Verification of Mondex electronic purses with KIV: from a security protocol to verified code

Holger Grandy, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Grandy, Holger, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, and Wolfgang Reif. 2008. "Verification of Mondex electronic purses with KIV: from a security protocol to verified code." In *FM 2008: Formal Methods: 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, proceedings*, edited by Jorge Cuellar, Tom Maibaum, and Kaisa Sere, 165–80. Berlin: Springer. https://doi.org/10.1007/978-3-540-68237-0_13.

Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code

Holger Grandy, Markus Bischof, Kurt Stenzel,
Gerhard Schellhorn, and Wolfgang Reif

{grandy,stenzel,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. We present a verified JavaCard implementation for the Mondex Verification Challenge. This completes a series of verification efforts that we made to verify the Mondex case study starting at abstract transaction specifications, continuing with an introduction of a security protocol and now finally the refinement of this protocol to running source code. We show that current verification techniques and tool support are not only suitable to verify the original case study as stated in the Grand Challenge but also can cope with extensions of it resulting in verified and running code. The Mondex verification presented in this paper is the first one that carries security properties proven on an abstract level to an implementation level using refinement.

1 Introduction

The Mondex [22] case study is a significant contribution to the Verified Software Repository [4] [37] which has its origin in the Grand Challenge in Software Verification [18]. Mondex is an electronic purse application for smart cards. It was originally implemented by Mastercard and became famous for being one of the first applications to be verified according to the highest criteria of ITSEC [8]. The challenge is the machine assisted verification of Mondex smart cards security properties. It was first done by paper and pencil proofs by Stepney, Cooper and Woodcock [34]. A lot of groups recently showed that their verification tools and methods can cope with the case study (e.g. [25] [2] [17] [38] [20]). Some small errors were found in the original case study. Our group also solved the challenge in [30] and [29] using Abstract State Machines (ASM) [6] and ASM Refinement [5] [26] [27] with the interactive theorem prover KIV [1]. Furthermore, we extended the case study by introducing a suitable cryptographic security protocol in [15], while the original specifications do not deal with explicit cryptography, but only assume messages to be unforgeable. We also introduced an UML-based modelling framework for security protocols in general and used it to model Mondex in [24].

In this paper we adopt our refinement method for security protocol implementations (already presented in [12]), to the verification of a Mondex implementation. The code we are verifying is running on Java smart cards [35]. Besides the original Mondex challenge, this paper addresses especially the problems of

Data Abstraction and of Complex Heap Data Structures as stated in a current verification challenge [21].

We prove that our implementation preserves the Mondex security properties. First, security is proved for the abstract levels. Then, the refinement theory carries the properties over to the concrete level. All proofs and the implementation are available on the Web [19]. Fig. 1 shows an overview of our specification and verification layers for Mondex. The levels 1 and 2 are the A and C levels of the original case study, using ASMs as the specification language. The third layer introduces cryptography and was not present in the original specification. Level 4 is the JavaCard implementation level. The refinement from level 3 to 4 is the main content of this paper.

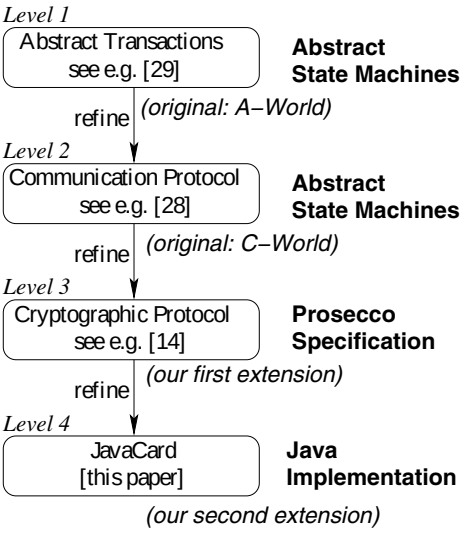

Fig. 1. Our Mondex layers

Sect. 2 will introduce Mondex, Sect. 3 describes the source code. Sect. 4 introduces Java in KIV and the refinement framework, Sect. 5 explains the proof strategy. Sect. 6 compares the approach to related work.

2 Mondex in a Nutshell

Mondex smart cards are electronic purses, that store an amount of money. They can be used to pay by transferring money from one purse (called the FROM Purse) to another one (the TO Purse). Those transactions are assumed to be possibly faulty. While the most abstract level in the case study only uses nondeterministic choice between successful money transfer or loss of money (both possible in one atomic step), the first refinement introduces a protocol using five different messages, together forming a transaction. All messages can be lost during transmission, thereby leading to an error on any of the cards in any state. Level 2 also uses an ether of messages which are currently in transit. Receiving a message is basically taking an arbitrary one out of the ether. Thereby replay attacks are modeled by adding the possibility of taking the same message out more than once. Besides that, there is no explicit attacker analyzing messages or generating new ones. Also there is no cryptography on this level: all messages are assumed to be unforgeable. Any error during one protocol run (like receiving a replayed message) leads to logging of the current transaction. The first refinement shows that the log entries correctly represent the lost money on level 1. This was shown e.g. in [30].

