OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

A systematic verification approach for Mondex electronic purses using ASMs

Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius, Wolfgang Reif

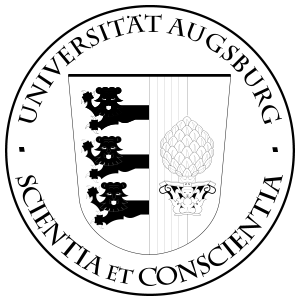
Angaben zur Veröffentlichung / Publication details:

Schellhorn, Gerhard, Holger Grandy, Dominik Haneberg, Nina Moebius, and Wolfgang Reif. 2006. "A systematic verification approach for Mondex electronic purses using ASMs." Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use: **licgercopyright**

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these 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



A Systematic Verification Approach for Mondex Electronic Purses using ASMs

**G.Schellhorn, H.Grandy, D.Haneberg,
N.Moebius, W.Reif**

Report 2006-27 December 2006

Institut für Informatik
D-86135 Augsburg

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

A Systematic Verification Approach for Mondex Electronic Purses using ASMs

Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany
{schellhorn,grandy,haneberg,moebius,reif}@informatik.uni-augsburg.de

Abstract. In [SGHR06] we have solved the challenge to mechanically verify the Mondex challenge about the specification and refinement of an electronic purse as defined in [SCJ00]. In this paper we show, that the verification can be made more systematic and better automated using ASM refinement instead of the original data refinement. This avoids to define a lot of properties of intermediate states during protocol runs. The systematic development of a generalized forward simulation also uncovered a weakness of the protocol, that could be exploited in a denial of service attack.

1 Introduction

Mondex smart cards implement an electronic purse [MCI]. They have become famous for having been the target of the first ITSEC evaluation of the highest level E6 [CB99], which requires formal specification and verification.

Such formal specifications were given in [SCJ00] using the Z specification language [Spi92]. 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. To mechanize the security and refinement proofs in [SCJ00] has been recently proposed as a challenge for theorem provers (see [Woo06] for more information on the challenge and its relation to 'Grand Challenge 6').

In [SGHR06] we have solved the challenge: we have tried to repeat the case study as faithful as possible by formalizing the underlying data refinement theory given in [CSW02] and by using the original backward simulation and invariant. A detailed description of this verification (including the extra protocol for archiving exception logs) is currently in preparation for [CJ07].

When we solved the original challenge, we found that the backward simulation and the invariant needed for the concrete level[1] look rather ad hoc and specific to Mondex. We could find no system in the properties listed. Much more

[1] formally, the verification of an invariant is encoded in [SCJ00] as a second refinement of a between level, that assumes the invariant to a concrete level, that does not.

work than the 4 weeks we needed to do the mechanical verification surely was necessary to develop this invariant by incrementally adding properties.

Therefore, in this paper we show how to develop a simulation relation and an invariant systematically. We do this using Abstract State Machines (ASM, [Gur95], [BS03]) as specification language. In [SGHR06] we have already given ASMs for the abstract and concrete level of Mondex and shown that the proof for the main backward simulation condition for this ASM is the same as the one for data refinement, but with a lot of technical overhead removed. Therefore we feel justified to use the simplified version here.

Using ASMs naturally leads to the use of ASM refinement ([BR95], [Sch01], [Bör03], [Sch05]) and (generalized) forward simulations.

This paper is organized as follows: Sect. 2 introduces the ASMs used in the case study, and gives an informal idea why the refinement is correct.

Section 3 develops a forward simulation for Mondex systematically using two core ideas of ASM refinement: focussing the simulation relation on *states of interests*, which in this case naturally are those *future* states where all protocols have completed, and *localizing invariants* to individual purses. We show that the main proof obligation, a commuting diagram for a local invariant can be verified fully automatically in KIV.

Development of a systematic invariant for the concrete level turned out to be much harder than the development of a simulation relation. The main reason is that the protocol is vulnerable to a certain kind of denial of service attack described in Section 4. Although the attack does not violate the security properties defined in [SCJ00] (no money is lost), it came into sharp focus when we applied ASM techniques to develop an invariant in Sect. 5. While using *states of interests* is possible, in this case the *past* states at the beginning of the protocol, fully *localizing* the invariant to individual purses is impossible due to this denial of service attack. We use *lazy development* for the invariant and show that the main proof only requires a few KIV interactions.

Finally, Sect. 6 gives related work and Sect. 7 concludes. Full details on all proofs (including those we did for [SGHR06] are available as a Web presentation [KIVa]. The project [KIVb] contains the ASM refinement.

2 The ASM Specifications of Mondex

In the following we describe the specifications of the smart cards involved in the Mondex case study. To be self-contained in this paper we here repeat a slightly modified version of the description given in [SGHR06]. The changes are purely cosmetic to have shorter formulas in the proof obligations.

The specification is given on two levels: An abstact level which defines an atomic transaction for transfering money, and a concrete level which defines a protocol. Both levels are defined using abstract state machines (ASMs, [Gur95], [BS03]) and algebraic specifications as used in KIV [RSSB98].

2.1 The Abstract Level

The abstract specification of a purse consists of a function abalance from purse names to their current balance. Since the transfer of money from one to another purse may fail (due to the card being pulled abruptly from the card reader, or for internal reasons like lack of memory) the state of an abstract purse also must log the amount of money that has been lost in such failed transfers.

In the formalism of ASMs this means that the abstract state astate consists of two dynamic functions $\mathsf{abalance} : \mathsf{name} \to \mathbb{N}$ and $\mathsf{lost} : \mathsf{name} \to \mathbb{N}$.

Purses may be faked, so we have a finite number of names which satisfy a predicate authentic. How authenticity is checked (using secret keys, pins etc.) is left open on both levels of the specification, so the predicate is left unspecified.

Transfer of money between purses is done with the simple ASM rule ASTEP#[2]:

ASTEP#
choose from, to, value, fail?
with authentic(from) $\wedge$ authentic(to) $\wedge$ from $\neq$ to
$\wedge$ value $\leq$ abalance(from)
in if $\neg$ fail? **then** TRANSFEROK#
else TRANSFERFAIL#
ifnone skip //do nothing, if there is no authentic pair of purses

TRANSFEROK#
abalance(from) := abalance(from) $-$ value
abalance(to) := abalance(to) $+$ value

TRANSFERFAIL#
abalance(from) := abalance(from) $-$ value
lost(from) := lost(from) $+$ value

The rule chooses two authentic, different names from and to, and an amount value which should be transfered from the from purse to the to purse. The from card must have enough money left for the transfer (value $\leq$ abalance(from)). Additionally a boolean value fail? indicates whether the actual transaction will complete regularly or will nondeterministically fail for internal reasons. If the step completes normaly, the rule TRANSFEROK# subtracts value from the from purse and adds it to the to purse in one step. Otherwise, the rule TRANSFERFAIL# subtracts the money from the from purse and logs it in the lost(from) state function instead.

2.2 The Concrete Level

On the concrete level transferring money is done using a protocol with 5 steps. To execute the protocol, each purse needs a status that indicates how far it has progressed executing the protocol. 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

[2] By convention our rule names end with a # sign to distinguish them from predicates.

