DOI 10.1007/s00165-007-0054-3
BCS © 2007
Formal Aspects of Computing (2008) 20: 101–116

Formal Aspects of Computing

Specification, proof, and model checking of the Mondex electronic purse using RAISE

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

[1]International Institute for Software Technology, United Nations University, Macao, UNU-IIST, PO Box 3058, Macao SAR, China.
E-mail: cwg@iist.unu.edu
[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 against 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. The work is part of a year-long project contributing to the international grand challenge in verified software engineering.

Keywords: Formal methods; RAISE; PVS; SAL; Verification; Mondex

1. Introduction

As a response to the international grand challenge in verified software engineering [Hoa03, HM05, Hoa06, JOW06] a number of research groups started in January 2006 to work on the same verification problem, the Mondex case study, in a year-long project [Woo06] initiated by Jim Woodcock. This paper reports on how the Mondex problem has been approached using the Rigorous Approach to Industrial Software (RAISE) formal method.

In the remaining part of this introduction, we briefly describe the Mondex case study, provide some information about RAISE and give a survey of the paper. For a more detailed description of the Mondex case study, descriptions of the work made by the other research groups and a comparison of the different approaches we refer to the other papers in this special issue of the FAC journal [FACJ].

1.1. The Mondex case study

Mondex [Inc, IE97] is a 10-year-old 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.

It is rather important that such a financial system does not contain bugs. Therefore Logica (a commercial software house) in collaboration with the University of Oxford specified and refined the Mondex smart card system in the Z [Spi92] formal specification language, and proved by hand that the system satisfies some required security properties. This resulted in an assurance at ITSEC [ITS] level E6, ITSEC's highest granted security-level

Correspondence and offprint requests to: C. George, E-mail: cwg@iist.unu.edu

classification which is equivalent to Common Criteria level EAL7. This work is documented in [SCW00]. Recently Woodcock and Freitas mechanised the proof [WF06] using the Z/Eves theorem prover.

During 2006 a number of research groups worked in parallel on the Mondex case study using different formalisms, methods and tools: Alloy, ASM/KIV, Event-B, OCL, PerfectDeveloper, π-calculus, RAISE and Z. The purpose was to see and compare what the current state of the art is in mechanising specification and proof, and more generally to contribute to the international grand challenge in verified software engineering. The results will be stored in the verified software repository [BHW06].

In this paper we report on our contribution using the RAISE method.

1.2. The Mondex protocol and key problems

The Mondex protocol and key problems are completely described in [SCW00]. Here we just provide enough information for being able to read the remainder of this paper.

The Mondex protocol for transferring an amount (value), *v*, from one purse, *P1* (the "from purse"), to another purse, *P2* (the "to purse"), 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"; money may be lost from the purse balances but not from the system as a whole.

In the complete, concrete description of Mondex there are altogether 11 operations:

- *startFrom* and *startTo* set up the purses *P1* and *P2* to make a transfer; *StartTo* also sends the request message (step 1 above).
- *req* and *val* perform steps 2 and 3 respectively.
- *ack* performs the receipt of the acknowledgement.
- *abort* aborts any existing transfer.
- *readLog*, *clearLog*, *authoriseClear*, and *archive* are operations concerned with reliably moving purse logs to a central archive.
- *increase* is an abstraction of other purse activities, and in our model only increases the purse's internal sequence number. Monotonically increasing sequence numbers in the purses are used to ensure that messages sent by purses are fresh, i.e. not possible duplicates of previous messages.

The first five of these operations are the ones intended for use in transferring money between purses. They are each triggered by receipt of a corresponding message, i.e. we have *startFrom*, *startTo*, *req*, *val* and *ack* messages. There are also other messages concerned with log archiving which we do not discuss further in this paper.

The key problems are:

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

1.3. RAISE background

The RAISE 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 may be 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 [ORS92], SAL [dMOS03] and C++.

1.4. Paper survey

First, in Sect. 2, we describe how a specification of the Mondex communication protocol and proof obligations has been stepwise developed. Then, in Sects. 3 and 4, we explain how the proof obligations have been proved by translation to PVS and SAL, respectively, and in Sect. 5 we outline how a prototype implementation has been created by translation to C++. Finally, in Sect. 6, we draw some conclusions.

There are many excerpts from the specifications: full specifications can be found in the technical report [GH07].

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 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 use 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

[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.

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 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.