The original verification of Mondex ends at our level 2. Level 3 now introduces cryptography for this protocol using the Prosecco specification approach [14] [13], which is also based on Abstract State Machines. We chose this approach because Prosecco already provides lots of specification libraries for security protocols and is well integrated into the KIV system. Prosecco contains an explicit Dolev-Yao attacker [9] who is analyzing and building messages. The ether of level 2 is now modelled using explicit input queues. Also additional participants (like the terminal or the card holder) are introduced to get closer to reality. We proved that this protocol is a correct refinement of level 2 [29]. A symmetric secret key shared between all the authentic purses is used to encrypt most messages and thereby ensures that the attacker cannot generate forged critical messages. Some details on level 2 and level 3 were already given in [15]. Our Java implementation of Mondex on level 4 is a correct refinement of the abstract ASM specification on level 3.

The protocol of level 3 and 4 is shown in Fig. 2. After the purses' data was queried, the two messages STARTTO and STARTFROM set up the transaction by sending one purse the data of the other. Every purse has a name and a sequence number. The latter is used to avoid replay attacks. The first messages establish a transaction context called PayDetails, consisting of both purse names, both sequence numbers and the money in transfer. The messages REQ(uest), VAL(ue), and ACK(knowledge) are used to transfer money. All messages contain the PayDetails of the current transaction. The FROM purse withdraws money from its balance after receiving a correct REQ and before sending VAL, whereas the TO Purse deposits the same amount after receiving VAL and before sending ACK. Sending STARTTO as a response to STARTFROM is a slight modification of the original protocol of [34], because we found a possible attack in [29]. Another modification is the addition of an explicit getData message to query the sequence number, the name and the current balance of a purse, which is necessary in a realistic environment. Furthermore the messages STARTTO, REQ, VAL and ACK are secured by symmetrically encrypting them with a shared secret key (also shown by italic font and prefix enc in Fig. 2). STARTFROM and getData messages are not encrypted. Those

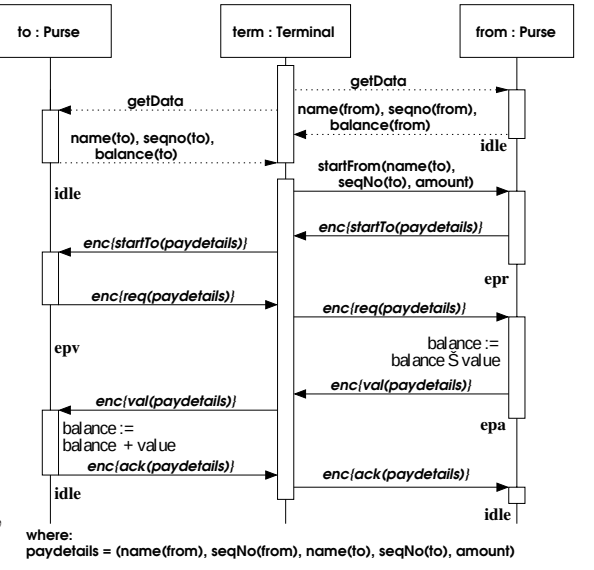

Fig. 2. The Mondex Protocol

protocol modifications and extensions do not incur any problems with the security properties since we proved that this is a correct refinement of level 2 [29].

3 An Implementation of Mondex

3.1 Data Types and Communication

One of the main problems for a correct refinement from an abstract specification is the correct mapping of data types from specification to implementation. Level 3 uses a algebraic data type called Document with various subtypes for modelling messages. For Mondex, we will need the following ones: A Document is either (an empty document), an IntDoc representing an arbitrary large integer, or the result of cryptographic encryption of a document doc using the cryptographic key key (i.e. EncDoc(key, doc)). Arbitrary protocol data (like the purse's name or sequence number) is modeled using the IntDoc type. Messages in communication protocols are composed of those basic data types. To model composition the Document type also contains the Doclist constructor, which contains a list of documents (Documentlist). The Documentlist type itself can either be the empty list [ ] or a composition of a Document and another Documentlist. This gives a mutual recursive algebraic specification of a free data type Document:

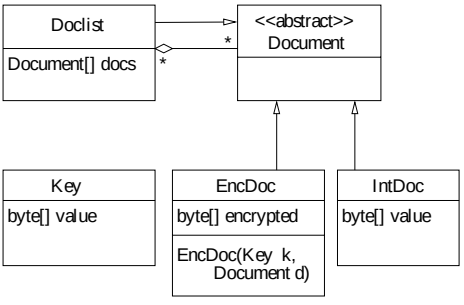

Fig. 3. Mondex Document classes

$$\begin{aligned}
\mathsf{Document} = \quad & | \\
& \mathsf{IntDoc}(\mathsf{value} : \mathsf{int}) \mid \\
& \mathsf{EncDoc}(\mathsf{key} : \mathsf{Key}, \mathsf{doc} : \mathsf{Document}) \mid \\
& \mathsf{Doclist}(\mathsf{docs} : \mathsf{Documentlist}) \\
\mathsf{Key} = \mathsf{mkKey}(\mathsf{value} : \mathsf{int}) & \\
\mathsf{Documentlist} = \quad & [\,] \mid .+.(\text{“}\mathsf{rst} : \mathsf{Document}, \mathsf{rest} : \mathsf{Documentlist})
\end{aligned}$$