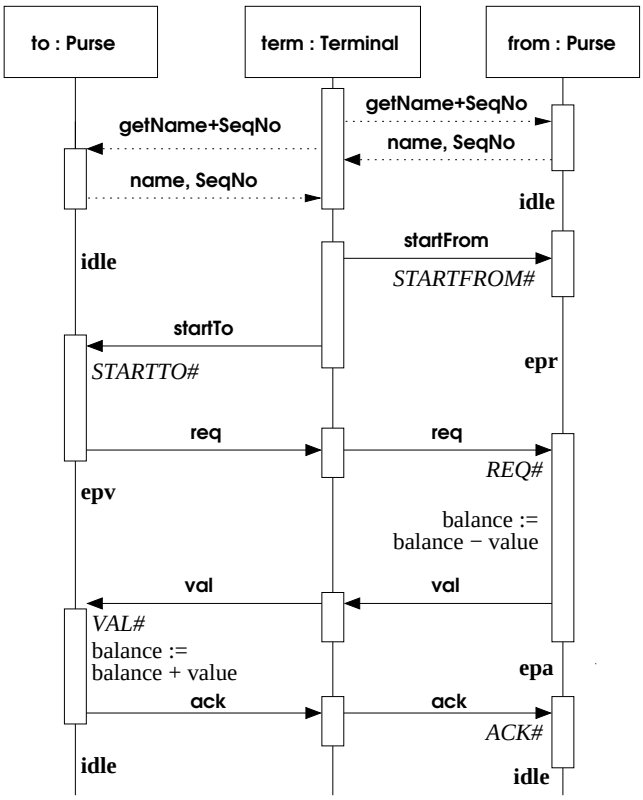


Fig. 1. An overview of the Mondex protocol

sequence number nextSeqNo that is used in the next transaction. This number is incremented at the start of every protocol run. During the run of the protocol each purse stores the current payment details in a variable pdAuth of type 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. In KIV we define a free data type

PayDetails = mkpd(.from:name; .fromno:nat; .to:name; .tono:nat; .value:nat)

with postfix selectors (so pd.from is the name of the from purse stored in payment details pd). The state of a purse finally contains a log exLog of failed transfers represented by their payment details. The protocol is executed by sending messages between the purses. The ether collects all messages that are currently available. A purse receives a message by selecting a message from the ether. Since the environment of the card is assumed to be hostile the message received may be any message that has already been sent, not just one that is directed to the card (this simple model of available messages is also used in many abstract specifications

of security protocols, e.g. the traces of [Pau98]). The state of the concrete ASM, abbreviated cstate in the following, therefore is

balance : name → ℕ
state : name → status
pdAuth : name → PayDetails
exLog : name → set(PayDetails)
ether : set(message)

An overview of the protocol on the concrete level is shown in Fig. 1. The message getName+SeqNo (shown by dashed lines) is needed in a real implementation of the Mondex scenario, since the terminal must be able to get the information about card names and their sequence numbers. This information is used in the following protocol steps. For the specification of Mondex those messages are not modelled, even the terminal itself is not modelled explicitly. Instead, all messages needed to start a protocol run are assumed to be initially contained in a set of messages available to the purses, called the ether.

The protocol is started with messages startFrom(msgna, value, msgno) and startTo(msgna, value, msgno) which are sent to the from and to purse respectively by the interface device. These two messages are assumed to be always available, so the initial ether already contains every such message. The arguments msgna and msgno of startFrom(msgna, value, msgno) are assumed to be the name and nextSeqNo of the to purse, value is the amount of value transfered. Similarly, for startTo(msgna, value, msgno) msgna and msgno are the corresponding data of the from purse. The messages req(pdAuth(name)), val(pdAuth(name)) and ack(pdAuth(name)) are responsible for the actual money transfer. After receiving a req, the from purse withdraws money from its internal balance. After receiving the corresponding val, the to purse deposits money on its internal balance. The ack message is used to ackknowledge the successful transfer. We now describe the ASM rule CSTEP#, which executes all the protocol steps, and the individual protocol steps in detail:

CSTEP#
choose msg, receiver fail? **with** msg ∈ ether ∧ authentic(receiver)
  in LCSTEP#

LCSTEP#
  if isOKstartFrom(msg) ∧ ¬ fail?
    then STARTFROM#
  else if isOKstartTo(msg) ∧ ¬ fail?
    then STARTTO#
  else if msg = req(pdAuth(receiver)) ∧ state(receiver) = epr ∧ ¬ fail?
    then REQ#
  else if msg = val(pdAuth(receiver)) ∧ state(receiver) = epv ∧ ¬ fail?
    then VAL#
  else if msg = ack(pdAuth(receiver)) ∧ state(receiver) = epa ∧ ¬ fail?
    then ACK#
  else ABORT#

where

$$\begin{aligned} &\mathsf{isOKstartFrom(msg)} \\ :\leftrightarrow\ &\mathsf{isStartFrom(msg)} \wedge \mathsf{state(receiver)} = \mathsf{idle} \wedge \mathsf{authentic(msg.msgna)} \\ &\wedge\ \mathsf{receiver} \neq \mathsf{msg.msgna} \wedge \mathsf{msg.value} \leq \mathsf{balance(receiver)} \end{aligned}$$

$$\begin{aligned} &\mathsf{isOKstartTo(msg)} \\ :\leftrightarrow\ &\mathsf{isStartTo(msg)} \wedge \mathsf{state(receiver)} = \mathsf{idle} \\ &\wedge\ \mathsf{authentic(msg.msgna)} \wedge \mathsf{receiver} \neq \mathsf{msg.msgna} \end{aligned}$$

The ASM rule CSTEP# chooses an authentic receiver for some message msg from ether. Additionally the purse may fail (e.g. for internal reasons), denoted by the flag fail?. The rule LCSTEP# executes the different protocol steps. It checks whether the incoming message is wellformed regarding to the current internal state. For example, when the purse is in state epr ("expecting request"), it will only accept messages of the form req(pdAuth(receiver)). Every other message received in epr will result in an ABORT# operation (which resets the purse state and logs the current transaction as faulty if necessary (see below).

On receiving a startFrom message msg from ether, the purse receiver[3] first checks whether it is in the idle state and if the message is syntactically correct. This means it must be of the right message type (isStartTo(msg)) and the contained card name is authentic. Additionally the transmitted msg.msgna (the name of the other purse) must be different from receiver. Finally receiver must have enough money stored (msg.value $\leq$ balance(receiver)). Then receiver executes the following step:

```
STARTFROM#
choose n with nextSeqNo(receiver) < n in
in pdAuth(receiver) := mkpd(receiver, nextSeqNo(receiver),
                            msg.msgna, msg.msgno, msg.value)
   state(receiver) := epr
   nextSeqNo(receiver) := n seq
   let outmsg := ⊥ in SENDMSG#
```

The purse stores the requested transfer in its pdAuth component, using its current nextSeqNo number as one component and proceeds to the epr state. Thereby it becomes the from purse of the current transaction. nextSeqNo is incremented to make it unavailable in further transactions. An empty output message $\perp$ is generated that will be added to the ether (see SENDMSG# below).