2.2.1. 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:

```
value
  totalCirculating : World → Nat
  totalCirculating(w) ≡ inPurses(w) + inTransit(w),

  totalAccounted : World → Nat
  totalAccounted(w) ≡
    inPurses(w) + lost(w) + inTransit(w),

  noIncreasedCirculation : World × World → Bool
  noIncreasedCirculation(w, w1) ≡
    totalCirculating(w1) ⩽ totalCirculating(w),

  allValueAccounted : World × World → Bool
  allValueAccounted(w, w1) ≡
    totalAccounted(w) = totalAccounted(w1)
```

Types *Op* and *Pre* for protocol operations (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
```

The arrow in the type of *Op* indicates that it is a partial function; that for *Pre* shows it is total.

We define what it means for an operation to be *correct*, by which we will mean only that it satisfies *noIncreasedCirculation* and *allValueAccounted*:

```
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 value in the input message; one that satisfies *isTransferRight* reduces *inTransit* and increases *inPurses* by the value in the input message; one that satisfies *isNo_op* leaves the three observers unchanged; and one that satisfies *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). Here is *isTransferLeft* for example:

```
value
  isTransferLeft : Pre × Op → Bool
  isTransferLeft(p, op) ≡
    (∀ n : Name, m : Message, w, w1 : World •
      p(n, m, w) ⇒
        op(n, m, w) = w1 ⇒
          inPurses(w1) = inPurses(w) − valu(m) ∧
          (lost(w1) = lost(w) ∧
            inTransit(w1) = inTransit(w) + valu(m) ∨
            lost(w1) = lost(w) + valu(m) ∧
            inTransit(w1) = inTransit(w))),
```

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 protocol 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*.

2.2.2. 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 • isPurse(p) |}

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

PurseBase is an abstract type; *Purse* a subtype of it satisfying *isPurse*, which is defined (not shown here) in terms of relations between the *PurseBase* observers: *isPurse* is an invariant on purses since, as we shall see, the purse operations are specified to take *Purse* values as arguments and return *Purse* values as results.

Purse state changing operations receiving and sending a message are axiomatically specified, and their preconditions are explicitly defined. Here we show the signature, the precondition, and one of the axioms for the 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) ≡ m = req(pdAuth(p)) ∧ status(p) = epr,

axiom /∗ observer generator axioms ∗/
  [ balance_req ]
    ∀ m : Message, p : Purse •
```

```
    canReq(m, p) ⇒
      balance(req(m, p)) = balance(p) − valu(pdAuth(p)),
  ...
```

The precondition *canReq* states that the received message must be a *req* message containing the same pay details as held by the purse, and the purse must have status *epr*.

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

The alert reader will notice we have overloaded *balance* so that it can apply to a pair of purse and message (as returned by *req*) as well as to a purse.

WORLD1 introduces new *World* observers giving the state of each purse, the sets of pay details logged by "to purses" and "from purses", respectively, when a protocol is interrupted, the set of all previous messages sent, and the set of those previous messages sent that are visible (i.e. not lost):

```
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 and guards, of types *Op* and *Pre* (identical to those types in level 0), are promoted to *World* operations:

```
value /∗ World operations and guards ∗/
  req : Op,
  canReq : Pre,
  ...
axiom /∗ explicit definitions of guards ∗/
  [ canReqAxiom ]
    ∀ n : Name, m : Message, w : World •
      canReq(n, m, w) ≡ ...,
  ...
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. Implementation is fully defined in the RAISE method book [RAI95], but here it suffices to note that if P is a property of WORLD0, and WORLD1 implements WORLD0, then P is a property of WORLD1. So, taking the operation *req* as an example:

1. *req* in WORLD1 satisfies *isTransferLeft* (asserted in THEORY1)
2. WORLD1 implements WORLD0 (asserted in THEORY1)
3. In WORLD0, any operation satisfying *isTransferLeft* satisfies *isCorrect* (asserted in THEORY0)

Item 2 above allows us to lift item 3 to a statement about WORLD1, and then item 1 implies that *req* in WORLD1 satisfies *isCorrect*, provided we prove THEORY0 and THEORY1. Similarly for the other ten operations. We restate this argument slightly more formally in Sect. 3.3.

2.2.3. 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.

For later discussion, we show how the *World* type is defined as a subtype of a concrete type:

```
type
  World = {| w : WorldBase • isWorld(w) |},
  WorldBase ::
    purses : PursesMap
    ether : Message-set
    visible : Message-set
    archive : LogBook
```

The state invariant *isWorld* is explicitly specified to satisfy 14 conjuncts.

Here is the specification or *req* and *canReq* in WORLD2:

