation classes for encoding of *Document* instances to APDUs and vice versa are not contained here. They can be downloaded from our website [KIVa].

```
/*
 * Copyright (C) 2006 Holger Grandy, Department of Software
 * Engineering, University of Augsburg, Germany This program
 * is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details. You should have received a copy of the
 * GNU General Public License along with this program; if
 * not, write to the Free Software Foundation, Inc., 51
 * Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA
 */
/*
 * Mondex Electronic Purse Implementation using symmetric
 * cryptography and pre shared keys Author: Holger Grandy,
 * Department of Software Engineering, University of
 * Augsburg, Germany This is an example implemenation for
 * the Mondex electronic purse scenario. This implementation
 * is a case study for research and for academic purposes
 * only. It has nothing to do with the real implementation
 * of Mondex Electronic Purses owned by Mastercard
 * International. There is absolutely no relationship
 * between the authors of this source code and National
 * Westminster Bank, Mastercard International or any other
 * institution involved in the Mondex Smart Cards. The
 * underlying specification for this implementation can be
 * found at
 * http://www.informatik.uni-augsburg.de/swt/projects/mondex.html
 * in Project "Mondex Specification using the Prosecco
 * Approach"
 */
package mondexOnCard;
```

```java
import javacard.framework.*;

public class Purse implements TransformApplication {
  // private static Purse theinstance;
  // private static SimpleComm initcomm;
  // private static Document initdata;
  // CONSTANTS
  private static final byte STATE_IDLE      = 1;
  private static final byte STATE_EPR       = 2;
  private static final byte STATE_EPV       = 3;
  private static final byte STATE_EPA       = 4;
  // INSBYTES
  private static final byte INS_START_FROM = 1;
  private static final byte INS_START_TO    = 2;
  private static final byte INS_REQ         = 3;
  private static final byte INS_VAL         = 4;
  private static final byte INS_ACK         = 5;
  private static final byte INS_GET_BAL     = 6;
  private static final byte INS_GET_DATA    = 7;
  private static final byte INS_GET_STATE   = 8;
  // FIELDS OF PURSE
  private byte[]             name;
  private short              sequenceNo;
  private short              balance;
  private byte               state;
  private short              exLogCounter;
  private Doclist[]          exLog;
  // MESSAGES
  private Doclist            pd;
  private Doclist            reqvalack_msg;
  private Doclist            startto_msg;
  private Doclist            getdata_msg;
  private EncDoc             enc_msg;
  private IntDoc             balstate_msg;
  // COMMUNICATION
  private SimpleComm         comm;
  // KEY
  private SessionKey         key;

  /**
   * constructor, init fields and messages
   */
  public Purse(SimpleComm initcomm, Document initdata) {
        // check initdata:
        // document must be of type:
        // doclist(intdoc1,intdoc2,intdoc3)
        // intdoc1 must have 8 digits (thename)
        // intdoc2 must have 2 digits (initbal)
        // intdoc3 must have 2 digits (loglen)
        Document thename = initdata.getPart((short) 1);
        Document initbal = initdata.getPart((short) 2);
        Document loglen = initdata.getPart((short) 3);
        if (!(thename != null && thename.is_intdoc()
            && thename.getValue().length == 8
            && initbal != null && initbal.is_intdoc()
            && initbal.getValue().length == 2 && loglen != null
            && loglen.is_intdoc() && loglen.getValue().length == 2))
```

```
            return;
        short theloglen = Util.getShort(loglen.getValue(),
            (short) 0);
        short thebalance = Util.getShort(initbal.getValue(),
            (short) 0);
        if (theloglen <= 0 || thebalance <= 0)
            return;
        // init all fields
        initExLog(theloglen);
        initPaydetails();
        initSimpleFields();
        initSessionKey();
        initOutMessages();
        // save name, balance and comminterface
        Util.arrayCopy(thename.getValue(), (short) 0, name,
            (short) 0, (short) 8);
        balance = thebalance;
        comm = initcomm;
}

/**
 * init fields for exlog
 *
 * @param len
 *            length of exlog
 */
private void initExLog(short len) {
        exLog = new Doclist[len];
        exLogCounter = (short) 0;
