

ASM Refinement Preserving Invariants

Gerhard Schellhorn

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

Abstract. This paper gives a definition of ASM refinement suitable for protocol verification that has been formalized in the interactive theorem prover KIV. We used this definition as the basis of the verification of the refinements of the Mondex case study [SGH⁺07], [HSGR07]. The refinement definition we give differs from the ones we gave in [Sch01] which preserve partial and total correctness assertions. The reason is that the main goal of the refinement of the Mondex protocol is to preserve an invariant, while total correctness is not preserved. To preserve invariants, the definition of generalized forward simulation is limited to the use of “small” m:1 diagrams, and we show a technique that enables us to recover the idea of “big” m:n diagrams.

1 Introduction

This paper gives a formal definition of the ASM refinement theory underlying the mechanical verification of the Mondex protocol in KIV that is described in [SGH⁺07] and also used in the second refinement to a security protocol in [HSGR07].

Although the theory developed is rather different, our work can be viewed as a parallel to [CSW02], which also defines a suitable data refinement theory for the original paper-and-pencil verification of Mondex in [SCW00].

The Mondex case study deals with Mondex smart cards that implement an electronic purse (see [MCI]). The main content of the case study is a refinement of the atomic action “transfer money” into a communication protocol between two purses. More details on this protocol are given in Section 2. On first glance the refinement looks very similar to problems in compiler verification which typically also refine one atomic step — execution of one source code instruction — to several steps of executing corresponding assembler code.

But there are three key differences: the refinement must preserve an invariant, the presence of an attacker means that total correctness is not preserved, and protocol steps are local steps of individual purses which can be interleaved.

Therefore we had to modify our formalization [Sch01] and [Sch05] of ASM refinement, which was based on Börger’s concepts ([Bör90], [Bör03]) that he introduced into the framework of Gurevich’s ASMs ([Gur95], [BS03]).

The original formalization is summarized in Section 3 for comparison and to provide the necessary notation. The new definition of *invariant preserving ASM refinement* is given in Section 4.

To preserve invariants forces the use of “small” $n:1$ (usually $0:1$ and $1:1$) commuting diagrams in the definition of forward simulations. Section 5 shows a technique to define simulation relations that either look into the future or the past of a protocol run. This technique is a key technique that allows to get back to the verification of the natural “big” $1:n$ diagrams that prove that 1 abstract transaction is implemented by n protocol steps.

Finally, Section 6 gives related work and Section 7 concludes.

2 Mondex Purses

Mondex smart cards implement an electronic purse [MCI]. They are intended to replace cash money when paying in shops: typing in some amount into a terminal with two smart card slots, money is transferred from one card to another.

Mondex cards are famous for having been the target of the first ITSEC evaluation of the highest level E6 [CB99], which requires formal specification and verification.

The formal specification and paper-and-pencil proofs were done in [SCW00] using the Z specification language. Two models of the 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. A suitable data refinement theory was developed in [CSW02].

The Mondex case study has been recently proposed as a challenge for theorem provers that we and several other groups [JW07] have solved. Initially we followed the original approach and verified the original backward simulation [SGHR06] which used data refinement with KIV. As a second step we used ASM refinement to develop another proof [SGH⁺07]. The resulting (generalized forward) simulation relation and invariants were more systematic and proofs could be better automated in KIV, than those of data refinement. The experiment using ASMs and ASM refinement also lead us to discover a weakness of the protocol, that required a small change.

While [SGH⁺07] concentrates on the case study and the difficulties of verification this paper focuses on the underlying ASM refinement theory. Therefore we only give a short summary of the main ideas of the Mondex protocol refinement.

Both levels of the refinement are defined as abstract state machines (ASM, [Gur95], [BS03]). The abstract level consists of the following simple rule that describes money transfer between two smart cards.

```

TRANSFER#
choose from, to, value, fail?
with from  $\neq$  to  $\wedge$  value  $\leq$  balance(from)
in if  $\neg$  fail? then balance(from) := balance(from) - value
                    balance(to) := balance(to) + value
                    else balance(from) := balance(from) - value
                        lost(from) := lost(from) + value

```

The ASM rule chooses two different purses `from` and `to` and an amount `value` of money, that should be transferred from the `from` purse to the `to` purse. The amount is checked not to be higher than the available money `balance(from)` of the `from` card. If the randomly chosen boolean variable `fail?` is false, the transfer is successful and appropriate updates are done on `balance(from)` and `balance(to)`. The transfer may also fail for various reasons, like the card being pulled out of the card reader, power failure, or lack of sufficient memory on the card. An attack on the transfer protocol using faked cards or a faked terminal might also cause the money transfer to fail. In the abstract rule all these cases are represented as `fail? = true`, and `value` is added to `lost(from)` in this case. One key goal of the refinement is to design a protocol, that will keep track of the money in `lost` (so that it is never truly lost!). The main advantage of the abstract specification is that the two key security properties of the final protocol are trivially provable as invariants: first, money transfer will never generate money: the sum of all values `balance(purse)` for all authentic purses will never increase. Second, the sum of all balances and lost values will remain constant in all transfers, meaning all lost money has been accounted.

