Specification and Proof of the Mondex Electronic Purse

Chris George[1] and Anne E. Haxthausen[2]

[1] International Institute for Software Technology,
United Nations University, Macao
[2] Informatics and Mathematical Modelling
Technical University of Denmark, Lyngby, Denmark

Abstract. This paper describes how the communication protocol of Mondex electronic purses can be specified and verified wrt. desired security properties. The specification is developed by stepwise refinement using the RAISE formal specification language, RSL, and the proofs are made by translation to PVS and SAL.
Keywords: Formal methods, RAISE, PVS, SAL, verification, Mondex

1 Introduction

1.1 The problem and the protocol

Mondex [Inc,IE97] is a financial system that utilizes smart cards as electronic purses. Each card stores financial value (equivalent to cash) as electronic information on a microchip and provides operations for making financial transactions with other cards via a communication device. The system is distributed without any central controller.

The protocol for transferring an amount (value), *v*, from one purse, *P1*, to another purse, *P2*, includes the following steps:

1. *P2* sends a request about transferring *v* from *P1* to *P2*
2. *P1* receives the request and sends the amount *v* to *P2*
3. *P2* receives the amount *v* and sends an acknowledgement to *P1*

It should be noted that the protocol can be stopped at any point by either purse aborting the transaction, such as by a card losing power. If e.g. *P2* aborts after *P1* has sent the amount but before *P2* has received it, it must be logged that the amount is "lost"; it may be lost to the purse balances but not to the system as a whole.

1.2 Past and ongoing work

It is rather important that such a financial system does not contain bugs, and therefore the NatWest Team that had the development task asked Logica to verify this using formal models.

The key problems were:

1. to specify the protocol in detail
2. to prove that each operation satisfies two conditions:
   (a) the value in circulation in the system will not increase
   (b) all value is accounted in the system

Originally [SCW00], the Mondex smart card problem was specified and refined in the Z [Spi92] formal specification language, and proved correct by hand. This resulted in an assurance at ITSEC [ITS] level E6, ITSEC's highest granted security-level classification which is equivalent to Common Criteria level EAL7.

Currently a number of research groups are working on the same problem in parallel using different formalisms, methods and tools. The purpose is to see what the current state of the art is in mechanising specification and proof.

In this paper we report on how we have done that using the RAISE method.

1.3 RAISE background

The RAISE ("Rigorous Approach to Industrial Software") method [RAI95] is based on stepwise refinement using the invent and verify paradigm. In each step a new specification is constructed and proved to be a refinement of the specification of the previous step.

The specifications are formulated in the RAISE specification language (RSL) [RAI92]. RSL is a formal, wide-spectrum specification language that enables the formulation of modular specifications which are algebraic or model-oriented, applicative or imperative, and sequential or concurrent. A basic RSL specification is called a class expression and consists of declarations of types, values, variables, channels, and axioms. Specifications may also be built from other specifications by renaming declared entities, hiding declared entities, or adding more declarations. Moreover, specifications may be parameterized.

With the method and language there is a suite of tools supporting the construction, checking and verification of specifications and development relations, and for translating specifications in certain subsets of RSL into several other languages including PVS and SAL.

1.4 Paper survey

First, in section 2, we describe how a specification of the Mondex communication protocol and proof obligations has been stepwise developed. Then, in sections 3–4, we explain how the proof obligations have been proved by translation to PVS [ORS92] and SAL [dMOS03], respectively. Finally, in section 5, we draw some conclusions.

2 RSL specification of Mondex

In this section first we describe the original Z approach to specification of the Mondex problem, as this specification served as inspiration for our work. Then we describe our approach using RAISE.

2.1 The Z approach

The Z specification [SCW00] of Mondex electronic purses consists of three models at different levels of abstraction:

1. An *abstract model* that describes the global state (world of purses) and operations on this for atomic value transfer between purses.
2. A *between model* that describes (1) the state of a single purse and the single purse operations, and, (2) the global state as a collection of purses, an ether of messages and an archive of logs, and global world operations defined by promoting the single purse operations.
3. A *concrete model* that is similar to the between model, but with no state invariants. Loss of messages is also introduced.

The abstract model describes the functional requirements (independent of any protocol) for value transfer between purses, while the concrete model describes the actual Mondex communication protocol. The between model was introduced for very technical reasons to ease the proof that the concrete model refines the abstract one.