```
value
  req : Name × Message × World ~→ World
  req(n, m, w) ≡
    let (p1, m1) = P.req(m, n, purses(w)(n)) in
      mk_WorldBase(
        purses(w) † [ n ↦ p1 ], ether(w) ∪ {m1},
        hideSome(visible(w) ∪ {m1}), archive(w))
    end
  pre canReq(n, m, w),

  canReq : Name × Message × World → Bool
  canReq(n, m, w) ≡
    n ∈ dom purses(w) ∧ m ∈ visible(w) ∧
    P.canReq(m, n, purses(w)(n))
```

Here *P* is the object instantiating PURSES2, so *P.req* and *P.canReq* are the corresponding functions in PURSES2. We see that *req*, if its precondition is true (the purse *n* exists, the message *m* is visible, and *P.canReq* is true):

- calculates a new purse *p1* and message *m1*
- updates the map of purses with the new purse *p1*
- adds *m1* to the ether
- adds *m1* to the visible messages, and then allows some of those to be lost: *hideSome* is a function defined only by the axiom that for any set *ms* of messages $hideSome(ms) \subseteq ms$
- leaves the archive unchanged

A theory, THEORY2, asserts that WORLD2 implements WORLD1, and so inherits all its properties and hence its theories. Hence, by a similar argument to that for level 1, if we can also prove THEORY2, we will have proved that all the 11 concrete operations of WORLD2 satisfy *isCorrect*.

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 by the translator:

Existence TCCs Existence *type-correctness condition*s (TCCs) are generated when there is a need to show that a type is non-empty. We had very few of these, and they were easily discharged.

Subtype TCCs These are generated when there is a need to show that an expression is in a subtype. There were many of these, because there are a number of subtypes in the RSL specification and also many 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 any 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:

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

   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).

Table 1. Numbers of proofs by level and by difficulty

Level	Automatic	Trivial	Complex	Total
0	4	0	0	4
1	41	11	46	98
2	166	9	42	217
Total	211	20	88	319

3.4. The proofs

3.4.1. Effort

Table 1 summarises the proof effort. Here "Automatic" proofs were those done automatically by a tactic built-in to PVS (like `subtype-tcc` or `grind`) or by a simple tactic. We used a simple tactic, essentially

```
(then
 (skosimp* :preds? t)
 (repeat* (decompose-equality))
 (iff)
 (branch (grind :if-match nil) ((grind :if-match all))))
```

which dealt with a number of lemmas in which we needed to "decompose" equalities between sets. Since sets are predicates, proving an expression like `s1 = s2` involving sets of type `T`, say, is usually done by decomposition to