The concrete level of Mondex defines a communication protocol between two purses. A successful protocol run is shown as an activity chart (drawn horizontally) in Fig. 1. The figure shows that the protocol uses five messages: the `startFrom`, `startTo`, `req(uest)`, `val(ue)` and `ack(nowledge)` message. When a message is received the corresponding process step is taken, e.g. ASM rule `REQ#` is executed to process a request and to produce a value message. To keep track how far a purse has progressed in the protocol, the ASM uses a control state (indicated on the swim lane of the purse): e.g. control state `epa` (“expecting acknowledge”) indicates that the `from` purse has just sent the `val` message. A purse not running a protocol is in `idle` state.

The two first messages exchange relevant information to authenticate the two purses. The `val` message carries the money, so `REQ#` subtracts `value` from the `from` purse, and `VAL#` adds `value` to `balance(to)`. Communication is done indirectly using a global set of messages, called the `ether`. Receiving a message is done by picking some message from `ether`, sending a message adds the message to `ether`. Receiving *any* message from `ether` and not just picking the one that was previously sent implicitly models an attacker that may record, replay or delete arbitrary messages. If a purse picks a message other than the one shown in the successful run in Fig. 1, it aborts the protocol.

When a `to` purse has sent a `req` message and entered state `epv`, but does not get a `val` message back it aborts the protocol run writing an exception log. In this case either `req` or `val` has been lost. Dually, the `from` purse creates an exception log, when it has received a `req` message and has sent the `val`, but does not get an `ack` in state `epa`. The key idea is that *both* exception logs are created if and only if the `val` message is lost. Losing the `val` message is the only case where money is lost, and is equivalent to the step on the abstract layer where money is added to `lost`. Therefore the abstract `lost` component is implemented as the

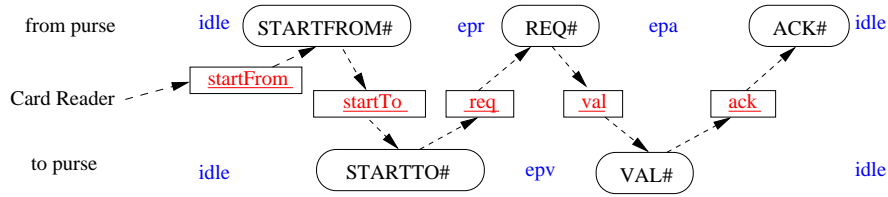


Fig. 1. Activity chart for the Mondex protocol

sum of all values contained in pairs of matching exception logs. Purse owners who suffered from a failed transaction can go to the bank, show their two cards with matching exception logs, and the bank can deduce that it should give them back their money.

Details on the data structures used as local components of each purse state and the full ASM rules can be found in [SGH⁺07]. For the ASM refinement theory the following facts should be remembered:

- Successful transfers are implemented by protocol runs as shown in Fig. 1
- Failed transfers are implemented by aborted protocol runs. There are aborted protocol runs which do not lose money. Those which do lose money create two matching exception logs.
- Protocols can be interleaved: A purse which has done its last protocol step can start a new protocol run even before the other purse has finished the run.
- The state of the ASM is composed of the states of the local purses, with the exception of the global set of messages `ether`
- The abstract level satisfies the invariant „no money generated or really lost”, and the refinement should preserve this invariant.

3 ASM Refinement in Compiler Verification

ASM refinement theory was formalized in [Sch01] [Sch05] with compiler verification (in particular the Prolog-WAM compiler of [BR95]) as the intended target. In this section we will repeat key definitions and informally give the results of this work. This is done to establish the necessary notations for ASMs and commuting diagrams. They are needed to define the modifications necessary for the main result of the next section needed on protocol verification.