The security properties are expressed in terms of the abstract model and proved to be correct already at that level. It is proved that the between model refines the abstract model and the concrete model refines the between model. Hence, it is proved that the concrete model satisfies the security properties.

2.2 The RAISE approach

First we considered staying very close to the structure of the Z approach. Technically this would have been possible, but we decided to using an approach more in keeping with the RAISE method and the RAISE notion of refinement[1]. However the details we describe follow closely the details described in Z.

We also specified the system at three levels as described in the subsections below, but there is not a one to one correspondence between the three Z levels and the three RAISE levels.

One of the main differences is that different levels in Z use different types to represent the global state (*AbWorld* and *ConWorld*), while in RAISE we use the same type *World* at all levels of the specification. In the abstract RAISE model the type is abstract (a sort with some associated observer functions), while in the concrete one it is given an explicit representation. In the intermediate level it is still abstract, but additional observers are introduced.

Another difference is that we do not have one set of atomic operations at the abstract level and another set of operations (the protocol operations) at the

[1] Refinement in Z is relational: A model B refines a model A if any B state transition simulates an A state transition. Simulation is defined in terms of a retrieve relation between the A and B states: seen through this retrieve relation the states reachable in B should be a subset of those reachable in A. In RAISE, refinement basically is model inclusion (or theory extension) and not a simulation subrelation.

intermediate and concrete levels which then should be related to each other. Rather from the very beginning our goal is to define the protocol operations. At the abstract level, we just define what it means for an operation to be correct. At the intermediate level we axiomatically specify the behaviour of the operations and at the final, concrete level, the operations are explicitly defined. Operations corresponding to the atomic Z operations are defined at the intermediate level as compositions of the protocol operations. In the Z work we have not found a corresponding demonstration that a complete transfer is possible in terms of the protocol operations.

Level 0 The initial specification WORLD0 specifies the global state as a sort *World* with three fundamental observers that give the total sum of values stored in purses, the total value in transit between purses, and the total value logged as lost, respectively:

```
type World

value /∗ World observers ∗/
  inPurses : World → Nat,
  inTransit : World → Nat,
  lost : World → Nat
```

The two desired relations *noIncreasedCirculation* (the sum of *inPurses* and *inTransit* does not increase) and *allValueAccounted* (the sum of all three is constant) between pre and post Worlds of operations are explicitly defined in terms of these fundamental observers.

Types *Op* and *Pre* for state changing operations (taking a purse name and a message as parameters) and their preconditions (protocol guards) are introduced, but no concrete operations are introduced at this level.

```
type
  Op = Name × Message × World ~→ World,
  Pre = Name × Message × World → Bool
```

We just define what it means for an operation to be correct:

```
value
  isCorrect : Pre × Op → Bool
  isCorrect(p, op) ≡
    (∀ n : Name, m : Message, w, w1 : World •
      p(n, m, w) ⇒
        op(n, m, w) = w1 ⇒
          noIncreasedCirculation(w, w1) ∧ allValueAccounted(w, w1)),
```

Finally four predicates for classifying operations according to how they change the observable values are defined. An operation that (satisfies) *isTransferLeft* reduces *inPurses* and increases *inTransit* or *lost* by the same amount; one that

isTransferRight reduces *inTransit* and increases *inPurses* by the same amount; one that *isNo_op* leaves the three observers unchanged; and one that *isAbort* reduces *inTransit* and increases *lost* by the same amount (which may in fact be zero: whether an abort actually causes a definite increase in *lost* depends on the point in the protocol when the abort of a purse happens).

A theory, THEORY0, asserts that the four classes of operation satisfy *isCorrect*. Our strategy for development is to show that each of the 11 concrete operations, when viewed abstractly in terms of the three fundamental observers, satisfy one of *isTransferLeft*, *isTransferRight*, *isNo_op*, *isAbort*, and hence that they all satisfy *isCorrect*.

Level 1 The next level consists of two parts: PURSE1 and WORLD1.