```
ALL (x : T): s1(x) = s2(x)
```

The command `iff` converts the (Boolean) equality in this expression to `IFF` which is then split automatically by `grind` into two implications.

"Trivial" proofs are those done manually, but only needing a few proof commands, and where the way to prove them is obvious. The rest are classified as "Complex", though the amount of effort and skill involved in proving them varied considerably.

3.4.2. Difficulties

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

1. At level 2, the proofs that the 11 operations preserve *isWorld* each involves proving 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, a value *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 theorems (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.

3.4.3. 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 impractical 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.

4. Mondex in SAL

We are currently developing a translator from RSL to SAL [PG06], and applied this translator to the concrete (level 2) Mondex specification.

4.1. SAL

SAL provides a set of commands for the analysis of specifications. The SAL language is similar to the input language of some other verification tools and describes transition systems in terms of initialisation and state transitions. One of SAL's components implements a symbolic model checker called sal–smc, which allows users to specify assertions in linear temporal logic (LTL). For invalid LTL-assertions, a counter-example is produced.

4.2. Motivation

One might ask why we bothered with a translation to SAL since the SAL version 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 trying to prove 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.
- It is good also to demonstrate some liveness properties. It is not clear that Mondex has any classical liveness properties—nothing is guaranteed to happen eventually (apart perhaps for the unbounded increasing of sequence numbers)—but we can use model checking to demonstrate simple liveness in terms of some possible sequences of events. In particular, we used model checking to prove 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 to the empty purse can be made.

It would be possible to prove this result with PVS, but it is complicated to state because of the inherent nondeterminism of the system through the possible loss of messages. It would involve asserting that if there is a (consistent) state in which one purse has a positive balance, and there is another purse, then there is a sequence of states which are each possible results of *startFrom*, *startTo*, *req*, *val*, and then *ack* applied to the previous state such that in the final state some amount, at most the balance, is transferred to another purse. You cannot just assert that this sequence of operations models a (definite) transfer, since in general it models a transfer or an abort.
In fact we did specify a multiple operation to do a definite transfer of money, but it was rather an artificial exercise. We could "prove" in PVS that the operation would effect a transfer, but the "proof" exploits a feature of the classical PVS logic that the epsilon (set choice) operator is deterministic: this is not true in the richer RSL logic. Determinism of epsilon enables us to state a precondition of the operation that predicts what messages will be lost during it.
Model checking provides a much more convenient method of demonstrating such properties, either of classical liveness (something will eventually happen) or simple liveness (something is possible).

- We will see below in Sect. 4.5 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.

4.3. Simplifications

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

- There are only two purses.
- Money is 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 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.
- Sequence numbers (used in the concrete version to ensure uniqueness of messages, and modelled as natural numbers) are in the range *0..3*.

The more we reduce the model in order to model check it, the harder it is to justify that results from the model checker actually apply to the full model. The last simplification seems particularly drastic, because it reduces the number of "runs" of the protocol: if the sequence numbers are restricted to *0..n* then at most *n* transfers can take place. So, for example, if *Money* were in the range *0..4* we could "prove" that a purse with a balance of four would never become empty!

We also made some changes to split the state into smaller components. Such modifications do not reduce the size of the state, or change the behaviour, but they make the specification more accessible to the optimisations 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.
- Some data types are split into a number of disjoint subtypes, and assertions that are universal quantifications over them are split into a number of assertions each over one of the subtypes. For example, the type we use to represent pay-details was split up into four disjoint subtypes and assertions modified accordingly.
- Extra state variables are introduced to simplify the LTL assertions. For example, a pair of state components hold the total money in circulation in the previous and current states, allowing for a simple formulation of the assertion that money in circulation does not increase.
- Pay-details are injectively mapped to a range of natural numbers, and this range used as the domain of the predicates representing sets of pay-details.

Even with the simplifications and modifications listed above the SAL checking of the assertions mentioned in Sect. 4.4 grows to a maximum memory usage of 1,133 MB.

4.4. Results

We have been able to prove (for the simplified specification):

- the original Mondex requirements that money in circulation does not increase and that all money is accounted,
- simple liveness in the sense that
  - an empty purse can become non-empty,
  - a non-empty purse can become empty, and,
  - money can be lost,
- all of the state invariants *isWorld*, *isWorldAxiom* and *isPurse*,
- all of the confidence conditions (see Sect. 4.5).

4.5. 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 $balance(p) \geqslant 1$ 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. Prototype implementation

We also produced a prototype implementation of the specification using the RSL to C++ translator [AG00]. This involved a few changes to the specification, the first two to make it translatable, the next two for convenience:

- Things that were left abstract in the specification, such as the type *Name* and the definition of *increase* for sequence numbers, were made concrete.
- Some set comprehensions and quantified expressions were rewritten slightly so that the comprehension or quantification is over a set.