In compiler verification the semantics of the source code language is defined by giving an ASM $AM = (AS, AIN, ARULE, AOUT)$ that acts as an abstract interpreter of the source code language: AS are the states (algebras) of an ASM AM . The program source code and an initial program counter are stored in the possible initial states $AIN \subseteq AS$ of AM . One application of the ASM rule $ARULE$ typically executes one source code instruction. Execution may terminate in a final state in $AOUT$. The compiler transforms the program to

byte code or assembler code (again stored in the initial state CIN). An ASM $CM = (CS, CIN, CRULE, COUT)$ executes these instructions. A relation IR between the initial states gives compiler assumptions (this generalizes the case where a compiler function is given).

Typically one source code instruction is compiled to several assembler instructions (1:n diagrams), but occasionally it is useful to consider the more general case of m:n diagrams, where m source code instructions are compiled to n assembler instructions (see [BR95], [SA98]). Corresponding states at the beginning and end of the diagrams are called *states of interest*. Refinement is defined relative to a correspondence relation IO that should hold for states of interest. IO typically states that inputs read and outputs produced are the same (relative to possible differences in representation) for both ASMs or that relevant parts of the states match. Given a finite concrete run, refinement requires that there must be a finite abstract run, such that IO holds initially and at the end of every diagram. For final states IO must imply some output relation OR, which typically says that executing both programs must have produced the same result. For infinite runs refinement requires that IO must hold infinitely often.

Fig. 2 gives the two situations of finite and infinite runs. It can be shown that ASM refinement preserves partial and total correctness assertions modulo the relation IO.

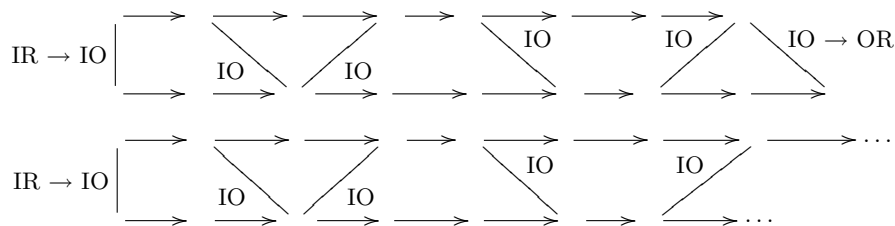


Fig. 2. Refinement in Compiler Verification

To verify refinements a generalized forward simulation INV is defined. This relation always strengthens IO and sometimes splits the diagrams of Fig. 2 into smaller ones. The proof obligation for generalized forward simulation propagates INV forwards and reads informally as:

Given two states as and cs of AM and CM for which $INV(as, cs)$ holds, every run of CM starting with cs must reach a state cs' such that a commuting diagram can be closed: a run of AM from as to some as' must exist, such that $INV(as', cs')$ holds again.

Typically the number of steps m and n of CM and AM that are needed to get the next commuting m:n diagram can be determined from the states as, cs . Of course m:n must not both be zero. Triangular diagrams are allowed, but infinite

chains of 0:n and m:0 diagrams must be forbidden using well-founded orders $<_{0n}$ and $<_{m0}$, otherwise the correspondence of *finite* runs to *finite* runs cannot be established. Also, if an application of CRULE fails (due to clashes, or recursion in TurboASMs; see [BS03]) it must be possible to construct a run of AM that leads to a failed application of ARULE. The four cases of the proof obligation are shown in Fig.3. ARULE^+ is a positive number of applications of ARULE and the \perp in case (C) of Fig.3 indicates a failed rule. Dashed lines, as well as states and formulas after “ \vdash ” have to be shown to exist or to be provable.

Formal proof obligations using either temporal operators or Dynamic Logic can be found in [Sch05].

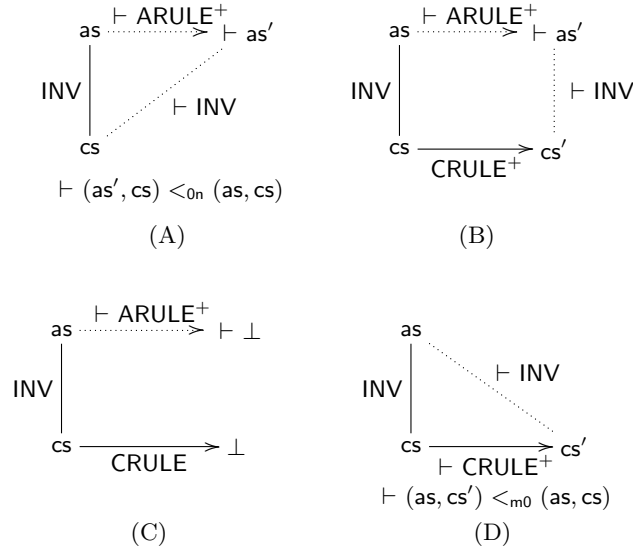


Fig. 3. The commuting diagrams of ASM refinement

4 ASM Refinement Preserving Invariants

At first glance, verification of the Mondex protocol looks quite similar to compiler verification: an atomic abstract step – money transfer – is refined by n smaller, concrete steps (here the 5 steps of the Mondex protocol), but there are three key differences.