}

/**
 * init paydetailsfield
 */
private void initPaydetails() {
        pd = new Doclist(new Document[] {
            new IntDoc(new byte[8]), new IntDoc(new byte[2]),
            new IntDoc(new byte[8]), new IntDoc(new byte[2]),
            new IntDoc(new byte[2]) });
}

/**
 * init simple fields (shorts, bytes)
 */
private void initSimpleFields() {
        sequenceNo = (short) 0;
        state = STATE_IDLE;
        name = new byte[8];
}

/**
 * init sessionkey
 */
private void initSessionKey() {
        key = new SessionKey(new byte[] { 1, 2, 3, 4, 5, 6, 7,
            8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4 });
}
```

```
/**
 * init all messages used for communication
 */
private void initOutMessages() {
      startto_msg = new Doclist(new Document[] {
          new IntDoc(new byte[2]),
          new Doclist(new Document[] {
              new IntDoc(new byte[8]),
              new IntDoc(new byte[2]),
              new IntDoc(new byte[2]) }) });
      reqvalack_msg = new Doclist(new Document[] {
          new IntDoc(new byte[2]),
          new Doclist(new Document[] {
              new IntDoc(new byte[8]),
              new IntDoc(new byte[2]),
              new IntDoc(new byte[8]),
              new IntDoc(new byte[2]),
              new IntDoc(new byte[2]) }) });
      balstate_msg = new IntDoc(new byte[2]);
      getdata_msg = new Doclist(new Document[] {
          new IntDoc(new byte[8]), new IntDoc(new byte[2]) });
      enc_msg = new EncDoc(new byte[56]);
}

/**
 * Set paydetails
 *
 * @param fromPurse
 *            fromPurse involved in transfer
 * @param nextSeqNoFromPurse
 *            seqno of fromPurse
 * @param toPurse
 *            toPurse involved in transfer
 * @param nextSeqNoToPurse
 *            seqno of toPurse
 * @param value
 *            value to transfer
 */
private void mkpd(byte[] fromPurse,
        short nextSeqNoFromPurse, byte[] toPurse,
        short nextSeqNoToPurse, short value) {
      byte[] theval1 = pd.getPart((short) 1).getValue();
      byte[] theval2 = pd.getPart((short) 2).getValue();
      byte[] theval3 = pd.getPart((short) 3).getValue();
      byte[] theval4 = pd.getPart((short) 4).getValue();
      byte[] theval5 = pd.getPart((short) 5).getValue();
      Util.arrayCopy(fromPurse, (short) 0, theval1,
          (short) 0, (short) 8);
      Util.setShort(theval2, (short) 0, nextSeqNoFromPurse);
      Util.arrayCopy(toPurse, (short) 0, theval3, (short) 0,
          (short) 8);
      Util.setShort(theval4, (short) 0, nextSeqNoToPurse);
      Util.setShort(theval5, (short) 0, value);
}

/**
 * return next seq no
 */
```

```java
private short nextSeqNo() {
        if (sequenceNo < 32767)
          return sequenceNo++;
        else return -1;
}

/**
 * chech structure of indoc
 *
 * @param indoc
 *              document to check
 * @return document or null
 */
private Document checkIndoc(Document indoc) {
        if (indoc == null)
          return null;
        // decrypt if necessary
        if (indoc.is_encdoc()) {
          indoc = Crypto.getCrypto().decrypt(key, indoc);
          // check assures, that indoc is a correct req, val,
          // ack or startto message
          // (follows from abstract invariant)
          if (indoc != null)
                return indoc;
          return null;
        } else if (indoc.is_intdoc()) {
          byte[] theval = indoc.getValue();
          if (theval != null && theval.length == 2) {
                short insb = Util.getShort(theval, (short) 0);
                if (insb >= 0) {
                  byte b = (byte) insb;
                  if (b == INS_GET_BAL || b == INS_GET_DATA
                          || b == INS_GET_STATE) {
                        return indoc;
                  }
                }
          }
        } else if (indoc.is_doclist()) {
          Document[] docs = ((Doclist) indoc).getDocs();
          if (docs.length == 2) {
                Document part1 = indoc.getPart((short) 1);
                Document part2 = indoc.getPart((short) 2);
                if (part1.is_intdoc() && part2.is_doclist()) {
                  byte[] part1value = part1.getValue();
                  Document[] part2docs = ((Doclist) part2)
                          .getDocs();
                  if (part1value.length == 2 && part1value[0] == 0
                          && part1value[1] == INS_START_FROM) {
                        if (part2docs.length == 3
                            && part2docs[0].is_intdoc()
                            && part2docs[1].is_intdoc()