PURSE1 introduces a purse state with names, balances, pay details, statuses:

```
type PurseBase, Purse = {| p : PurseBase • ... |}

value /∗ Purse observers ∗/
  balance : PurseBase → Nat,
  pdAuth : PurseBase → PayDetails,
  status : PurseBase → Status,
  name : PurseBase → Name
```

Purse state changing operations receiving and sending a message are axiomatically specified, and their preconditions explicitly defined. Here we show it for an operation *req* that formalizes the request step in the communication protocol:

```
value /∗ Purse operations and guards ∗/
  req : Message × Purse ~→ Purse × Message,

  canReq : Message × Purse → Bool
  canReq(m, p) ≡ ...,

axiom /∗ observer generator axioms ∗/
  [ balance_req ]
    ∀ m : Message, p : Purse •
      canReq(m, p) ⇒
        balance(req(m, p)) =  balance(p) − valu(pdAuth(p)),
  ...
```

The term *valu(pdAuth(p))* is the amount of value requested to be transferred. The first axiom states that the balance of the purse is decreased by that amount.

In WORLD1 new *World* observers are introduced:

```
value
  purses : World → PursesMap,
  toLogs : World → PayDetails-set,
  fromLogs : World → PayDetails-set,
```

ether : World → Message-**set**,
visible : World → Message-**set**

and the observers from WORLD0 are explicitly defined in terms of these. An axiom *isWorldAxiom* specifies a state invariant (required relations between the observers). For example, in any *World*, *visible* is a subset of *ether*, reflecting that messages may be lost.

The Purse operations are promoted to World operations:

```
value /∗ World operations and guards ∗/
  req : Op,
  canReq : Pre,
  ...
axiom /∗ observer generator axioms ∗/
  [ purses_req ]
    ∀ n : Name, m : Message, w : World •
      canReq(n, m, w) ⇒ purses(req(n, m, w)) = ...
...
```

A theory, THEORY1, asserts that each of the 11 *World* operations satisfies one of the four classification conditions (introduced in WORLD0) and that WORLD1 implements WORLD0 and so inherits all its properties and hence its theories.

Level 2 At the final level, in WORLD2 and PURSE2, all types and operations are explicitly defined apart from a few things that are intentionally underspecified. We have the same degree of concreteness as in the concrete Z model.

A theory, THEORY2, asserts that WORLD2 implements WORLD1 and so inherits all its properties and hence its theories.

3 Mondex in PVS

The Mondex RSL specification was translated to PVS using the RSL to PVS translator [DG02]. Each of the three levels was translated together with the associated theory.

3.1 Strategy

The aim of the Mondex project is to see what can be done automatically. We could probably have developed automatic strategies to deal with all the proofs involved here, but that, we feel, is not really the purpose: rather we are interested in how much can tools deal with proofs guided by software engineers with some experience and some heuristics to guide them. Scripted proof strategies should only be employed if they are fairly general, with guidelines about when to use them, or (as in this case) there are a number of very similar proofs and it is fairly simple to develop a useful strategy that is particular to the problem.

3.2 Proof obligations

A number of different kinds of proof obligation were generated:

Existence TCCs We had very few of these, and they were easily discharged.

Subtype TCCs There were many of these, because there are a number of subtypes in the RSL specification and also a lot of functions with preconditions, which become subtypes in PVS. A large collection of proofs in this category appears at level 2, where we have to prove that *isWorld* is invariant for each of the 11 operations.

Finiteness There are a number of sets defined by comprehension which need to be finite because we sum over them.

Lemmas Many of the statements in the THEORY modules are lemmas to be used later, in particular expressing quantities that will be needed in the proofs of later assertions.

Refinement In THEORY1 we find the assertion "WORLD1 $\preceq$ WORLD0". In the translation to PVS such refinement relations are expanded into a collection of predicates sufficient to imply refinement. In fact, WORLD1 is essentially an extension of WORLD0, and the relation reduces immediately to **true**. In THEORY2 the corresponding relation expands into 113 lemmas to be proved, arising from the axioms in WORLD1 and PURSE1 that now have to be proved from the concrete definitions in WORLD2 and PURSE2. Most of them are trivial to prove.