The implementation uses a Java class type Document to implement this abstract type, resulting in the classes of Fig. 3. We use byte[] as the representation of the integer values in the abstract type. Those classes and the mapping from the abstract to the concrete world was described in more detail in [11].

We implement a communication interface, which contains send and receive methods for the Document type. This is the SimpleComminterface:

```
1  public interface SimpleComm {
2      public Document receive();
3      public void send(Document d);}
```

An embedding of this SimpleComminterface into JavaCard will be introduced in Sect. 3.3

3.2 Purse Functionality

Using the communication interface, the Purse functionality benefits from the high-level Java class type Document instead of low-level byte sequences. The resulting implementation works as follows[1]:

```
1  public class Purse{
2    SimpleComm comm; Document payDetails, name;
3    short exLogCounter; Document[] exLog;
4    ...
5    public void step() {
6      Document outdoc = null;
7      if(exLogCounter < exLog.length){
8        Document indoc = comm.receive();
9        ... //decrypt and check indoc
10       switch(getInsByte(indoc)) {
11         case START_FROM: outdoc = startFrom(indoc); break;
12         ... // same for other steps
13         case ACK:                ack(indoc);       break;
14         default:                 abort();          break;}
15       if(outdoc!=null) comm.send(outdoc);}}}
```

The Purse class uses the SimpleComm.receive() method to receive the next input (line 8). Since in a smart card implementation the exception log must have a bounded length (it is unbounded in the original case study, the refinement to bounded lengths is done from level 2 to 3), it is first checked, whether the log is already full (line 7). The exception log is implemented using a field Document[] exLog of class Purse. We use an additional field exLogCounter, which stores the index of the next free exception log entry. A full log can be checked by comparing this field to the maximum log length constant. If the log is full, no further step is performed (the restriction to fixed length exception logs itself is already introduced on the abstract level 3). If space is available, the input document structure is decrypted (if necessary, line 9) and its structure is checked. Level 3 already introduced a document format for all the Mondex messages of Fig. 2. For example, the ACK message is:

$$
\left.\begin{array}{l}
\mathsf{EncDoc}(\\
\quad \mathsf{mkKey}(\mathsf{THESECRETKEY}), \qquad \left.\begin{array}{l}\\ \end{array}\right\} \text{Crypto Key} \\
\quad \mathsf{Doclist}(\\
\qquad \mathsf{IntDoc}(\mathsf{ACK}) \qquad \left.\begin{array}{l}\\ \end{array}\right\} \text{Type Flag} \\
\qquad + \mathsf{Doclist}(\quad \mathsf{IntDoc}(\mathsf{name}_{\mathsf{from}}) + \mathsf{IntDoc}(\mathsf{seqno}_{\mathsf{from}}) \\
\qquad\qquad + \mathsf{IntDoc}(\mathsf{name}_{\mathsf{to}}) + \mathsf{IntDoc}(\mathsf{seqno}_{\mathsf{to}}) \\
\qquad\qquad + \mathsf{IntDoc}(\mathsf{amount})))) \qquad \left.\begin{array}{l}\\ \\ \\ \end{array}\right\} \text{Pay Details}
\end{array}\right\} \text{Encrypted Message}
$$

The other messages (except STARTFROM, which is slightly shorter) have the same structure. This structure maps directly to Java Documents.

[1] For the sake of readability we slightly pretty printed the programs for this paper. The original verified programs can be found on the web [19].

After checking the input structure, the getInsByte method (line 9 above) returns the type of the input message and the correct protocol step method (e.g. ack(...), line 13) is chosen. The ack() method now has to check whether the PayDetails are correct and that it is no replayed message from an older protocol run. Since the PayDetails are also implemented using the Document type (using a field Document payDetails in class Purse), we can do this by calling a generic equals Method on Document (line 3 below):

```
1  private void ack(Document indoc) {
2    indoc = getPaydetails(indoc);
3    if(!payDetails.equals(indoc)){ abort(); return; }
4    state = STATE_IDLE;}
```

