Algebraic Approaches to Formal Analysis of the Mondex Electronic Purse System

Weiqiang Kong, Kazuhiro Ogata, and Kokichi Futatsugi
Graduate School of Information Science
Japan Advanced Institute of Science and Technology
March 23, 2007
IS-RR-2007-004

Algebraic Approaches to Formal Analysis of the Mondex Electronic Purse System

Weiqiang Kong, Kazuhiro Ogata, and Kokichi Futatsugi

Graduate School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1, Asahidai, Nomi, Ishikawa 923-1292, Japan
{weiqiang,ogata,kokichi}@jaist.ac.jp

March 23, 2007

Abstract

Mondex is a payment system that utilizes smart cards as electronic purses for financial transactions. The paper first reports on how the Mondex system can be modeled, specified and interactively verified using an equation-based method – the OTS/CafeOBJ method. Afterwards, the paper reports on, as a complementarity, a way of automatically falsifying the OTS/CafeOBJ specification of the Mondex system, and how the falsification can be used to facilitate the verification. Differently with related work, our work provides alternative ways of (1) modeling the Mondex system using an OTS (Observational Transition System), a kind of transition system, and (2) expressing and verifying (and falsifying) the desired security properties of the Mondex system directly in terms of invariants of the OTS.

Keywords: Mondex electronic purse, the OTS/CafeOBJ method, Maude search command, verification, falsification

Contents

1 Introduction **3**

2 Overview of the Mondex Electronic Purse System **3**

3 The OTS/CafeOBJ Method **4**
- 3.1 CafeOBJ: An Algebraic Specification Language 4
- 3.2 Observational Transition Systems (OTSs) 5
- 3.3 Specification of OTSs in CafeOBJ 5
- 3.4 Verification of Invariants of OTSs 6

4 Formalization of the Mondex System **7**
- 4.1 Basic Data Types 7
- 4.2 OTS Model and Its CafeOBJ Specification 9

5 Verification of the Mondex System **14**
- 5.1 Formal Definitions of the Properties 14
- 5.2 Verification of the Properties 16
- 5.3 Summarization of the Specification and Verification 19

6 Falsification of the Mondex System **20**
- 6.1 Maude Specification of the Mondex System 20
- 6.2 Falsification of the Mondex System 22
- 6.3 Some Further Issues about Verification and Falsification 23

7 Related Work **24**

8 Conclusion **25**

A Mondex Specifications including Data Types **27**

B Invariants Proved **31**

C A Sample Proof Score **37**

1 Introduction

Mondex [1] is a payment system that utilizes smart cards as electronic purses for financial transactions. The system has recently been chosen as a challenge for formal methods [2, 4], after it was originally specified and manually proved for correctness (of refinement) using the Z notations described in [5]. The purpose of setting up this challenge is to see what the current state-of-the-art is in mechanizing the specification, refinement, and proof, and ultimately to contribute to the Grand Challenge – Dependable Software Evolution [2, 3, 4]. As a response, different formal methods have been applied to tackle this same problem, which include, for example, KIV [6, 7], RAISE [8], Alloy [9] etc.

In this paper, we report on how this problem can be tackled by using an equation-based method – the OTS/CafeOBJ method [10]. Specifically, we describe how the Mondex system is modeled as an OTS (Observational Transition System), a kind of transition system that can be straightforwardly written in terms of equations; and how to specify the OTS in CafeOBJ [11, 12], an algebraic specification language; and finally how to express the desired security properties of the Mondex system as invariants of the OTS, and to interactively verify the invariants by writing and executing proof scores using CafeOBJ system.

As a complementarity of the interactive verification of the OTS/CafeOBJ method, we also report on a way of automatically falsifying (finding counterexamples) the OTS/CafeOBJ specification of the Mondex system by using Maude `search` command [13], which is achieved through an automatic translation from the OTS/CafeOBJ specification into corresponding Maude one [14, 15]. The falsification has been shown, from our experience, to be useful for facilitating the the OTS/CafeOBJ method in its different verification stages.

Differently with related work, our work provides an alternative way of modeling the Mondex system in an operational style (in terms of transition system), which is inspired by the work [6, 7], rather than in a relational style as used in [5, 8, 9]; and our work also provides an alternative way of expressing and verifying (and falsifying) desired properties of the Mondex system directly in terms of invariants of an OTS, rather than the refinement construction and proof that are originally used in the Z methods [5] and then used in [6, 7, 8, 9]. This work therefore provides a different way of viewing the Mondex analysis problem and can be used to compare different modeling and proof strategies.

The rest of the paper is organized as follows: Sect. 2 outlines the main part of the Mondex electronic purse system. Sect. 3 introduces the OTS/CafeOBJ method. Sect. 4 and 5 describe how to model and specify the Mondex system, and how to express the desired security properties of the Mondex system as invariants and their corresponding verification method. Sect. 6 discusses the motivation of falsifying the OTS/CafeOBJ specification of the Mondex system and our proposed way to do this. Sect. 7 discusses related work. And finally Sect. 8 concludes the paper and mentions future work.

2 Overview of the Mondex Electronic Purse System

In the Mondex system, the cards, which are used as electronic purses, store monetary value as electronic information, and exchange value with each other through a communication device without using a central controller (such as a remote database). The communication protocol, which is used for transferring electronic value between two cards, say `FromPurse` (the paying purse) and `ToPurse` (the receiving purse), is as follows:

1. The communication device ascertains a transaction by collecting cards' information and sending two messages *startFrom* and *startTo*.

2. `FromPurse` receives the *startFrom* message that contains information of the `ToPurse`, and the amount of value to be transferred.

3. `ToPurse` receives the *startTo* message that contains information of the `FromPurse`, and the amount of value to be transferred. As a result, `ToPurse` sends a *Req* message to `FromPurse` for requesting the amount of value.

4. `FromPurse` receives the *Req* message and decreases its balance, and then sends a message *Val* to `ToPurse` for transferring value.

5. `ToPurse` receives the *Val* message and increases its balance, and then sends a message *Ack* to `FromPurse` for acknowledging the transaction.

Although the communication protocol seems to be simple, it is complicated by several facts as pointed in [5, 9]: (1) the protocol can be stopped at any time, either due to internal reasons of cards, or due to card-holders intentionally doing so; (2) a message can be lost and replayed in the communication channel, and (3) a message can be read by any card. Note, however, that it is assumed that the *Req*, *Val* and *Ack* messages cannot be forged, which is guaranteed by some (unclear) means of cryptographic system [5].

Two key security properties demanded by the Mondex system are that [5]:

(1) No value may be created in the system, namely that the sum of all purses' balances does not increase;

(2) All value is accounted for in the system (no value is lost), namely that the sum of all purses' balances and lost components does not change.

Note that in this paper, we omit another protocol of the Mondex system that deals with uploading exception logs[1] onto a central archive, since it is not directly related to the above properties.

3 The OTS/CafeOBJ Method

3.1 CafeOBJ: An Algebraic Specification Language

Abstract machines as well as abstract data types can be specified in CafeOBJ [11, 12] mainly based on hidden and initial algebras. CafeOBJ has two kinds of sorts: visible and hidden sorts that denote abstract data types and the state spaces of abstract machines, respectively. There are two kinds of operators to hidden sorts: action and observation operators. Action operators denote state transitions of abstract machines, and observation operators let us know the situation where abstract machines are located. Both an action operator and an observation operator take a state of an abstract machine and zero or more data, and return the successor state of the state, and respectively, a value that characterizes the situation where the abstract machine is located.

Declarations of action and observation operators start with `bop`, and those of other operators with `op`. Declarations of equations start with `eq`, and those of conditional ones with `ceq`. The CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules. The CafeOBJ command `red` is used to rewrite a given term.

Basic units of CafeOBJ specifications are modules. The CafeOBJ built-in module `BOOL` that specifies proposition logic is automatically imported by almost every module unless otherwise

[1] Exception logs are used to record information of those failed transactions in which value may be lost (detailed in Sect. 4).

stated. In the module `BOOL`, visible sort `Bool` denoting truth values, and the constants `true` and `false`, and some logical operators such as `not_` (negation), `_and_` (conjunction), and `_implies_` (implication) are declared. The operator `if_then_else_fi` is also available. An under-score `_` indicates the place where an argument is put.

`BOOL` plays an essential role in verification with the CafeOBJ system. If the equations available in the module are regarded as left-to-right rewrite rules, they are complete wrt propositional logic [16]. Any term denoting a propositional formula that is always true (or false) is surely rewritten to `true` (or `false`).

3.2 Observational Transition Systems (OTSs)

Observational Transition Systems (OTSs) [10] is a definition of transition systems that can be straightforwardly written in equations. We assume that there exists a universal state space called Υ, and also that data types used, including the equivalence relation (denoted by $=$) for each data type, have been defined in advance. An OTS $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$, where:

- $\mathcal{O}$: A finite set of observers. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$, where D is a data type and may differ from observer to observer. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence (denoted by $v_1 =_{\mathcal{S}} v_2$) between them wrt $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}, o(v_1) = o(v_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A finite set of conditional transition. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon \rightarrow \Upsilon$, provided that $\tau(v_1) =_{\mathcal{S}} \tau(v_2)$ for each $[v] \in \Upsilon / =_{\mathcal{S}}$ and each $v_1, v_2 \in [v]$. $\tau(v)$ is called the successor state of $v \in \Upsilon$ wrt. τ. The condition c_τ of τ is called the effective condition. For each $v \in \Upsilon$ such that $\neg c_\tau(v)$, $v =_{\mathcal{S}} \tau(v)$.

Reachable states wrt $\mathcal{S}$ are inductively defined: (1) each $v_0 \in \mathcal{I}$ is reachable, and (2) for each $\tau \in \mathcal{T}$, $\tau(v)$ is reachable if $v \in \Upsilon$ is reachable. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states wrt $\mathcal{S}$. An invariant wrt $\mathcal{S}$ is a state predicate $p : \Upsilon \rightarrow$ `Bool`, which holds in all reachable states wrt $\mathcal{S}$, namely that $\forall v \in \mathcal{R}_{\mathcal{S}}.p(v)$.

Observers and transitions may be parameterized. Generally, observers and transitions are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, provided that $m, n \geq 0$ and there exists a data type D_k such that $k \in D_k$ ($k = i_1, \ldots, i_m, j_1, \ldots, j_n$).

3.3 Specification of OTSs in CafeOBJ

The universal state space Υ is denoted by a hidden sort, say H. An observer $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operator and declared as `bop` $o : H\ V_{i_1} \ldots V_{i_m}$ `->` V, where $V_{i_1}, \ldots, V_{i_m}$ and V are visible sorts.

Any initial state in $\mathcal{I}$ is denoted by a constant, say *init*, which is declared as `op` *init* : `->` H . The equation expressing the initial value of $o_{i_1,\ldots,i_m}$ is as follows:

`eq` $o(init, X_{i_1}, \ldots, X_{i_m}) = f(X_{i_1}, \ldots, X_{i_m})$.

X_k is a CafeOBJ variable of V_k, where $k = i_1, \ldots, i_m$, and $f(X_{i_1}, \ldots, X_{i_m})$ is a CafeOBJ term denoting the initial value of $o_{i_1,\ldots,i_m}$.

A transition $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator and declared as `bop` a : $H\ V_{j_1} \ldots V_{j_n}$ `->` H, where $V_{j_1}, \ldots, V_{j_n}$ are visible sorts. $\tau_{j_1,\ldots,j_n}$ may change the value returned by $o_{i_1,\ldots,i_m}$ if it is applied in a state v such that $c_{\tau_{j_1,\ldots,j_n}}(v)$, which can be written generally as follows:

$$\texttt{ceq}\ o(a(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = e\text{-}a(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})\ \texttt{if}\ c\text{-}a(S, X_{j_1}, \ldots, X_{j_n})\ .$$

S is a CafeOBJ variable for H and X_k is a CafeOBJ variable of V_k, where $k = i_1, \ldots, i_m, j_1, \ldots, j_n$. $a(S, X_{j_1}, \ldots, X_{j_n})$ denotes the successor state of S wrt $\tau_{j_1,\ldots,j_n}$. $e\text{-}a(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$ denotes the value returned by $o_{i_1,\ldots,i_m}$ in the successor state. $c\text{-}a(S, X_{j_1}, \ldots, X_{j_n})$ denotes the effective condition $c_{\tau_{j_1,\ldots,j_n}}$.

$\tau_{j_1,\ldots,j_n}$ changes nothing if it is applied in a state v such that $\neg c_{\tau_{j_1,\ldots,j_n}}(v)$, which can be written generally as follows:

$$\texttt{ceq}\ a(S, X_{j_1}, \ldots, X_{j_n}) = S\ \texttt{if}\ \mathit{not}\ c\text{-}a(S, X_{j_1}, \ldots, X_{j_n})\ .$$

If the value returned by $o_{i_1,\ldots,i_m}$ is not affected by applying $\tau_{j_1,\ldots,j_n}$ in any state (regardless of the truth value of $c_{\tau_{j_1,\ldots,j_n}}$), the following equation may be declared:

$$\texttt{eq}\ o(a(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = o(S, X_{i_1}, \ldots, X_{i_m})\ .$$

3.4 Verification of Invariants of OTSs

We describe the verification method of invariants (safety properties) of OTSs and refer interested readers to [17] for the verification method of liveness properties of OTSs.