The action of a purse receiving a startTo message in idle state is similar except that it enters epv state ("expecting value") and becomes the to purse of the transaction. Additionally it sends a request message to the from purse.

[3] receiver is always a purse receiving a message. This can be a from purse sending money as well as a to purse receiving money and should not be confused with the latter.

STARTTO#
choose n **with** nextSeqNo(receiver) $<$ n
in pdAuth(receiver) := mkpd(msgna, msgno, receiver,
nextSeqNo(receiver), value)
state(receiver) := epv
nextSeqNo(receiver) := n **seq**
let outmsg := req(pdAuth(receiver)) **in** SENDMSG#

The request req(pdAuth(receiver)) contains the payment details of the current transaction. Although this is not modeled, the message is assumed to be securely encrypted. Since an attacker can therefore never guess this message before it is sent, it is assumed that the initial ether does not contain any request messages. When the from purse receives the request in state epr, it executes

REQ#
balance(receiver) := balance(receiver) − pdAuth(receiver).value
state(receiver) := epa
let outmsg := val(pdAuth(receiver)) **in** SENDMSG#

The message is checked to be consistent with the current transaction stored in pdAuth and if this is the case the money is sent with an encrypted value message val(pdAuth(receiver)). The state changes to epa ("expecting acknowledge"). On receiving the value the to purse executes

VAL#
balance(receiver) := balance(receiver) + pdAuth(receiver).value
state(receiver) := idle
let outmsg := ack(pdAuth(receiver)) **in** SENDMSG#

It adds the money to its balance, sends an encrypted acknowledge message back and finishes the transaction by going back to state idle. When this acknowledge message is received, the from purse finishes similarly.

ACK#
state(receiver) := idle
let outmsg := $\perp$ **in** SENDMSG#

Finally a rule for adding the sent messages to the ether is needed. Additionally the ether is assumed to lose messages randomly (due to an attacker or technical reasons like power failure). Both is now done in the rule SENDMSG# used in all the rules above.

SENDMSG#
choose ether′ **with** ether′ $\subseteq$ ether $\cup$ {outmsg} **in** ether := ether′

If a purse is sent an illegal message $\perp$ or a message for which it is not in the correct state, the current transaction is aborted by

ABORT#
choose n **with** nextSeqNo(receiver) $\leq$ n
in LOGIFNEEDED#
  state(receiver) := idle
  nextSeqNo(receiver) := n
  let outmsg := $\perp$ **in** SENDMSG#

LOGIFNEEDED#
if state(receiver) = epa $\vee$ state(receiver) = epv
then exLog(receiver) := exLog(receiver) $\cup$ {pdAuth(receiver)}

This action logs if money is lost due to aborting a transaction. The idea is that the lost money of the abstract level can be recovered if both the from and to purse have a log of the failed transaction. Logging takes place if either the purse is a to purse in the critical state epv or a from purse in critical state epa. Note that aborting in states idle and epr needs no exception log. Logging achieves, that in states where all purses are currently idle, balances and the lost money are related by

$$\mathsf{abalance(na)} = \mathsf{balance(na)} \wedge \mathsf{lost(na)} = \Sigma\ (\mathsf{fromLogged(na)} \cap \mathsf{toLogged}) \quad (1)$$

where Σ takes the sum of all values of a set of payment details,
fromLogged(na) := {pd : pd $\in$ exlog(na) $\wedge$ pd.from = na},
toLogged(na$'$) := {pd : pd $\in$ exlog(na$'$) $\wedge$ pd.to = na$'$} and
toLogged := $\bigcup_{\mathsf{na}'}$ toLogged(na$'$). For future use fromLogged is defined similarly.

The definitions of the abstract state machines used in the refinement proof can be found in the Web presentation [KIVb] in specifications *AASM* and *CASM*. As a small difference to the presentation of this paper, abbreviations fromLogged, toLogged (and also toInEpv toInEpa as defined in the next section) are always used in their expanded form.

3 Systematic Development of a Forward Simulation

One of the key ideas of ASM refinement (and also verification of ASM invariants) is *not* to consider all intermediate states of runs of the ASM but to focus on *states of interest* and to define properties φ only for these. In general, there is a choice to use future or past states. Either we can say:

- Future: From every state a state of interest that will satisfy φ is reachable via some ASM rule applications
- Past: Every state is reachable from some state of interest that satisfies φ

Applied to Mondex the obvious states of interest are states where a purse with name na does not participate in a protocol, i.e. where state(na) = idle. For the development of the simulation relation, we will consider future states of interest, in section 5 we will develop the invariant based on past states of interest.

There are two problems we have to solve. First, what is the simulation relation for states of interest? This question was already answered by formula (1) at the end of the previous section. Second, we have to show how a state of interest is reachable from any state. For the concrete level this is easy: simply call ABORT# for all purses. To have an equivalent state on the abstract level we must execute failing transactions for all those purses where money will be lost on the concrete level by executing the ABORT#'s. This money, which is currently in transit can be characterized by the set of relevant payment details. This set was already central to the correctness consideration of the original Mondex case study [SCJ00]. It is called maybelost and defined as

$$(\mathsf{fromInEpa} \cap \mathsf{toInEpv}) \cup (\mathsf{fromInEpa} \cap \mathsf{toLogged}) \cup (\mathsf{fromLogged} \cap \mathsf{toInEpv})$$

where

$$\mathsf{fromInEpa} = \{\mathsf{pdAuth(na)} : \mathsf{authentic(na)} \wedge \mathsf{state(na)} = \mathsf{epa}\}$$
$$\mathsf{toInEpv} = \{\mathsf{pdAuth(na)} : \mathsf{authentic(na)} \wedge \mathsf{state(na)} = \mathsf{epv}\}$$

The definition is based on the idea that money is lost in ABORT# iff a new pair of matching exception logs is created, which happens if either both purses log, or one logs and the other has already logged.
Putting everything together we get the following formula of Dynamic Logic[4]:

$$\begin{array}{l}\langle \textbf{forall } \mathsf{authentic(na)} \textbf{ do } \mathsf{ABORT\#(na;\ cstate)};\\ \ \textbf{forall } \mathsf{pd} \in \mathsf{maybelost} \textbf{ do}\\ \quad \mathsf{TRANSFERFAIL\#(pd.from,\ pd.to,\ pd.value;\ \ abalance,\ lost)}\rangle\\ \qquad (\quad \mathsf{abalance} = \mathsf{balance}\\ \qquad\ \wedge\ \mathsf{lost} = \lambda\,\mathsf{na}.\ \Sigma(\mathsf{fromLogged(na)} \cap \mathsf{toLogged})\end{array}$$

Informally this says: After executing the necessary ABORT#'s to get to an idle state and the necessary TRANSFERFAIL#'s to get to a corresponding abstract state we have the simple correspondence of balances and lost vs. exLog's as stated in Section 2.