The abort() method now only has to check whether the current state is critical (line 2 below). Money can only be lost if the TO Purse is in state EPV or the FROM Purse is in state EPA (see Fig. 2). If we are in a critical state, the current PayDetails have to be copied to the exception log exLog (using the method copyLogPDs, line 3):

```
1  private void abort() {
2   if(state==STATE_EPA || state == STATE_EPV) {
3     exLog[exLogCounter] = copyLogPDs();
4     exLogCounter++;}
5   state = STATE_IDLE;}
```

ack() is relatively short. In contrast, the startFrom() method has to perform more checks, since it has to set up the PayDetails correctly:

```
1  private Document startFrom(Document indoc) {
2    Document othername = checkName(indoc);
3    short value = checkBalance(indoc);
4    short otherSeqNo = checkSeqNo(indoc);
5    if(   32767 == sequenceNo || otherSeqNo == -1
6       || value == -1 || othername == null) {
7     abort(); return null;}
8    if(state != STATE_IDLE) abort();
9    if(exLogCounter < exLog.length) {
10    setPaydetails(name,sequenceNo,othername,
11                  otherSeqNo,value);
12    sequenceNo++; state = STATE_EPR;
13    return generateOutmsg(START_TO);}
14   else return null;}
```

First it has to check, whether the transmitted name of the TO Purse is authentic and different from the FROM purse name (in this implementation all names with a length of 8 bytes are authentic[2]). This is checked in the method

[2] Authenticity of names is a concept introduced in the abstract levels of Mondex, used to distinguish real Mondex cards from faked ones. The original case study does not state which names are authentic. Note that a check of authenticity of names using e.g. cryptographic signatures does not add any security to the Mondex application. All security of Mondex is based on the encryption and the sequence numbers.

checkName (line 2). Also the amount of money to be deposited must be positive and smaller than the current balance (checkBalance, line 3) and the transmitted sequence number of the other purse must be reasonable (checkSeqNo, line 4). Since the sequence number is a short value, we only allow a further protocol run, if the maximum sequence number (32767) is not yet reached. This restriction was already made on the abstract Prosecco level 3. The three check methods return an error value (null or -1), if anything is wrong. In those cases, we simply abort and stop (line 7). If we receive a STARTFROM message when we are in not in the IDLE state, we abort and continue afterwards (line 8). This is an extension of the original case study where STARTFROM is only accepted in IDLE. The approach of the original case study would have the negative effect that every new transaction started directly after a previously interrupted transaction would fail, too. This is not what a user expects in reality. If all parameter checks are successful, we store the PayDetails, increment the sequence number and generate a STARTTO return message (lines 10 to 13).

3.3 Embedding in Javacard

The implementation of Mondex runs on Java smart cards. Those cards communicate with APDUs (Application Protocol Data Units), which are essentially byte arrays. To use Java Document classes on Java smart cards we provide a transformation layer that encodes and decodes instances of the Document classes to byte sequences and sends them over the APDU interface. This can be combined with certain checks on the structure of incoming and outgoing messages. An attacker generating for example non well-formed APDU messages has no chance of attacking the protocol, if all those invalid inputs are filtered out in a transformation layer before even starting with the real protocol functionality. We have implemented and proved correct such a transformation layer in [11] for normal network communication and adopted it here to the use on Java smart cards.

The Purse class uses the SimpleComminterface to communicate. The embedding of SimpleCommin the JavaCard world is is done schematically as described in the following code:

```
1  public class PurseAppletWrapper
2    extends javacard.framework.Applet implements SimpleComm{
3      Document input; Purse protocolimpl;
4      public void process(APDU apdu){
5          ...
6          input = decode(apdu.getBuffer());
7          protocolimpl.step();}
8      public Document receive(){ return input; }
9      public void send(Document d){
10         byte[] outbytes = encode(d);
11         ... //copy outbytes to apdu buffer and send them }
12     //for an implementation of encode and decode see [11]
13     private byte[] encode(Document d){...}
14     private Document decode(byte[] b){...}}
```

In JavaCard, every execution of a protocol step on the card is started by a call of the `Applet.process(APDU apdu)` method (line 4). Therefore the byte array input, which is wrapped in the parameter `apdu` of this method, is decoded into a Java `Document` pointer structure (line 6). Then a `step()` Method on the `Purse` implementation class is called (line 7). This method uses `receive()`, which returns the decoded input `Document`. It will also eventually call `send(Document)`, which encodes the output and sends it over the APDU interface (line 10-11).

Additionally, the encoding scheme can be used to deal with cryptography (not shown above), which is inherently necessary for Mondex: If we want to encrypt a `Document` object with a certain key, we can simply encode it to an array of bytes using the encoding scheme and then use the standard Java cryptographic architecture to encrypt this array using the given key's value. This is also the reason why our `EncDoc` class contains an array of bytes as the encrypted value. The only assumption we have to make here is the standard assumption about perfect cryptography used in almost all approaches to security protocol verification: cryptography can only be broken when knowing the right key.

All together, the `Purse` class implementation consists of over 600 lines of code, not counting the various `Document` classes and the encoding/decoding implementation. All classes verified for this case study have around 1800 lines, which is quite a large number for `interactive` source code verification.