Some invariants may be proved by case analysis only, but we often need to do (structural) induction on the reachable state space of an OTS $\mathcal{S}$, namely to show that the predicate to be proved invariant holds on any initial state and is preserved by each transition of the OTS. We describe how to prove a predicate p_1 is invariant to $\mathcal{S}$ by such induction through writing proof scores in CafeOBJ. The proof that p_1 is invariant to $\mathcal{S}$ often needs other predicates. We suppose that $p_2, \ldots, p_n$ are such predicates. We then prove $p_1 \wedge \ldots \wedge p_n$ invariant to $\mathcal{S}$. Let $x_{i_1}, \ldots, x_{i_{m_i}}$ whose types are $D_{i_1}, \ldots, D_{i_{m_i}}$ be all free variables in $p_i (i = 1, \ldots, n)$ except for v whose type is Υ.

We first declare and define the operators denoting $p_1, \ldots, p_n$ in a module `INV` (which imports the module where $\mathcal{S}$ is described) as follows:

$$\texttt{op}\ \mathit{inv}_i\ :\ H\ V_{i_1}\ \ldots\ V_{i_{m_i}}\ \texttt{->}\ \texttt{Bool}$$
$$\texttt{eq}\ \mathit{inv}_i(S, X_{i_1}, \ldots, X_{i_m})\ =\ p_i(S, X_{i_1}, \ldots, X_{i_{m_i}})\ .$$

where $i = 1, \ldots, n$. S is a CafeOBJ variable for the hidden sort H, and X_k $(k = i_1, \ldots, i_{m_i})$ is a CafeOBJ variable for the visible sort V_k. $p_i(S, X_{i_1}, \ldots, X_{i_{m_i}})$ is a CafeOBJ term denoting p_i.

In the module `INV`, we also declare a constant x_k denoting an arbitrary value of V_k $(k = 1, \ldots, n)$. These constants are constrained with equations, which make it possible to split the state space, or the case. For example, if we declare a constant x for `Nat` that is the visible sort for natural numbers, x can be used to denote an arbitrary natural number. Suppose that the case is split into two: one where x equals 0 and the other where x does not, namely that x is greater than 0. The former is expressed by declaring the equation "eq $x = 0$.", and the latter is expressed by declaring the equation "eq $(x > 0) = \mathit{true}$.".

We then declare the operators denoting basic formulas to show in the inductive cases (denoted by the transitions of $\mathcal{S}$) and their defining equations in a module `ISTEP` (which imports `INV`) as follows:

```
op istep_i : V_{i_1} ... V_{i_{m_i}} -> Bool
eq istep_i(X_{i_1}, ..., X_{i_{m_i}}) = inv_i(s, X_{i_1}, ..., X_{i_{m_i}}) implies inv_i(s', X_{i_1}, ..., X_{i_{m_i}}) .
```

where $i = 1, \ldots, n$. s and s' are constants of H, which denote an arbitrary state s and a successor state of s.

Now we are ready to show the way of conducting induction by writing proof scores in CafeOBJ. A proof score is composed of proof passages, which are temporal CafeOBJ modules that are created by the CafeOBJ command `open` with a module name as a parameter, and killed by the command `close`. In a proof passage, the reduction command `red` should be included, which reduces (via term rewriting) a term denoting a proposition to its truth values, and more generally to an exclusive-or normal form (in this case, case-splitting is needed).

Let *init* denotes any initial state of the OTS concerned. All we have to do to show that p_i holds on any initial state is to write a proof passage as follows:

```
open INV
  red inv_i(init, x_{i_1}, ..., x_{i_{m_i}}) .
close
```

The proof of each inductive case often needs case analysis. Let us consider the inductive case where it is shown that $\tau_{j_1,\ldots,j_n}$ preserves p_i. Suppose that the state space is split into l sub-spaces for the proof of the inductive case and each sub-space is characterized by a predicate $case_k$ $(k = 1, \ldots, l)$ such that $(case_1 \vee \ldots \vee case_l) \Leftrightarrow$ `true`. Also suppose that $\tau_{j_1,\ldots,j_n}$ is denoted by an action operator a and visible sorts $V_{j_1}, \ldots, V_{j_n}$ correspond to data types $D_{j_1}, \ldots, D_{j_n}$ of the parameters of $\tau_{j_1,\ldots,j_n}$. The proof for case $case_k$ looks like:

```
open ISTEP
  -- arbitrary objects
  op y_{j_1} : -> V_{j_1} .   ...   op y_{j_n} : -> V_{j_n} .
  -- assumptions
  Declarations of equations denoting case_k.
  -- successor state
  eq s' = a(s, y_{j_1}, ..., y_{j_n}) .
  -- check if the predicate is true
  red SIH_i implies istep_i(x_{i_1}, ..., x_{i_{m_i}}) .
close
```

where $i = 1, \ldots, n$. $y_{j_1}, \ldots, y_{j_n}$ are constants that are used as parameters of the CafeOBJ action operator a, and they denote arbitrary objects of intended sorts. The equation with s' as its left-hand side specifies that s' is the successor state after applying any transition denoted by a in the state s. SIH_i is a CafeOBJ term denoting what strengthens the inductive hypothesis $inv_i(s, X_{i_1}, \ldots, X_{i_{m_i}})$ and can be the (`and`) concatenation of different predicates ranging from $inv_1(\ldots)$ to $inv_n(\ldots)$. A comment starts with `--` and terminates at the end of the line.

4 Formalization of the Mondex System

4.1 Basic Data Types

Before describing the OTS model of the Mondex system (more precisely the communication protocol introduced in Sect. 2, which is the core part of the Mondex system), we first describe

some key data types that are used in the OTS, which include: `Purse`, `Message` and `Ether`[2]. Each `Purse` of the Mondex system is constructed using the CafeOBJ operator `mk-purse` that takes the following seven arguments:

(1) `Name`: the name of the purse. This component is the identifier of a purse.

(2) Previous `Balance`: the balance before a coming transaction. Note that this component is introduced and used by us only with the purpose to express and verify (and falsify) the desired properties directly as invariants, while this component is not used in the Z methods and its follow-up work. The value of this component is set (updated) to be equal to the current balance whenever a transaction is going to happen.

(3) Current `Balance`: the current balance of the purse.

(4) `Seqnum`: the sequence number, which is globally unique and is to be used in next transaction. This number is increased (through the operator `nextseqnum`) during any transaction, and thus it is necessary for avoiding replay attacks.

(5) `Status`: the status of the purse. Possible status of a purse is: `idle`, `epr`, `epv`, and `epa`. `idle` denotes that a purse is in a status of either before or after a transaction. The other three status denotes that a purse is expecting value requesting message, expecting value transferring message, and expecting acknowledging message, respectively.

(6) `Paydetail`: the payment detail of a transaction that the purse is currently involved in or just finished. A payment detail is constructed using the CafeOBJ operator `mk-pay` that takes five arguments: the name of the `from` purse and its sequence number, the name of the `to` purse and its sequence number, and the amount of value (also of sort `Bal` for simplicity) to be transferred. Given a payment detail `mk-pay(FN:Name, FS:Seqnum, TN:Name, TS:Seqnum, V:Bal)`, projection operators `from`, `fromno`, `to`, `tono`, and `value` are defined to obtain each of its components.

(7) `Exlog`: the exception log, which is a list of payment details of failed transactions. A transaction can be failed since a message may be lost and the cards may abort the transaction etc. If there are possibilities that money may be lost during a failed transaction, the current payment detail will be recorded into the exception log. A predicate `_/inexlog_` is defined to check whether a payment detail is in the exception log or not.

Given a purse `mk-purse(N:Name, PB:Bal, CB:Bal, SE:Seqnum, ST:Status, P:Paydetail, E:Exlog)`, projection operators `name`, `pbal`, `bal`, `seq`, `sta`, `pay`, and `exlog` are also defined to obtain each of its components.

According to the communication protocol, there are five kinds of `Messages`: `startfrom(N:Name, V:Bal, S:Seqnum)`, `startto(N:Name, V:Bal, S:Seqnum)`, `req(P:Paydetail)`, `val(P:Paydetail)`, and `ack(P:Paydetail)`. For each kind of messages, there exists a predicate to check the attribution of the messages, such as `isstartfrom` and `isreq` etc. For the first two kinds of messages, projection operators `nameofm`, `valueofm` and `seqofm` are defined, and for the remaining three kinds of messages, projection operators `pdofm` is defined. All of these projection operators return the corresponding parts of the messages. Note that we also assume, as the Z work did, that the messages can only be lost and replayed, but cannot be forged, which is guaranteed by some cryptographic means that is not considered here.

[2]Besides denoting data types, these names in `Typewriter` font (with capital initial) are also used, for simplicity, to denote sort names of the corresponding data types.

The Ether is considered as a bag (multi-set) of messages, which is used to formalize the communication channel. All the messages sent by the communication device and purses are put into the ether and the messages received by a purse are those selected from the ether. In this way, we model the fact that a message can be read by any card as mentioned in Sect. 2. Data constructors of Ether are CafeOBJ operators nil and _,_ (of Ethers, where Message is declared as a subsort of Ether). Two predicates _/in_ and empty? are defined for Ether for checking whether a message is in ether and whether the ether is empty. Another two operators get and top are defined to remove the first element and obtain the first element of ether, respectively.

All the above introduced data types, Purse, Message and Ether, together with those used for defining the three data types, such as Name, Bal etc, are defined in CafeOBJ modules. We describe the module defining data type Purse as a demonstration example, and others can be found in the Appendix section.

```
mod! ETHER {
  pr(MESSAGE)
  [Message < Ether]
  op nil : -> Ether
  op _,_ : Ether Ether -> Ether {assoc comm}
  --
  op _/in_ : Message Ether -> Bool
  op get : Ether -> Ether
  op top : Ether -> Message
  op empty? : Ether -> Bool
  --
  vars M M1 M2 : Message          vars E E1 E2 : Ether
  eq (M /in nil) = false .
  ceq (M1 /in (M2,E)) = true if (M1 = M2) .
  ceq (M1 /in (M2,E)) = (M1 /in E) if not(M1 = M2) .
  --
  eq get(M) = nil .               eq get(M,E) = E .
  eq top(M) = M .                 eq top(M,E) = M .
  eq empty?(nil) = true .         eq empty?(M) = false .
  eq empty?(M,E) = false . }
```

The keyword mod! indicates that the module is a tight semantics declaration, meaning the smallest model (implementation) that respects all requirements written in the module. The contents of the module are enclosed in the keywords { }. The keyword pr is used to import a module, here the module MESSAGE. The term [Message < Ether] expresses that a sort Ether is declared, and also that the sort Message declared in the imported module MESSAGE is a subsort of sort Ether. A subsort represents a subset of the elements of the sort. The keywords assoc and comm specifies that the operator _,_ is associative and commutative.

4.2 OTS Model and Its CafeOBJ Specification

The OTS model of the Mondex system is defined in a CafeOBJ module with the name MONDEX using the keyword mod*, which indicates that the module is a loose semantics declaration, meaning an arbitrary model (implementation) that respects all requirements written in the module. The module MONDEX imports all the data type modules defined in advance. A hidden sort Sys is declared in the module as *[Sys]* by enclosing it with *[and]*, which denotes the universal state space Υ of the OTS model.

In the MONDEX module, two observers denoted by CafeOBJ observation operators purse and ether are declared as follows:

```
bop purse : Sys Name -> Purse .
bop ether : Sys -> Ether .
```

Given a state of the OTS and a purse name, observer `purse` returns the content (components) of the purse in this state, and given a state of the OTS, observer `ether` returns the content (messages) of the ether in this state.

A constant `init` is declared as "`op init : -> Sys`" to denote any initial state of the OTS model of the Mondex system. The initial state is characterized by the following two equations:

```
eq purse(init,P)
   = mk-purse(P,ib(P,seedv),ib(P,seedv),is(P,seedn),idle,none,emptyexlog) .
eq ether(init) = nil .
```

In the first equation, variable `P:Name` denotes an arbitrary purse. The right-hand side of the equation describes the components of the purse `P`, which are composed using the operator `mk-purse`. `ib(P,seedv)` is a term denoting the previous balance of `P`, which is set to be equal to its current balance in initial state; `is(P,seedn)` is a term denoting the initial sequence number of `P`. The constants `seedv` and `seedn`, together with the variable `P`, are used as arguments of operators `ib` and `is` to generate these initial values. In addition, any purse denoted by `P` is initially in the status `idle`, and there are no payment detail and exception log for `P`, which are denoted by `none` and `emptylog`, respectively. The second equation says that initially the ether is empty (denoted by `nil`), namely that no message exists in the ether.

Nine transitions, which characterize sending and/or receiving messages, and also the security features of the Mondex system, are declared as follows:

```
bop startpay     : Sys Name Name Bal -> Sys
bop recstartfrom : Sys Name Message  -> Sys
bop recstartto   : Sys Name Message  -> Sys
bop recreq       : Sys Name Message  -> Sys
bop recval       : Sys Name Message  -> Sys
bop recack       : Sys Name Message  -> Sys
bop drop         : Sys -> Sys
bop duplicate    : Sys -> Sys
bop abort        : Sys Name -> Sys
```

(1) Transition denoted by the CafeOBJ action operator `startpay` characterizes that the communication device ascertains a transaction and sends the `startfrom` and `startto` messages.

```
op c-startpay : Sys Name Name Bal -> Bool
eq c-startpay(S,P1,P2,V)
   = sta(purse(S,P1)) = idle and sta(purse(S,P2)) = idle and not(P1 = P2) .
--
ceq purse(startpay(S,P1,P2,V),Q) = purse(S,Q)    if c-startpay(S,P1,P2,V) .
ceq ether(startpay(S,P1,P2,V))
    = startfrom(P2,V,seq(purse(S,P2))),
      startto(P1,V,seq(purse(S,P1))),ether(S)    if c-startpay(S,P1,P2,V) .
ceq startpay(S,P1,P2,V) = S                  if not c-startpay(S,P1,P2,V) .
```

The effective condition (the first equation) denoted by `c-startpay` demands that: the two purses denoted by `P1` and `P2` are in the `idle` status, namely that they are currently not involved in any other transactions; and they are different purses since it is not permitted to perform a transaction between a purse and itself. Note that we did not consider whether the two purses are authentic or not in our modeling, although adding a predicate `authentic` to check this, as other related work did, is simple. The reason is that no clear standards/constraints exist for

a purse being authentic, and we thus currently consider that all purses involved in our model are authentic. The condition did not check whether one of the purses (which is to be the `from` purse of this transaction) has enough value, and this checking is made in the next transition `startfrom` .