                            && part2docs[2].is_intdoc())
                          return indoc;
                  }
                }
          }
        }
        return null;
```

```
}

/**
 * copy local paydetails in d
 */
private void copyLocalPds() {
        byte[] pd1value = pd.getPart((short) 1).getValue();
        byte[] pd2value = pd.getPart((short) 2).getValue();
        byte[] pd3value = pd.getPart((short) 3).getValue();
        byte[] pd4value = pd.getPart((short) 4).getValue();
        byte[] pd5value = pd.getPart((short) 5).getValue();
        Document outmsg_pds = reqvalack_msg.getDocs()[(short) 1];
        byte[] out1value = outmsg_pds.getPart((short) 1)
                .getValue();
        byte[] out2value = outmsg_pds.getPart((short) 2)
                .getValue();
        byte[] out3value = outmsg_pds.getPart((short) 3)
                .getValue();
        byte[] out4value = outmsg_pds.getPart((short) 4)
                .getValue();
        byte[] out5value = outmsg_pds.getPart((short) 5)
                .getValue();
        Util.arrayCopy(pd1value, (short) 0, out1value,
                (short) 0, (short) 8);
        Util.arrayCopy(pd2value, (short) 0, out2value,
                (short) 0, (short) 2);
        Util.arrayCopy(pd3value, (short) 0, out3value,
                (short) 0, (short) 8);
        Util.arrayCopy(pd4value, (short) 0, out4value,
                (short) 0, (short) 2);
        Util.arrayCopy(pd5value, (short) 0, out5value,
                (short) 0, (short) 2);
}

/**
 * set the pd in the outmsg and set insbyte
 *
 * @param ins
 *            set the insbyte of outmsg to this byte
 * @return outmsg
 */
private Document generate_ReqValAck_msg(byte ins) {
        // copy pds in outmessage
        copyLocalPds();
        // set instructionbyte
        byte[] theval = reqvalack_msg.getPart((short) 1)
                .getValue();
        theval[0] = (byte) 0;
        theval[1] = ins;
        // encrypt
        enc_msg.setEncrypted(key, reqvalack_msg);
        return enc_msg;
}

/**
 * Generates encrypted startto-message
 *
 * @param ins
```

```
 *              set the insbyte of outmsg to this byte
 * @return outmsg
 */
private Document generate_Startto_msg() {
      // get information of paydetails
      byte[] fromname = pd.getPart((short) 1).getValue();// name
      // from
      byte[] fromseqno = pd.getPart((short) 2).getValue();// seqno
      // from
      byte[] thevalue = pd.getPart((short) 5).getValue();// value
      Document startto = startto_msg.getPart((short) 2);
      byte[] startto1value = startto.getPart((short) 1)
          .getValue();
      byte[] startto2value = startto.getPart((short) 2)
          .getValue();
      byte[] startto3value = startto.getPart((short) 3)
          .getValue();
      // copy from paydetails to startto_msg
      Util.arrayCopy(fromname, (short) 0, startto1value,
          (short) 0, (short) 8);
      Util.arrayCopy(thevalue, (short) 0, startto2value,
          (short) 0, (short) 2);
      Util.arrayCopy(fromseqno, (short) 0, startto3value,
          (short) 0, (short) 2);
      // set instructionbyte
      byte[] theval = startto_msg.getPart((short) 1)
          .getValue();
      theval[0] = (byte) 0;
      theval[1] = (byte) 2; // startto instruction
      // encrypt
      enc_msg.setEncrypted(key, startto_msg);
      return enc_msg;
}

/**
 * get the insbyte from a well formed document, insbyte
 * is returned only if insstruction is possible in
 * current state
 *
 * @param d
 *              well formed document
 * @return insbyte
 */
private byte getInsByte(Document d) {
      byte ins = 0;
      if (d == null)
        return ins;
      if (d.is_doclist())
        ins = d.getPart((short) 1).getValue()[1];
      else ins = d.getValue()[1];
      if (((ins == INS_START_FROM || ins == INS_START_TO) && state == STATE_IDLE)
          || (ins == INS_REQ && state == STATE_EPR)
          || (ins == INS_VAL && state == STATE_EPV)
          || (ins == INS_ACK && state == STATE_EPA))
        return ins;
      else if (ins == INS_GET_BAL || ins == INS_GET_DATA
          || ins == INS_GET_STATE)
        return ins;
```

```
        else return (byte) 0;
}

/**
 * check is name equals msgna
 *
 * @param msgna
 *            input
 * @return true if names are equal
 */
private Document checkName(Document dmsgna) {
        if (!(dmsgna.is_intdoc()))
          return null;
        byte[] theval = dmsgna.getValue();
        // check assures that msgna is authentic, because an