4 Refinement Method

We described our refinement framework for Java protocol implementations using another case study in [12]. The method is based on the Java calculus in KIV [32] [33]. For the Mondex verification, ASM Refinement theory [5] is used in a variant which is preserving invariants over the refinement [27] [3]. Every method call of the `step` method in the implementation corresponds to exactly one step of the abstract specification. Our refinement approach consists of the following steps (described in more detail in [12]):

1. `Specify the Implementation Level (Level 4, Fig. 1)`: First specify the implementation level as a copy of the abstract `Prosecco` level. Then replace the part of the abstract specification dealing with the Mondex purse steps with a call of the Java `Purse.step()` method. Further replace the abstract initialization with a constructor call for `Purse`. This is possible within KIV, since both ASM Verification and Java Verification are based on the same logical background framework, Dynamic Logic (DL) [16] and algebraic specifications. ASMs are modeled using the programs of DL , Java Verification is done by extending the program operators of DL by introducing an explicit memory model for the heap, as described later. So we can add

[3] In [12], standard Data Refinement theory is used. ASM Refinement is used in all the other levels of our Mondex refinements, and ASM Refinement was shown to be a generalization of Data Refinement [28]. So, technically this does not make any difference, because all our proof obligations are standard 1:1 refinement properties.

the heaps of the Java purses as an additional state function to the ASM. Then we execute the Java implementation using those heaps to define the concrete agents' behaviour.

2. Data Transformation for Inputs: Insert a data transformation function from abstract input Documents to Java objects before the actual call of the Java implementation Purse.step().
3. Data Transformation for Outputs: Insert a data transformation function from Java objects to abstract output Documents after the actual call of the Java implementation Purse.step().
4. Simulation Relation: Find a simulation relation R, that maps the abstract Prosecco state to the concrete Java state using data transformation functions that are similar to input/output transformation. Additionally, find suitable invariants for the abstract and concrete levels.
5. Prove Initialization: Prove that the Java constructor call of Purse leads to a Java state, where a corresponding initial abstract state can be found in which the simulation relation holds.
6. Prove Correctness: Prove that if the simulation relation holds and if we execute a sequence of data input transformation to Java, call of Purse.step() and data output transformation back to Documents, we then find a step of the abstract ASM purse specification which results in a state where the simulation relation holds again.

We will describe these steps in more detail now. The state of the Prosecco ASM is given by different state functions, which map an agent to some data, where agent is a free data type specifying the protocol participants:

Agent = purse(int : name) | terminal | user(int : name) | attacker

For the Mondex purses, the level 3 ASM specification uses the state functions:

inputs : agent	documentlist	input messages of each agent
seqNo : agent	int	current sequence numbers
balance : agent	int	current balances
payDetails : agent	Document	current transaction details
exLog : agent	Documentlist	current exception logs

To define the ASM rules for the agents on level 3, we use macro definitions MACRO#(input; output) with input parameters input and input/output parameters output. For an agent representing a purse, the rule is:

```
1  PURSE#(agent, ... ;inputs, exLog, balance, ...)
2     if inputs(agent) ≠ [] ∧ #exLog(agent) < MAXLENGTH
3     then let indoc = inputs(agent).first in
4      inputs(agent) := rest(inputs(agent));
5          //check the input and decrypt
6          ...
7          if is_startfrom(indoc) then STARTFROM#(...)
8          else if ... then ...
9          if is_ack(indoc) then ACK#(...)
10         else if insbyte = 0 then ABORT#(...);
11    SEND#(outdoc, ...;inputs)
```

The abstract specification first selects the next available input from the inputs state function (if one is available and the log is not yet full, line 2 and 3). Then the inputs state function is updated accordingly (line 4), and the received Document is cut off (rest(...)). After checking the input message, a case distinction over the type of the message is performed, and the matching ASM Macro for that protocol step is executed (just as in the implementation).

Now we have to define the concrete specification layer for the refinement. The purse ASM rule is now substituted with a Java implementation. Since no other agent protocol definitions are modified, and since those other definitions still use the Documents to communicate, we keep the inputs state function on the concrete level. We introduce data transformation functions from and to the Java world before and after our protocol step. This will be done by Macros TOSTORE and FROMSTORE. TOSTORE takes an input document from inputs and transforms it into the Java world, FROMSTORE does the inverse. For this, we have to take a look at how the state of Java programs is modeled in KIV [32] [33]. Since KIV is a very elaborated system for the verification using Dynamic Logic and algebraic specifications, we have a huge library of algebraically specified data types. The state of Java programs is modeled explicitly using an algebraic data type in KIV, too. This is the store data type. A store defines a mapping from a tuple of a reference (a pointer to an object or array) and a field (a field of a class or an array index) to a Java value. A Java value can be a primitive value like an int or short, or a reference representing a pointer to another object. This allows representation of arbitrary pointer structures. We write st[r.f] for the access to field f of reference r in store st.