If `startpay` is applied when the condition holds: the components of any purse denoted by `Q` are not changed (the second conditional equation); and two messages `startfrom` and `startto` are put into the ether (the third conditional equation). The last conditional equation says that even if `startpay` is applied when the condition does not hold, nothing changes (For simplicity, this last situation will not be explained in the following description of transitions).

(2) Transition denoted by the CafeOBJ action operator `recstartfrom` characterizes that a purse receives the message `startfrom`.

```
op c-recstartfrom : Sys Name Message -> Bool
eq c-recstartfrom(S,P,M)
   = M /in ether(S) and isstartfrom(M) and sta(purse(S,P)) = idle and
     not(P = nameofm(M)) and valueofm(M) <= bal(purse(S,P)) .
--
ceq purse(recstartfrom(S,P,M),Q)
    = mk-purse(Q,(if (P = Q) then bal(purse(S,Q)) else pbal(purse(S,Q)) fi),
               bal(purse(S,Q)),(if (P = Q) then nextseqnum(seq(purse(S,Q)))
                                           else seq(purse(S,Q)) fi),
               (if (P = Q) then epr else sta(purse(S,Q)) fi),
               (if (P = Q) then mk-pay(Q,seq(purse(S,Q)),
                                       nameofm(M),seqofm(M),valueofm(M))
                           else pay(purse(S,Q)) fi),
               exlog(purse(S,Q)))              if c-recstartfrom(S,P,M) .
ceq ether(recstartfrom(S,P,M)) = ether(S)      if c-recstartfrom(S,P,M) .
ceq recstartfrom(S,P,M) = S                  if not c-recstartfrom(S,P,M) .
```

The effective condition denoted by `c-recstartfrom` demands that: there exists a `startfrom` message in the ether; the purse `P` that is going to receive the message is in the status `idle`; the name argument of the `startfrom` message (which is assumed to be the `to` purse's name) is not equal to `P`, namely that `P` is not going to do transaction with itself; and last `P` has enough value for this value requesting.

If `recstartfrom` is applied when the condition holds: the previous balance of `P` is updated to its current balance, namely to record the current balance before a coming transaction as the previous balance; increase the sequence number; change the status of `P` to `epr`; and generate a payment detail. Note that two variables `P` and `Q` both denote purses. However, `P` denotes the purse receiving the message `startfrom` (executing the transition `recstartfrom`), and `Q` denotes the purse that the observer `purse` are "observing" on. After applying `recstartfrom`, `P` becomes the `from` purse of a transaction denoted by its payment detail.

(3) Transition denoted by the CafeOBJ action operator `recstartto` characterizes that a purse receives the message `startto`.

```
op c-recstartto : Sys Name Message -> Bool
eq c-recstartto(S,P,M)
   = M /in ether(S) and isstartto(M) and sta(purse(S,P)) = idle and
     not(P = nameofm(M)) .
--
ceq purse(recstartto(S,P,M),Q)
    = mk-purse(Q,(if (P = Q) then bal(purse(S,Q)) else pbal(purse(S,Q)) fi),
               bal(purse(S,Q)),(if (P = Q) then nextseqnum(seq(purse(S,Q)))
                                           else seq(purse(S,Q)) fi),
```

```
              (if (P = Q) then epv else sta(purse(S,Q)) fi),
              (if (P = Q) then mk-pay(nameofm(M),seqofm(M),
                                      Q,seq(purse(S,Q)),valueofm(M))
                          else pay(purse(S,Q)) fi),
              log(purse(S,Q)))                    if c-recstartto(S,P,M) .
ceq ether(recstartto(S,P,M))
    = req(pd(nameofm(M),seqofm(M),P,seq(purse(S,P)),valueofm(M))),
      ether(S)                                    if c-recstartto(S,P,M) .
ceq recstartto(S,P,M) = S                      if not c-recstartto(S,P,M) .
```

Equations defining effective condition and application of transition `recstartto` are similar to those of transition `recstartfrom`, except that: the condition demands a `startto` message in the ether; the status of the purse is changed to `epv`; and a `req` message is put into the ether. After applying `recstartto`, `P` becomes the `to` purse of the transaction denoted by its payment detail.

(4) Transition denoted by the CafeOBJ action operator `recreq` characterizes that a purse receives the message `req`.

```
op c-recreq : Sys Name Message -> Bool
eq c-recreq(S,P,M)
   = M /in ether(S) and isreq(M) and sta(purse(S,P)) = epr and
     pay(purse(S,P)) = pdofm(M) .
--
ceq purse(recreq(S,P,M),Q)
    = mk-purse(Q,pbal(purse(S,Q)),
               (if (P = Q) then (bal(purse(S,Q)) - value(pdofm(M)))
                           else bal(purse(S,Q)) fi),
               seq(purse(S,Q)),
               (if (P = Q) then epa else sta(purse(S,Q)) fi),
               pay(purse(S,Q)),log(purse(S,Q)))      if c-recreq(S,P,M) .
ceq ether(recreq(S,P,M)) = val(pdofm(M)),ether(S)    if c-recreq(S,P,M) .
ceq recreq(S,P,M) = S                              if not c-recreq(S,P,M) .
```

The effective condition denoted by `c-recreq` demands that: there exists a `req` message in the ether; the purse `P` that is going to receive the `req` message is in the status `epr`; and the payment detail of the `req` message is equal to the payment detail of `P`. If `recreq` is applied when the condition holds, the current balance of `P` is deceased with the requested amount of value; the status of `P` is changed to `epa`; and a `val` message is put into the ether.

(5) Transition denoted by the CafeOBJ action operator `recval` characterizes that a purse receives the message `val`.

```
op c-recval : Sys Name Message -> Bool
eq c-recval(S,P,M)
   = M /in ether(S) and isval(M) and sta(purse(S,P)) = epv and
     pay(purse(S,P)) = pdofm(M) .
--
ceq purse(recval(S,P,M),Q)
    = mk-purse(Q,pbal(purse(S,Q)),
               (if (P = Q) then (bal(purse(S,Q)) + value(pdofm(M)))
                           else bal(purse(S,Q)) fi),
               seq(purse(S,Q)),
               (if (P = Q) then idle else sta(purse(S,Q)) fi),
               pay(purse(S,Q)),log(purse(S,Q)))      if c-recval(S,P,M) .
ceq ether(recval(S,P,M)) = ack(pdofm(M)),ether(S)    if c-recval(S,P,M) .
ceq recval(S,P,M) = S                              if not c-recval(S,P,M) .
```

The effective condition denoted by `c-recval` demands that: there exists a `val` message in the ether; the purse `P` that is going to receive the message is in the status `epv`; and the payment detail of the `val` message is equal to the payment detail of the purse `P`. If `recval` is applied when the condition holds: the current balance of `P` is increased with the transferred amount of value; the status of `P` is changed to `idle`. which means that the transaction is completed at the `to` purse's side; and a `ack` message is put into the ether.

(6) Transition denoted by the CafeOBJ action operator `recack` characterizes that a purse receives the message `ack`.

```
op c-recack : Sys Purse Message -> Bool
eq c-recack(S,P,M)
   = M /in ether(S) and isack(M) and sta(purse(S,P)) = epa and
     pay(purse(S,P)) = pdofm(M) .
--
ceq purse(recack(S,P,M),Q)
    = mk-purse(Q,pbal(purse(S,Q)),bal(purse(S,Q)),seq(purse(S,Q)),
               (if (P = Q) then idle else sta(purse(S,Q)) fi),
               pay(purse(S,Q)),log(purse(S,Q))) if c-recack(S,P,M) .
ceq ether(recack(S,P,M)) = ether(S)             if c-recack(S,P,M) .
ceq recack(S,P,M) = S                        if not c-recack(S,P,M) .
```

The effective condition denoted by `c-recack` demands that: there exists a `ack` message in the ether; the purse `P` that is going to receive the `ack` message is in the status `epa`; and the payment detail of the `ack` message is equal to the payment detail of `P`. If `recack` is applied when the condition holds: the status of `P` is changed to `idle`, which denotes that a transaction is successfully completed.

In addition to the above described transitions that correspond to the sending and receiving messages of the communication protocol of the Mondex system, there are three more transitions to characterize security features of the Mondex system, which include: the ether is unreliable, and a transaction can be stopped at any time.

(7) To characterize that the messages in the ether may be lost and replayed, we define two more transitions: `drop` and `duplicate`. As long as the ether is not empty, transition `drop` can remove a message from the ether, and transition `duplicate` can duplicate a message and put it into the ether. Equations defining these two transitions are as follows:

```
op c-drop : Sys -> Bool
eq c-drop(S) = not empty?(ether(S)) .
--
ceq purse(drop(S),Q) = purse(S,Q)                  if c-drop(S) .
ceq ether(drop(S)) = get(ether(S))                 if c-drop(S) .
ceq drop(S) = S                                 if not c-drop(S) .

op c-duplicate : Sys -> Bool
eq c-duplicate(S) = not empty?(ether(S)) .
--
ceq purse(duplicate(S),Q) = purse(S,Q)           if c-duplicate(S) .
ceq ether(duplicate(S)) = top(ether(S)),ether(S) if c-duplicate(S) .
ceq duplicate(S) = S                          if not c-duplicate(S) .
```

(8) To characterize that a transaction can be stopped at any time, namely that a purse can abort a transaction at any time as the card-holder wishes, we define the transition `abort` as follows:

```
eq purse(abort(S,P),Q)
   = mk-purse(Q,pbal(purse(S,Q)),bal(purse(S,Q)),
              (if (P = Q) then nextseqnum(seq(purse(S,Q)))
                          else seq(purse(S,Q)) fi),
              (if (P = Q) then idle else sta(purse(S,Q)) fi),
              pay(purse(S,Q)),
              (if (P = Q) then
                  (if (sta(purse(S,Q)) = epa or sta(purse(S,Q)) = epv)
                              then pay(purse(S,Q)) @ log(purse(S,Q))
                              else log(purse(S,Q)) fi)
                          else log(purse(S,Q)) fi)) .
eq ether(abort(S,P)) = ether(S) .
```

Note that no effective condition is defined for transition `abort`, which means that the transition `abort` can be executed at any time. When a purse aborts the transaction, the status of the purse is changed to `idle`, and its sequence number is increased. In addition, if the purse aborts the transaction when it is in status of either `epa` or `epv`, which means that a `from` purse has transferred value or a `to` purse is waiting for value being transferred (has not received the value), namely that there exist possibilities that the value can be lost, the payment detail of this transaction has to be recorded to the exception log of the aborting purse (through concatenation operator `@`). Note that a same payment detail may be logged in both `from` and `to` purse, although a value is only lost once. The purpose of this is to analyze the exception logs in the future by comparing the two logs and refund value if value did be lost.

5 Verification of the Mondex System

5.1 Formal Definitions of the Properties

In the original Z work [5], and later in the KIV, RAISE and Alloy work [6, 7, 8, 9], the two security properties of the Mondex system are defined respectively in the forms look like:

(1) `totalBalanceofPurse'` $\leq$ `totalBalanceofPurse`, which states that the sum of the before (transaction) balances of all purses is greater or equal to the sum of the after (transaction) balances of all purses.

(2) `totalBalanceofPurse' + totalLostofPurse' = totalBanlanceofPurse + totalLostofPurse`, which states that the sum of the before balances and lost value of all purses is equal to the sum of the after balances and lost value (due to a possibly failed transaction) of all purses.

In our work, through making use of the introduced component "previous balance" of purses, we make the notion "before balance" explicit. In addition, we are also able to express the "after balance" (through the component "current balance" of purses) and the execution of any one transaction (through the components "status" and "payment detail" of purses). The two properties of the Mondex system can thus be defined as invariants of the OTS model following the above forms of related work. Formal definitions of the two properties are in the following.

1. For any reachable state s, any two purses p_1 and p_2:

   (sta(purse(s,p_1)) = idle and sta(purse(s,p_2)) = idle and
   pay(purse(s,p_1)) = pay(purse(s,p_2)) and not($p_1 = p_2$))
   implies
   (bal(purse(s,p_1)) + bal(purse(s,p_2)) <= pbal(purse(s,p_1)) + pbal(purse(s,p_2))).

In the premise of property 1, two arbitrary different purses denoted by p_1 and p_2 are both in the status `idle`, which means that p_1 and p_2 are currently not involved in any transactions; additionally the equality between their payment details expresses that either they are never involved in any transactions (thus their payment details are both `none`), or a transaction between them is just finished (finished normally or abnormally by aborting the transaction does not matter). Therefore, property 1 can be read as: for two arbitrary different purses, (1) if no transactions ever happen for each of the two purses, or (2) after any one transaction between them, the sum of their current balances is not increased (less or equal to the sum of their balances before the transaction). This implicitly implies the above description of property 1 that covers all possible purses for any possible number of transactions.