First, compiler verification typically preserves termination: executing the compiled code of a program should result in an infinite run only if the semantics of the original program allowed an infinite run too. But the Mondex refinement does not preserve termination because of failed protocol runs: those failed runs

which do not create two matching exception logs correspond to doing nothing on the abstract layer (in the terminology of data refinement they implement “skip” steps). This happens e.g. if an attacker deletes all request messages. Therefore the protocol refinement may create an infinite chain of triangular 0:n diagrams. It only preserves partial correctness, but not termination and total correctness.

Second, the main requirement of the Mondex refinement is not (as for compiled code) that the abstract and concrete states correspond in states of interest, where some code sequence has been executed. The critical properties the refinement must preserve are the two security invariants, that say that money is never generated nor lost.

The original ASM refinement only weakly preserves invariants: an invariant AINV of the abstract level will only hold (modulo the IO correspondence) in every concrete state of interest.

Unfortunately this is not sufficient to establish security on the concrete level, where *all* intermediate states in the middle of protocols should also satisfy the security invariant.

Finally, the third difference is that protocol runs can be interleaved. Intuitively, this does not matter since two interleaved protocol runs will modify disjoint pairs of purses, and the messages one protocol run generates will not be usable in other runs. But in particular the second fact is essential for security: replay attacks would be possible if it were false. Therefore it is not sufficient to prove that a non-interleaved protocol run implements an abstract step.

Therefore, we have to specialize our refinement theory to refinement that preserves invariants and to forward simulations that consist of n:1 diagrams only. Only these small diagrams propagate an abstract invariant (modulo IO) to *every* state of the protocol. For Mondex we will even use 0:1 and 1:1 diagrams only. We first define invariants:

Definition 1. [invariant]

A predicate $\text{AINV}(\text{as}_0, \text{as})$ is an invariant of ASM AM, if

- $\text{AINV}(\text{as}_0, \text{as}_0)$ holds for every initial state $\text{as}_0 \in \text{AIN}$.
- $\text{AINV}(\text{as}_0, \text{as})$ and $\text{ARULE}(\text{as}, \text{as}')$ imply $\text{AINV}(\text{as}_0, \text{as}')$.

Our definition of an invariant generalizes the usual definition by allowing it to refer to the current state *as as well as* to the initial state as_0 from where the run started. This is necessary since the invariant of the abstract Mondex specification indeed compares the sum of balances of the initial state with those of the current state.

Definition 2. [Preservation of invariants] Given two ASMs AM and CM, and correspondences IR, IO and OR between initial states, states of interest and final states, the refinement from AM to CM preserves invariants, if for every invariant AINV of AM

$$\text{CINV}(\text{cs}_0, \text{cs}) := \exists \text{as}_0, \text{as}. \text{IR}(\text{as}_0, \text{cs}_0) \wedge \text{IO}(\text{as}, \text{cs}) \wedge \text{AINV}(\text{as}_0, \text{as})$$

is an invariant of CM.

Note that for IR and IO both being identity, the formula can be simplified to show that AINV itself is an invariant of CM.

For the Mondex case study IR and IO are identity for the **balance** of purses, but intersections of exception logs replace the **lost** component. Therefore the resulting invariant for the protocol is slightly more complex than the one of the abstract layer. Informally, the sum of money that is constant throughout all protocol runs is the sum of

- all balances of purses,
- all money that has been sent and can still be received (is “in transit”), and
- all money contained in matching pairs of exception logs.

If all purses are idle, the second summand is empty. Some details on the simulation relation IO for Mondex that is necessary to derive this invariant (by applying Theorem 1 below) will be given in the next section. A formal definition using predicates *intransit* and *lostafterabort* defined in [SGH⁺07] can be found in the Web presentation of Mondex [KIV] in specification *Refinement-preserves-SecProp*.

