A Concept-Driven Construction of the Mondex Protocol using Three Refinements

Gerhard Schellhorn[1] and Richard Banach[2]

[1]Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de

[2]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk

Abstract. The Mondex case study concerns the formal development and verification of an electronic purse protocol. Several groups have worked on its specification and mechanical verification, their solutions being (as were ours previously), either one big step or several steps motivated by the task's complexity. A new solution is presented that is structured into three refinements, motivated by the three concepts underlying Mondex: a message protocol to transfer money over a lossy medium, protection against replay attacks, and uniqueness of transfers using sequence numbers. We also give an improved proof technique based on our theoretical results on verifying interleaved systems.

1 Introduction

Mondex smart cards implement an electronic purse [1]. They were the target of one of the first ITSEC evaluations at level E6 [2] (now EAL7 of Common Criteria [3]), which requires formal specification and verification. The formal specifications were given in [4] using Z [5], together with manual correctness proofs. Two models of electronic purses were defined: an abstract one which models the transfer of money between purses as elementary transactions, and a concrete level that implements money transfer using a communication protocol that can cope with lost messages using a suitable logging of failed transfers.

Mechanizing the security and refinement proofs of [4] was recently proposed as a challenge for theorem provers (see [6] for more information on the challenge and its relation to 'Grand Challenge 6'). Several groups took up the challenge. For a survey see [7] — more details on some are given in Section 7. Results to date have been focused on solving the problem either as closely as possible to the original, or by adapting the problem to fit the style of the tool, thereby simplifying it.

The first author works in the Augsburg group, which uses KIV. This has derived solutions for both styles. [8] gives a solution that formalizes the original data refinement theory of [9] and uses the original backward simulation. Alternatively, since KIV supports the Abstract State Machines (ASM, [10], [11]) style of specifying operations, we have also given a solutions using ASMs in [12], as one refinement that uses generalized forward simulations of ASMs ([13], [14], [15], [16], [17]). This solution simplified

the deduction problem by using purse-local invariants (inspired by [18]), and by using big commuting diagrams for full protocol runs, a technique used previously in ASM refinements. This approach also uncovered a weakness of the original protocol, which can be resolved by a small change. Still, the proof is monolithic, consisting of a single refinement.

Other authors, particularly [19] and [20], have tried to modularize the refinement into several to make deduction simpler, but from our point of view they have not isolated the Mondex *concepts* into separate refinements, allowing a clean explanation. However, their work has strongly influenced ours.

Isolating the Mondex concepts is a necessity when explaining the Mondex protocol live to an audience. This prompted the attempt to formalize them as separate refinements. The essential concepts of the Mondex protocol are the following:

- Implementing transfers by sending messages over a lossy transport medium.
- Adding checks that protect against replay attacks.
- A challenge-response system to ensure uniqueness of protocol runs.
- Choosing suitably strong cryptographic functions to encrypt messages.

This paper explains the first three concepts by putting them into three successive refinements. The fourth was absent in the original Mondex work: the Mondex concrete level *assumes* that suitable cryptography can be used to protect messages. Elsewhere [21], we have shown that suitable cryptography can indeed be added using another refinement, and that as an instance of a model-driven approach [22] the resulting ASM can be implemented using Java [23], so we do not repeat this here.

The next section recalls the Mondex abstract specification, and we then explain each of the refinements in turn in the following three sections. We also explain some of the simulation relations and invariants that are needed to verify each refinement with KIV (full details of all specifications and proofs are available at [24]). A final technical refinement, that slightly changes notation to be compatible with the original definitions completes the development. Finally, Section 7 gives related work and Section 8 concludes.

2 The Abstract Specification

The main protocol of Mondex implements electronic cash transfer, using either a device (wallet) with two slots, or an internet connection. Since the key idea is *cash*, the main security concern is that, even in a hostile environment, money cannot be created or destroyed (satisfying the security concerns of the bank and the customer, respectively).

The abstract specification formalizes atomic money transfer: a function balance : name $\rightarrow$ $\mathbb{N}$ gives the current balance of each named purse (type name). A predicate authentic distinguishes legitimate purses from impostors. Successful money transfer is described by the TRANSFEROK rule below. This rule chooses the two authentic participating purses from and to and the amount value to transfer, which should be less or equal than balance(from), and modifies the balances according to a successful money transfer.