        // authentic pursename must have 8 digits
        // and must be positive.
        if (theval.length != 8)
          return null;
        if (theval[0] < 0)
          return null;
        if (!Document.comparison.equals(theval, name))
          return dmsgna;
        else return null;
}

/**
 * check if balance-value<0
 *
 * @param value
 *            the value
 * @return the value or -1
 */
private short checkBalanceMinus(Document dvalue) {
        if (!(dvalue.is_intdoc()))
          return -1;
        byte[] theval = dvalue.getValue();
        if (!(theval.length == (short) 2))
          return -1;
        short value_short = Util.getShort(theval, (short) 0);
        if (balance < value_short || value_short < 0)
          return -1;
        else return value_short;
}

/**
 * check if balance+value <32767
 *
 * @param value
 *            the value
 * @return the value or -1
 */
private short checkBalancePlus(Document dvalue) {
        if (!(dvalue.is_intdoc()))
          return -1;
        byte[] theval = dvalue.getValue();
        if (!(theval.length == (short) 2))
          return -1;
```

```
        short value_short = Util.getShort(theval, (short) 0);
        if (value_short < 0
            || (short) (balance + value_short) < (short) 0)
          return -1;
        else return value_short;
}

/**
 * check is seqno of other purse is intdoc and >0
 *
 * @param seqno
 *            seqno of other purse
 * @return seqno or -1
 */
private short checkSeqNoOtherPurse(Document seqnoother) {
        if (!(seqnoother.is_intdoc()))
          return -1;
        byte[] theseqno = seqnoother.getValue();
        if (!(theseqno.length == (short) 2))
          return -1;
        short seqno_s = Util.getShort(theseqno, (short) 0);
        if (seqno_s < 0)
          return -1;
        else return seqno_s;
}

//
// ***************************************************************
//
// functions from paper "the mondex challange"
//
// ***************************************************************
//
/**
 * check which function to call
 *
 * @param indoc
 *            received document (with "ins"-byte in
 *            first document -> intdoc)
 * @return Document or null
 */
public void step() {
        Document outdoc = null;
        Document indoc = null;
        // check if there is a document in the inbox
        if (comm.available())
          indoc = comm.receive();
        else return;
        // nothing received
        if (indoc == null)
          return;
        // no memory available
        if (exLogCounter == exLog.length)
          return;
        indoc = checkIndoc(indoc);
        switch (getInsByte(indoc)) {
        case INS_START_FROM:
          outdoc = startFrom(indoc.getPart((short) 2));
```

```
        break;
      case INS_START_TO:
        outdoc = startTo(indoc.getPart((short) 2));
        break;
      case INS_REQ:
        outdoc = req(indoc.getPart((short) 2));
        break;
      case INS_VAL:
        outdoc = val(indoc.getPart((short) 2));
        break;
      case INS_ACK:
        ack(indoc.getPart((short) 2));
        break;
      case INS_GET_BAL:
        outdoc = getBalance();
        break;
      case INS_GET_DATA:
        outdoc = getData();
        break;
      case INS_GET_STATE:
        outdoc = getState();
        break;
      default:
        abort();
        break;
      }
      // send doc if outdoc is set
      if (outdoc != null)
        comm.send(outdoc);
}

/**
 * abort method change next sequenceno, set state to
 * idle and log (if needed) dynamic memory allocation!
 */
private void abort() {
      if (state == STATE_EPA || state == STATE_EPV) {
        // save pds
        Document[] docs = pd.getDocs();
        byte[] oldfromname = docs[0].getValue();
        byte[] oldfromseqno = docs[1].getValue();
        byte[] oldtoname = docs[2].getValue();
        byte[] oldtoseqno = docs[3].getValue();
        byte[] oldvalue = docs[4].getValue();
        byte[] fromseqno = new byte[2];
        Util.arrayCopy(oldfromseqno, (short) 0, fromseqno,
                (short) 0, (short) 2);
        IntDoc fromseqnoi = new IntDoc(fromseqno);
        byte[] toseqno = new byte[2];
        Util.arrayCopy(oldtoseqno, (short) 0, toseqno,
                (short) 0, (short) 2);
        IntDoc toseqnoi = new IntDoc(toseqno);
        byte[] thevalue = new byte[2];
        Util.arrayCopy(oldvalue, (short) 0, thevalue,
                (short) 0, (short) 2);
        IntDoc valuei = new IntDoc(thevalue);
        byte[] fromname = new byte[8];