This is already all that is required to define the simulation relation. To efficiently prove it, we simplify it: the only relevant modification of ABORT# is that of the the exception log (in LOGIFNEEDED#) and TRANSFERFAIL# only modifies abalance and lost of the from purse. We also apply the idea to *localize* the simulation relation to one individual purse with name na. We get the following definition of the simulation relation ACINV and its localized version LACINV:

$$\begin{array}{l}\quad \mathsf{ACINV(astate, cstate)}\\ :\leftrightarrow\ \forall\ \mathsf{authentic(na).LACINV(na, astate, cstate)}\end{array}$$

$$\begin{array}{l}\quad \mathsf{LACINV(na, astate, cstate)}\\ :\leftrightarrow\ \langle \mathsf{exLog'(na)} := \mathsf{exLog(na)} \cup \textbf{if } \mathsf{state(na)} = \mathsf{epv} \vee \mathsf{state(na)} = \mathsf{epa}\\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \{\mathsf{pdAuth(na)}\} \textbf{ else } \emptyset\rangle\\ \qquad \mathsf{abalance(na)} = \mathsf{balance(na)} + \mathsf{intransit(na, exLog'(na), state, pdAuth})\\ \quad \wedge\ \mathsf{lost(na)} = \mathsf{lostafterabort(na, exLog'(na), exLog))}\end{array}$$

[4] $\langle\alpha\rangle\,\varphi$ means, that there is a terminating execution of α after which φ holds

where intransit (money that is in transit) and lostafterabort (money that will be added to lost when all purses abort) are defined recursively over the set of payment details:

$$
\begin{array}{l}
\mathsf{intransit}(\mathsf{na}, \emptyset, \mathsf{state}, \mathsf{pdAuth}) = \mathsf{lostafterabort}(\mathsf{na}, \emptyset, \mathsf{exLog}) := 0 \\
\quad \mathsf{intransit}(\mathsf{na}, \mathsf{pds} \dot\cup \{\mathsf{pd}\}, \mathsf{state}, \mathsf{pdAuth}) \\
:= \quad \mathsf{intransit}(\mathsf{na}, \mathsf{pds}, \mathsf{state}, \mathsf{pdAuth}) \\
\quad + \textbf{if } \mathsf{pd.from} = \mathsf{na} \wedge \mathsf{state}(\mathsf{pd.to}) = \mathsf{epv} \wedge \mathsf{pdAuth}(\mathsf{pd.to}) = \mathsf{pd} \\
\quad \textbf{then } \mathsf{pd.value} \textbf{ else } 0 \\
\quad \mathsf{lostafterabort}(\mathsf{na}, \mathsf{pds} \dot\cup \{\mathsf{pd}\}, \mathsf{exLog}) \\
:= \quad \mathsf{lostafterabort}(\mathsf{na}, \mathsf{pds}, \mathsf{exLog}) \\
\quad + \textbf{if } \mathsf{pd.from} = \mathsf{na} \wedge \mathsf{pd} \in \mathsf{exLog}(\mathsf{pd.to}) \textbf{ then } \mathsf{pd.value} \textbf{ else } 0
\end{array}
$$

This definition is considerably simpler to use than the original (backward) simulation of [SCJ00], which uses and has to expand maybelost: the definitions of intransit and lostafterabort can be directly used as rewrite rules.

The main proof obligation of ASM refinement now requires, that one concrete step corresponds to some abstract proof steps. In our case this will be one or no abstract step. Written in Dynamic Logic the proof obligation reads:

$$
\begin{array}{l}
\quad \mathsf{CINV}(\mathsf{cstate}) \wedge \mathsf{ACINV}(\mathsf{astate}, \mathsf{cstate}) \\
\rightarrow \langle\!|\mathsf{CSTEP\#}(\mathsf{cstate})|\!\rangle \ \langle \mathsf{ASTEP\#}(\mathsf{astate}) \vee \textbf{skip} \rangle \ \mathsf{ACINV}(\mathsf{astate}, \mathsf{cstate})
\end{array} \tag{2}
$$

It assumes an additional (yet unknown) invariant CINV(cstate) of the concrete level. $\langle\!|\alpha|\!\rangle\, \varphi$ means "all applications of ASM rule α terminate and yield a state for which φ holds".

The proof obligation can be localized by replacing ACINV with LACINV for some purse na, and the generic CSTEP# with LCSTEP# for some purse receiver (where receiver and na may be different or the same). For this localized setting we can replace the nondeterministic choice between TRANSFEROK#, TRANSFERFAIL# (within ASTEP#) and **skip** with a deterministic statement by the following reasoning: our simulation relation says, that "after executing ABORT#'s we reach a concrete state of interest related to some abstract state". Therefore, only when executing ABORT#'s before and after CSTEP# leads to a *different* state of interest, an abstract transition must be executed. There are three cases where this is true:

- If receiver is a to purse that successfully receives a value in VAL#, an ABORT# before the operation will cause the money to be lost, but has no effect after the operation. Therefore a TRANSFEROK# is necessary.
- If receiver is a to purse that aborts in epv, when the from purse already has sent money (i.e. is in state epa or has logged the payment details) then the money will be lost by this action (the ABORT# of the to purse before the step has the effect to lose money, executing another ABORT# after the ABORT# has no effect).

– If receiver is a from purse that accepts a request in REQ#, sending a value to a to purse which has already aborted, then the ABORT# before the step will not create an exception log, while executing ABORT# afterwards will create the second exception log needed for money to be lost. Therefore this case requires a TRANSFERFAIL# to be executed too.

Altogether proof obligation (2) can be reduced to the following lemma, where pd abbreviates pdAuth(receiver) (oldstate saves the initial state of the receiver to be used after the step):

$$
\begin{array}{l}
\quad \mathsf{LACINV}(\mathsf{na},\mathsf{astate},\mathsf{cstate}) \wedge \mathsf{CINV}(\mathsf{cstate}) \wedge \mathsf{oldstate} = \mathsf{state}(\mathsf{receiver}) \\
\rightarrow \langle\!|\mathsf{CSTEP}\#(\mathsf{msg},\mathsf{receiver},\mathsf{fail}?;\ \mathsf{cstate})|\!\rangle \\
\quad \langle \textbf{if}\ //\ \textit{to: epv} \rightarrow \textit{VAL} \rightarrow \textit{idle:} \\
\qquad \mathsf{oldstate} = \mathsf{epv} \wedge \mathsf{msg} = \mathsf{val}(\mathsf{pd}) \wedge \neg\ \mathsf{fail}? \\
\quad \textbf{then}\ \mathsf{TRANSFEROK}\#(\mathsf{from},\mathsf{to},\mathsf{value}) \\
\quad \textbf{else if}\quad //\ \textit{to: epv} \rightarrow \textit{ABORT} \rightarrow \textit{idle:} \\
\qquad\qquad\qquad \mathsf{oldstate} = \mathsf{epv} \wedge (\mathsf{msg} \neq \mathsf{val}(\mathsf{pd}) \vee \mathsf{fail}?) \\
\qquad\qquad\quad \wedge\ \mathsf{pd} \in \mathsf{fromInEpa} \cup \mathsf{fromLogged} \\
\qquad\qquad\quad //\ \textit{from: epr} \rightarrow \textit{REQ} \rightarrow \textit{epa, when toLogged} \\
\qquad\qquad \vee \quad \mathsf{oldstate} = \mathsf{epr} \wedge \mathsf{msg} = \mathsf{req}(\mathsf{pd}) \\
\qquad\qquad\quad \wedge \neg\ \mathsf{fail}? \wedge \mathsf{pd} \in \mathsf{exLog}(\mathsf{pd.to}) \\
\qquad\quad \textbf{then}\ \mathsf{TRANSFERFAIL}\#(\mathsf{pd.from},\mathsf{pd.to},\mathsf{pd.value}) \\
\qquad\quad \textbf{else skip}\rangle \\
\quad \mathsf{LACINV}(\mathsf{na},\mathsf{astate},\mathsf{cstate})
\end{array}
\tag{3}
$$