3.3 The argument for correctness

We need to be clear about exactly what we have proved. We defined the basic relations *noIncreasedCirculation* and *allValueAccounted* in WORLD0, and defined a function *isCorrect* that returns **true** for an operation (state transformer in this case) that maintains these two relations when its precondition is true. This reduces the problem to showing for each operation and associated precondition that we define, that they satisfy *isCorrect*. But we need to be aware that *isCorrect* is defined in WORLD0, and we need proofs for the concrete operations defined in WORLD2. We will take the operation *req* as an example. The argument proceeds as follows:

1. We assert in THEORY0 and prove that an operation satisfying *isTransferLeft* satisfies *isCorrect*.
2. We axiomatise *req* in WORLD1, and assert in THEORY1 and prove that *req* satisfies *isTransferLeft*. Now we appeal to a meta-theorem for refinement in RAISE:

$$\frac{\text{S0} \vdash \text{P},\ \text{S1} \preceq \text{S0}}{\text{S1} \vdash \text{P}} \tag{1}$$

which states that if predicate P is proved for specification *S0*, and specification *S1* refines *S0*, then P is proved for *S1*. We can appeal to this rule to

show that *req* must satisfy *isCorrect* at level 1. (We did in fact assert and prove this separately at level 1, but theorem (1) shows that such an assertion and proof is unnecessary.)
Theorem (1) follows from the definition of refinement in RAISE, which says that *S1* $\preceq$ *S0* iff (a) the signature of *S1* includes that of *S0* (from which it follows that if *P* is well-formed in the context *S0* then it will be well-formed in the context *S1*) and (b) all the properties of *S0* are true in *S1*. It follows that if *P* can be proved from the properties of *S0* then it can be proved from the properties of *S1*.

3. We define *req* explicitly in WORLD2 and prove that WORLD2 $\preceq$ WORLD1. The final result that *req* must satisfy *isCorrect* at level 2 follows by another appeal to theorem (1).

3.4 The proofs

The following (groups of) proofs were the most difficult.

1. At level 2, the proofs that the 11 operations preserve *isWorld*. *isWorld* contains 14 conjuncts, and this was a very substantial proof effort, and the one for which it was worth producing a tactic to assist. While, obviously, each of the operations changes the state in different ways, so requiring its own proof, the overall structure of the 11 proofs was similar. We wrote a tactic to do most of the work, basically by doing one proof by hand and using its proof, structured with `then` and `split`, as the tactic, then trying it on the next one, stepping through the tactic and improving it and generalising it as necessary. Then we just ran it on the rest and completed the proofs by hand. The basic tactic completed 81% of the proof branches — but this still left 70 proofs to be done!
   The branches of the tactic end with `assert`, `flatten`, `assert`, and `assert`. It was tempting to use something more powerful than `assert`, but the size of the specification meant that `grind` could generate a very large number of cases (we once generated 1580) and even `grind :if-match nil` could generate 70. Some might have generated more: we often lost patience and interrupted a seemingly endless proof.
2. The proof that the invariant *isWorldAxiom* at level 1 was implied by the invariant *isWorld* at level 2 was quite substantial: the two invariants have very different structures, that at level 1 being designed to help the proofs of the correctness of the operations, and that at level 2 being designed to support the proofs that the operations are correctly implemented, which depends on having a suitable tightly restricted state space. This proof was quite large, but not difficult.
3. The proofs of the finiteness of a number of sets needed care. Proving finiteness from first principles (by the existence of a bijection to a finite range of integers) is almost never a good idea, and existing useful theorems need to be found (or created). For example, *toInEpv* (used to calculate *inTransit*) of a *World* *w* is defined by the expression

{P.pdAuth(purses(w)(T.to(pd))) | pd : T.PayDetails •
T.to(pd) ∈ **dom** purses(w) ∧ P.status(purses(w)(T.to(pd))) = T.epv}

and it is easy to see that this will be a subset of

{P.pdAuth(purses(w)(n)) | n : Name • n ∈ **dom** purses(w)}

which is the image of a function (λ *n : Name • P.pdAuth(purses(w)(n))*) applied to the set **dom** *purses(w)*, which is finite by definition. The necessary theories (a subset of a finite set is finite, and the functional image of a finite set is finite) are in the PVS prelude.
More difficult was proving that logs are finite. *fromLogs*, for example, which is used to calculate *inTransit* and *lost*, is a subset of the pay details found in purse logs or the archive. The archive is stored as a finite map from purse name to finite sets of pay details, and the purse logs effectively have the same structure. So we had to prove that mapping the set union operator through a finite collection of finite sets gives a finite set. We did this by defining such a structure in RSL — a parameterised scheme MAPUNION defining the map type, and a function *mapunion*, defined by a comprehension, to generate the set. This was asserted to have a finite result, and so generated the condition as a TCC. The TCC was proved by complete set induction (included in the RSL prelude). Then the finiteness of *fromLogs*, for example, is proved by showing it is a subset of the union of two instances of *mapunion*: finiteness of the union of two finite sets is included in the PVS prelude.