2. For any reachable state s, any two purses p_1 and p_2:

```
(sta(purse(s,p1)) = idle and sta(purse(s,p2)) = idle and
pay(purse(s,p1)) = pay(purse(s,p2)) and not(p1 = p2))
implies
(if pay(purse(s,p1)) /inexlog log(purse(s,p1)) and
    pay(purse(s,p2)) /inexlog log(purse(s,p2))
 then bal(purse(s,p1)) + bal(purse(s,p2)) + lost(pay(purse(s,p1)))
      = pbal(purse(s,p1)) + pbal(purse(s,p2))
 else bal(purse(s,p1)) + bal(purse(s,p2))
      = pbal(purse(s,p1)) + pbal(purse(s,p2)) fi).
```

The premise of property 2 is exactly same as property 1, which states that two arbitrary different purses are either never involved in any transaction or a transaction between them is just finished. To understand the conclusion part of property 2, let us see the following table, which analyzes, under the property's premise, whether value is lost or not during a transaction.[3]

from \ **to**		abort	non-abort
abort	log	*lost* (a)	not lost (b)
	non-log	not lost (c)	*impossible* (d)
non-abort		*impossible* (e)	not lost (f)

A `from` purse can be in the status `idle`, `epr` and `epa`, and a `to` purse can be in the status `idle` and `epv`. Since aborting of either the `from` purse or the `to` purse in status `idle` only increases its sequence number, and the current and previous balances remain unchanged, we only analyze the situations that a purse aborts in the status `epr`, `epa` (for `from` purse) and `epv` (for `to` purse). `non-abort` in the table denotes that a purse finished successfully the transaction on its side, and `abort` denotes that a purse finished the transaction (on its side) by aborting it. `log` and `non-log` are used to distinguish that `from` purse aborts the transaction on status `epa` or `epr` (only aborting in `epa` will be logged). The `to` purse will always log the transaction when aborting the transaction (in `epv`). The items of the table labeled with (a) – (f) are explained as follows:

(a) The `from` purse aborts the transaction after it decreases its current balance and sends the `val` message (in `epa`), and the `to` purse aborts the transaction before it receives the `val` message (in `epv`). Therefore value is lost.

[3] As to the other situation denoted by the premise that two purses are never involved in any transactions, it is obviously that no value is lost. So this situation is omitted in the following discussion.

(b) The `from` purse aborts the transaction after it decreases its current balance and sends the `val` message, and the `to` purse does not abort the transaction. Since the `to` purse is in status `idle` (as the premise says), it has successfully received the `val` message and therefore no value is lost.

(c) The `from` purse aborts the transaction before it decreases its current balance (in `epr`), and the `to` purse aborts when it is waiting the `val` message (in `epv`). Therefore no value is lost.

(d) The `from` purse aborts the transaction before it decreases its current balance, and the `to` purse finishes the transaction successfully. This situation is impossible since no `val` message has ever been sent.

(e) The `from` purse successfully finished the transaction, and the `to` purse aborts the transaction when it is waiting the `val` message. This situation is impossible since no `ack` message has ever been sent.

(f) Both the `from` and the `to` purse finish the transaction successfully. Therefore no value is lost.

The above analyzed situations from (a) – (f) are reflected in the formula for property 2, in which `lost` is a function that counts the lost value of a transaction (denoted by the payment detail). Therefore, for any one transaction between two arbitrary different purses, if value is lost, the value is logged in the exception logs of both the `from` and `to` purses, and the sum of their current balances plus the lost value is equal to the sum of their previous balances before this transaction; otherwise, value is not lost, and the sum of their current balances is equal to the sum of their previous balances before this transaction.

5.2 Verification of the Properties

We describe the inductive proof of property 2 by writing and executing proof scores using CafeOBJ system. An inductive case of the proof, which shows that transition `recack` preserves the property, is selected (from eight inductive cases corresponding to the transitions of the OTS) and described as a demonstration example. The inductive case needs three other invariants[4] (called here as properties 3, 4 and 5) to strengthen the inductive hypothesis of property 2. Formal definition of properties 3, 4 and 5 are as follows:

3. For any reachable state s and any purse p :

   `sta(purse(`s`,`p`)) = epa`
   `implies`
   `bal(purse(`s`,`p`)) = pbal(purse(`s`,`p`)) - value(pay(purse(`s`,`p`))) .`

4. For any reachable state s, any two purses p_1 and p_2, and any message m :

   m `/in ether(`s`) and isack(`m`) and pay(purse(`s`,`p_1`)) = pdofm(`m`) and`
   `pay(purse(`s`,`p_2`)) = pdofm(`m`) and not(`$p_1 = p_2$`) and`
   `sta(purse(`s`,`p_1`)) = epa and sta(purse(`s`,`p_2`)) = idle`
   `implies`
   `bal(purse(`s`,`p_2`)) = pbal(purse(`s`,`p_2`)) + value(pay(purse(`s`,`p_2`))) .`

[4]Actually these three invariants are found during the proof of property 2. The method of finding these invariants is to be introduced in the following of this subsection, and also in Sect. 6.3.

5. For any reachable state s, any purse p :

$$\texttt{sta(purse(}s\texttt{,}p\texttt{))} = \texttt{epa implies not(pay(purse(}s\texttt{,}p\texttt{)) /inexlog log(purse(}s\texttt{,}p\texttt{)))} .$$

As introduced in Sect. 2, we declare the operators denoting properties 2, 3, 4 and 5 in the module INV as follows:

```
mod INV {
  pr(MONDEX)
  -- arbitrary objects
  ops p p1 p2 : -> Name
  -- declare invariants to prove
  op inv2 : Sys Name Name -> Bool            op inv3 : Sys Name -> Bool
  op inv4 : Sys Name Name Message -> Bool    op inv5 : Sys Name -> Bool
  -- CafeOBJ variables
  var S : Sys
  var P P1 P2 : Name                         var M : Message
  -- equations defining invariants
  eq inv2(S,P1,P2)   = ...                   eq inv3(S,P) = ...
  eq inv4(S,P1,P2,M) = ...                   eq inv5(S,P) = ... }
```

The omitted part "..." in the right-hand sides of equation definitions of properties are the corresponding terms for properties presented before (but replacing the symbols s, m, p_1 and p_2 with variables S, M, P1 and P2). In proof scores to be introduced later, although constants and variables both denote arbitrary objects of intended sorts, the scope of a constant is to the end of the proof score, while the scope of a variable is inside of an equation.

We then declare the operator denoting the basic formula to prove in each inductive case of the proof of property 2, and give their definitions in equation in the module ISTEP as follows:

```
mod ISTEP {
  pr(INV)
  -- arbitrary objects
  ops s s' : -> Sys
  -- declare predicates to proved in inductive cases
  op istep2 : Name Name -> Bool
  -- CafeOBJ variables
  vars P P1 P2 : Name              var M : Message
  -- equations defining the inductive cases
  eq istep2(P1,P2) = inv2(s,P1,P2) implies inv2(s',P1,P2) .
  eq istep3(P) = inv3(s,P) implies inv3(s',P) .
  eq istep4(P1,P2,M) = inv4(s,P1,P2,M) implies inv4(s',P1,P2,M) .
  eq istep5(P) = inv5(s,P) implies inv5(s',P) . }
```

where s and s' are constants of sort Sys, and s denotes an arbitrary state and s' denotes a successor state of s. Note that since properties 3, 4, and 5 should also be proved to complete the proof of property 2, operator denoting inductive cases of the proofs of properties 3, 4 and 5 should also be declared and defined, however their proofs will not be described here.

We show that property 2 holds on the initial state (the base case) by writing the following proof scores:

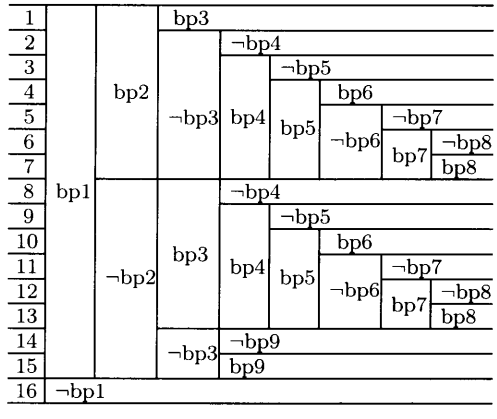
```
open INV
  red inv2(init,p1,p2) .
close
```

where the constant init is declared in the module MONDEX, and constants p1 and p2 are declared in the module INV. CafeOBJ system returns true for this proof score, meaning that property 2 holds on any initial state.

We then show that property 2 is preserved by each transition of the OTS, namely the inductive cases. In the selected inductive case denoted by transition `recack`, the case is split into sixteen sub-cases based on the following predicates:

```
bp1 ≝ c-recack(s,q,m)
bp2 ≝ p1 = q
bp3 ≝ p2 = q
bp4 ≝ sta(purse(s,p2)) = idle
bp5 ≝ pdofm(m) = pay(purse(s,p2))
bp6 ≝ pdofm(m) /inexlog log(purse(s,q)) and
      pdofm(m) /inexlog log(purse(s,p2))
bp7 ≝ bal(purse(s,q)) = pbal(purse(s,q)) - value(pdofm(m))
bp8 ≝ bal(purse(s,p2)) = pbal(purse(s,p2)) + value(pdofm(m))
bp9 ≝ pay(purse(s,p1)) /inexlog log(purse(s,p1)) and
      pay(purse(s,p2)) /inexlog log(purse(s,p2))
```

The constant `s` of sort `Sys` denoting an arbitrary state, is the one declared in module `ISTEP`; the constants `q`, `p1` and `p2` are of sort `Name` denoting arbitrary purses, where `p1` and `p2` are declared in module `INV`; and the constant `m` of sort `Message` denotes an arbitrary message. The case-splitting for the inductive case denoted by transition `recack` is shown in the following table.



Each case in the above table is denoted by the predicate obtained by connecting ones appearing in the row with conjunction. The proof passage for the sub-case 5, namely `bp1` $\wedge$ `bp2` $\wedge$ $\neg$`bp3` $\wedge$ `bp4` $\wedge$ `bp5` $\wedge$ $\neg$`bp6` $\wedge$ $\neg$`bp7`, which uses property 3 to strengthen the inductive hypothesis, is shown, as a demonstration example, as follows:

```
open ISTEP
-- arbitrary objects
  op q : -> Name .                  op m : -> Message .
-- assumption
  -- eq c-recack(s,q,m) = true .
  eq (m /in ether(s)) = true .      eq isack(m) = true .
  eq sta(purse(s,q)) = epa .        eq pay(purse(s,q)) = pdofm(m) .
  --
  eq p1 = q .                       eq (p2 = q) = false .
  eq sta(purse(s,p2)) = idle .      eq pay(purse(s,p2)) = pdofm(m) .
  eq (pdofm(m) /inexlog log(purse(s,q)) and
      pdofm(m) /inexlog log(purse(s,p2))) = false .
  eq (bal(purse(s,q)) = pbal(purse(s,q)) - value(pdofm(m))) = false .
```

```
-- successor state
  eq s' = recack(s,q,m) .
  -- check if the predicate is true.
  red inv3(s,q) implies istep2(p1,p2) .
close
```

Note that the predicate "`c-recack(s,q,m) = true`" is expanded into the first four equations to get more equations available for term rewriting. `inv3(s,q)` is used to strengthen the inductive hypothesis denoted by `inv2(p1,p2)`. Proof passages for the remaining sub-cases of the inductive case `recack` are written similarly. Property 4 is used in the proof passages for sub-cases 7 and 13, and property 5 is used in the proof passages for sub-cases 4 and 10.

We briefly introduce the idea of coming up with property 3 to prove this sub-case. By assuming the equations characterizing the sub-case, we first try to let CafeOBJ system reduce the inductive case `istep2(p1,p2)` directly. However CafeOBJ system does not return `true` as expected. By observing the equations of this sub-case, in particular the one "`sta(purse(s,q)) = epa`", we notice that the status of the purse denoted by `p` is `epa`. Our knowledge about the Mondex system tells us that whenever a purse is in the status `epa`, the purse has already payed money and its current balance has been reduced (which is what property 3 states), and this fact (actually to be proved as a fact) is contrary to the last equation defining predicate bp7. We thus use property 3 to strengthen the inductive hypothesis and this sub-case can be discharged.

We last mention an important observation that is found during the verification of property 1. An invariant called here as property 6 is used to strengthen the inductive cases of property 1, which is as follows:

6. For any reachable state s, any two purses p_1 and p_2 :

   `pay(purse(`s`,`p_1`)) = pay(purse(`s`,`p_2`)) and not(`$p_1 = p_2$`)`
   `implies`
   `bal(purse(`s`,`p_1`)) + bal(purse(`s`,`p_2`)) <= pbal(purse(`s`,`p_1`)) + pbal(purse(`s`,`p_2`))` .

Property 6 is interesting in the sense that it is stronger than the original property 1. It reveals the fact of our Mondex specification that: at any point of a transaction following the communication protocol, the sum of the current balances of purses is equal or smaller than the sum of their previous balances before the transaction. In other words, no value is created at any point of a transaction (not only after the transaction as stated in property 1).

5.3 Summarization of the Specification and Verification

We give a brief summarization of the OTS/CafeOBJ specification and verification of the Mondex system. The CafeOBJ specification of the OTS model of the Mondex system is approximately 1100 lines. And 55 other invariant properties are proved and used as lemmas to prove the two desired properties of the Mondex system, and the whole proof scores are approximately 47000 lines. Although the proof scores seem to be long, most of the work is "copy-and-paste" work, and the difficult task in verification is to come up with some of those 55 lemmas. It took about 5 minutes to have the CafeOBJ system load the CafeOBJ specification and execute all the proof scores on a desktop computer with 3.2GHz processor and 2GB memory. It took a couple of weeks to complete the case study. The system specification including those data type specifications, and the definitions of the 55 invariants can be found from Appendix section.