Note that this proof obligation has no quantified formulas (and no nondeterministic programs, which would also lead to instantiating quantifiers), so the proof is fully automatic by symbolic execution of the involved programs. It has 641 proof steps. To lift the lemma to the simulation theorem is easy, but harder to automate: 107 proof steps and 30 interactions are necessary. Compared to the 197 interactions needed for the original backward simulation proof in [SGHR06] (which had 655 steps) this is a vast improvement (especially when considering that the proof of the main lemma 3 has to be iterated on corrections). The proof was developed in about a week, once the right approach had been found (see the remarks at the end of Section 5).

The proof ends in premises of the form $\mathsf{CINV} \rightarrow \varphi$ which give requirements for the invariant of the concrete level that we will develop in Section 5. We close these premises automatically by defining around 40 rewrite rules. Some of them are inter-derivable leaving a basic set of 26 rewrite rules. Our development of the invariant of CINV is *lazy* in the sense that it is done after the main simulation proof, guided by the requirement that the rewrite rules must be provable from the definition.

The full proofs of the main lemma (3) and the main simulation theorem (2) can be found in the Web presentation [KIVb] in specification *ASM-refine* as *LACINV-lem* and *ACINV-lem*. That they imply (together with initialization

and finalization conditions) that the Mondex refinement is a correct ASM refinement is proved in *Mondex-is-ASM-refinement*, which imports a variant of ASM refinement [Sch01] as a library. This variant of ASM refinement comes with a theorem that invariants are preserved, so given the proofs of security properties in *Mondex-SecProp* we can show in *Refinement-preserves-security* that the communication protocol also preserves security: no money is lost in protocol runs, and all money — except the money that is currently in transit (as given by the intransit function) — is accounted.

4 A Denial of Service Attack

In the previous section we derived a number of properties that the invariant CINV of the concrete level will have to imply. In the next section we will define such an invariant systematically. Like for the simulation relation, we tried to localize the invariant to individual purses. Several attempts to do this did not work, basically since we did not understand that the protocol allows a particular kind of a kind of denial of service attack. The attack does not violate the original security requirements "no money lost" and "all money accounted" of [SCJ00], our proofs show that they are correct and preserved by the refinement. But the attack shows, that the protocol violates the property that an attacker should not be able to *systematically* create exception logs. This section describes the relevant scenario and the global property it makes necessary for the invariant.

The scenario is as follows: we assume an attacker that has a faked card and knows the name and current sequence number of some purse called from (names and sequence numbers are not secret, they can be requested freely with getName+SeqNo and are used openly in startFrom and startTo).

Now, without the from purse involved in any way, on the next connection to a to purse the attacker can pose as from purse: he will answer the getName+SeqNo request of the terminal with from and a *future* sequence number n of the from purse. The terminal will then start a protocol with a startTo(from, n, value) message to the to purse, which will send a request message back. Since to does not receive a value message as response it has no choice but to abort the protocol.

Repeating the attack several times, the to purse will create an exception log each time. This will fill up the limited amount of space reserved for exception logs quickly: in reality, only a very small number of exception logs is allowed[5]. Since exception logs can also be created by accidents (power failures or an impatient card holder pulling his card too early out of the card reader) the original Mondex specification in [SCJ00] has an additional protocol, where the customer shows his card at the bank and gets his exception logs moved from the card to a central archive of the bank[6]. Therefore the fact, that an attacker can systematically (and

[5] our refinement of the communication protocol to a security protocol based on abstract cryptography described in [CJ07] enforces such a limit too.

[6] Archiving will be considered in our contribution to [CJ07]. Proofs are available in the Web presentation [KIVa]. Since the protocol is small and independent of the main protocol it is not considered in the formalization here.

not accidentally) create exception logs on the to purse can be seen as a mere inconvenience, since the to purse does not lose money.

But the scenario can be taken one step further: if the attacker connects to several to purses as described above, each time posing as the from purse he can collect the request messages he receives (although he can not decrypt them!).

In a second stage he then connects to the from purse, this time posing as one of the various to purses: he can send startFroms and all the collected request messages from the to purses to the from purse (provided he has used suitable sequence numbers in the first stage). This will cause the from purse to lose an arbitrary amount of money *immediately* and to write exception logs. Although money is then recoverable at the bank[7] and all security properties are kept intact by the attack (the attacker just damages the from purse, he does not benefit) we think this behavior is undesirable. The owner of the from purse must go to the bank and force *every* to purse to do the same: the bank will only be to able to give him back the money when it detects matching pairs of exception logs. The motivation for the owners of to purses will be low to do that, since they have not lost any money. They will not notice that the exception log their purse carries does some damage to a from purse, they did not even communicate with.

A proposal to remedy the situation is to send an encrypted startTo message only as a response to startFrom. This would not allow an attacker to create exception logs without having both purses available at the same time (or by pulling out one card in front of his owner). Another solution would be to force purses to respond to a challenge from the terminal to prove, that they are indeed the authentic purses with the correct name. We prefer the first solution, since the second depends on the authenticity of terminals and requires to include them explicitly in the formal model. An analysis of this proposed modification should simplify our invariant. We leave the actual verification for further work.

Summarizing, the effect of the scenario for our verification task is, that any purse na must expect req(pd) messages with na = pd.from and future sequence numbers pd.fromno in the ether. All those messages may be used in future protocol runs, that may fail since pd has already been logged by the to purse pd.to. Note that such future *request* messages are the only ones that may be relevant for a purse in idle state (i.e. in a state of interest): all other encrypted messages satisfy the property that they contain only past sequence numbers.