Importance of replay We produced some 11 versions of the specification, doing some proof for all of them. This amount of experimentation would have been impossible without the ability to replay proofs from a previous version, either blindly, just to see how much was left, or more carefully stepping through the proof and adjusting it as necessary. It would be impossible, we think, to complete an exercise of this size without this ability.

Some statistics The RSL specifications and theories for the three levels is 2237 lines (in pretty-printed form; there are very few comments). These translated to 4007 lines of PVS (not pretty-printed — the pretty printer in version 3.1 seems to be broken). There are 7 proofs at level 0, 120 at level 1, and 242 at level 2: 369 in total. The proof scripts for these total 55009 lines (measured from the .prf files, a large proportion of which is dependency information).

4 Mondex in SAL

We are currently developing a translator from RSL to SAL, and applied this translator to the concrete (level 2) Mondex specification.

4.1 Purpose of SAL version

One might ask why we bothered with a SAL version since it must be finite with a bounded size, and we already have a formal proof of an unbounded version. There are several reasons:

- Model checking is much easier than proof, and so much more available to software engineers.
- Model checking can be used to check things before a great deal of effort is invested in proving them. For example, we wasted a lot of time on one version with an appealing clause in the invariant which turned out not to be true. We used it on the proofs of many operations before we stumbled on the operation for which it was not invariant. Model checking at the beginning would almost certainly have exposed the problem and saved a lot of work.
- The properties we are trying to prove are safety properties. It is good also to demonstrate some liveness properties, for example that a transfer of money between purses can occur. We did this by starting the model with one purse empty, the other not, and asserting that the empty purse remains forever empty. The model checker generates a counter-example to the (deliberately invalid) assertion, showing how a transfer can be made.
- We will see below in section 4.3 that we can use model checking to check for violations of subtype conditions and preconditions, again avoiding possible wasted effort in trying to prove things that are false.

4.2 Simplifications

We made a number of simplifications in order to reduce the size of the state space to something SAL can handle:

- two purses
- money in the range *0..3* only
- transfers are always of one unit of money (so that the amount does not need to be included in messages)
- all the operations and messages concerned with archiving logs are removed (as we think these operations could be model checked separately)
- the possible loss of messages is not modelled (to reduce possible changes to the ether)
- the technique inherited from the Z specification of including all the *startTo* and *startFrom* messages in the ether is replaced by the (equivalent) technique of including none of them. The *nil* message is also removed: it serves no purpose apart from allowing all operations to have the same signature.
- sequence numbers (used in the concrete version to ensure uniqueness of messages, and modelled as natural numbers) are the range *0..3*.

We also made some changes to split the state into smaller components. Such modifications do not reduce the size of the state, but they make the specification more accessible to the optimizations used by SAL:

– The ether containing a set of three possible message types is replaced by three sets each containing the contents of one message type.
– The logs are separated from the purses.

Reducing the range of sequence numbers reduces the number of "runs" of the protocol. We could "prove" with this model that a purse with a balance of four would never become empty! (In general, if the sequence numbers are restricted to *0..n* then at most *n* transfers can take place.)

At the time of writing we are able to prove that:

– All money is accounted.
– The amount of money in circulation does not increase.
– *isWorld*, *isWorldAxiom* and *isPurse* are state invariants.
– Liveness in the sense that
  • An empty purse can become non-empty.
  • A non-empty purse can become empty.
  • Money can be lost.

For all of the above assertions the SAL process grows to a maximum memory usage of between 80MB and 260MB.

Modifying the specification and the assertions to make SAL succeed in checking them we:

– Split up some data types into a number of disjoint subtypes and reformulated some assertions as a number of assertions each over one of the subtypes. For example, the type we use to represent pay-details was split up into 4 disjoint subtypes and assertions modified accordingly.
– Introduced extra state variables, and so shifted computation from assertions to transitions of the transition system, making the results readily available for lookup in the state. For example, two state components hold the total money in circulation in the previous state as well as the money in circulation in the current state, allowing for a simple formulation of the assertion that money in circulation does not increase.
– Injectively mapped pay-details to a range of natural numbers, and used this as the domain of the predicates representing sets of pay-details.