- The operations concerned with archiving purse exception logs were removed, just to keep the prototype and its interface simple.
- The top-level specification WORLD2 was extended to provide an imperative interface: a variable of type *World* was defined and functions defined for the operations that would update this variable. This makes it simpler to define an interface that allows the user to repeatedly call these functions to run the system. Some other functions useful in the interface, such as one to remove a message from the visible ether according to a parameter generated randomly, were also included.

A simple command line interface was then defined by hand in C++ (just 90 lines of code). This is a simple loop that repeatedly:

- displays the states of the purses, the messages currently in the ether, and the current values of *inPurses*, *inTransit*, and *lost*, plus the contents of the message which disappeared if one disappeared at the previous step (see below)
- asks the user to choose an operation and a purse to run it on
- if there is a visible message in the ether such that the operation's precondition is true, then the operation is performed using that message, updating the state accordingly, and adding any output message to the ether; otherwise it tells the user the operation could not be performed
- a random number is generated and used to possibly select a message to disappear from the visible messages; if such a message is selected its disappearance is reported to the user.

This provides a simple means to exercise the specification and show the results. Building the prototype is just a case of running the translator to C++, using `#include` to incorporate the translated code into the hand-written file, and compiling.

As a default, the translator generates code to check the confidence conditions we mentioned in Sect. 4.5. While running the prototype we are therefore also checking that preconditions are not violated, that the state invariant (subtype) is maintained, etc.

Running the prototype exposed a possible "denial of service" attack on the protocol. There is a user option to start a transfer by inputting the from and to purse and the amount. This *start_transfer* operation places a pair of *startFrom* and *startTo* messages in the ether. The *startTo* message to be read by the to purse, for example, includes the identity and sequence number of the from purse. If there are several such messages in the ether (with different sequence numbers, left by previous runs) the to purse cannot tell which is appropriate, since it has no other information about the current sequence numbers of other purses, and this can cause the protocol to fail later because the sequence number of the from purse will be wrong if the to purse selected an earlier message. Thus an attacker can cause the protocol to possibly fail by recording and replaying earlier such messages. These messages are sent in the clear, so the attack could also work by the attacker generating such messages. But since it need only involve replay and not decryption, this attack could work even if the *startFrom* and *startTo* messages were encrypted.

We changed the prototype so that the *start_transfer* operation removed from the ether any existing *startFrom* and *startTo* messages.

There is an obvious issue here about whether this attack was foreseen by the designers, and whether resilience against such attacks could be in the requirements. The specification described in this paper (along with the original Z specification and other Mondex specifications) abstracts away from the message delivery mechanism. They merely posit an "ether" of messages that have been sent (and that have not been lost), and the preconditions of the operations merely state the availability of an appropriate message. The actual triggering of operations may, of course, involve much stronger conditions, probably only reading one message on a transmission channel, and then checking this against the precondition. But the prototype takes the simple view that the precondition is sufficient, and part of this is "if there is a suitable message available, then use it". This notion of an ether of messages means that the proofs of correctness also hold in an environment where messages are replayed by an attacker.

What was not noticed by the authors of this paper, at least in terms of possible significance, until the problem with running the prototype, is that most operations depend on a unique message: the one of the right type for

this operation that contains a payment details identical to that held by the purse. Any other message can be ignored. But the *startFrom* and *startTo* messages cannot be validated by their recipient in this way. This affects the prototype's behaviour; whether it affects the real system's behaviour depends on how message transmission and reception is handled. For example, if the real protocol for each operation is to read one message and act on it (aborting if it is not what was expected) then denial of service can be achieved by constantly resending older messages of any type. If the real protocol is to read messages up to some time limit or until an acceptable one is received, then denial of service using older *startFrom* and *startTo* messages can work because older ones can be accepted.

6. 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 succeeded in specifying the complete protocol and in proving mechanically the two security properties of the Z work. 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.

We also model checked the protocol using SAL, and checked the security properties, the confidence conditions (subtype correctness and precondition satisfaction), and some simple liveness properties, in particular showing that transfers of money between purses are possible.

We also built and exercised a prototype in C++. It is worth noting that the "denial of service" attack described in Sect. 5 only became apparent when we used the prototype to conduct more than one run of the protocol. Testing does not provide conclusive evidence of correctness, but it often uncovers problems that were not apparent through formal analysis because they had not been considered and hence were not included in that analysis.

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

Consistency 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 doing the proof manually 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.[2] Since level 2 is mostly constructive it is less likely than more abstract descriptions to contain a contradiction. 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. The translation to and running of an executable prototype also demonstrates that there is no contradiction in the specification.