We therefore have to characterize requests with future sequence numbers with a global protocol invariant. All other properties defined in the next section will be defined local to one purse and its communication partner.

$$
\begin{aligned}
& \mathsf{reqsok}(\mathsf{cstate}) \\
:\leftrightarrow\ & \forall\ \mathsf{pd}. \quad \mathsf{req}(\mathsf{pd}) \in \mathsf{ether} \wedge \mathsf{authentic}(\mathsf{pd.from}) \\
& \qquad \wedge\ \mathsf{nextSeqNo}(\mathsf{pd.from}) \leq \mathsf{pd.fromno} \\
& \qquad \rightarrow \quad \neg\ \mathsf{Val}(\mathsf{pd}) \in \mathsf{ether} \wedge \mathsf{pd} \in \mathsf{toInEpv} \cup \mathsf{toLogged}
\end{aligned}
$$

The predicate says that for all req(pd) messages with a future sequence number, (so the from purse has not yet been involved) a Val answer cannot have

[7] Michael Butlers case study in [CJ07] explicitly considers such a recovery step.

been created and the to purse has either just sent the message or has logged its payment details. All other messages are in the past, they satisfy

$$
\begin{array}{lll}
 & \mathsf{pastether(cstate)} & \\
:\leftrightarrow \forall\ \mathsf{pd}. & (& \mathsf{Val(pd)} \in \mathsf{ether} \vee \mathsf{Ack(pd)} \in \mathsf{ether} \\
 & & \rightarrow \mathsf{authentic(pd.from)} \wedge \mathsf{pd.fromno} < \mathsf{nextSeqNo(pd.from)}) \\
 & \wedge (& \mathsf{req(pd)} \in \mathsf{ether} \vee \mathsf{val(pd)} \in \mathsf{ether} \vee \mathsf{ack(pd)} \in \mathsf{ether} \\
 & & \rightarrow \mathsf{authentic(pd.to)} \wedge \mathsf{pd.tono} < \mathsf{nextSeqNo(pd.to)})
\end{array}
$$

5 Systematic Development of an Invariant

Like for the simulation relation, the basic idea for the development is again to focus on states of interest, i.e. idle states, and to use local invariants for purses. Our local invariant LCINV will use the last past idle state to say

> "the current state cstate of the purse named na is the result of executing some steps of the protocol starting from an idle state oldcstate"

Which steps have been executed can be determined from state(na): if $\mathsf{state(na)} = \mathsf{idle}$ the purse has done no steps, then $\mathsf{oldcstate} = \mathsf{cstate}$. If the state is epr then the purse has successfully executed (i.e. $\mathsf{okstartFrom} \wedge \neg\ \mathsf{fail?}$ holds) a STARTFROM#. Similar clauses result for $\mathsf{state(na)} = \mathsf{epv}, \mathsf{epa}$. Formally LCINV has the following form ($\varphi_{\mathsf{epr,epv,epa}}$ will be explained below):

$$
\begin{array}{l}
\mathsf{LCINV(na, oldcstate, cstate)} \\
:\leftrightarrow\ \textbf{case}\ \mathsf{state(na)}\ \textbf{of} \\
\quad \mathsf{idle} : \mathsf{oldcstate} = \mathsf{cstate} \\
\quad \mathsf{epr} : \quad \mathsf{isOKstartFrom(msg)} \wedge \neg\ \mathsf{fail?} \\
\qquad \wedge \langle \mathsf{STARTFROM\#(msg, na;\ oldcstate)\#} \rangle\ (\mathsf{oldcstate} = \mathsf{cstate} \wedge \varphi_{\mathsf{epr}}) \\
\quad \mathsf{epv} : \quad \mathsf{isOKstartTo(msg)} \wedge \neg\ \mathsf{fail?} \wedge \\
\qquad \wedge \langle \mathsf{STARTTO\#(msg, na;\ oldcstate)\#} \rangle\ (\mathsf{oldcstate} = \mathsf{cstate} \wedge \varphi_{\mathsf{epv}}) \\
\quad \mathsf{epa} : \quad \mathsf{isOKstartFrom} \wedge \neg\ \mathsf{fail?} \\
\qquad \wedge \langle \mathsf{STARTFROM\#(msg, na;\ oldcstate);} \\
\qquad\ \ \mathsf{REQ\#(msg, na;\ oldcstate)} \rangle\ (\mathsf{oldcstate} = \mathsf{cstate} \wedge \varphi_{\mathsf{epa}})
\end{array}
$$

Using this local approach has several advantages, First, LCINV is trivially invariant for all steps into the protocol (STARTFROM#, STARTTO# and REQ#) (and oldcstate will stay the same before and after the step). Second for the steps finishing a protocol run (where oldcstate after the step will be chosen to be the final state of the step), we will essentially prove properties of *full protocol runs of one purse*: e.g. executing an ACK# in state epa yields a proof obligation that considers a full execution STARTFROM#; REQ#; ACK# of a from purse. In essence we will have to verify a "big" diagram consisting of one abstract step and 3 concrete steps.

Compared to the original invariant effort can be concentrated to get the invariant right for full protocol runs. There is no need to explicitly define properties of intermediate protocol states. E.g. property

P-1: $\mathsf{state(na)} = \mathsf{epr} \rightarrow \mathsf{pdAuth(na).value} < \mathsf{balance(na)}$

is implied by LCINV since it is are established by the purse executing STARTFROM# to get into state epr. The same is true for all properties of purses in [SCJ00], p. 26 and also some of the properties of BetweenWorld on p. 42f..

The approach works for the *local* state of purses. It is not sufficient for the (global) ether. Two things are necessary for the ether. First, the global invariant is necessary, that we already derived in the last section.

Second, for each intermediate state of a purse na, we need to characterize the state of its communication partner, which we call other. This can be done by abstracting from the Mondex protocol to any protocol that sends messages forth and back and which has a status that changes on each step. Assuming the protocol enters a state s and sent a message m where the other purse has a response m', we need to say the following:

1. if the response m' has been sent (i.e. is in ether) , then the other purse is either still in the state it reached by sending m' or it has aborted in this state.
2. All messages after m' are not yet in the ether.

For Mondex this means that abbreviating pdAuth(na) as pd we have:

$$\begin{aligned}\varphi_{\mathsf{epr}} :\leftrightarrow\ & \mathsf{req}(\mathsf{pd}) \in \mathsf{ether} \\ & \rightarrow (\mathsf{pd} \in \mathsf{toInEpv} \cup \mathsf{toLogged}) \wedge \neg\, \mathsf{val}(\mathsf{pd}) \in \mathsf{ether} \wedge \neg\, \mathsf{ack}(\mathsf{pd}) \in \mathsf{ether} \\ \varphi_{\mathsf{epv}} :\leftrightarrow\ & \mathsf{val}(\mathsf{pd}) \in \mathsf{ether} \\ & \rightarrow (\mathsf{pd} \in \mathsf{fromInEpa} \cup \mathsf{fromLogged}) \wedge \neg\, \mathsf{ack}(\mathsf{pd}) \in \mathsf{ether}\end{aligned}$$

If m is the last protocol message of one participant (here $\mathsf{m} = \mathsf{val}(\mathsf{pd})$) is the last message of na, then the situation is slightly different. We have to say:

1. the other purse has not aborted
2. if the other has sent the final response (here: ack(pd)), then it is either in state idle, or it has started a new protocol run with a new pdAuth(other).