Now back to refinement, we store the states of the Java purses using another state function cstore : agent store. The functions seqNo, balance, ... of the abstract level are not present on the refined level, since their values are contained in the corresponding fields of the Purse inside the store. Formally, we have the state functions:

inputs : agent	documentlist	current input messages of each agent
cstore : agent	store	current Java heaps

Java programs are now integrated into the logic by extending the Box and Diamond operators of Dynamic Logic (shown here only for Diamond):

st;	states that Java program terminates if executed in the context of store st and afterwards formula holds

With those operators, KIV provides a sequent calculus for the complete sequential part of Java [10]. We do not perform any transformation on the Java code we verify, the running original source code is verified. With those modified DL operators, the concrete purse step is defined as:

```
1  JAVAPURSE(agent, ... ;inputs, cstore, ...)
2      TOSTORE(agent, inputs; cstore);
3      st0 := cstore(agent);
4      choose st with ⟨st0; Purse.instance.step();⟩ (st0 = st) in
5        cstore(agent) := st;
6        FROMSTORE(agent, st; outdoc);
7        SEND(outdoc, ...; inputs)
```

The current Purse object is stored in a static field Purse.instance which is set by the constructor as a Singleton. First the input from inputs(agent) is transformed into cstore using TOSTORE (line 2), then step() is called (line 4) on a Purse object in the context of the heap of that agent (which is $st_0 = cstore(agent)$, line 3). As described later, step() then calls SimpleComm.receive() to get the input and calls SimpleComm.send() to produce some output, which we transform afterwards from the store into variable outdoc using FROMSTORE (line 6). Finally, the SEND macro is used (as on the abstract level) to update the inputs function for the receiver of the document (line 7).

The next step is to define, how concrete Java states and abstract ASM states relate to each other. This is done in the simulation relation R. It defines that the Java state is the same as the corresponding abstract state. For example, for the exception log in the JavaCard program, this means that the array of exception log entries exLog from index 0 to index exLogCounter is (transformed to the abstract world) equal to the abstract exLog state function. For Document classes, a generic transformation function $java2doc : reference \times store \rightarrow Document$ is defined for this purpose. It takes a store and a reference pointing to some Document Java object and constructs the corresponding abstract Document. Using a simple recursion, this is lifted to lists of references $java2doc : referencelist \times store \rightarrow Documentlist$. Additionally, we use a function $getarray : reference \times startindex \times length \times store \rightarrow referencelist$ to extract the references contained in an array in the store. Using such functions, the property for the exception log is:

$$exLogCorrect(cstore, exLog) \Leftrightarrow (agent.is\_purse(agent) \Rightarrow java2doc(getarray(cstore(agent)[Purse.instance.exLog], 0, cstore(agent)[Purse.instance.exLogCounter], cstore(agent)), cstore(agent)) = exLog(agent))$$

Similar definitions are needed for all the different state functions. Besides such value definitions, a lot of invariants, both for the concrete and the abstract level are needed: a good example for the abstract level is the property that the exLog is always shorter or equally long as the MAXLENGTH. Additionally, all exception log entries have the correct format of PayDetails. Another example is that all purses always share the same secret key, and that the attacker never knows it.

On the concrete level, one needs the property that the exLogCounter is always smaller than the exLog.length. Here again all exception log entries have to be well-formed. This is a lot harder to formulate than on the abstract level since we are now talking about pointer structures. For example, one has to deal with properties like sharing among the pointers or cyclic structures. Those structures must be ruled out.

The whole invariant for the concrete level is way too long to be presented here. It can be viewed on the web [19]. It consists of 87 properties, all of them again divided into lots of different formulas. For example, the pure value mapping between abstract and concrete world requires 8 properties, the invariant on the abstract level 21 and the invariant on the Java level requires 58 different properties.

Using such a simulation relation now directly translates the security properties of the abstract world to the implementation. It follows directly from refinement theory that for the concrete balances and for the concrete exception log entries the same properties hold as on the abstract level. This is because the simulation relation simply states that their values are equal (modulo transformation from Java pointer structures to abstract data types). Thereby, all security properties (which are all invariants on the state) of the abstract world hold for the implementation as well.

5 Proof Strategy and Experiences

The proof strategy for the case study is symbolic execution of the Java program, extended by the use of lemmata for every method that is called on the way. The main proof obligation for correct refinement of the Purse step is (abbreviating the abstract state to astate and the concrete state to cstate): In every abstract state astate, in which the simulation relation R holds with some concrete state cstate, we have to show that if we do a step of the concrete JAVAPURSE , then a step of abstract PURSE exists, after which the simulation relation holds again with the new abstract and concrete states. In Dynamic Logic, this is:

$$R(astate, cstate) \rightarrow |JAVAPURSE(cstate)| \; PURSE(astate) \; R(astate, cstate)$$

We now can use symbolic execution for JAVAPURSE, leading to a formula that contains a sequence of TOSTORE, Purse.step and FROMSTORE. Using further symbolic execution on the step method at some point will lead to the switch case distinction of the implementation deciding which protocol method to call. Each protocol step is now treated by formulating a lemma, which discards the Java method call for the protocol functionality and the corresponding abstract specification ASM rule. Those lemmata have to state that the Java method, when given the same input, behaves the same as the corresponding abstract ASM macro. E.g. for STARTFROM we have schematically:

$$R(astate, cstate) \rightarrow \; ...// \text{ some more preconditions about structure of inputs and state}$$
$$st = cstore(agent) \;\; java2doc(in, st) = indoc \rightarrow$$
$$\langle st/Document\ out = Purse.instance.startfrom(in); \rangle$$
$$STARTFROM\#(agent, indoc; outdoc, astate)$$
$$(java2doc(out, st) = outdoc \;\; R(astate, setStore(agent, cstate, st))$$

where setStore(agent, cstate, st) updates cstate by setting the new Java store st for the given agent.