TRANSFEROK =
choose from, to, value
with authentic(from) $\wedge$ authentic(to) $\wedge$ from $\neq$ to $\wedge$ value $\leq$ balance(from)
in balance(from) := balance(from) $-$ value
balance(to) := balance(to) + value

In reality, transfers may fail, since power can fail, card memory can run out, and cards may detach from the protocol prematurely. According to the security requirement no money should be lost, so a mechanism must be implemented which saves information about failed transfers. The definition of this information will be the task of the first refinement. At the abstract level, it is simply assumed that there is another field lost: name $\rightarrow \mathbb{N}$ on each card, which saves all outgoing money that is lost in failed transfer attempts, so that it can subsequently be recovered. The rule for failed transfer attempts is then simply

TRANSFERFAIL =
choose from, to, value
with authentic(from) $\wedge$ authentic(to) $\wedge$ from $\neq$ to $\wedge$ value $\leq$ balance(from)
in balance(from) := balance(from) $-$ value
lost(from) := lost(from) + value

With this rule it is obvious that the sum of all balance and lost values of all authentic purses never changes, so money can neither be created nor *truly* lost and the security goal is satisfied. This completes the description of the abstract specification. Runs of the system repeatedly apply ARULE = TRANSFEROK $\vee$ TRANSFERFAIL which chooses nondeterministically between the two possibilities.

3 From Atomic Transfers to Messages

The first refinement towards the Mondex protocol is concerned with implementing atomic transfer using a protocol that sends messages between the two purses, despite the fact that these messages may get lost for any number of reasons.

Sending a single Val(from,value) message from the from purse to the to purse containing the value and its sender will not do, since the message could be lost, and neither party would be able to prove that this happened.

An additional Ack(to,value) message acknowledging that the to purse has received the money improves matters: if the to purse sends this message when receiving money, but the from purse does not receive it, the from purse can write an *exception log* proving that something did not work: either the Val message was not processed properly or the Ack was lost. Dually, if the to purse sends a Req(to,value) message that requests the money, and the from purse only sends Val on receiving it, then the to purse can know that something did not work: either the request was not processed properly, or the Val was lost. Using all three messages enables detection of Val(from,value) message loss by inspecting both cards: the Val(from,value) message has been lost iff *both* purses have a suitable log entry.

Being able to detect failed transfers by checking both purse logs has one caveat: the two log entries must reliably belong to the *same transfer*. Otherwise a first attempt could lose the Ack(to,value) message, creating a from log entry, and a second attempt could lose the Req(to,value) message, creating a fictitious "matching" pair. Therefore

we will assume that each attempt to transfer money is equipped with a unique transfer identifier tid. The implementation of tid by sequence numbers is deferred to the third refinement; in this refinement we just assume there is a global finite set tids : set(tid) that stores the used identifiers and that it is always possible to choose a fresh one.

So, for the protocol we need messages Req(to, value, tid),Val(from, value, tid) and Ack(to, value, tid). Triples consisting of a name, a value and a tid are called *payment details*. They form the content of messages. Payment details are also remembered in exception logs which now replace the lost component. The logs are functions exLogfrom : name $\rightarrow$ set(paydetails) and exLogto : name $\rightarrow$ set(paydetails). To compute the abstract lost(from) we have to sum all values of payment details (to,value,tid) $\in$ exLogfrom(from) for which a matching (from,value,tid) in exLogto(to) exists (and this is already the main information needed for the simulation relation).

To allow message exchange, each purse now has a "message box" of messages awaiting processing. This is the function inbox : name $\rightarrow$ set(message). We first tried an inbox that contained one rather than several messages, but this turned out to be too restrictive, enforcing message sequencing between purses. Losing messages is realized by the following simple rule which may be invoked at any time by a purse receiver:

LOSEMSG =
choose msgs **with** msgs $\subseteq$ inbox(receiver) **in** inbox(receiver) := msgs

Finally, a purse has to know which message it sent last so as to react to missing answers appropriately; function outbox : name $\rightarrow$ message does this. An outbox that is *not* in the middle of a protocol run, can contain either the last sent Ack, or the special value none, when it has not yet sent a message or successfully received an Ack. Both cases are checked with the predicate isNone.