Correct modelling Is our model correct? We gain more confidence as we get results from proof, from model checking, from prototyping, etc., but these are no defence against we, the authors of the specification, not understanding the requirements. If we did not specify the right thing then all our results are useless. This can only be resolved by someone else 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.

Correct final implementation Is our "concrete" specification correctly implemented? Our level 2 specification is still an abstraction of the actual code. Is the chip itself safe from attack by micro-probing [Mon97]?

Correct tools Are the tools correct? We rely on translators from RSL to PVS, SAL, and C++, and on the PVS and SAL engines and C++ compiler (gcc): none of these have been proved correct.

Correct transition system 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.

[2] This is a consequence of the RSL implementation relation.

Correctness of model check version While model checking does not require interaction, using SAL to automatically check the Mondex specification written in RSL required substantial modification and rewriting of the specification. First, we had to reduce the number of possible states. Secondly, we had to further rewrite the RSL specification to help the SAL optimizer. If we were relying more heavily than in this example on model checking to check correctness then we would need more evidence that these changes maintained the protocol's behaviour.

Accessibility of model checking 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 in particular, 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.

Invariant discovery A 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, including for example any 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.

The critical requirements What are the important properties? The two security 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. The original security properties might be characterised as "security for the bank" rather than "security for the user". Simple liveness properties such as the ability to actually transfer money, or to clear a purse log, might also be interesting. The abstract specification was designed to make it possible to prove the properties stated in the original problem. Proving other properties might require choosing a new abstraction and redoing the development and proofs. The size of the proof task makes this a daunting prospect.

For liveness properties, especially in nondeterministic systems, model checking is much more convenient than proof. 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, either by expressing (classical) liveness conditions directly in LTL, or, for simple liveness (possibility), 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.

Should we have used PVS and SAL directly, instead of starting in RSL? The immediate answer is no, because there is no notion of refinement in PVS or SAL, and our strategy is based on refinement, on starting from an abstract specification of the properties to be proved and showing that a concrete specification refines it. But there is a second issue, of whether specifications written directly in PVS or SAL could have made the proofs easier, more amenable to automation, easier to model check? 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 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.

Acknowledgments

We are indebted to Jim Woodcock and the other people working on Mondex for inspiration, to Leonard Moura and Shankar at SRI for advice on using SAL and PVS respectively, and to Marko Schütz of The University of the South Pacific for his help in applying SAL to the Mondex problem. Anonymous referees also helped us improve the paper.

References

[AG00] Ahn U, George C (2000) C++ Translator for RAISE Specification Language. Technical Report 331, UNU-IIST, P.O.Box 3058, Macau, November 2000

[BHW06] Bicarregui J, Hoare CAR, Woodcock JCP (2006) The verified software repository: a step towards the verifying compiler. Formal Aspects Comput 18(2):143–151

[DG02] Dasso A, George CW (2002) Transforming RSL into PVS. Technical Report 256, UNU/IIST, P.O. Box 3058, Macau, May 2002

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

[FACJ] The Mondex Challenge Project. Special Issue. Formal Aspects Comput J

[GH07] George C, Haxthausen AE (2007) Specification, proof, and model checking of the Mondex Electronic Purse using RAISE. Technical Report 352, UNU-IIST, P.O.Box 3058, Macau, February 2007

[HM05] Hoare T, Milner R (2005) Grand challenges for computing research. Comput J 48(1):49–52

[Hoa03] Hoare CAR (2003) The verifying compiler: a grand challenge for computing research. J ACM 50(1):63–69

[Hoa06] Hoare T (2006) The ideal of verified software. In 18th international conference on computer aided verification (CAV2006), vol 4144 of Lecture Notes in Computer Science, Seattle, August 2006. Springer, Heidelberg

[IE97] Ives B, Earl M (1997) Mondex international: Reengineering money. Technical Report CRIM CS97/2, London Business School

[Inc] MasterCard International Incorporated. Mondex

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

[JOW06] Jones C, O'Hearn PW, Woodcock J (2006) Verified software: a grand challenge. IEEE Comput 39(4):93–95

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

[ORS92] Owre S, Rushby JM, Shankar N (1992) PVS: a prototype verification system. In: Kapur D (ed) 11th International Conference on Automated Deduction (CADE), volume 607 of Lecture Notes in Artificial Intelligence. Saratoga, June 1992. Springer, Heidelberg, pp 748–752

[PG06] Perna JI, George C (2006) Model checking RAISE specifications. Technical Report 331, UNU-IIST, P.O.Box 3058, Macau, November 2006

[RAI92] The RAISE Language Group (1992) The RAISE specification language. BCS Practitioner Series. Prentice Hall, Englewood Cliffs

[RAI95] The RAISE Method Group (1995) The RAISE development method. BCS Practitioner series. Prentice Hall, Englewood Cliffs. Available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book

[SCW00] Stepney S, Cooper D, Woodcock JCP (2000) An electronic purse: specification, refinement, and proof. Technical Monograph PRG-126, Oxford University Computing Laboratory

[Spi92] Spivey JM (1992) The Z Notation: a reference manual, 2nd edn. Prentice Hall International Series in Computer Science, Englewood Cliffs

[WF06] Woodcock J, Freitas L (2006) Z/Eves and the Mondex electronic purse. In: Theoretical aspects of computing—ICTAC 2006, third international colloquium, vol 4281 of Lecture Notes in Computer Science, Tunis, November 2006. Springer, Heidelberg, pp 15–34

[Woo06] Woodcock J (2006) First steps in the verified software grand challenge. IEEE Comput 39(10):57–64

Received 19 January 2007
Accepted in revised form 29 October 2007 by J. C. P. Woodcock
Published online 5 December 2007