4.3 Checking RSL confidence conditions

Confidence conditions are conditions that a specification must satisfy if unintended use of RSL constructs are to be avoided. The main ones in applicative specifications are satisfaction of preconditions and "subtype correctness" - constants should be in subtypes, and arguments of functions and their results should be in subtypes. Here the term "functions" includes both user-defined functions and operators and the built-in ones. So detecting confidence conditions includes, for example, division by zero. SAL and PVS do not support preconditions, but they can be translated into subtypes (since SAL and PVS support the notion of dependent type). However, subtype checks are not performed by SAL.

The translator from RSL to SAL generates a separate set of SAL files which include confidence condition (CC) checking, and replaces the user-defined LTL assertions with a single one that says no "NaV" ("Not a Value") occurs in any of the state variables. The term "NaV"is inspired by the IEEE's floating-point standard (IEEE 754) which involves the use of "NaN", "Not a Number", for such things as the result of under- or overflow, or attempting to calculate the square root of a negative number. A NaV is generated whenever a confidence condition is violated, and all functions and operators in this version are strict: if a NaV is generated in a function it is returned as the result. This is done by "lifting" all types into SAL `DATATYPES`, so that for any defined type `T` there is also defined a lifted type:

```
T_cc: TYPE = DATATYPE
  T_cc(T_val: T),
  T_nav(T_nav_val: Not_a_value_type)
```

The type `Not_a_value_type` is generated by the translator as an enumeration of identifiers which indicate where the confidence condition violation was detected, as we see from the example below.

When we ran the CC version of Mondex no violations were detected. To check that it was actually capable of detection we introduced some CC violations and checked they were detected. For example, we changed one clause of *canStartFromEaFrom* from $balance(p) \geq 1$ to $balance(p) > 1$. The results of normal model-checking were unaltered, but the CC version produced a counter-example in which the state of the second purse is a NaV:

```
purse2 =
PurseBase_nav(
 Precondition_of_function_WORLD2INV_startFromEaFrom_not_satisfied)
```

This indicated that the error was the non-satisfaction of the precondition of `startFromEaFrom` in the context `WORLD2INV`.

Use of the CC version of SAL increases our confidence in the quality of the specification. It also increases our confidence that the proofs of confidence conditions — which are partly generated as TCCs in PVS after translation, and otherwise generated by the translator to PVS as extra lemmas to be proved — can be done. Again model-checking is a useful precursor to the effort of proof.

5 Conclusion

The aim of this work is to see, for this example, (a) whether the Z proof could be mechanically proved and so confirmed, and (b) how much can be automated. We are clearly some way yet from having an automated proof: a lot of this proof had to be done by hand. It is also not clear how much of the expertise required to do such proofs can be taught to software engineers.

There are also a number of other problems that are not solved by automation of the proof and model checking processes alone:

- Is there a contradiction in our specification somewhere? If there is, effectively, **axiom false** somewhere in our specification then all our proofs could have gone through. One can argue that automating proofs increases this risk: a user might notice. In fact this possibility is less likely than it might seem. Since we know that WORLD2 implements WORLD1, and WORLD1 implements WORLD0, then any such contradiction must exist at level 2 if it exists anywhere. Since level 2 is mostly constructive it is much less likely to be present without being spotted. The RSL to PVS translator includes confidence conditions in the translation as lemmas (when these are not subsumed by PVS TCCs). These conditions therefore have to be proved, and their proof removes the possibility of many possible sources of contradiction. We have also seen that the special CC version of the translator to SAL can be used to give us confidence that these conditions are not violated.
- Is our model correct? If we didn't specify the right thing then all our results are useless. This can only be resolved by someone carefully checking the specification by reading it. Hence it is important that our language (and our style of using it) be readable, expressing concepts at an appropriate level, with an intuitive semantics, etc.
- Is our "concrete" specification correctly implemented? Our level 2 specification is still an abstraction of the actual code. Is the chip istself safe from attack by microprobing [Mon97]?
- Are the tools correct? We rely on translators from RSL to PVS and SAL, and on the PVS and SAL engines: none of these have been proved correct.
- For model checking, if the only result we see is "proved", is anything actually happening, or are we only visiting a small set of states because the guards of some transitions are never true? Here we can be much more confident if, as we did here, some expectedly false assertions are included, when the output traces that accompany failure show us the transitions that have occurred.
- While model checking does not require interaction guiding the system to the proof, using SAL to automatically check the Mondex specification written in RSL required substantial modification and rewriting. We had to substantially change the RSL specification in order to reduce the number of possible states. Furthermore, we had to rewrite the RSL specification after these changes to help the SAL optimizer.
- Our state reduction raises the question of whether results of model checking are applicable to the full specification. Our rewrites for SAL's optimizer raise the question of whether in practice software engineers can be expected to learn enough about tailoring specifications to the requirements of model checking in general and about the internal operation of SAL to successfully use it or whether the optimizations used in SAL can be improved to be applicable to a wider range of inputs. In order to successfully employ SAL in practice software engineers will need a good selection of techniques to make their specifications accessible to the model checker.
- The key to being able to do a proof of such a system is finding the appropriate invariant. Unfortunately there is no single "correct" invariant. Anything which can be proved from an invariant is also an invariant of course, and so