To verify that a refinement preserves invariants we need a generalized forward simulation INV that only uses n:1 diagrams. This means that *all* concrete states will have a corresponding abstract state from which the invariant can be deduced using IO. As in [Sch01] we first consider ASM rules that never fail. For these the semantics of the ASM rule is a relation between states and the main proof obligation necessary to have a commuting now looks like

$$\begin{aligned} & \text{INV}(\text{as}, \text{cs}) \wedge \text{cs} \notin \text{COUT} \\ \rightarrow & \text{EF}^+(\text{as}, \lambda \text{as}'. \text{INV}(\text{as}', \text{cs}) \wedge (\text{as}', \text{cs}) <_{m0} (\text{as}, \text{cs})) & (1) \quad (\text{VC1}) \\ & \vee (\forall \text{cs}'. \text{CRULE}(\text{cs}, \text{cs}') \rightarrow \text{EF}(\text{as}, \lambda \text{as}'. \text{INV}(\text{as}', \text{cs}))) & (2) \end{aligned}$$

For a pair of states with INV(as, cs), where cs is not a final state in COUT the proof obligation requires, that one of the two possible commuting diagrams (1) and (2) may be added

- either there is a positive number of executions (temporal operator EF⁺) of ARULE such that the state as' reached satisfies INV(as', cs) and some well-founded order <_{m0} has been decremented (this prevents an infinite chain of m:0 diagrams) or
- for every state cs' that can be reached from cs by executing CRULE, there are some (possibly zero) abstract steps (operator EF) that lead from as to as' and INV(as', cs') holds at the end.

The first case allows finitely many m:0 diagrams just as case (A) of the original definition given in Fig. 3 does. The second case allows m:1 diagrams with m ≥ 0. This corresponds to cases (B) and (D) in Fig. 3. The requirement that a well-founded order decreases in 0:n diagrams has been dropped, since there is no need to preserve termination. In diagrams (B) and (D) there must now be one concrete step (CRULE) instead of any positive number (0:1 and m:1 diagrams instead of 0:n and m:n diagrams with n > 0).

To be able to use syntactic ASM rules¹ $\text{CRULE}\#$ instead of a semantic relation CRULE , we recast the proof obligation in Dynamic Logic ([HKT00]):

$\text{CRULE}(\text{cs}, \text{cs}') \rightarrow \varphi(\text{cs}')$ is replaced by $[\text{CRULE}\#(\text{cs})]\varphi(\text{cs})$
 (“all executions of $\text{CRULE}\#$ result in a state where φ holds”) and

$\text{EF}^+(\text{as}, \lambda \text{as}', \psi(\text{as}'))$ becomes $\exists i. \langle \text{ARULE}\#(\text{as})^{i+1} \rangle \psi(\text{as})$
 (“some (pos.) iterated execution of $\text{ARULE}\#$ will lead to a state with ψ ”).

For more general ASM rules that may fail we have to use a set of states which in addition to the algebras over some signature includes a special \perp element to indicate failure. The semantic relation of an ASM rule is then a strict relation over such states. The fact, whether an ASM rule may fail or must fail can be encoded in Dynamic Logic² using divergence to represent failure:

$\text{mayfail}(\text{cs}) := \neg \langle \text{CRULE}\#(\text{cs}) \rangle \text{true}$, $\text{mustfail}(\text{cs}) := \neg \langle \text{CRULE}\#(\text{cs}) \rangle \text{true}$

(informally: “not every run of $\text{CRULE}\#$ terminates (in a state where true holds)” and “not any run of $\text{CRULE}\#$ terminates”).

Using these definitions, the proof obligation using states with \perp can be simplified such that all occurrences of \perp are removed. The process of removing \perp from proof obligations is basically the same as in data refinement, it is more complex since m:n instead of 1:1 diagrams are used, and because ASM rules already have a semantics over states with \perp : data refinement operations are usually embedded using either the behavioral or the contract embedding (see [DB01] for the definitions), which in essence yields two specific types of ASM rules as described in [Sch05]. We get the following proof obligation:

$$\begin{aligned} & \text{INV}(\text{as}, \text{cs}) \wedge \text{as} = \text{as}_0 \wedge \neg \text{cs} \in \text{COUT} \\ \rightarrow & \quad \exists i. \langle \text{ARULE}\#(\text{as})^{i+1} \rangle (\text{INV}(\text{as}, \text{cs}) \wedge (\text{as}, \text{cs}) <_{\text{m0}} (\text{as}_0, \text{cs})) \quad (\text{VC2}) \\ & \vee \quad [\text{CRULE}\#(\text{cs})] \exists i. \langle \text{ARULE}\#(\text{as})^i \rangle \text{INV}(\text{as}, \text{cs}) \\ & \quad \wedge (\text{mayfail}(\text{cs}) \rightarrow \exists i. \langle \text{ARULE}\#(\text{as})^i \rangle (\neg \text{final}(\text{as}) \wedge \text{mayfail}(\text{as}))) \end{aligned}$$

The proof obligation has an additional case (last line) which corresponds to case (C) of Fig. 3.