With these data structures, we derive four rules: for sending requests (STARTTO), for receiving a request and sending a value (REQ), for receiving a value and sending an acknowledgement (VAL), and finally for receiving an acknowledgement (ACK).

Like LOSEMSG above, the STARTTO and REQ rules assume that an authentic purse receiver has been chosen to execute the rule. Note that STARTTO chooses a new transfer identifier and is possible only when the purse is currently not involved in a protocol (i.e. when isNone(outbox(receiver))). Postfix selectors msg.pd, msg.value, msg.tid select the full payment details, the value and the tid contained in a message. **seq** is ASM notation to indicate sequential execution (usually all assignments are executed in parallel).

STARTTO =
if isNone(outbox(receiver))
then choose na, value, tid
with tid $\notin$ tids $\wedge$ authentic(na) $\wedge$ na $\neq$ receiver
in inbox(na) := inbox(na) $\cup$ {Req(receiver, value, tid)}
  outbox(receiver) := Req(na, value, tid)
  tids := tids $\cup$ {tid}

```
REQ =
choose msg
with    msg ∈ inbox(receiver) ∧ isReq(msg) ∧ authentic(msg.na)
      ∧ msg.na ≠ receiver ∧ msg.value ≤ balance(receiver)
      ∧ isNone(outbox(receiver)) in
   inbox(msg.na) := inbox(msg.na) ∪ {Val(receiver, msg.value, msg.tid)}
   outbox(receiver) := Val(msg.pd)
   balance(receiver) := balance(receiver) − msg.value seq
   inbox(receiver) := inbox(receiver) \ {msg}
```

The VAL rule is similar to REQ: the input is checked to be a Val(pd) message, where the outbox must be Req(pd) with the same payment details pd, and the sent message placed in inbox(msg.na) is an Ack. Also msg.value is added to the balance instead of subtracted. The ACK rule is similar too, but does not change the balance, does not write any output message and sets the outbox to none.

```
VAL =
choose msg
with    msg ∈ inbox(receiver) ∧ isVal(msg) ∧ isReq(outbox(receiver))
      ∧ msg.pd = outbox(receiver).pd in
   inbox(msg.na) := inbox(msg.na) ∪ {Ack(receiver, msg.value, msg.tid)}
   outbox(receiver) := Ack(msg.pd)
   balance(receiver) := balance(receiver) + msg.value seq
   inbox(receiver) := inbox(receiver) \ {msg}


ACK =
choose msg
with    msg ∈ inbox(receiver) ∧ isAck(msg) ∧ isVal(outbox(receiver))
      ∧ msg.pd = outbox(receiver).pd in
   outbox(receiver) := none
   inbox(receiver) := inbox(receiver) \ {msg}
```

Finally, a purse can abort a protocol (for whatever reason); it then executes

```
ABORT =
if isReq(outbox(receiver))
then exLogto(receiver) := exLogto(receiver) ∪ {outbox(receiver).pd} seq
if isVal(outbox(receiver))
then exLogfrom(receiver) := exLogfrom(receiver) ∪ {outbox(receiver).pd} seq
outbox(receiver) := none
```

The full specification chooses an authentic receiver and nondeterministically executes one of the above rules

```
IRULE = choose receiver with authentic(receiver) in
            LOSEMSG ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT
```

In [12], [17] we have proposed the use of purse-local simulation relations and invariants to verify refinements that split up an atomic action into several protocol steps. The approach described there for the simulation relation could be used unchanged. The invariants used in the approach state that "each purse has executed some (maybe no) steps into the protocol". Such invariants are easily expressible in KIV's Dynamic Logic.

Our research in [25, 26] has established a general framework, that suggests invariants should be *protocol-local*, not *purse-local*. Therefore we generalized the approach to using the following idea for invariants:

> "For every protocol already running (identified by a tid $\in$ tids), there are two purses from and to that have executed some of the steps of the protocol. These steps determine the part of the state involving tid"

To get a formal definition involving ASM rules we have to do two things. Firstly, we have to formalize "some protocol steps". For the Mondex protocol these are

(1) no step.
(2) STARTTO and possibly an ABORT of the to purse.
(3) STARTTO, then REQ, then possibly ABORT(from) or ABORT(to) or both.
(4) STARTTO, REQ, and VAL and then possibly ABORT(from).
(5) The full protocol STARTTO, REQ, VAL and ACK.