6 Falsification of the Mondex System

As a complementarity of the interactive verification of the OTS/CafeOBJ method, we report on a way of automatically falsifying the Mondex system by employing Maude model checking facilities (in particular the Maude `search` command [13]) to take advantage of (1) the fully automatic verification/falsification procedure, and (2) informative counterexamples.

An implemented prototype translator [15] that translates the OTS/CafeOBJ specifications into corresponding Maude ones is used as the basis for this falsification. As a sibling language of CafeOBJ, Maude is a specification and programming language based on rewrite logic, which is equipped with model checking facilities. The primary reason of choosing Maude is that it supports model checking on abstract data types including inductively defined data types and does not require the state space of a system to be finite, although the reachable state space of the system should be finite. This finiteness restriction can be abandoned when using Maude `search` command to explicitly explore a finite reachable state space of a system for counterexamples (namely falsification).

One may wonder why we need falsification of the Mondex system since we have already verified it using the OTS/CafeOBJ method. The reasons are that falsification can be used to facilitate, in different stages, the interactive verification of the two security properties:

1. Before carrying out the interactive verification, falsifying the two properties can help obtain a certain degree of confidence of the correctness (within a finite reachable state space) of the system and property specifications.

2. During conducting the interactive verification, generating good and correct lemmas is not a simple task. Falsification can help, in this stage, filter out those generated but essentially incorrect lemmas.

6.1 Maude Specification of the Mondex System

The translated Maude specifications of data types are very similar to the original CafeOBJ ones due to being as two sibling algebraic specification languages. We show, as an example, the translated Maude functional module (for defining data types) of the data type `Ether` introduced in Sect. 4.1 as follows:

```
fmod ETHER is
  pr MESSAGE .
  sort Ether .
  subsort Message < Ether .
  op nil : -> Ether .
  op _,_ : Ether Ether -> Ether [assoc comm] .
  ---
  op _/in_ : Message Ether -> Bool .
  op get : Ether -> Ether .
  op top : Ether -> Message .
  op empty? : Ether -> Bool .
  ---
  vars M M1 M2 : Message .      vars E E1 E2 : Ether .
  eq (M /in nil) = false .
  ceq (M1 /in (M2,E)) = true if (M1 = M2) .
  ceq (M1 /in (M2,E)) = (M1 /in E) if not(M1 = M2) .
  ---
  eq get (M) = nil .            eq get (M,E) = E .
  eq top(M) = M .               eq top(M,E) = M .
  eq empty?(nil) = true .       eq empty?(M) = false .      eq empty?(M,E) = false .
endfm
```

The translation for OTS module is not straightforward as the one for data type modules. We first briefly introduce the main idea of obtaining a finite model-checkable model from the infinite OTS one, and then show some of the translated Maude specification. More technical details (translation rules and soundness proof wrt counterexamples[5]) can be found in [14, 15].

The reachable state space of an OTS is generally infinite. We carry out two steps to obtain a finite model from a potentially infinite OTS to make use of Maude model-checking facilities: (1) setting a bound to restrict the number of executions of transitions [18], namely to make the depth of a rewriting tree finite, which is inspired by Bounded Model Checking [19]; and (2) instantiating some necessary data types to make the number of observers and transitions finite, namely to make the breadth of a rewriting tree finite.

Consider two different purses denoted by Maude constants `p1` and `p2` of sort `Name`, the translated Maude specification of the initial state of the OTS/CafeOBJ specification is as follows:

```
eq init = (purse[p1] : mk-purse(p1,ib(p1,seedv),ib(p1,seedv),is(p1,seedn),idle,none,emptyexlog))
          (purse[p2] : mk-purse(p2,ib(p2,seedv),ib(p2,seedv),is(p2,seedn),idle,none,emptyexlog))
          (ether : nil) (steps : 0) .
```

The initial state consists of four terms of observations in the form `(obName : obValue)`, where `obName` is the observer name possibly with parameters enclosed with `[` and `]`, and `obValue` is the value returned by the observer on a certain state. The first two terms describe the two purses `p1` and `p2`, and the third describes the ether. In the last term, `steps` is an newly introduced observer with the purpose to restrict the number of executions of transitions, and its initial value is 0.

The translated Maude specification of transition `recack` of the OTS/CafeOBJ specification is shown, as a demonstration example, as follows:

```
crl[recack_p1]:
   (purse[p1] : PS1) (purse[p2] : PS2) (ether : (M,EH)) (steps : C)
    =>
   (purse[p1] : mk-purse(p1,pbal(PS1),bal(PS1),seq(PS1),idle,pay(PS1),log(PS1)))
   (purse[p2] : mk-purse(p2,pbal(PS2),bal(PS2),seq(PS2),sta(PS2),pay(PS2),log(PS2)))
   (ether : (M,EH)) (steps : (C + 1))
if (isack(M) and sta(PS1) = epa and pay(PS1) = pdofm(M) and C < bound) .
```

The set of equations of the OTS/CafeOBJ specification that characterizes the transition `recack` is translated into Maude conditional rewrite rules. `crl` is the keyword to declare a conditional rewrite rule, and `recack_p1` in the bracket is the label of this rule, which denotes that `p1` receives the message `ack`.

The left-hand side of the rule (before `=>`) denotes the current state of the OTS, which consists of four terms of observations. Maude variables `PS1` and `PS2` of sort `Purse` denotes the return values of observer `purse` on purses `p1` and `p2`, respectively. The term `(M,EH)` of sort `Ether` denotes that current ether consists of a message `M` and the remaining part `EH` of the ether. The right-hand side of the rule (after `=>`) denotes the successor state of the OTS wrt the execution of the rule. The component `status` of purse `p1` is changed to `idle`, and other components remain unchanged. The return value of observer `purse` on purse `p2`, and the return value of observer `ether` remain unchanged.

Note that the return value of the newly introduced observer `steps` is added by 1 after the execution of the rule. Through defining a predicate `C < bound` in the condition of the rule, we can restrict execution of the OTS within finite steps (less than `bound`, which is a natural number predetermined by human verifiers).

[5] i.e. for any counterexample reported by Maude for the translated specification, there exists a corresponding one in the original OTS/CafeOBJ specification.

Predicates in the condition of the rule check that: there exists a `ack` message in the ether; the purse `p1` that is going to receive the message is in the status `epa`; and the payment detail of the `ack` message is equal to the payment detail of `p1`.

Another similar Maude rewrite rule is also generated to characterize the situation that purse `p2` receives the `ack` message. And the sets of equations describing the other seven transitions of the OTS/CafeOBJ specification are translated similarly.

6.2 Falsification of the Mondex System

We show the translated Maude specification of property 1 of the Mondex system as follows and property 2 is translated similarly:

```
search [1] in MONDEX :
init =>* (purse[P1] : PS1) (purse[P2] : PS2) S
         such that not((sta(PS1) = idle and sta(PS2) = idle and
                        pay(PS1) = pay(PS2) and not(P1 = P2))
                        implies
                       (bal(PS1) + bal(PS2) <= pbal(PS1) + pbal(PS2))) .
```

Maude `search` command explores the tree of possible rewrites starting at an initial state `init` to a final state that matches pattern `(purse[P1] : PS1) (purse[P2] : PS2) S` and satisfies the condition denoted by the term after `such that`. In the above command, `MONDEX` is a Maude module that describes the OTS of the Mondex system (in which equation defining `init` and those rewrite rules are defined). `P1` and `P2` are variables of sort `Name` denoting two arbitrary purses. `S` is a variable of sort `Sys` denoting the remaining terms of an arbitrary state of the OTS. Note that in the condition part, we use the negation operator `not` in front of the term denoting property 1 since we aim at falsification of the property.

Setting `bound` to 9, and considering two purses `p1` and `p2` in the initial state `init`, we feed the above `search` command into Maude system, and `No Solution` is returned, which denotes that no counterexample is found.

We now give a simple example showing that the falsification can help filter out a lemma generated during interactive verification of the OTS/CafeOBJ method. The lemma named here as property 7 is as follows:

7. For any reachable state s, any two purses denoted by p_1 and p_2:

   `pay(purse(`s`,`p_1`)) = none and from(pay(purse(`s`,`p_2`))) =` p_1 `and not(`$p_1 = p_2$`)`
   `implies`
   `fromno(pay(purse(`s`,`p_2`))) = seq(purse(`s`,`p_1`))`.

Intuitively property 7 says that: if a purse `p1`'s payment detail is `none`, and the `from` component of the payment detail of another purse `p2` is equal to `p1`, then the `fromno` component of the payment detail of `p2` is equal to `p1`'s current sequence number. This seems to be reasonable since when `p1`'s payment detail is `none`, it means that `p1` has never involved in any transactions. And thus `p1`'s sequence number is never increased. The property describes the situation that two purses `p1` and `p2` are going to have a transaction, and `p2` has received the `startto` message, but `p1` has not received the `startfrom` message.

However, property 7 is actually incorrect because that even if `p1`'s payment detail is `none`, it can execute the `abort` transition freely before it receives the `startfrom` message since no condition is defined for `abort`. Therefore, `p1`'s sequence number can be increased. A correct conclusion of property 7 should be `fromno(pay(purse(`s`,`p_2`))) <= seq(purse(`s`,`p_1`))`.

To realize this incorrectness of property 7 by using the interactive verification of the OTS/CafeOBJ method, a certain amount of proof effort is needed, however, the incorrectness can be immediately reported by Maude system as a counterexample as follows:

```
state 0: ...
       ===[ crl ... [label startpay_p1_p2_con] ]===>
state 1: ...
       ===[ crl ... [label recstartto_p2] ]===>
state 8: ...
       ===[ crl ... [label abort_p1] ]===>
state 51: ...
```

where `state 0` denotes the initial state and `state 51` denotes the state (reached from `state 0` by applying rewrite rules) where a counterexample is found. The omitted parts after each numbered states are terms denoting corresponding states, and the omitted parts after `crl` are terms denoting the rewrite rules with corresponding labels. For example, the label `startpay_p1_p2_con` denotes that two purses `p1` and `p2` are going to do a transaction with value `con` (a declared Maude constant). In state 51, the `fromno` component of `p2` is `is(p1,seedn)`, but the sequence number of `p1` is `nextseqnum(is(p1, seedn))`, which is contrary to property 7.

6.3 Some Further Issues about Verification and Falsification

A possible question that one may ask about the above introduced falsification method is that: what if the depth of an existing counterexample of a predicate is deeper than the predetermined bound, namely that the counterexample cannot be found within the bounded reachable state space? Trivially increasing the value of bound may not work due to the state-explosion problem.

We have proposed a procedure called Induction-Guided Falsification (IGF) [20] to solve this problem. Assume a state predicate p to be proved invariant wrt an OTS, which however, has a counterexample of depth $n + m$, the procedure IGF first employs Maude `search` command (or Maude model checker) to explore a bounded, say n, reachable state space of the OTS for a counterexample. If no counterexample is found, IGF employs (structural) induction of the OTS/CafeOBJ method to try to verify p, during which some other state predicates called *necessary lemmas* may be obtained. When the CafeOBJ reduction result is `false` for a sub-case of an inductive case of p in a proof passage, a necessary lemma can be constructed by negating the conjunction of all the equations characterizing the sub-case. The necessary lemma can be used to discharge this `false` case, and its basic idea is that the sub-case may not be possible, or in other words, the states characterized by the sub-case may not be reachable.

Two important features of necessary lemmas are briefly that: (1) If p has a counterexample of length $n + 1$, then one of its necessary lemmas has a counterexample of length n, and (2) if a necessary lemma has counterexamples, then p also has counterexamples. Based on these two features of necessary lemmas, IGF repeats induction and searching counterexamples for *each* of these state predicates (p and the recursively constructed necessary lemmas) until a counterexample is found or p is proved.

The procedure IGF can be used very systematically. An algorithm for IGF has been described in [20], and we have also investigated several issues related to automating IGF in [21]. One limitation of IGF is that it is not suitable for proving a state predicate is invariant, since necessary lemmas used to discharge `false` cases are the weakest state predicates to strengthen inductive hypothesis and may not be appropriate ones.

We now briefly discuss a simple idea for systematically generating candidate lemmas (such as those of properties 3, 4 and 5 in Sect. 5.2), in which falsification can be very useful. A sub-case of an inductive case is characterized by a set of equations, say E. When CafeOBJ system reduces to `false` for this sub-case, a necessary lemma in the form $\neg(\bigwedge_{e \in E} e)$ can be

constructed. Note that from this set E of equations, we can also systematically construct other state predicates in the form $\neg(\bigwedge_{e' \in E'} e')$, where $E' \in 2^E \wedge E' \neq E$, and these state predicates are stronger than the necessary lemma since $\neg(\bigwedge_{e' \in E'} e') \Rightarrow \neg(\bigwedge_{e \in E} e)$. Basically, all of these state predicates are candidates that can be used, instead of the necessary lemma and maybe more appropriately, to strengthen inductive hypothesis. Falsification can be used here to filter out those incorrect candidates. This idea is part of a procedure called Combined Falsification and Verification (CFV) described in [22].

7 Related Work

The Mondex system has been originally specified and manually proved for correctness using the Z methods [5]. In [5], two models of the Mondex system are developed, where the first is an abstract model that models value exchanges between purses as atomic transaction, and the second is a concrete model that models value exchanges between purses following the communication protocol. It is then proved that the two security properties hold for the abstract model, and the concrete model is a refinement of the abstract one (actually an intermediate model is introduced to ease the proof).

Following the original Z work, a number of other formal methods, such as KIV [6, 7], RAISE [8] and Alloy [9] etc, have been employed to the Mondex problem. We discuss these related work wrt the aspects of modeling, refinement proof (or verification) and falsification, respectively.