Theorem 1. [Generalized forward simulation] A refinement from AM to CM preserves invariants if

- $\forall \text{cs} \in \text{CIN}. \exists \text{as} \in \text{AIN}. \text{IR}(\text{as}, \text{cs})$ (“initialize”)
- $\text{IR}(\text{as}, \text{cs}) \rightarrow \text{INV}(\text{as}, \text{cs})$ (“establish invariant”)
- verification condition (VC1)/(VC2) holds for nonfailing/potentially failing rules (“preserve simulation relation”)
- $\text{INV}(\text{as}, \text{cs}) \rightarrow \text{IO}(\text{as}, \text{cs})$ (“simulation relation implies refinement relation”)

¹ Adding a $\#$ sign is the KIV convention to distinguish ASM rules from predicates

² Original Dynamic Logic does not have formulas $\langle \alpha \rangle \varphi$ but they can be added and are equivalent to $\text{wp}(\alpha, \varphi)$ from Dijkstra’s wp calculus

Although the proof is rather tricky to do formally, since it involves infinite runs (the axiom of choice and a diagonalization argument are necessary to construct infinite abstract runs) the intuition behind the proof is simple: compose the commuting diagrams that (VC1) resp. (VC2) provide. The proof proceeds like the one in [Sch05], it is slightly simpler, since using $m:n$ diagrams with $n=1$ allows to construct a corresponding abstract run to a finite concrete run by induction over the length of the concrete run.

5 Simulation Relations that Look into the Future/Past

The refinement theory of the previous section restricts possible commuting diagrams to “small” $m:1$ diagrams (where in fact m will be 0 or 1) and to define a simulation relation that relates every intermediate protocol state to some abstract state. This is unfortunate, since the basic idea of protocol verification is to show commutativity of a “big” $1:5$ diagram that consists of abstract money transfer and the 5 protocol steps.

The small diagrams are needed only to preserve the security invariant for every intermediate state, while the states of interest in protocol verification are only those states where every purse has control state `idle`, i.e. no protocol is running. In this section we show a technique that allows to essentially verify such a big diagram again. Although we give some idea how the formulas look in the Mondex case, the reader should have no trouble to see that the idea is applicable for any protocol refinement which consists of the local states of the individual participants and a global communication state. Section 6 will give several other examples where the same idea has been successfully applied. In Mondex the local states are the local purse states, and the global communication state is the set `ether` of messages that were already sent.

A first technical observation is, that even to verify the “big” $1:n$ diagram we will need an invariant `CINV` of the protocol ASM. Formally, an invariant is easily integrated into the forward simulation `INV` by defining

$$\text{INV}(\text{as}, \text{cs}) := \text{ACINV}(\text{as}, \text{cs}) \wedge \text{CINV}(\text{cs})$$

where `ACINV` is the “real” simulation relation (which is identical to the refinement relation `IO` in this case). Splitting `INV` allows to develop `ACINV` and `CINV` separately, both times using an instance of the technique we describe now. We will talk about a “(coupling) invariant” when we mean both `ACINV` and `CINV` in the following.

The idea is to first define a *simple* (coupling) invariant `SCINV/SACINV` that only needs to hold, when purses are `idle`. For Mondex, `SACINV` states that abstract and concrete `balances` are equal and that `lost` is the intersection of exception logs. `SCINV` needs to say nothing about local purse states, as they are irrelevant when no protocol is running. But it needs to state the security assumption for the global `ether`, which says that messages are suitably encrypted such that the `ether` never contains messages that will be used in future protocol runs. Indeed, if such messages would be available to an attacker, he could load

these messages on a faked card, and could use this card to gain money. This property was not completely true for the original Mondex protocol, which lead us to discover the attack described in [SGH⁺07]: the protocol described in 2 has therefore been slightly changed (the `startTo` message is now an encrypted answer to `startFrom`) from the original protocol in [SCW00]).

The main question to solve now is: how do we derive a (coupling) invariant for intermediate states of the protocol? The idea is to do this schematically using a (coupling) invariant that either *looks into the future or the past* of a protocol run. Either we can say:

- Future: from the current state a future state is reachable, where the simple (coupling) invariant holds or
- Past: Every state is reachable from some state of interest, where the simple (coupling) invariant held

Looking into the future or past is easy to express in Dynamic Logic, since it just corresponds to iterated execution of `CRULE#`. For `ACINV` the resulting formulas are

- $\exists i. \langle \text{CRULE\#}(cs) \rangle \text{SACINV}(cs, as)$
 (“all runs of protocol steps will eventually reach a state with `SACINV`”)
- $\exists i, cs_0. \text{SACINV}(cs_0, as) \wedge \langle \text{CRULE\#}(cs_0)^i \rangle cs_0 = cs$
 (“the current state has been reached by steps that started in a state of interest `cs0` where `SACINV` was true.”)