The states reached by executing some steps of the protocol therefore correspond directly to the final states st1 of the nondeterministic program

```
SOMESTEPS(st,from,to) =
(1)   skip ∨
(2)   STARTTO; {skip ∨ ABORT(to) ∨
(3)      REQ; {{skip ∨ ABORT(to)};{skip ∨ ABORT(from)} ∨
(4)         VAL; {skip ∨ ABORT(from) ∨
(5)            ACK}}}
```

when started some initial state st, where tid was still unused (tid $\notin$ tids). Note that the parameters from and to were dropped where it was obvious: STARTTO is really STARTTO(to), i.e. the to purse is used in the STARTTO rule in place of receiver. The fact, that st1 is a final state of some terminating run of SOMESTEPS is expressed, using Dynamic Logic [27] in KIV, as

```
⟨SOMESTEPS(st,from,to)⟩ st = st1
```

but the approach is not tied to the ASM formalism and Dynamic Logic: using a relational encoding of ASM rules, a relation somesteps could be defined similarly to the SOMESTEPS program above, using relational composition instead of compounds. The Dynamic Logic formula would then be equivalent to

```
somesteps(st,from,to,st1)
```

Secondly, we have to give a formal equivalent of the assertion "the protocol steps determine the part of the state involving tid". The part of the state that involves tid is easy to define. It consists of those messages in in- and outboxes, and those exception logs, that have tid in their payment details. To define what it means for the protocol steps to determine the part of the state that involves tid, we define a predicate eqtid(tid,st1,st2). This predicate compares two states. The first state st1 is the final state of running *just* the protocol steps involving tid from some initial state. It is a possible result of running SOMESTEPS. The second state st2 is the result of running these protocol

steps interleaved with an arbitrary number of protocol steps of other protocol instances. eqtid(tid,st1,st2) specifies that indeed the parts of the state involving tid of st1 and st2 are equal, since other protocol instances cannot interfere with the current protocol instance. There are two small exceptions: LOSEMSG may delete messages from inboxes, and a final Ack-message in an outbox may be overwritten after the current protocol has finished.

Putting together the SOMESTEPS program and the eqtid predicate we get the following invariant that encodes the informal idea given above:

$$\begin{array}{l} \mathsf{INV(st2)} \leftrightarrow \\ \quad \forall\, \mathsf{tid} \in \mathsf{tids2}.\ \exists\, \mathsf{from, to, st, st1}. \\ \qquad \mathsf{tid} \notin \mathsf{tids} \wedge \langle \mathsf{SOMESTEPS(st)} \rangle\ \mathsf{st} = \mathsf{st1} \wedge \mathsf{eqtid(tid,st1,st2)} \end{array}$$

The formula states, that for every protocol currently running (tid $\in$ tids2) there was an initial state st before the protocol was started and two participants from,to, such that the current part of the state involving tid is nearly the same (eqtid) as the state resulting from some terminating run of SOMESTEPS(st).

This is already the full invariant that is needed, except for the trivial invariant stating that no messages in inboxes, outboxes or exception logs, mention a tid that is not yet in tids.

The invariance proof reduces to proofs for every single protocol instance (i.e. for every tid), and for every protocol step. It has two cases: either the protocol step executed is one of the steps of the protocol instance for tid or it is a step of some other protocol instance. In the first case we essentially get trivial proof obligations, since "some steps of the protocol have been executed" is trivially invariant when executing yet another step. Essentially these proof obligations check that SOMESTEPS indeed encodes *all* possible protocol runs. For the second case we have to prove that steps of other protocol instances will not create messages or exception logs involving tid. The proof obligations check that eqtid correctly captures all potential interference from the other protocol instances.

Compared to the earlier proof of the full refinement in one step [12], which used purse-local invariants (which already simplified the many invariants needed for the original proof [4] that we used in [21]), the invariant has again been simplified: the purse-local approach required predicate logic properties that related the two states of the from and to purses participating in a protocol run. These are not needed any more.

4 Protection against Replay Attacks

Our next refinement is concerned with protection against replay attacks. The original development assumed that Req, Val, Ack messages are cryptographically protected, so we do the same. So an attacker cannot create such messages.