        Util.arrayCopy(oldfromname, (short) 0, fromname,
```

```java
                (short) 0, (short) 8);
        IntDoc fromnamei = new IntDoc(fromname);
        byte[] toname = new byte[8];
        Util.arrayCopy(oldtoname, (short) 0, toname,
                (short) 0, (short) 8);
        IntDoc tonamei = new IntDoc(toname);
        exLog[exLogCounter++] = new Doclist(
                new Document[] { fromnamei, fromseqnoi, tonamei,
                    toseqnoi, valuei });
    }
    // nextSeqNo();
    state = STATE_IDLE;
}

/**
 * startfrom method check received document and save
 * paydetails. Move to state EPR
 *
 * @param indoc
 *            received document
 * @return null
 */
private Document startFrom(Document indoc) {
    // get indocs
    Document dmsgna = checkName(indoc.getPart((short) 1));
    if (dmsgna == null)
      return null;
    short value_short = checkBalanceMinus(indoc
        .getPart((short) 2));
    if (value_short == -1)
      return null;
    short nextSeqNoToPurse = checkSeqNoOtherPurse(indoc
        .getPart((short) 3));
    if (nextSeqNoToPurse == -1)
      return null;
    short seqno = nextSeqNo();
    if (seqno < 0)
      return null;
    // paydetails, state
    mkpd(name, seqno, dmsgna.getValue(), nextSeqNoToPurse,
        value_short);
    state = STATE_EPR;
    // generate starttomsg
    return generate_Startto_msg();
}

/**
 * startto method check received document and save
 * paydetails. Generate a req-message and move to state
 * EPV
 *
 * @param indoc
 *            received document
 * @return req message
 */
private Document startTo(Document indoc) {
    // get indocs
    Document msgna = checkName(indoc.getPart((short) 1));
```

```java
        if (msgna == null)
          return null;
        short value_short = checkBalancePlus(indoc
            .getPart((short) 2));
        if (value_short == -1)
          return null;
        short nextSeqNoFromPurse = checkSeqNoOtherPurse(indoc
            .getPart((short) 3));
        if (nextSeqNoFromPurse == -1)
          return null;
        short seqno = nextSeqNo();
        if (seqno < 0)
          return null;
        // paydetails, state and outmsg
        mkpd(msgna.getValue(), nextSeqNoFromPurse, name, seqno,
            value_short);
        state = STATE_EPV;
        Document d = generate_ReqValAck_msg(INS_REQ);
        return d;
}

/**
 * req method check received document and generate a
 * val-message. Change balance and move to state EPA
 *
 * @param indoc
 *            received document
 * @return val message
 */
private Document req(Document indoc) {
        if (!pd.equals(indoc))
          return null;
        balance = (short) (balance - Util.getShort(pd.getPart(
            (short) 5).getValue(), (short) 0));
        state = STATE_EPA;
        return generate_ReqValAck_msg(INS_VAL);
}

/**
 * val method check received document and change
 * balance. Move to state IDLE.
 *
 * @param indoc
 *            received document
 * @return ack message
 */
private Document val(Document indoc) {
        if (!pd.equals(indoc))
          return null;
        balance = (short) (balance + Util.getShort(pd.getPart(
            (short) 5).getValue(), (short) 0));
        state = STATE_IDLE;
        return generate_ReqValAck_msg(INS_ACK);
}

/**
 * ack method
 *
```

```
     * @param indoc
     *              received document
     * @return null
     */
    private void ack(Document indoc) {
        if (!pd.equals(indoc))
            return;
        state = STATE_IDLE;
    }

    /**
     * return the actual state of the purse
     *
     * @return state
     */
    private Document getState() {
        byte[] theval = balstate_msg.getValue();
        theval[0] = 0;
        theval[1] = state;
        return balstate_msg;
    }

    /**
     * return the seq no and the name of the purse used in
     * the next transaction
     *
     * @return seqno
     */
    private Document getData() {
        Document[] thedocs = getdata_msg.getDocs();
        byte[] nameval = thedocs[0].getValue();
        byte[] seqnoval = thedocs[1].getValue();
        Util.arrayCopy(name, (short) 0, nameval, (short) 0,
            (short) 8);
        Util.setShort(seqnoval, (short) 0, sequenceNo);
        return getdata_msg;
    }

    /**
     * return the current balance of the purse
     *
     * @return bal
     */
    private Document getBalance() {
        Util.setShort(balstate_msg.getValue(), (short) 0,
            balance);
        return balstate_msg;
    }
}
```