Formally:

$$\begin{aligned}\varphi_{\mathsf{epa}} :\leftrightarrow (\ & \mathsf{ack}(\mathsf{pd}) \in \mathsf{ether} \\ & \rightarrow \mathsf{pd} \notin \mathsf{toLogged}(\mathsf{pd}) \wedge (\mathsf{state}(\mathsf{pd.to}) = \mathsf{idle} \vee \mathsf{pd} \neq \mathsf{pdAuth}(\mathsf{other}))\end{aligned}$$

Finally, we have to characterize states of interest. All we need for them is that $\mathsf{state}(\mathsf{na}) = \mathsf{idle}$ (of course) and that exception logs have sequence numbers in the past. Again, due to the scenario of the previous section, exception logs in exLog(na) with $\mathsf{pd.from} = \mathsf{na}$ may have future to. Therefore our invariant is defined as:

$$\begin{aligned}&\mathsf{CINV}(\mathsf{cstate}) \\ :\leftrightarrow\ & \mathsf{pastether}(\mathsf{cstate}) \wedge \mathsf{reqsok}(\mathsf{cstate}) \\ & \wedge\, \forall\, \mathsf{authentic}(\mathsf{na}).\exists\, \mathsf{oldcstate}. \quad \mathsf{LCINV}(\mathsf{na}, \mathsf{oldcstate}, \mathsf{cstate}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \mathsf{oldstate}(\mathsf{na}) = \mathsf{idle}\end{aligned}$$

where

$$
\begin{array}{l}
\quad \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate}) \\
:\leftrightarrow \forall\ \mathsf{pd}.\quad (\mathsf{pd} \in \mathsf{fromLogged} \rightarrow \mathsf{pd.fromno} < \mathsf{oldnextSeqNo}(\mathsf{pd.from})) \\
\qquad\qquad \wedge\ (\quad \mathsf{pd} \in \mathsf{toLogged} \\
\qquad\qquad\quad \rightarrow \quad \mathsf{pd.fromno} < \mathsf{oldnextSeqNo}(\mathsf{pd.from}) \\
\qquad\qquad\qquad \wedge\ \mathsf{pd.tono} < \mathsf{oldnextSeqNo}(\mathsf{pd.to}))
\end{array}
$$

Note that it is not necessary to prove $\mathsf{pd.from} \neq \mathsf{pd.to}$ for exception logs as in the original invariant.

To verify that CINV is a global invariant for CSTEP# we again reduce this property to lemmas about the local invariant. In this case we need two lemmas: one for the case where the purse na of the local invariant is the same as the receiver of the message, and one where it is different. Both lemmas need the local invariant for na and its communication partner other:

$$
\begin{array}{l}
\quad \mathsf{LCINV}(\mathsf{na}, \mathsf{oldcstate}, \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate}) \\
\wedge\ \mathsf{LCINV}(\mathsf{other}, \mathsf{oldcstate}', \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}', \mathsf{cstate}) \\
\wedge\ \mathsf{pastether}(\mathsf{cstate}) \wedge \mathsf{reqsok}(\mathsf{cstate}) \wedge \mathsf{msg} \in \mathsf{ether} \\
\wedge\ \mathsf{authentic}(\mathsf{receiver}) \wedge \mathsf{oldstate}(\mathsf{na}) = \mathsf{idle} \wedge \mathsf{na} = \mathsf{receiver} \\
\rightarrow \langle\!|\mathsf{LCSTEP\#}(\mathsf{msg}, \mathsf{receiver}, \mathsf{fail?};\ \mathsf{oldcstate})|\!\rangle \\
\qquad \langle \mathbf{if}\ \mathsf{state}(\mathsf{receiver}) = \mathsf{idle}\ \mathbf{then}\ \mathsf{oldcstate} := \mathsf{cstate} \rangle \\
\qquad\quad (\quad \mathsf{LCINV}(\mathsf{na}, \mathsf{oldcstate}, \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate}) \\
\qquad\quad \wedge\ \mathsf{pastether}(\mathsf{cstate}) \wedge \mathsf{reqsok}(\mathsf{cstate}) \wedge \mathsf{oldstate}(\mathsf{na}) = \mathsf{idle})
\end{array}
\tag{4}
$$

$$
\begin{array}{l}
\quad \mathsf{LCINV}(\mathsf{na}, \mathsf{oldcstate}, \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate}) \\
\wedge\ \mathsf{LCINV}(\mathsf{other}, \mathsf{oldcstate}', \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}', \mathsf{cstate}) \\
\wedge\ \mathsf{pastether}(\mathsf{cstate}) \wedge \mathsf{reqsok}(\mathsf{cstate}) \wedge \mathsf{msg} \in \mathsf{ether} \\
\wedge\ \mathsf{authentic}(\mathsf{receiver}) \wedge \mathsf{oldstate}(\mathsf{na}) = \mathsf{idle} \wedge \mathsf{na} \neq \mathsf{receiver} \\
\rightarrow \langle\!|\mathsf{LCSTEP\#}(\mathsf{msg}, \mathsf{receiver}, \mathsf{fail?};\ \mathsf{oldcstate})|\!\rangle \\
\qquad \mathsf{LCINV}(\mathsf{na}, \mathsf{oldcstate}, \mathsf{cstate}) \wedge \mathsf{pastexlog}(\mathsf{oldcstate}, \mathsf{cstate})
\end{array}
\tag{5}
$$

The first lemma now explicitly states, that oldcstate must be changed iff the step of receiver leads to idle state. Note also that the two purses na and other have started their protocol run in (potentially) different states oldcstate and oldcstate′. The lemmas have one level of quantification hidden in the definitions of pastether, pastexlog and reqsok. We cope with this level of quantification by using the simple heuristic that after symbolic execution of the ASM rules unfolds the definitions and instantiates the resulting quantifier with identity (i.e. to prove $\varphi(\mathsf{pd})$ in the postcondition, we need the precondition exactly for this same pd).

The proofs are nearly fully automatic: two interactions are needed for lemma (4), one for a case split that is not found automatically and one to unfold reqsok in the one place where it is needed. The second lemma (5)needs one interaction to distinguish the four cases of state(na) right at the start, and two interactions for case splits.

Compared to the original proof, which had 447 steps proofs are much bigger: the first lemma has 698 steps, the second 2233. The increase of proof size stems from the finer granularity of symbolic execution, which dominates these proofs: symbolic execution counts each step individually, while purely algebraic reasoning used predominantly in the original proof, applies an arbitrary number of rewrite rules in one step. On the other hand, the original proof was much more complex: it had 71 interactions, since each of the 20 properties of the original invariant is a quantified formula, and their invariance proof is heavily interrelated: each case needs other preconditions to prove a certain property invariant, as can be seen from the informal proofs in [SCJ00] as well. Therefore the quantifiers had to be instantiated interactively. Even, when adding the 19 interactions for lifting the two local lemmas to the invariance theorem (118 steps) this is still a significant improvement.

Working out the proof, once the right aproach was found, took around 2 weeks. The main effort here was to find the right approach to solve the problem. Several weeks were spent trying to find out, *why* purely local invariants always failed to work and to figure out the worst-case scenario of the previous section. Summarizing, to work out the full case study using ASM refinement required about 2 person months of work.

Full proofs can be found again in the Web presentation [KIVb]: specification *CINV* contains the main invariance theorem *CINV-lem*, together with the two lemmas (4) and (5) named *CINV-receiver-lem* and *CINV-not-receiver-lem*. Specification *CINV-props* defines the rewrite rules for CINV used to close the premises of the simulation proofs.