Using such lemmata the method calls in the Java program are discarded one after another together with the corresponding abstract specification. Finally the simulation relation holds on the resulting states. The same strategy is then applied to prove those lemmata themselves, meaning that for STARTFROM more lemmata about checkName or setPaydetails are formulated and proven the same way until we reach methods not containing other method calls. This is quite similar to the approach used in Design by Contract [23], but making it more specific by linking Java methods to abstract ASM program definitions as shown above.

The verification described was done using the KIV system for the protocol steps STARTFROM and ACK. Although these are only 2 of 6 different protocol steps, the verification nevertheless covered almost 85% of the total lines of code of the Purse class. This is because the other protocol steps are nearly symmetric to those two (STARTTO is nearly the same as STARTFROM, REQ and VAL are equal to ACK). So, we can state that the verification done gives a representative insight on the case study. One of the hardest parts was getting the invariant right. This is not really a matter of difficulty, more a matter of complexity, because the state of the Java program and of the abstract specification is not trivial. The most important part of the proofs was the formulation of suitable lemmata, which can divide the complexity of the overall case study into smaller parts. Especially sometimes a method called early in the program ensured properties needed late in the program, but this was not predictable in the first iteration of the proofs. All together, the Java KIV project for Mondex, consisting of the abstract specification, its invariants, the concrete specification and finally the refinement proof, took around 4200 lines of specification and 600 lines of Java code for Purse, around 1800 in total. Over 1700 theorems where formulated, which took 85000 proof steps (with an automation degree of around 70%). This is not counting any libraries, like the transformation functions for abstract Documents to Java and vice versa and basic libraries for Prosecco, refinement theory or simple data types. Nevertheless, the Mondex case study heavily accounted to the growth of those libraries too. The time needed for verifying the case study is hard to measure, but it certainly was more than half a year of verification for one person. Most of the pure verification work was done in a master's thesis.

6 Related Work

A lot of work has already been done for Mondex on more abstract specification levels as mentioned in the introduction. Since all of them are not focusing on source code verification we omit a detailed explanation of those approaches in this paper.

The most important work that is closely related to what we are presenting in this paper is the verification by Tonin and Schmitt presented in [31] and in more detail in a technical report [36]. They also claim to verify an implementation of the Mondex case study for smart cards in JavaCard. Furthermore, they state to have verified the security properties of the original Mondex case study on the code level. Their approach uses JML[7] annotations for every method and class and generates proof obligations from those annotations to be discarded using the KeY Verification System [3]. There is no abstract specification and no refinement theory. However, their implementation does not really implement the case study in the sense that one could use the code on a real smart card and it would be secure. That is because they do not use cryptography in their implementation and therefore have very strong assumptions about the environment the cards are used in. Without using cryptography, a malicious attacker can easily generate a faked smart card and use it for payment, thereby generating money. They also assume that a terminal exists which generates all the messages for the protocol

instead of generating the messages by the cards themselves. But the latter, combined with the need for cryptography in a real implementation, is postulated in the original case study. The corresponding original technical monograph [34] clearly states on p. 1 that "All security measures have to be implemented on the card" and "Once released into the field, each purse is on its own: it has to ensure the security of all its transactions without recourse to a central controller.". In the work of Schmitt and Tonin, every card only answers "OK" or "Error" to every input. Such a smart card can easily be faked. Also their formalization of the security properties of Mondex does not capture the original work in all their aspects. The property "All value accounted" (which states that no money is really lost but correctly logged in the exception logs on the cards) is formulated without using the exception logs in the formalization. They argue that this is because JML lacks the abilily to formulate causality between different operations, and argue that their formalization still captures the essence of the security property. But actually, their implementation contains a bug in the handling of the exception logs. All previous exception log entries are changed by side effect when adding a new exception log entry because there is a problem with pointer sharing. So the security property "All value accounted", which states that the exception logs really captures exactly the lost money due to failed transactions cannot hold in their implementation. They did not find that bug, because their proof obligations do not state anything about the contents of the log entries. Summarizing, in our opinion their work can be viewed as an additional specification of the Mondex case study using the Java programming language as a kind of specification language, rather than an actual implementation of it, which is usable and secure in the real world. Our aim in this work, however, was to achieve the latter.