Bibliography

[Bör03] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15 (1–2):237–257, November 2003.

[BR95] E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. North-Holland, Amsterdam, 1995.

[BS03] Egon Börger and Robert F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[Car94] Ulf Carlsen. Generating formal cryptographic protocol specifications. In *IEEE Symposium on Research in Security and Privacy*, pages 137–146. IEEE Computer Society, 1994.

[GHRS06] Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. Developing Provably Secure M-Commerce Applications. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.

[GSR06a] Holger Grandy, Kurt Stenzel, and Wolfgang Reif. A refinement method for java programs. Technical Report 2006-29, University of Augsburg, December 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[GSR06b] Holger Grandy, Kurt Stenzel, and Wolfgang Reif. Refinement of Security Protocol Data Types to Java. In *PASSWORD Workshop 2006 at ECOOP 2006*, Nantes, France, 2006. URL: `http://research.ihost.com/password/`.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford Univ. Press, 1995.

[HGRS05] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Security Protocols: An ASM Approach. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th Int. Workshop on Abstract State Machines, ASM 05*. University Paris 12 – Val de Marne, Créteil, France, March 2005.

[HSGR06] Dominik Haneberg, Gerhard Schellhorn, Holger Grandy, and Wolfgang Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. Technical Report 2006-32, University of Augsburg, December 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[KIVa] Web presentation of the mondex case study in KIV. URL: `http://www.informatik.uni-augsburg.de/swt/projects/mondex.html`.

[KIVb] KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.

[MCI] MasterCard International Inc. *Mondex*. URL: `http://www.mondex.com`.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.

[Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://www.jucs.org`.

[Sch05] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.

[SCW00] S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. URL: `http://www-users.cs.york.ac.uk/~susan/bib/ss/z/monog.htm`.

[SGH+06] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Möbius, and Wolfgang Reif. A systematic verification Approach for Mondex Electronic Purses using ASMs. Technical Report 2006-27, Universität Augsburg, 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[SGH+07] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Möbius, and Wolfgang Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In U. Glässer J.-R. Abrial, editor, *Proceedings of the Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, LNCS. Springer, 2007. (submitted).

[SGHR06] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, and Wolfgang Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods 2006, Proceedings*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.

[SGR06] Kurt Stenzel, Holger Grandy, and Wolfgang Reif. Reasoning about Pointer Structures in Java. Technical Report 2006-30, University of Augsburg, December 2006. URL: `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/`.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Ste04] Kurt Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[Ste05] Kurt Stenzel. *Verification of Java Card Programs.* PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, 2005.

[Woo06] Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.