The RAISE and Alloy work seem to intentionally follow closely the modeling methods of the original Z work while keeping their own features. The KIV work provides an alternative operational style formalization of the Mondex system using (two) abstract state machines, and makes several simplifications and modifications, which include, for example: removed the global input while obtaining the input from the ether; removed the `ignore` operation that does nothing (which is needed by the refinement theory used in Z work); merged the purses' two idling status `eaFrom` and `eaTo` into one `idle` status, etc.

Our work of modeling the Mondex system as an OTS in an operational style is inspired by the KIV work, which from our point of view is simpler to the Z modeling method (similar statement is made in the KIV work). In addition, we made several further modifications to the KIV modeling as follows:

1. Since messages existing in ether can be lost, we abandoned the assumption made in KIV modeling that `startfrom` and `startto` messages are always available in ether. In our modeling, no message exists in the initial ether.

2. To reflect that a purse can abort a transaction at any time as the card-holder wishes or due to purses' internal reasons, we did not define any effective condition for the transition `abort`, while a condition was defined for `abort` in KIV modeling.

3. We explicitly defined two transitions `duplicate` and `drop` to characterize that messages in the ether can be replayed and lost. The KIV modeling used `ether'` $\subseteq$ `ether` to characterize that messages can be lost, but did not explicitly show that messages can be replayed.

To show the correctness of the properties to the Mondex system, refinement proofs are developed in Z, KIV, RAISE and Alloy work in different forms and with different features. Although the refinement construction and proof strategy is reasonable and suitable for the Mondex problem, we employ an alternative way of expressing and verifying the security properties of the Mondex system directly as invariants of an OTS through using an introduced component "previous balance" of purses. Note, however, that even if different proof strategies are used, we share

some similar or exactly same proof obligations. First, for the property of payment details that `from` and `to` components should be different (Sect. 4.3.2 of [5]), and for the properties P-2 to P-4 for purses (Sect. 4.6 of [5]), which are used in the refinement proofs of the Z and KIV work, we have proved and used as lemmas exactly same properties in our verification; and second, for some of the properties B-2 to B-12 expressing constraints on ether (Sect. 5.3 of [5]), we have proved and used as lemmas very similar properties.

In the RAISE and Alloy work, two different ways of falsification of the Mondex system are described by means of translating the RSL (RAISE Specification Language) specification of the Mondex system into the input of SAL model checker, and respectively, appealing the Alloy analyzer (model-finding technique). In our work, Maude `search` command is used for conducting falsification through a translation into Maude specification of the Mondex system. Our work is similar to the above two work in the sense that we all consider a finite reachable state space (called finite scope in Alloy terminology), such as finite number of purses. However, our work is different with the RAISE work in the sense that we do not need to make those changes of the Mondex system as RAISE work did: (1) the possible loss of messages was not modeled in RAISE work to reduce possible changes to the ether, and (2) ranges of money and sequence numbers were restricted to 0..3, etc. One possible reason for these may be that we are able to do falsification on inductively defined data types. For example, `Ether` is defined using data constructors `nil` and `_,_`. This point is also a possible difference between our work and the Alloy work.

8 Conclusion

We have described two algebraic approaches to both verification and falsification of the Mondex system, and how the falsification can be used to facilitate the verification. We have employed alternative ways of (1) modeling the Mondex system in an operational style, rather than in a relational style, and (2) expressing and verifying (and falsifying) security properties of the Mondex system directly in terms of invariants. This work therefore provides a different way of viewing the Mondex analysis problem and can be used to compare different modeling and proof strategies. In addition, our model of the Mondex system makes several simplifications to the original Z model (as inspired by the KIV model), and several further modifications to the KIV model to keep closer to the real problem.

In our modeling and verification of the Mondex system, we did not consider intruder purses that may send faked *Req*, *Val* and *Ack* messages based on possibly gleaned information. This is because that it is assumed that those messages cannot be forged, which is guaranteed by some (unclear) means of cryptographic system. In the KIV work, a possible communication protocol using cryptographic algorithm is developed. Our first future work is to extend our modeling and verification by considering possible intruder purses under a cryptographically secured communication protocol, in which it should be proved that the three messages cannot be forged rather than assuming it.

Our second future work relates to falsification. We are going to investigate the technical issue that how many entities (such as purses) are enough to uncover possible counterexamples when the number of the entities has to be made finite for falsification. Furthermore, we are also going to extensively investigate ways of utilizing falsification to facilitate verification of the OTS/CafeOBJ method, and implement tools that automate the procedure IGF, and possibly also the procedure CFV.

Acknowledgements

This research is conducted as a program for the "21st Century COE Program" in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology. We would like to thank Chris George and Anne E. Haxthausen for kindly sharing their RSL specification of the Mondex problem with us.

References

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

[2] Mondex Case Study. URL: http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy.

[3] UK Computing Research Committee, Grand Challenges in Computer Research. URL: http://www.ukcrc.org.uk/grand_challenges/index.cfm

[4] J. Woodcock, Grand Challenges 6: Dependable Systems Evolution. URL: http://www.fmnet.info/gc6/.

[5] S. Stepney, D. Cooper and J. Woodcock. An electronic purse specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July, 2000.

[6] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex challenge: machine checked proofs for an electronic purse. Technical Report, University of Augsburg, 2006.

[7] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex challenge: machine checked proofs for an electronic purse. In: FM 2006, LNCS Vol.4085, Springer (2006), 16-31

[8] A. Haxthausen, C. George, and M. Schütz. Specification and proof of the Mondex electronic purse. In: AWCVS'06, UNU-IIST Report No. 347, 2006, 209-224

[9] T. Ramananandro. Mondex, An Electronic Purse: Specification and refinement checks with the Alloy model-finding method. Internship Report, 2006. http://www.eleves.ens.fr/home/ramanana/work/mondex/.

[10] K. Ogata and K. Futatsugi: Proof scores in the OTS/CafeOBJ method. In: FMOODS 2003, LNCS Vol.2884, Springer (2003), 170-184

[11] CafeOBJ Web Site. URL: http://www.ldl.jaist.ac.jp/cafeobj/, 2007

[12] R. Diaconescu and K. Futatsugi: CafeOBJ report. AMAST Series in computing, 6. World Scientific, Singapore, 1998.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (Version 2.2). URL: http://maude.cs.uiuc.edu/maude2-manual/, 2007

[14] M. Nakamura, W. Kong, K. Ogata and K. Futatsugi. A complete specification translation from OTS/CafeOBJ into OTS/Maude. IEICE Technical Report, SS2006, 13, 2006, 1-6

[15] W. Kong, K. Ogata, and K. Futatsugi. A lightweight integration of theorem proving and model checking for system verification. In: APSEC'05, IEEE CS, 2005, 59-66

[16] J. Hsiang and N. Dershowitz. Rewrite methods for clausal and nonclausal theorem proving. In ICALP 1983, LNCS Vol.154, Springer (1983), 331-346

[17] K. Ogata and K. Futatsugi. Proof score approach to verification of liveness properties. In SEKE 2005, 608-613

[18] K. Ogata, W. Kong and K. Futatsugi. Falsification of OTSs by searches of bounded reachable state spaces. In SEKE 2006, 440-445

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu. Bounded Model Checking. Advances in Computers, Vol. 58, Academic Press, 2003.

[20] K. Ogata, M. Nakano, W. Kong and K. Futatsugi. Induction-Guided Falsification. In ICFEM 2006, LNCS Vol. 4260, Springer (2006), 114-131

[21] W. Kong, K. Ogata and K. Futatsugi. A review of Induction-Guided Falsification and towards its automation. In AWCVS 2006, UNU-IIST Technical Report 347, 2006, 48-59

[22] W. Kong. Facilitating inductive verification with counterexample discovery capability. PhD thesis, JAIST, 2006.

A Mondex Specifications including Data Types

```
-- Bal, a subsort of Int, is used to represent balances of purses
mod! BAL {
  pr(INT)
  [Bal < Int]
  op seedv : -> Bal
  op _=_ : Bal Bal -> Bool {comm}
  --
  vars I J M N : Bal
  eq (I = I) = true .
  eq (I >= 0) = true .
  ceq (I <= J) = true if (I = J) .
  eq (I - I) = 0 .
  ceq ((I + J) <= (M + N)) = true if ((I <= M) and (J <= N)) .
  --
  op _=_ : Int Int -> Bool {comm}
  vars X Y P Q : Int
  eq (X = X) = true .
  ceq (X <= Y) = true if (X = Y) .
  ceq ((X + P) <= (Y + Q)) = true if ((X <= Y) and (P <= Q)) .
  eq (X = X - I) = false .
  ceq ((X - P) <= Y) = true if (X <= Y) .
}

-- Name is used to represent the name of purses
mod! NAME Name {
  [Name]
  op _=_ : Name Name -> Bool {comm}
  var N : Name
  eq (N = N) = true .
}

-- Ibal is used to generate the initial balance of purses.
mod! IBAL {
  pr(NAME + BAL)
```

```
  [Ibal < Bal]
  op ib : Name Bal -> Ibal
  op nofibal : Ibal -> Name
  op valofibal : Ibal -> Bal
  op _=_ : Ibal Ibal -> Bool {comm}
  --
  var N : Name                          var P : Bal
  vars I I1 I2 : Ibal
  eq nofibal(ib(N,P)) = N .
  eq valofibal(ib(N,P)) = P .
  eq (I = I) = true .
  eq (I1 = I2) = (nofibal(I1) = nofibal(I2) and valofibal(I1) = valofibal(I2)) .
}

-- Seqnum is used to represent the sequence number to be used by purses
-- in next transaction.
mod! SEQNUM {
  pr(NAT)
  [Seqnum < Nat]
  op seedn : -> Seqnum
  op _=_ : Seqnum Seqnum -> Bool {comm}
  op nextseqnum : Seqnum -> Seqnum
  --
  vars S S1 : Seqnum
  eq (S = S) = true .
  eq (S = nextseqnum(S)) = false .
  eq (S < nextseqnum(S)) = true .
  --
  eq (S < S) = false .
  ceq (S < nextseqnum(S1)) = true if (S < S1) .
  --
  ceq (S < nextseqnum(S1)) = true if (S <= S1) .
  ceq (S = nextseqnum(S1)) = false if (S <= S1) .
  eq (nextseqnum(S) <= S) = false .
}

-- Inum is used to generate the initial seqnum of purses.
mod! INUM {
  pr(NAME + SEQNUM)
  [Inum < Seqnum]
  op is : Name Seqnum -> Inum
  op nofinum : Inum -> Name
  op sofinum : Inum -> Seqnum
  op _=_ : Inum Inum -> Bool {comm}

  vars I I1 I2 : Inum
  var N : Name                          var S : Seqnum
  eq nofinum(is(N,S)) = N .             eq sofinum(is(N,S)) = S .
  eq (I = I) = true .
  eq (I1 = I2) = (nofinum(I1) = nofinum(I2) and sofinum(I1) = sofinum(I2)) .
}

-- Status is used to represent the status of purses
mod! STATUS {
  [Status]
  ops idle epr epv epa : -> Status
  op _=_ : Status Status -> Bool {comm}
  var PS : Status
  eq (PS = PS) = true .
  eq (idle = epr) = false .             eq (idle = epv) = false .
  eq (idle = epa) = false .             eq (epr = epv) = false .
```

```
  eq (epr = epa) = false .                eq (epv = epa) = false .
}

-- Paydetail is used to represent the Paydetail of purses
mod! PAYDETAIL {
  pr(BAL + NAME + SEQNUM)
  [Emptypd < Paydetail]
  op none : -> Emptypd
  op _=_ : Paydetail Paydetail -> Bool {comm}
  op mk-pay : Name Seqnum Name Seqnum Bal -> Paydetail
  op from   : Paydetail -> Name
  op fromno : Paydetail -> Seqnum
  op to     : Paydetail -> Name
  op tono   : Paydetail -> Seqnum
  op value  : Paydetail -> Bal
  --
  vars F T : Name                          vars FN TN : Seqnum
  vars V : Bal                             var PD : Paydetail
  eq from(mk-pay(F,FN,T,TN,V)) = F .
  eq fromno(mk-pay(F,FN,T,TN,V)) = FN .
  eq to(mk-pay(F,FN,T,TN,V)) = T .
  eq tono(mk-pay(F,FN,T,TN,V)) = TN .
  eq value(mk-pay(F,FN,T,TN,V)) = V .
  --
  eq (PD = PD) = true .
  eq (mk-pay(F,FN,T,TN,V) = none) = false .
}

-- Exlog is used to represent the exception log of purses,
-- Exlog is essentially is list of paydetails.
mod! EXLOG {
  pr(PAYDETAIL + INT)
  [Paydetail < Exlog]
  op emptyexlog : -> Exlog
  op _@_ : Exlog Exlog -> Exlog
  op _/inexlog_ : Paydetail Exlog -> Bool
  --
  vars EXLOG E1 E2 : Exlog
  vars PD PD1 PD2 : Paydetail
  eq PD1 /inexlog emptyexlog = false .
  ceq PD1 /inexlog (PD2 @ EXLOG) = true if (PD1 = PD2) .
  ceq PD1 /inexlog (PD2 @ EXLOG) = PD2 /inexlog EXLOG if not(PD1 = PD2) .
  --
  op _=_ : Exlog Exlog -> Bool
  eq (EXLOG = EXLOG) = true .
  ceq ((PD1 @ E1) = (PD2 @ E2)) = false if not(PD1 = PD2) .
  ceq ((PD1 @ E1) = (PD2 @ E2)) = (E1 = E2) if (PD1 = PD2) .
  eq ((PD @ E1) = emptyexlog) = false .
  --
  op lost : Paydetail Exlog -> Int
  eq lost(PD,EXLOG) = (if (PD /inexlog EXLOG) then value(PD) else 0 fi) .
}

-- Purse is used to represent the purses themselves.
mod! PURSE {
  pr(STATUS + EXLOG + INT)
  [Purse]
  op mk-purse : Name Int Int Seqnum Status Paydetail Exlog -> Purse
  op _=_ : Purse Purse -> Bool {comm}
  op name : Purse -> Name
  op pbal : Purse -> Int
```

```
  op bal  : Purse -> Int
  op seq  : Purse -> Seqnum
  op sta  : Purse -> Status
  op pay  : Purse -> Paydetail
  op log  : Purse -> Exlog
  --
  var P : Purse
  var N : Name                          vars V1 V2 : Int
  var S : Seqnum                        var ST : Status
  var PD : Paydetail                    var EXLOG : Exlog
  --
  eq name(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = N .
  eq pbal(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = V1 .
  eq bal(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = V2 .
  eq seq(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = S .
  eq sta(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = ST .
  eq pay(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = PD .
  eq log(mk-purse(N,V1,V2,S,ST,PD,EXLOG)) = EXLOG .
  --
  eq (P = P) = true .
}

-- Message is used to represent messages transferred
mod! MESSAGE {
  pr(PAYDETAIL)
  [Message]
  op startfrom : Name Bal Seqnum -> Message
  op startto : Name Bal Seqnum -> Message
  op req : Paydetail -> Message
  op val : Paydetail -> Message
  op ack : Paydetail -> Message
  --
  op isstartfrom : Message -> Bool
  op isstartto : Message -> Bool
  op isreq : Message -> Bool
  op isval : Message -> Bool
  op isack : Message -> Bool
  --
  op nameofm : Message -> Name
  op valueofm : Message -> Bal
  op seqofm : Message -> Seqnum
  op pdofm : Message -> Paydetail
  --
  op _=_ : Message Message -> Bool {comm}
  var N : Name                          var I : Bal
  var S : Seqnum                        var PD : Paydetail
  vars M M1 M2 : Message
  --
  eq isstartfrom(startfrom(N,I,S)) = true .
  eq isstartfrom(startto(N,I,S)) = false .
  eq isstartfrom(req(PD)) = false .
  eq isstartfrom(val(PD)) = false .
  eq isstartfrom(ack(PD)) = false .
  --
  eq isstartto(startto(N,I,S)) = true .
  eq isstartto(startfrom(N,I,S)) = false .
  eq isstartto(req(PD)) = false .
  eq isstartto(val(PD)) = false .
  eq isstartto(ack(PD)) = false .
  --
  eq isreq(req(PD)) = true .
```

```
  eq isreq(startfrom(N,I,S)) = false .
  eq isreq(startto(N,I,S)) = false .
  eq isreq(val(PD)) = false .
  eq isreq(ack(PD)) = false .
  --
  eq isval(val(PD)) = true .
  eq isval(startfrom(N,I,S)) = false .
  eq isval(startto(N,I,S)) = false .
  eq isval(req(PD)) = false .
  eq isval(ack(PD)) = false .
  --
  eq isack(ack(PD)) = true .
  eq isack(startfrom(N,I,S)) = false .
  eq isack(startto(N,I,S)) = false .
  eq isack(req(PD)) = false .
  eq isack(val(PD)) = false .
  --
  eq nameofm(startfrom(N,I,S)) = N .
  eq nameofm(startto(N,I,S)) = N .
  eq valueofm(startfrom(N,I,S)) = I .
  eq valueofm(startto(N,I,S)) = I .
  eq seqofm(startfrom(N,I,S)) = S .
  eq seqofm(startto(N,I,S)) = S .
  eq pdofm(req(PD)) = PD .
  eq pdofm(val(PD)) = PD .
  eq pdofm(ack(PD)) = PD .
  --
  eq (pdofm(M) = none) = false .
  --
  eq (M = M) = true .
  ceq (M1 = M2) = (isstartfrom(M2) and nameofm(M1) = nameofm(M2) and
                   valueofm(M1) = valueofm(M2) and seqofm(M1) = seqofm(M2))
                  if isstartfrom(M1) .
  ceq (M1 = M2) = (isstartto(M2) and nameofm(M1) = nameofm(M2) and
                   valueofm(M1) = valueofm(M2) and seqofm(M1) = seqofm(M2))
                  if isstartto(M1) .
  ceq (M1 = M2) = (isreq(M2) and pdofm(M1) = pdofm(M2)) if isreq(M1) .
  ceq (M1 = M2) = (isval(M2) and pdofm(M1) = pdofm(M2)) if isval(M1) .
  ceq (M1 = M2) = (isack(M2) and pdofm(M1) = pdofm(M2)) if isack(M1) .
}

-- Ether contains messages, is used to represent the communication channel
mod! ETHER {
  -- The remaining parts are those introduced in Sect. 4.1.
}

-- The Mondex communication protocol
mod* MONDEX {
  pr(IBAL + INUM + PURSE + ETHER)
  *[Sys]*
  -- The remaining parts are those introduced in Sect. 4.2.
}
```