But even with this assumption, an attacker could destroy money on a from purse by saving and replaying Req and Ack messages. Indeed our first protocol is vulnerable to such an attack. For the new level, we assume an attacker who can intercept (and/or delete) messages, save them, and replay them. To model this formally, we assume a global set of messages ether : set(message) that contains at most all messages that

were sent so far. Since the union of all inboxes is a subset of ether, we can delete the inboxes altogether from the ASM state and let purses pick a message directly from ether. This corresponds to the attacker's ability to intercept and replace the message sent to a purse. Placing messages into a global ether instead of the inbox of the recipient has as immediate consequence: the intended recipient of the message must now be a component of the payment details of messages, and must be checked to be correct by the actual recipient. Otherwise the attacker could redirect messages from one purse to another. Since the attacker can still delete messages and messages might still be lost, LOSEMSG becomes

LOSEMSG = **choose** msgs **with** msgs $\subseteq$ ether **in** ether := msgs

To protect against replay attacks the states of purses must be enhanced with usedTids : name $\rightarrow$ set(tid), which gives the tids a purse receiver has seen previously. When a purse receives a Req, it saves the tid, and subsequently rejects messages with these transfer ids. Note that it is not necessary to add the tid of a Val message to the usedTids: accepting such a message only when the last sent message was a Req with the same payment details (and which must therefore have had a new tid!) is enough. This gives the following new rules for sending and receiving messages:

STARTTO =
choose na, value, tid **with** tid $\notin$ tids $\wedge$ authentic(na) $\wedge$ na $\neq$ receiver **in**
if isNone(outbox(receiver))
then ether := ether $\cup$ {Req(na, receiver, value, tid)}
outbox(receiver) := Req(na, receiver, value, tid)
tids := tids $\cup$ {tid}

REQ =
choose msg **with** msg $\in$ ether **in**
if isReq(msg) $\wedge$ msg.from = receiver $\wedge$ authentic(msg.to)
$\wedge$ msg.to $\neq$ receiver $\wedge$ msg.value $\leq$ balance(receiver)
$\wedge$ msg.tid $\notin$ usedTids(receiver) $\wedge$ isNone(outbox(receiver))
then ether := ether $\cup$ {Val(msg.pd)}
outbox(receiver) := Val(msg.pd)
balance(receiver) := balance(receiver) $-$ msg.value
usedTids(receiver) := usedTids(receiver) $\cup$ {msg.tid}

VAL =
choose msg **with** msg $\in$ ether **in**
if isVal(msg) $\wedge$ isReq(outbox(receiver)) $\wedge$ msg.pd = outbox(receiver).pd
then ether := ether $\cup$ {Ack(msg.pd)}
outbox(receiver) := Ack(msg.pd)
balance(receiver) := balance(receiver) + msg.value

ACK =
choose msg **with** msg $\in$ ether **in**
if isAck(msg) $\wedge$ isVal(outbox(receiver)) $\wedge$ msg.pd = outbox(receiver).pd
then outbox(receiver) := none

Aborting can now be simplified slightly: since the payment details contain the names of both purses, there is no further need to distinguish exLogfrom and exLogto. A single exLog: name → paydetails will do, and ABORT becomes

ABORT = **if** isReq(outbox(receiver)) ∨ isVal(outbox(receiver))
then exLog(receiver) := exLog(receiver) ∪ {outbox(receiver).pd} **seq**
outbox(receiver) := none

All together we have:

ERULE = **choose** receiver **with** authentic(receiver) **in**
LOSEMSG ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT

Whereas the previous refinement splits atomic steps into a protocol, this one is a typical data refinement: abstract and concrete rules correspond pairwise.

The simulation relation needed for verification consists of three parts:

- The union of all inboxes is always a subset of the ether.
- All requests in ether can only be in an inbox, if they have a tid that is not in usedTids(from) of the from purse that this message is sent to.
- Enhancing the union of the two logs exLogfrom(receiver) and exLogto(receiver) with "receiver" as a new component of the payment details gives exLog(receiver) for each authentic purse receiver.

Three invariants are needed for the concrete level:

- ether contains only messages with authentic names of different purses.
- tid's saved in the outbox-, exLog- or usedTids- field of an authentic purse are always also in tids.
- outbox(receiver) has payment details enhanced with "receiver" as to/from component for Req and Ack/Val.