7 Conclusion

We presented a verification of the Mondex case study starting at abstract specifications and ending at the proof of correctness and security of an implementation in Java. The result is based on several techniques ranging from refinement theories, implementation techniques, encoding and decoding of messages in the implementation, modelling and specification of security protocols on an abstract level. It is, to our best knowledge, the largest and most comprehensive approach to the Mondex case study. We succeded showing that current elaborated verification tools like KIV can cope with the challenge of verifying such applications.

References

1. Balser, M., Reif, W., Schellhorn, G., Stenzel, K.: KIV 3.0 for Provably Correct Systems. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, Springer, Heidelberg (1999)
2. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the purse: The balance enquiry quandary, and generalised and (1,1) forward refinements. Fundamenta Informaticae 77 (2006)

3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Bicarregui, J., Hoare, C.A.R., Woodcock, J.C.P.: The verified software repository: a step towards the verifying compiler. Formal Aspects Computing 18(2), 143–151 (2006)
5. Börger, E.: The ASM Refinement Method. Formal Aspects of Computing 15(1–2), 237–257 (2003)
6. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
7. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3) (2005)
8. UK ITSEC Certification Body. UK ITSEC SCHEME CERTIFICATION REPORT No. P129 MONDEX Purse. Technical report, UK IT Security Evaluation and Certification Scheme (1999)
9. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Proc. 22th IEEE Symposium on Foundations of Computer Science, pp. 350–357. IEEE, Los Alamitos (1981)
10. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, Reading (1996)
11. Grandy, H., Bertossi, R., Stenzel, K., Reif, W.: ASN1-light: A Verified Message Encoding for Security Protocols. In: Software Engineering and Formal Methods, SEFM, IEEE Press, Los Alamitos (2007)
12. Grandy, H., Stenzel, K., Reif, W.: A Refinement Method for Java Programs. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 221–235. Springer, Heidelberg (2007)
13. Haneberg, D.: Sicherheit von Smart Card – Anwendungen. PhD thesis, University of Augsburg, Augsburg, Germany (in German) (2006)
14. Haneberg, D., Grandy, H., Reif, W., Schellhorn, G.: Verifying Smart Card Applications: An ASM Approach. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 313–332. Springer, Heidelberg (2007)
15. Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. Formal Aspects of Computing 20(1) (January 2008)
16. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
17. Haxthausen, A.E., George, C., Schütz, M.: Specification and Proof of the Mondex Electronic Purse. In: 1st Asian Working Conference on Verified Software, AWCVS 2006, UNU-IIST Reports 348, Macau (2006)
18. Hoare, S.T.: The Ideal of Verified Software. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 5–16. Springer, Heidelberg (2006)
19. Web presentation of the Mondex case study in KIV. URL, `http://www.informatik.uni-augsburg.de/swt/projects/mondex.html`
20. Kong, W., Ogata, K., Futatsugi, K.: Algebraic Approaches to Formal Analysis of the Mondex Electronic Purse System. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 393–412. Springer, Heidelberg (2007)
21. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing 19(2) (June 2007)
22. MasterCard International Inc., `http://www.mondex.com`
23. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)

24. Moebius, N., Haneberg, D., Schellhorn, G., Reif, W.: A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In: International Conference on Software Engineering Advances 2007, IEEE Press, Los Alamitos (2007)
25. Ramananadro, T., Jackson, D.: Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method (2006), http://www.eleves.ens.fr/home/ramanana/work/mondex/
26. Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. Journal of Universal Computer Science (J.UCS) 7(11) (2001)
27. Schellhorn, G.: ASM Refinement Preserving Invariants. In: Proceedings of the 14th International ASM Workshop, ASM 2007, Grimstad, Norway (2007)
28. Schellhorn, G.: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. Journal of Theoretical Computer Science 336(2-3), 403–435 (2005)
29. Schellhorn, G., Grandy, H., Haneberg, D., Moebius, N., Reif, W.: A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In: Dagstuhl seminar on Rigorous Methods for Software Construction and Analysis, LNCS. Springer, to appear (older version available as Techn. Report 2006-27 at [19] (2008)
30. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 16–31. Springer, Heidelberg (2006)
31. Schmitt, P.H., Tonin, I.: Verifying the Mondex case study. In: Software Engineering and Formal Methods, SEFM, IEEE Press, Los Alamitos (2007)
32. Stenzel, K.: A Formally Verified Calculus for Full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
33. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik (2005)
34. Stepney, S., Cooper, D., Woodcock, J.: AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory (July 2000)
35. Sun Microsystems Inc. Java Card 2.2 Specification (2002), http://java.sun.com/products/javacard/
36. Tonin, I.: Verifying the mondex case study. The KeY approach. Techischer Bericht 2007-4, Fakultät für Informatik, Universität Karlsruhe (2007)
37. Woodcock, J.: First steps in the verified software grand challenge. IEEE Computer 39(10), 57–64 (2006)
38. Woodcock, J., Freitas, L.: Z/Eves and the Mondex Electronic Purse. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 15–34. Springer, Heidelberg (2006)