B Invariants Proved

Property 1 shown in the paper corresponds to inv100 and property 2 to inv440. Besides, property 3, 4, 5 and 6 corresponds to inv130, inv140, inv470 and inv330. These invariants are defined in module `INV`, and their definitions for basic formulas to be proved in inductives cases are defined in module `ISTEP`.

```
-- proved with inv110, inv120, inv130, inv140, and inv330
eq inv100(S,P1,P2) =
   ((sta(purse(S,P1)) = idle and sta(purse(S,P2)) = idle and
   pay(purse(S,P1)) = pay(purse(S,P2)) and not(P1 = P2))
   implies
   ((bal(purse(S,P1)) + bal(purse(S,P2)))
                    <= (pbal(purse(S,P1)) + pbal(purse(S,P2))))) .

-- proved with inv600, inv610, inv620, inv650, inv150, inv160, inv200,
-- inv170 and inv180
eq inv110(S,P1,P2,M) =
   ((M /in ether(S) and isval(M) and pay(purse(S,P1)) = pdofm(M) and
   sta(purse(S,P1)) = epv and pay(purse(S,P2)) = pdofm(M) and
   not(P1 = P2)) implies
   (bal(purse(S,P2)) = (pbal(purse(S,P2)) - value(pay(purse(S,P2)))))) .

-- proved by itself
eq inv120(S,P) =
   (sta(purse(S,P)) = epv implies (bal(purse(S,P)) = pbal(purse(S,P)))) .

-- proved with inv150
eq inv130(S,P) =
   ((sta(purse(S,P)) = epa)
   implies
   (bal(purse(S,P)) = pbal(purse(S,P)) - value(pay(purse(S,P))))) .

-- proved with inv260, inv120, inv160, inv200 and inv270
-- inv220, inv290, inv510, inv560, inv590
eq inv140(S,P1,P2,M) =
   ((M /in ether(S) and isack(M) and pay(purse(S,P2)) = pdofm(M) and
   sta(purse(S,P2)) = idle and sta(purse(S,P1)) = epa and
   pay(purse(S,P1)) = pdofm(M) and not(P1 = P2))
   implies
   (bal(purse(S,P2)) = (pbal(purse(S,P2)) + value(pay(purse(S,P2)))))) .

-- proved by itself
eq inv150(S,P) =
   ((sta(purse(S,P)) = epr)
   implies (bal(purse(S,P)) = pbal(purse(S,P)))) .

-- proved with inv210
eq inv160(S,P,P1,P2,PD) =
   ((pay(purse(S,P1)) = PD and pay(purse(S,P2)) = PD and not(PD = none)
   and not(P1 = P2) and not(P = P1) and not(P = P2))
   implies not(pay(purse(S,P)) = PD)) .

-- proved with inv190
eq inv170(S,P1,P2) =
   ((sta(purse(S,P1)) = epv and sta(purse(S,P2)) = epv and
   pay(purse(S,P1)) = pay(purse(S,P2))) implies (P1 = P2)) .

-- proved with inv150
eq inv180(S,P) =
   ((sta(purse(S,P)) = epa)
   implies
   (bal(purse(S,P)) = pbal(purse(S,P)) - value(pay(purse(S,P))))) .

-- proved by itself
eq inv190(S,P) =
   ((sta(purse(S,P)) = epv) implies (to(pay(purse(S,P))) = P)) .
```

```
-- proved by itself
eq inv200(S,P) =
   ((sta(purse(S,P)) = epv) implies not(pay(purse(S,P)) = none)) .

-- proved with inv220
eq inv210(S,P,P1,P2,PD) =
   ((not(P1 = P) and not(P2 = P) and pay(purse(S,P1)) = PD and
   pay(purse(S,P2)) = PD and not(P1 = P2) and not(PD = none))
   implies (not(from(PD) = P) and not(to(PD) = P))) .

-- proved by itself
eq inv220(S,P,PD) =
   ((pay(purse(S,P)) = PD and not(PD = none))
   implies (from(PD) = P or to(PD) = P)) .

-- proved by itself
eq inv230(S,P) =
   ((sta(purse(S,P)) = epr) implies (from(pay(purse(S,P))) = P)) .

-- proved with inv220 and inv230
eq inv240(S,P1,P2) =
   ((sta(purse(S,P1)) = epr and pay(purse(S,P1)) = pay(purse(S,P2))
   and not(P1 = P2)) implies (to(pay(purse(S,P2))) = P2)) .

-- proved by itself
eq inv250(S,P) =
   ((sta(purse(S,P)) = epa or sta(purse(S,P)) = epr)
   implies not(pay(purse(S,P)) = none)) .

-- proved with inv280, inv120, inv160, inv200, inv230, and inv290
-- inv560, inv590
eq inv260(S,P1,P2,M) =
   (((M /in ether(S)) and isack(M) and pay(purse(S,P1)) = pdofm(M) and
   sta(purse(S,P1)) = idle and pay(purse(S,P2)) = pdofm(M) and
   not(P1 = P2) and sta(purse(S,P2)) = epr)
   implies (bal(purse(S,P1)) = (pbal(purse(S,P1)) + value(pdofm(M))))) .

-- proved with inv300
eq inv270(S,P1,P2) =
   ((sta(purse(S,P1)) = epa and sta(purse(S,P2)) = epa and
   pay(purse(S,P1)) = pay(purse(S,P2))) implies (P1 = P2)) .

-- proved with inv120, inv320 and inv290, inv560, inv220, inv590
eq inv280(S,P,M) =
   (((M /in ether(S)) and isack(M) and
   pay(purse(S,P)) = pdofm(M) and sta(purse(S,P)) = idle and
   not(P = from(pdofm(M))))
   implies (bal(purse(S,P)) = (pbal(purse(S,P)) + value(pdofm(M))))) .

-- proved with inv230
eq inv290(S,P) =
   ((sta(purse(S,P)) = epa) implies (from(pay(purse(S,P))) = P)) .

-- proved with inv290 and inv310
eq inv300(S,P1,P2) =
   ((sta(purse(S,P1)) = epa and sta(purse(S,P2)) = epr and
   pay(purse(S,P1)) = pay(purse(S,P2))) implies (P1 = P2)) .

-- proved with inv230
eq inv310(S,P1,P2) =
   ((sta(purse(S,P1)) = epr and sta(purse(S,P2)) = epr and
```

```
   pay(purse(S,P1)) = pay(purse(S,P2))) implies (P1 = P2)) .

-- proved with inv220 and inv190
eq inv320(S,P1,P2) =
   ((sta(purse(S,P1)) = epv and pay(purse(S,P1)) = pay(purse(S,P2))
   and not(P1 = P2)) implies (from(pay(purse(S,P2))) = P2)) .

-- deduced with inv340 and inv350
eq inv330(S,P1,P2) =
   ((pay(purse(S,P1)) = pay(purse(S,P2)) and not(P1 = P2))
   implies ((bal(purse(S,P1)) + bal(purse(S,P2)))
                    <= (pbal(purse(S,P1)) + pbal(purse(S,P2))))) .

-- proved by itself
eq inv340(S,P1,P2) =
   ((pay(purse(S,P1)) = pay(purse(S,P2)) and not(P1 = P2) and
   pay(purse(S,P1)) = none) implies
   ((bal(purse(S,P1)) + bal(purse(S,P2)))
                    <= (pbal(purse(S,P1)) + pbal(purse(S,P2))))) .

-- proved with inv360, inv370, inv110, inv120, inv630, inv580.
eq inv350(S,P1,P2) =
   ((pay(purse(S,P1)) = pay(purse(S,P2)) and not(P1 = P2) and
   not(pay(purse(S,P1)) = none)) implies
   ((bal(purse(S,P1)) + bal(purse(S,P2)))
                    <= (pbal(purse(S,P1)) + pbal(purse(S,P2))))) .

-- proved with inv230, inv420
eq inv360(S,P1,P2) =
   ((from(pay(purse(S,P2))) = P1 and pay(purse(S,P1)) = none and
   not(P1 = P2)) implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv430 and inv190.
eq inv370(S,P1,P2) =
   ((to(pay(purse(S,P2))) = P1 and pay(purse(S,P1)) = none and
   not(P1 = P2)) implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv390
eq inv380(S,P1,P2) =
   ((from(pay(purse(S,P2))) = P1 and not(pay(purse(S,P2)) = none))
   implies (fromno(pay(purse(S,P2))) <= seq(purse(S,P1)))) .

-- proved by itself
eq inv390(S,P,M) =
   ((M /in ether(S) and isstartto(M) and nameofm(M) = P)
   implies (seqofm(M) <= seq(purse(S,P)))) .

-- proved with inv410
eq inv400(S,P1,P2) =
   ((to(pay(purse(S,P2))) = P1 and not(pay(purse(S,P2)) = none))
   implies (tono(pay(purse(S,P2))) <= seq(purse(S,P1)))) .

-- proved by itself
eq inv410(S,P,M) =
   ((M /in ether(S) and isstartfrom(M) and nameofm(M) = P)
   implies (seqofm(M) <= seq(purse(S,P)))) .

-- proved with inv230, inv250
eq inv420(S,P,M) =
   ((M /in ether(S) and isval(M) and from(pdofm(M)) = P)
   implies not(pay(purse(S,P)) = none)) .
```

```
-- proved by itself
eq inv430(S,P,M) =
   ((M /in ether(S) and isreq(M) and to(pdofm(M)) = P)
   implies not(pay(purse(S,P)) = none)) .

-- proved with inv110, inv120, inv130, inv140, inv150, inv470,
-- inv450, inv520, inv220, inv200, inv190, inv510, inv530, inv180,
-- inv540, inv550, inv560, inv570, inv480, inv250, inv290.
eq inv440(S,P1,P2) =
   ((sta(purse(S,P1)) = idle and sta(purse(S,P2)) = idle and
   pay(purse(S,P1)) = pay(purse(S,P2)) and not(P1 = P2))
   implies
   (if (pay(purse(S,P1)) /inexlog log(purse(S,P1))) and
       (pay(purse(S,P2)) /inexlog log(purse(S,P2)))
    then ((bal(purse(S,P1)) + bal(purse(S,P2)) +
           lost(pay(purse(S,P1)),log(purse(S,P1))))
                 = (pbal(purse(S,P1)) + pbal(purse(S,P2))))
    else ((bal(purse(S,P1)) + bal(purse(S,P2)))
                 = (pbal(purse(S,P1)) + pbal(purse(S,P2)))) fi)) .

-- proved with inv460
eq inv450(S,P) =
   (sta(purse(S,P)) = epv
   implies not(pay(purse(S,P)) /inexlog log(purse(S,P)))) .

-- proved with inv190, inv200, inv220, inv250, inv290, inv450, inv470,
-- inv490, inv500 and inv510.
eq inv460(S,P,PD) =
   ((PD /inexlog log(purse(S,P)))
   implies (if from(PD) = P then (fromno(PD) < seq(purse(S,P)))
                            else (tono(PD) < seq(purse(S,P))) fi)) .

-- proved with inv480
eq inv470(S,P) =
   (sta(purse(S,P)) = epa
   implies not(pay(purse(S,P)) /inexlog log(purse(S,P)))) .

-- proved with inv460
eq inv480(S,P) =
   (sta(purse(S,P)) = epr
   implies not(pay(purse(S,P)) /inexlog log(purse(S,P)))) .

-- proved by itself
eq inv490(S,P) =
   (sta(purse(S,P)) = epv
   implies tono(pay(purse(S,P))) < seq(purse(S,P))) .

-- proved with inv660
eq inv500(S,P) =
   (sta(purse(S,P)) = epa
   implies fromno(pay(purse(S,P))) < seq(purse(S,P))) .

-- proved by itself
eq inv510(S,P) =
   (not(pay(purse(S,P)) = none) implies
   not(from(pay(purse(S,P))) = to(pay(purse(S,P))))) .

-- proved with inv460, inv480, inv450, inv190, inv200, inv510 and inv130
eq inv520(S,P) =
   ((pay(purse(S,P)) /inexlog log(purse(S,P)) and
```

```
   from(pay(purse(S,P))) = P) implies
   (bal(purse(S,P)) = pbal(purse(S,P)) - value(pay(purse(S,P)))))) .

-- proved with inv580, inv170, inv590, inv560, inv150
eq inv530(S,P1,P2) =
   ((sta(purse(S,P2)) = idle and sta(purse(S,P1)) = epv and
   pay(purse(S,P1)) = pay(purse(S,P2)) and
   not(pay(purse(S,P2)) /inexlog log(purse(S,P2))) and not(P1 = P2))
   implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv480, inv450, inv120, inv290, inv510, inv250
eq inv540(S,P) =
   ((pay(purse(S,P)) /inexlog log(purse(S,P)) and
   to(pay(purse(S,P))) = P)
   implies (bal(purse(S,P)) = pbal(purse(S,P)))) .

-- proved with inv560, inv510, inv230, inv290, inv250, inv450, inv480, inv120
eq inv550(S,P) =
   ((sta(purse(S,P)) = idle and
   not(pay(purse(S,P)) /inexlog log(purse(S,P)))
   and to(pay(purse(S,P))) = P and not(pay(purse(S,P)) = none))
   implies
   (bal(purse(S,P)) = pbal(purse(S,P)) + value(pay(purse(S,P))))) .

-- proved by itself
eq inv560(S,P) =
   (sta(purse(S,P)) = idle or
   (if from(pay(purse(S,P))) = P
    then (sta(purse(S,P)) = epr or sta(purse(S,P)) = epa)
    else sta(purse(S,P)) = epv fi)) .

-- proved with inv630, inv310, and inv640
eq inv570(S,P1,P2) =
   ((sta(purse(S,P1)) = epr and pay(purse(S,P1)) = pay(purse(S,P2))
   and not(P1 = P2)) implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv400, inv610 and inv620
eq inv580(S,P1,P2) =
   ((to(pay(purse(S,P2))) = P1 and
   tono(pay(purse(S,P2))) = seq(purse(S,P1))
   and not(P1 = P2) and not(pay(purse(S,P2)) = none))
   implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv600 and inv170
eq inv590(S,P,M) =
   ((M /in ether(S) and isack(M) and pay(purse(S,P)) = pdofm(M))
   implies (not(sta(purse(S,P)) = epv))) .

-- proved with inv610 and inv190
eq inv600(S,P,M) =
   ((M /in ether(S) and isack(M) and to(pdofm(M)) = P)
   implies tono(pdofm(M)) < seq(purse(S,P))) .

-- proved with inv250, inv230, inv510, inv620
eq inv610(S,P,M) =
   ((M /in ether(S) and isval(M) and to(pdofm(M)) = P)
   implies tono(pdofm(M)) < seq(purse(S,P))) .

-- proved by itself
eq inv620(S,P,M) =
   ((M /in ether(S) and isreq(M) and to(pdofm(M)) = P)
```

```
   implies tono(pdofm(M)) < seq(purse(S,P))) .

-- proved with inv380, inv650 and inv230
eq inv630(S,P1,P2) =
   ((from(pay(purse(S,P2))) = P1 and
   fromno(pay(purse(S,P2))) = seq(purse(S,P1))
   and not(P1 = P2) and not(pay(purse(S,P2)) = none))
   implies (bal(purse(S,P2)) = pbal(purse(S,P2)))) .

-- proved with inv650 and inv310
eq inv640(S,P,M) =
   ((M /in ether(S) and isval(M) and pay(purse(S,P)) = pdofm(M))
   implies not(sta(purse(S,P)) = epr)) .

-- proved with inv230 and inv660
eq inv650(S,P,M) =
   ((M /in ether(S) and isval(M) and from(pdofm(M)) = P)
   implies fromno(pdofm(M)) < seq(purse(S,P))) .

-- proved by itself
eq inv660(S,P) =
   (sta(purse(S,P)) = epr
   implies fromno(pay(purse(S,P))) < seq(purse(S,P))) .
```