6 Related Work

Prior to our work [BJPS06] showed that it is possible to define a forward simulation for the Mondex scenario restricted to exactly one from and one to purse. This work also suggested using generalized forward simulations. Although the complexity of interleaving protocol runs is absent in this scenario, and some of the reasoning of the paper is informal, this work was rather influential for ours. It is interesting to see, that our forward simulation when restricted to the scenario with two purses differs slightly from theirs: While in ours a transition from epr to epa implements TRANSFERFAIL#, when the to purse has already logged, in their refinement it implements **skip** and the failed transfer only happens for the subsequent ABORT#. We think our solution is more natural since this money is already definitely lost[8]. Their solution can be derived using our approach, but is more complex, since it would require to execute TRANSFERFAIL# backwards for definitely lost money in the simulation (to recover this money).

Parallel to our work in [SGHR06] several other groups have successfully formalized and verified the Mondex case study. Their results will all be published in [CJ07][9]: [TR06] demonstrates that Alloy and bounded model checking can find

[8] this is the set definitelylost as defined in the original case study [SCJ00].

[9] many of the results can already be found as talks at the Mondex workshop [Mo0].

all the problems we found and one more in the proof structure we did not use. The RAISE development in [HGS06] shows an interesting alternative to develop the protocol: it starts with a send and receive instead of a transfer operation (our ASTEP#). The case study develops the communication protocol with two refinements using a variant of forward simulation. This development has a rather well structured invariant, and many of the formulas used in this development are quite close to the ones we use (e.g. φ_{epv} can be found in this case study). There are also differences: the formalism used is purely algebraic, and many properties are defined for intermediate states. Nevertheless proving the Mondex refinement in two steps, one that splits money transfer into send and receive, and another that develops the protocol could be an improvement for our development too.

The idea of splitting the refinement into smaller steps is taken to its extreme by Michael Butler's group: their development splits the original development into 9 very small refinements that could be verified in very short time and with very good automation using the B4free tool.

A large part of the proof of the Mondex refinement has also been done by David Crocker using the resolution based prover Perfect Developer.

[JW06] use the original Z specification and also the original proofs within the Z-Eves tool. By avoiding any translation into another formalism, their approach found a lot of small problems, that no one else could find.

Our idea of using local invariants is a common idea in ASM refinement, it is used e.g. as the core idea in [BM96], which verifies a refinements from sequential to pipelined execution instruction of instructions of the DLX processor using localized invariants for each pipeline stage. The idea is also not specific to ASM refinement, it can be found in other refinement notions, e.g. in work that relates promotion in Z specifications and data refinement (see [DB01] for an overview).

Our use of states of interest on the other hand seems rather particular to ASM refinement. It was used informally in [BR95] for the compilation of Prolog to WAM, in our formal proofs to verify them [SA98] and was a key notion in the formalization of ASM refinement [Sch01]. The term *states of interest* itself was coined in [Bör03]. The only related refinement notion outside of ASM refinement we are aware of is coupled refinement [DW03] which uses past states of interest (as shown in [Sch05]).

For invariants the idea is closely related to old ideas of using invariants that "sometimes" instead of "always" hold [Bur74], that we used in KIV for a long time [HRS89].

Recently states of interest were also used in [HGRS06], [Han06] to analyze security protocols. The idea is to focus on states after all possible attacks have been tried.

7 Conclusion

In this paper we have shown how techniques of ASM refinement, namely focusing on *states of interest* and defining *local* invariants can be used to systematically define a simulation and an invariant for the Mondex refinement.

Our technique has resulted in a simple forward simulation for the Mondex case study, that can be verified with a very high degree of automation.

The systematic definition of an invariant has led us to discover a weakness of the protocol with respect to denial of service attack. It should be debated, whether the weakness is serious enough to change the protocol as suggested.

Although this largely remains future work, we hope that the techniques we used are also applicable for a wider range of security protocols. A first result in this direction is that they can be used in security proofs on an abstract cryptography level [HGRS06].

Our work is part of the more ambitious goal to develop verified JavaCard code for Mondex: a refinement of the communication protocol to a protocol using abstract cryptography has been verified and will be described in our contribution to [CJ07], and we are currently working on a refinement to Java Code.

References

[BJPS06] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the purse: The balance enquiry quandary, and generalised and (1,1) forward refinements. *Fundamenta Informaticae 77*, 2006.

[BM96] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1996.

[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.

[Bur74] R. M. Burstall. Program proving as hand simulation with a little induction. *Information processing 74*, pages 309–312, 1974.

[CB99] UK ITSEC Certification Body. UK ITSEC SCHEME CERTIFICATION REPORT No. P129 MONDEX Purse. Technical report, UK IT Security Evaluation and Certification Scheme, 1999. URL: http://www.cesg.gov.uk/site/iacs/itsec/media/certreps/CRP129.pdf.

[CJ07] J. Woodcock C. Jones, editor. *(no title yet)*. Formal Aspects of Computing, 2007. (Journal, to be published).

[CSW02] D. Cooper, S. Stepney, and J. Woodcock. Derivation of Z Refinement Proof Rules: forwards and backwards rules incorporating input/output refinement. Technical Report YCS-2002-347, University of York, 2002. URL: http://www-users.cs.york.ac.uk/∼susan/bib/ss/z/zrules.htm.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001.

[DW03] J. Derrick and H. Wehrheim. Using Coupled Simulations in Non-atomic Refinement. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651, pages 127–147. Springer LNCS, 2003.

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

[Han06] Dominik Haneberg. *Sicherheit von Smart Card – Anwendungen*. PhD thesis, University of Augsburg, Augsburg, Germany, 2006. (in German).

[HGRS06] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. Technical Report 2006-08, Universität Augsburg, 2006.

[HGS06] A. E. Haxthausen, C. George, and M. Schütz. Specification and Proof of the Mondex Electronic Purse. In M. Reed C. Xin, Z. Liu, editor, *Proceedings of 1st Asian Working Conference on Verified Software, AWCVS'06, UNU-IIST Reports 348, Macau*, Nov. 2006.

[HRS89] Maritta Heisel, Wolfgang Reif, and Werner Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.

[JW06] L. Freitas J. Woodcock. Z/eves and the mondex electronic purse. In A. Cerone K. Barkaoui, A. Cavalcanti, editor, *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium*, LNCS 4281, pages 14 – 34, Tunis, 2006. Springer.

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

[KIVb] Web presentation of the asm refinement for mondex in KIV. URL: http://www.informatik.uni-augsburg.de/swt/projects/mondex-ASM.html.

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

[Mo0] Talks of the third mondex workshop in york, 5.-6. october 2006. URL: http://www.gc6.clrc.ac.uk/gc6wiki/ThirdMondexWorkshopAgenda.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6:85–128, 1998.

[RSSB98] Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[SA98] Gerhard Schellhorn and Wolfgang Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, pages 165 – 194. Kluwer Academic Publishers, Dordrecht, 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://hyperg.iicm.tu-graz.ac.at/jucs/.

[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.

[SCJ00] S. Stepney, D. Cooper, and Woodcock J. 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.

[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.

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

[TR06] D. Jackson T. Ramananadro. Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. URL: http://www.eleves.ens.fr/home/ramanana/work/mondex/, 2006.

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