5 Sequence Numbers as Challenges

The next refinement guarantees the uniqueness of protocol runs without using the global data structure tids. Instead we use a challenge-response scheme, like session keys, to ensure uniqueness. Mondex uses sequence numbers, which are used only once and then incremented. An alternative design decision would be to use random numbers ("nonces"). The state is now enhanced with a new component nextSeqNo : name → **N**, while the global set tids and the usedTids of each purse are removed. To be secure, both purses participating in a protocol run provide and increment their nextSeqNo, guaranteeing that each abstract tid is implemented by a unique (fromseqno(tid), toseqno(tid)) pair; the two functions fromseqno and toseqno are the essence of the simulation relation. To ensure no faked sequence numbers get used, we need to send the sequence number as a challenge to both purses. For the from purse, Req can be used for the purpose. For the to purse a new message startTo(from,nextSeqNo(from),to, nextSeqNo(to),value), which is assumed to be encrypted too, is needed. On receiving a startTo/Req message, the to/from purse must check whether it contains the correct sequence number; both checks together guarantee, that Req and Val are never sent

on faked sequence numbers. Finally, for the from purse to send startTo, we need a startFrom(to,nextSeqNo(to),value) message, that sends nextSeqNo(to) to the from purse. This comes from the terminal, when the transfer amount has been entered. It need not be encrypted; at worst an invalid startTo message gets rejected by the to purse. For our ASM, we assume *all* startFrom messages are in the ether initially, modelling the ability of the attacker to generate such messages at will.

Note that this model deviates slightly from the original Mondex protocol [4], which assumes an unencrypted startTo, sent together with the startFrom, from the terminal. The original protocol cannot guarantee that a Req contains a correct nextSeqNo(to), and leads to the weakness described in [12].

The ASM of the resulting protocol is:

```
SRULE =
choose receiver with authentic(receiver) in
  LOSEMSG ∨ STARTFROM ∨ STARTTO ∨ REQ ∨ VAL ∨ ACK ∨ ABORT


STARTFROM =
choose msg, n with msg ∈ ether ∧ nextSeqNo(receiver) < n in
if   isStartFrom(msg) ∧ authentic(msg.name) ∧ msg.name ≠ receiver
  ∧ msg.value ≤ balance(receiver) ∧ isNone(outbox(receiver))
then outbox(receiver) :=
          startTo(receiver, nextSeqNo(receiver)
                  msg.name, msg.nextSeqNo, msg.value)
nextSeqNo(receiver) := n
ether := ether ∪ {outbox(receiver)}


STARTTO =
choose msg, n with msg ∈ ether ∧ nextSeqNo(receiver) < n in
if   isStartTo(msg) ∧ authentic(msg.from) ∧ msg.from ≠ receiver
  ∧ msg.to = receiver ∧ msg.tono = nextSeqNo(receiver)
  ∧ isNone(outbox(receiver))
then outbox(receiver) := Req(msg.pd)
     nextSeqNo(receiver) := n
     ether := ether ∪ {Req(msg.pd)}


REQ =
choose msg with msg ∈ ether in
if   isReq(msg) ∧ isStartTo(outbox(receiver))
  ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := Val(msg.pd)
     balance(receiver) := balance(receiver) − msg.value
     ether := ether ∪ {Val(msg.pd)}


VAL =
choose msg with msg ∈ ether in
if isVal(msg) ∧ isReq(outbox(receiver)) ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := Ack(msg.pd)
     balance(receiver) := balance(receiver) + msg.value
     ether := ether ∪ {Ack(msg.pd)}
```

```
ACK =
choose msg with msg ∈ ether in
if isAck(msg) ∧ isVal(outbox(receiver)) ∧ outbox(receiver).pd = msg.pd
then outbox(receiver) := none


ABORT =
choose n with nextSeqNo(receiver) ≤ n in
if isReq(outbox(receiver)) ∨ isVal(outbox(receiver))
then exLog(receiver) := exLog(receiver) ∪ {outbox(receiver).pd} seq
     nextSeqNo(receiver) := n
     outbox(receiver) := none


LOSEMSG =
choose newether with newether ⊆ ether in ether := newether
```

The rules are largely unchanged except that tid's are replaced by pairs of sequence numbers. ABORT is now allowed to increment nextSeqNo to conform to the final Mondex protocol.