C A Sample Proof Score

To give an impression of the proof using proof score technique, we show the whole proof score for inv120 that is proved without using other invariants. Each proof passage in the proof score is labeled with `[x.y].z`, where `x` is the number of the property, `y` is the number of an inductive case, and `z` is a list of bit numbers separated with dot "." denoting sub-cases (`1` denotes a predicate is `true`, and `0` denotes a predicate is `false`).

```
-- eq inv120(S,P) =
--    (sta(purse(S,P)) = epv implies (bal(purse(S,P)) = pbal(purse(S,P)))) .

--> I) Base case
--> [120.0] init
open INV
  red inv120(init,p) .
close

-- II) Inductive cases
--> [120.1] startpay
--> [120.1].1 c-startpay(s,q1,q2,v) = true .
open ISTEP
-- arbitrary objects
  ops q1 q2 : -> Name .
  op v : -> Bal .
-- assumption
  eq c-startpay(s,q1,q2,v) = true .
  --
-- successor state
  eq s' = startpay(s,q1,q2,v) .
-- check if the predicate is true.
  red istep120(p) .
close

--> [120.1].0 c-startpay(s,q1,q2,v) = false .
open ISTEP
```

```
-- arbitrary objects
  ops q1 q2 : -> Name .
  op v : -> Bal .
-- assumption
  eq c-startpay(s,q1,q2,v) = false .
  --
-- successor state
  eq s' = startpay(s,q1,q2,v) .
-- check if the predicate is true.
  red istep120(p) .
close

--> [120.2] recstartfrom
--> [120.2].1 c-recstartfrom(s,q,m) = true .
--> [120.2].1.1 p = q .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recstartfrom(s,q,m) = true .
  --
  eq p = q .
-- successor state
  eq s' = recstartfrom(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.2].1.0 (p = q) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recstartfrom(s,q,m) = true .
  --
  eq (p = q) = false .
-- successor state
  eq s' = recstartfrom(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.2].0 c-recstartfrom(s,q,m) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recstartfrom(s,q,m) = false .
  --
-- successor state
  eq s' = recstartfrom(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.3] recstartto
--> [120.3].1 c-recstartto(s,q,m) = true .
--> [120.3].1.1 p = q .
```

```
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recstartto(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isstartto(m) = true .
  eq sta(purse(s,q)) = idle .
  eq (q = nameofm(m)) = false .
  --
  eq p = q .
-- successor state
  eq s' = recstartto(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.3].1.0 (p = q) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recstartto(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isstartto(m) = true .
  eq sta(purse(s,q)) = idle .
  eq (q = nameofm(m)) = false .
  --
  eq (p = q) = false .
-- successor state
  eq s' = recstartto(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.3].0 c-recstartto(s,q,m) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recstartto(s,q,m) = false .
  --
-- successor state
  eq s' = recstartto(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.4] recreq
--> [120.4].1 c-recreq(s,q,m) = true .
--> [120.4].1.1 p = q .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recreq(s,q,m) = true .
  eq (m /in ether(s)) = true .
```

```
  eq isreq(m) = true .
  eq sta(purse(s,q)) = epr .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq p = q .
-- successor state
  eq s' = recreq(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.4].1.0 (p = q) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recreq(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isreq(m) = true .
  eq sta(purse(s,q)) = epr .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq (p = q) = false .
-- successor state
  eq s' = recreq(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.4].0 c-recreq(s,q,m) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recreq(s,q,m) = false .
  --
-- successor state
  eq s' = recreq(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.5] recval
--> [120.5].1 c-recval(s,q,m) = true .
--> [120.5].1.1 p = q .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recval(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isval(m) = true .
  eq sta(purse(s,q)) = epv .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq p = q .
-- successor state
  eq s' = recval(s,q,m) .
```

```
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.5].1.0 (p = q) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recval(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isval(m) = true .
  eq sta(purse(s,q)) = epv .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq (p = q) = false .
-- successor state
  eq s' = recval(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.5].0 c-recval(s,q,m) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recval(s,q,m) = false .
  --
-- successor state
  eq s' = recval(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.6] recack
--> [120.6].1 c-recack(s,q,m) = true .
--> [120.6].1.1 p = q .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recack(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isack(m) = true .
  eq sta(purse(s,q)) = epa .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq p = q .
-- successor state
  eq s' = recack(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.6].1.0 (p = q) = false .
open ISTEP
-- arbitrary objects
```

```
  op q : -> Name .
  op m : -> Message .
-- assumption
  -- eq c-recack(s,q,m) = true .
  eq (m /in ether(s)) = true .
  eq isack(m) = true .
  eq sta(purse(s,q)) = epa .
  eq pay(purse(s,q)) = pdofm(m) .
  --
  eq (p = q) = false .
-- successor state
  eq s' = recack(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.6].0 c-recack(s,q,m) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
  op m : -> Message .
-- assumption
  eq c-recack(s,q,m) = false .
  --
-- successor state
  eq s' = recack(s,q,m) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.7] drop
--> [120.7].1 c-drop(s) = true .
open ISTEP
-- arbitrary objects

-- assumption
  eq c-drop(s) = true .
  --
-- successor state
  eq s' = drop(s) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.7].0 c-drop(s) = false .
open ISTEP
-- arbitrary objects
-- assumption
  eq c-drop(s) = false .
  --
-- successor state
  eq s' = drop(s) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.8] duplicate
--> [120.8].1 c-duplicate(s) = true .
open ISTEP
-- arbitrary objects
-- assumption
```

```
  eq c-duplicate(s) = true .
  --
-- successor state
  eq s' = duplicate(s) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.8].0 c-duplicate(s) = false .
open ISTEP
-- arbitrary objects
-- assumption
  eq c-duplicate(s) = false .
  --
-- successor state
  eq s' = duplicate(s) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.9] abort
--> [120.9].1 p = q .
open ISTEP
-- arbitrary objects
  op q : -> Name .
-- assumption
  eq p = q .
-- successor state
  eq s' = abort(s,q) .
  -- check if the predicate is true.
  red istep120(p) .
close

--> [120.9].0 (p = q) = false .
open ISTEP
-- arbitrary objects
  op q : -> Name .
-- assumption
  eq (p = q) = false .
-- successor state
  eq s' = abort(s,q) .
  -- check if the predicate is true.
  red istep120(p) .
close

-- Q.E.D --
```