For `CINV` the abstract state `as` has to be dropped. For a (coupling) invariant looking into the future verifying a commuting diagram for all but the first step of a protocol run creates a trivial 0:1 diagram, since after one step of the protocol the states that will be reached at the end of the protocol are still the same states that were reachable before the step. The only interesting steps to verify are initial states of a protocol, since the future states reachable after one step into the protocol are the ones at the end of the diagram. This means the proof obligation for this case will be exactly the 1:n diagrams that we intended to verify. Looking into the past dually creates trivial 0:1 diagrams for all but the last step of a protocol run, and the 1:n diagram for the last step.

For the simulation relation of Mondex, looking into the future is the simpler approach, since all purses can simply abort the protocol in one step. On the abstract level, corresponding actions are failed transfers for those purses *from* and *to*, where a `val` message containing an amount of money `value` has been sent by *from*, but not yet received by *to*. This set is called `maybelost` in Mondex. Therefore, we define `ACINV(as, cs)` as

$$\langle \text{forall purses do ABORT\#(purse)} \rangle \\ \langle \text{forall (from, to, value) } \in \text{maybelost do TFAIL\#} \rangle \text{SACINV}(as, cs)$$

where `TFAIL#` is the case of `TRANSFER#` from Section 2 where `fail? = true`. The steps can be done in parallel (we do not have to consider the results of various

different execution orders) since they just set the local control state to `idle` and create (again local!) exception logs (in states `epv` and `epa`). The expression can be simplified to get the simulation relation we already described informally in Section 2. The resulting formal definition is given in [SGH⁺07].

The approach of looking into the future does not work for the invariant needed, since by resetting the control state to `idle` the abort step forgets all information about the current state, so the invariant will be trivial and not provide the information we need to verify the big 1:n diagram.

To get this information, it is necessary to look into its past. This can be done locally for each purse, except that we have to consider the global ether. We have

$$\text{CINV}(\text{cs}) \leftrightarrow \forall \text{purse} . \exists \text{ps}_0 . \text{LCINV}(\text{ps}_0, \text{cs}(\text{purse}), \text{ether})$$

where ps_0 is the old local state of the purse at the beginning of the current protocol run and $\text{cs}(\text{purse})$ is the current local state. This state including the control state of the purse, which is enough to give complete information as to how we reached the current state:

$$\begin{aligned} & \text{LCINV}(\text{ps}_0, \text{ps}, \text{ether}) \\ & := \text{case controlstate of} \\ & \quad \text{idle} : \text{ps}_0 = \text{ps} \\ & \quad \text{epv} : \langle \text{STARTFROM}\#(\text{ps}_0) \rangle (\text{ps}_0 = \text{ps} \wedge \varphi_{\text{epv}}) \\ & \quad \text{epa} : \langle \text{STARTTTO}\#(\text{ps}_0) \rangle (\text{ps}_0 = \text{ps} \wedge \varphi_{\text{epa}}) \\ & \quad \text{epa} : \langle \text{STARTFROM}\#(\text{ps}_0); \text{REQ}\#(\text{ps}_0) \rangle (\text{ps}_0 = \text{ps} \wedge \varphi_{\text{epa}}) \end{aligned}$$

The formula just says (compare to Fig. 1): if the control state is `idle`, the purse has executed no step since the last state of interest ps_0 , so the current state ps is equal to ps_0 . For state `epv` a `STARTFROM#` has been executed to reach the current state, and similarly for the two other cases.

This approach would work without any problems if we had local state only, the tricky bit is to handle the global state `ether` in the formulas φ_{epv} , φ_{epa} and φ_{epa} . Fortunately, the property that has to be stated about the ether in intermediate states can be derived systematically for every protocol that sends messages forth and back between two participants: it simply consists of three parts saying which messages *must* have been sent to reach the current control state, which response *may* have been sent, and which messages *have not* been sent in the current protocol.

Consider e.g. a purse p in control state `epv` (compare Fig. 1 again) that communicates with a purse p' . φ_{epv} says:

- `startFrom, startTo` and `req` have been sent
- `val` has been sent, iff the opposite purse p' has sent it entering state `epa` and is either still there or has aborted the protocol run
- `ack` has not yet been sent

The other two formulas φ_{epv} and φ_{epa} are similar (for full formal definitions see [SGH⁺07].).