To verify the refinement we consider 1:1 diagrams for the common operations. The new STARTFROM step implements an abstract skip. The simulation relation asserts that two functions fromseqno and toseqno with domain = tids exist with the following three properties:

- outboxes, messages in ether and exception logs of the concrete level have tid replaced with fromseqno(tid) and toseqno(tid). There are two exceptions: an outbox of the concrete level may already contain a startTo of a new protocol run when the abstract outbox still satisfies isNone. The concrete ether may contain additional startFrom and startTo messages.
- If tid_1 and tid_2 appear in payment details of the abstract level with the same purses from and to, then $fromseqno(tid_1) \neq fromseqno(tid_2)$ or $toseqno(tid_1) \neq toseqno(tid_2)$. This guarantees that every protocol run between the same two purses uses a different pair of sequence numbers.
- If on the concrete level outbox(receiver) = startTo(pd) and Req(pd) ∈ ether, then there is a corresponding Req(pd) (with tid instead of sequence numbers) in the abstract ether and it's tid is not in usedTids(receiver). This property describes the new situation after sending a startTo message.

The concrete ASM also needs an invariant stating:

- outboxes never contain startFrom messages.
- The nextSeqNo of each purse is larger than any sequence number contained in any payment details in messages, inboxes, outboxes and exLogs.
- If outbox(receiver) contains a startTo, then the value of the message is less than or equal to balance(receiver).

6 Renaming to Use the Original Data Structures

The final refinement step is a purely technical one. It adjusts two small differences between SRULE and the final Mondex protocol. Since the full ASM was already given earlier in [12], we just give a short description of the differences.

In the real protocol, the outbox information is split into two: a pdAuth component which stores the payment details, and a status field, which stores the type of the last sent message: epr ("expecting request") for a startTo message, epv ("expecting value") for a Req message, epa ("expecting acknowledge") for a Val message, idle for an Ack message or none.

The second difference is a small change in control structure: nondeterministic choice between SRULE's disjuncts is replaced by deterministic choice over the type of message; if the test of the rule fails, an ABORT is executed. Finally, losing messages is done while adding a message to ether.

7 Related Work

The work of this paper is heavily based on the original work in [4] and the mechanized proofs in [7]. Several of the solutions described therein are monolithic (including our own); however, two structured the development into several refinements.

We first discuss the work of M. Butler and D. Yadav [19], since it is closest to ours. Their development uses Event-B, which like ASMs uses an operational style of specification (in contrast to the original Z which is relational). Event-B is based on the idea of structuring a development into many small steps to achieve a high degree of automation. So [19] used 9 refinements to develop a Mondex-like protocol. One key idea in their work is to view Mondex protocol runs as instances of transactions, viewing the state of all the purses as a kind of database (our work in [25, 26] also picks up on this idea). Because of this, their first refinements do not introduce messages (like ours), but define transactions and status information. This leads to an elegant development with small steps and a high degree of automation, but the price to pay is that intermediate levels use concepts (like a purse being in several transactions simultaneously), which are not present in the Mondex protocol.

Our goal in this paper was different: we wanted to cleanly separate the concepts present in the original Mondex protocol, and made no attempt to generalize. We also did not attempt to automate proofs further than in our earlier work. In fact, the effort for proving the 4 refinements of this paper was slightly higher than for the single refinement [12], due to revisions of intermediate levels.

Despite the different aims of these papers, there is one key idea we also used: abstract tid's to identify protocol runs (or transactions), since it abstracts nicely from the use of sequence numbers to identify protocol runs. Use of tid's leads to similarities between the Event-B machines and our ASMs. Although there are differences (no startTrans in our development; at this stage, our protocol has three messages), the biggest similarities are between the Event-B machines derived after around 6 refinements, and the one that our first refinement derives. This agrees with our experience, that the first refinement is still the most complex to verify. Also, their refinements 6 and 7 introduce sequence numbers, which we define in the third refinement.

The other work on Mondex that uses a structured development is the one of C. George and A.E. Haxthausen [20]. The work is based on the RAISE specification language and derives the Mondex protocol using two refinements, starting from a specification that can be viewed as a first refinement of our abstract specification. The key idea of this specification is: to transfer money from one purse to another there has to be a sending step (called transferLeft which either puts money "in transit" or moves it to lost), a successful receiving step (called transferRight, which moves money from in transit to balance(to)), and a step which moves money from in transit to lost (called Abort). The two steps of the refinement then show that all steps of the Mondex protocol implement one of these steps (e.g. REQ, that sends the Val message, implements transferLeft). This development has the advantage that the propagation of the security goals to the refined machines becomes easy. However the resulting refinement steps are rather different from the ones we give here.