is any weaker expression, obtained for example by selecting just a subset of our 14 conjuncts. An invariant has to be strong enough to enable the proofs, but preferably no stronger, because that would require more proof effort (and, incidentally, make our system harder to maintain if we have to change it because of changed requirements). It also has, of course, to be an actual invariant and not an erroneous one! Finding such an invariant is not easy, and probably requires deep knowledge of why something works, in this case of the ideas behind the design of the protocol. There are techniques for discovering invariants from code or specifications, but of course one discovered from the specification may be erroneous if the specification is. Model checking is very useful in checking that proposed invariants are indeed invariant, but does not help one to find them.

– What are the important properties? The two properties we proved are the fundamental ones, according to the original problem, but they are also quite weak. They say nothing about individual purses, for example, so there is nothing to say that a transfer from A to B, say, does not involve stealing from C. Liveness properties such as the ability to actually transfer money, or to clear a purse log, might also be interesting. The size of the proof task makes it difficult to easily extend a model designed for proof of particular properties, as this model is, at least at the abstract levels, to prove other properties. We did show it was possible to transfer money, by carefully designing a multiple operation to do so, but it was rather an artificial exercise[2]. For liveness properties, especially in nondeterministic systems, model checking is much more convenient. Once the model has been set up then it is comparatively little effort to devise and check formulae for new properties, and liveness demonstrations can easily be done by getting the model checker to find a counter-example to a claim for the property's negation. So proving fundamental properties, and using a model checker to explore a range of other properties, makes a powerful combination in practice.

Could we have done better by writing specifications in PVS and SAL directly? Could we, for example, have made the PVS proofs easier, more amenable to automation? We think not. The translators to PVS and to SAL are quite straightforward: there is mostly a one-to-one correspondence between RSL constructs and PVS/SAL ones. We chose SAL as the target language for model checking RSL because of its comparatively rich type system, enabling us for example to translate RSL variant types as SAL `DATATYPE`s, rather than having to encode everything using only ranges of integers and Booleans, as some model checkers would force us to do. Hence we think (a) that the translators generate PVS and SAL code that is very close to what would be written by hand in the other languages, and (b) that problematic constructs in either target language can be avoided by writing the RSL in a suitable way. There are some issues here, for

[2] It is in fact impossible, because of the nondeterminism involved, to specify in advance the conditions under which a transfer will take place. The "proof" we give exploits a feature of the PVS logic that the epsilon (choice) operator is deterministic: this is not true in the RSL logic.

example the currently poor support for modularity in SAL, and of course its requirement for finiteness, but generally we found little contradiction between good style in RSL and good translated code.

References

[DG02] Aristides Dasso and Chris W. George. Transforming RSL into PVS. Technical Report 256, UNU/IIST, P.O. Box 3058, Macau, May 2002.

[dMOS03] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, SRI International, 2003. Available from `http://sal.csl.sri.com`.

[IE97] Blake Ives and Michael Earl. Mondex international: Reengineering money. Technical Report CRIM CS97/2, London Business School, 1997.

[Inc] MasterCard International Incorporated. Mondex.

[ITS] ITSEC. `http://en.wikipedia.org/wiki/ITSEC`.

[Mon97] Mondex's Pilot System Broken. `http://jya.com/mondex-hack.htm`, September 1997.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[RAI92] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

[RAI95] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. Available by ftp from `ftp://ftp.iist.unu.edu/pub/RAISE/method_book`.

[SCW00] S. Stepney, D. Cooper, and J. C. P. Woodcock. An Electronic Purse: Specification, Refinement, and Proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, 2000.

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