The proof that the invariant `CINV` is preserved in steps of purse q then reduces to a lemma that the local `LCINV` is preserved for every p . When $\text{p} = \text{q}$, `LCINV` is

trivially invariant for all steps into the protocol. The only interesting steps are aborting the protocol and finishing it with `VAL#` or `ACK#`. These reduce to a proof of the invariance of `SCINV` for full protocol runs.

If p and q are different, and q is not the opposite purse p' to p in the current protocol, the invariance can be proved, since the states of the purses are disjoint and since the protocol run of q does not create usable messages for the protocol run of p and p' . For $q = p'$ the critical properties to be shown are the properties φ_{epr} , φ_{epv} and φ_{epa} of the ether. E.g. when p' sends the `val` message and p expects it in `epv`, it enters `epa` so φ_{epv} remains true.

6 Related Work

Our work on Mondex is based on [SCW00] and a lot of related work will be available in [JW07] (see also [SGH⁺07] for more work on this case study).

The definition of ASM refinement preserving invariants and its proof obligations for generalized forward simulation are closely related to refinement of IO automata [LV95]: basically an IO automaton can be directly encoded as an ASM, and using the identity relation on actions as IO shows that preserving partial correctness for the ASM is equivalent to refinement of the IO automata. The $n:1$ diagrams given here for forward simulation are the same as those for IO automata. A detailed comparison is beyond the scope of this paper and will be published separately.

The 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 refinements from sequential to pipelined execution of instructions of the DLX processor using local 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 and the use of $m:n$ diagrams 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]. The term *states of interest* itself was coined in [Bör03]. Past and future simulation relations are also particular to ASM refinement. One instance of a future simulation relation was used already in the proof of ASM refinement correctness (Corollary 2 in [Sch01]). The only related refinement notion outside of ASM refinement we are aware of is coupled refinement [DW03] which uses past states of interest [Sch05].

For invariants the idea is closely related to using invariants that “sometimes” instead of “always” hold [Bur74], which we have done in KIV for a long time [HRS89]. Recently we used an invariant that looks into the future to analyze security protocols [HGRS07]. The idea there is to focus on future states *after* all possible attacks have been tried.

The general topic of refining atomic actions into protocols has similarities to many other refinement problems, e.g. the refinement of distributed transactions (this idea was taken up in the Event-B work on Mondex (see again [JW07]).

7 Conclusion and Further Work

In this paper we have defined invariant preserving ASM refinement, the theory underlying the Mondex protocol refinement in KIV. To preserve an invariant for all states, the theory is limited to m:1 diagrams. We have shown a systematic way, how to get from these small diagrams back to the intuitive 1:n diagrams, that refine one abstract step by the steps of a protocol run. The method is based on using simulation relations and invariants that look into the future or the past. The method results in extra conditions which show in essence, that interleaved executions of protocols do not interfere. Although the details of the invariants and simulation relations that result applying the method are specific to the Mondex protocol, all indications we have from other case studies suggest that the idea is applicable in many contexts.

References

- [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.
- [Bör90] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.
- [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>.
- [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.
- [HGRS07] Dominik Haneberg, Holger Grandy, Wolfgang Reif, and Gerhard Schellhorn. Verifying Smart Card Applications: An ASM Approach. In *Proceedings of Integrated Formal Methods (IFM) 2007*, 2007. (accepted for publication).
- [HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.
- [HRS89] Maritta Heisel, Wolfgang Reif, and Werner Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitlin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.
- [HSGR07] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 2007. (submitted, extended version available as Techn. Report 2006-32 at [KIV]).
- [JW07] C. Jones and J. Woodcock, editors. (*no title yet*). Formal Aspects of Computing, 2007. (Journal, to be published).
- [KIV] Web presentation of the mondex case study in KIV. URL: <http://www.informatik.uni-augsburg.de/swt/projects/mondex.html>.
- [LV95] N. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. also: Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, MIT.
- [MCI] MasterCard International Inc. *Mondex*. URL: <http://www.mondex.com>.
- [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, 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://www.jucs.org>.
- [Sch05] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 2005.
- [SCW00] S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. URL: <http://www-users.cs.york.ac.uk/~susan/bib/ss/z/monog.htm>.
- [SGH⁺07] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, Nina Moebius, and Wolfgang Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In U. Glässer J.-R. Abrial, editor, *Proceedings of the Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, LNCS. Springer, 2007. (submitted, extended version available as Techn. Report 2006-27 at [KIV]).
- [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.