8 Conclusion

In this paper we have analyzed the core concepts of the Mondex protocol, and we have shown that it is possible to place each concept into one concept-specific refinement. We have also given a slight improvement of the technique of *purse-local* invariants, explained in [17], by using *protocol-local* simulation relations, as suggested by our recent results on a framework for interleaved protocols [25]. This has led to the verification of each protocol run as one big commuting diagram, which moves much of the complexity of the first refinement into generic theory. The generic framework has now been verified in KIV [26], and holds promise for further extension and application.

Acknowledgement We would like to thank Bogdan Tofan, who has done many of the KIV proofs that ensure correctness of this work.

References

1. MasterCard International Inc.: Mondex. URL: http://www.mondex.com.
2. UK ITSEC Certification Body: UK ITSEC Scheme Certification Report No. P129 MONDEX Purse. Technical report (1999)
   URL: http://www.cesg.gov.uk/site/iacs/itsec/media/certreps/CRP129.pdf.
3. CCIB: Common Criteria for Information Technology Security Evaluation, Version 3.1 (ISO 15408). (November 2007) URL: http://csrc.nist.gov/cc.
4. Stepney, S., Cooper, D., Woodcock, J.: An Electronic Purse: Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Lab (2000) URL: http://www-users.cs.york.ac.uk/susan/bib/ss/z/monog.htm.
5. Spivey, J.M.: The Z Notation: A Reference Manual. 2nd edn. PHI (1992)
6. Woodcock, J.: First Steps in the Verified Software Grand Challenge. IEEE Computer **39**(10) (2006) 57–64
7. Jones, C., Woodcock, J., eds.: Formal Aspects of Computing. Volume 20 (1). Springer (January 2008)
8. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Proc. FM 2006. Volume 4085 of LNCS., Springer (2006) 16–31

9. Cooper, D., Stepney, S., Woodcock, J.: Derivation of Z Refinement Proof Rules. Technical Report YCS-2002-347, University of York (2002)
URL: http://www-users.cs.york.ac.uk/susan/bib/ss/z/zrules.htm.
10. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford Univ. Press (1995) 9–36
11. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
12. 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 (2008) (older version available as Techn. Report 2006-27 at [24]).
13. Börger, E., Rosenzweig, D.: The WAM—Definition and Compiler Correctness. In: Logic Programming: Formal Methods and Practical Applications. Studies in CS and AI 11. North-Holland (1995) 20–90
14. Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation. J.UCS **7**(11) (2001) 952–979
15. Börger, E.: The ASM Refinement Method. FAC **15 (1-2)** (2003) 237–257
16. Schellhorn, G.: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. TCS **336** (2005) 403–435
17. Schellhorn, G.: ASM Refinement Preserving Invariants. Proceedings of the ASM workshop 2007, Grimstad, Norway (2008) (to appear in J.UCS).
18. Banach, R., Jeske, C., Poppleton, M., Stepney, S.: Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1,1) Forward Refinements. Fund. Inf. 77 (2007) 29–69
19. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. FAC **20**(1) (January 2008)
20. Haxthausen, A., George, C.: Specification, Proof, and Model Checking of the Mondex Electronic Purse using RAISE. FAC **20**(1) (January 2008)
21. 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)
22. 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 (ICSEA), IEEE Press (2007) URL: http://ieeexplore.ieee.org.
23. Grandy, H., Bischof, M., Schellhorn, G., Reif, W., Stenzel, K.: Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. Accepted at 15th International Symposium on Formal Methods (FM 2008) (2008)
24. Mondex KIV: Web presentation of the Mondex case study in KIV
URL: http://www.informatik.uni-augsburg.de/swt/projects/mondex.html.
25. Banach, R., Schellhorn, G.: On the Refinement of Atomic Actions. Proceedings of REFINE 2007, ENTCS **201** (2007) 3–30
26. Banach, R., Schellhorn, G.: Atomic Actions, and their Refinements to Isolated Protocols. FAC (2008)
27. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)