CSC2014 Coursework
Electronic Purse Model
2018

Issued: 19 February 2018

Due: 20:00 (8pm), Thursday 15th March 2018

Submission: via Ness

1 Aims

This coursework develops and assesses your skills in modelling using VDM-SL. It provides practice with reading and understanding a specification and with using a variety of modelling techniques and abstractions. The coursework is marked out of 16: it is worth 16% of the module mark for CSC2021.

2 Submission

Submit to Ness your final vdmsl specification file containing your final model.

- Name your file "mondexNNNNNNNN.vdmsl" where NNNNNNNN is replaced by your student number (i.e. if your student number is 12345678 your file will be "mondex12345678.vdmsl").
- Include in a comment at the head of the file with your name and student number, for example

```
-- CSC2021 Coursework
-- Name: .....
-- StdNo: .....
```

3 Scenario

The coursework concerns a model of a "cashless card" system (known as Mondex), which is to be trialled by a bank[1]. This was a real system in 1996 that was the grandfather of chip-and-pin. The complete Mondex specification[2] is about 250 pages long!

The bank intends to issue all of its customers with smart **purses,** which have a unique identifier and record the current balance on the card. Money is transferred between cards through use of a terminal known as a **wallet**.

[1] This scenario is an adaptation of the Mondex card system, as described in: Jones, CB and Pierce, KG. *What Can the pi-calculus Tell Us About the Mondex Purse System?* University of Newcastle upon Tyne, Computing Science Technical Report Series, No. CS-TR-1185, 2010.

[2] The complete specification is part of an Oxford University technical report PRG126, and various papers have been published on it on the Formal Aspects of Computing Journal (Volume 20:1 2008).

There are three key security properties to be preserved in this system:

1. All value is accounted for – the total sum of money in the system does not change
2. Authentic purses – only authorised purses can participate in transactions
3. Sufficient funds – purses can only transfer up to the current balance

There are also key environmental assumptions about the system:

1. Mondex is a closed system; so once purses are issued and some initial cash "created" there is no way of "creating" more money. This accounts for the first security property.
2. There is no security property about money being stolen: so if purse A user steals £10 from purse B, still the total sum of money remains constant
3. There are no central controls authorising transactions: once issued purses / wallets are independent entities.

The following is an initial model in VDM-SL for **purses**, **wallets** and the overall state (modelled as the **World**). Your task is to answer the questions given in order to develop the model further.

Initial model

A **Purse** holds a cash balance and has a unique key, which is used for authentication. The details of the authentication scheme are beyond the scope of this coursework. A purse that has been issued to a customer is publicly identified by a unique **PurseId**, which is used as the domain for a mapping to represent all purses.

The structure of **PurseId**, **PurseKey** is unimportant for the model and so these types are represented as tokens.

```
PurseId = token;
PurseKey = token;

Purse :: balance : nat
         key     : PurseKey;

PursesMap = map PurseId to Purse;
```

A **Wallet** is the means by which cards are authenticated and cash transferred between purses. Each wallet has a unique identifier, an authorisation value that is used to authorise each purse, and two slots in which to insert the cards. **WalletId** and **KeyAuth** are also represented as tokens.

```
WalletId = token;
KeyAuth = token;

Wallet :: id: WalletId
          fromSlot : [PurseId]
          toSlot : [PurseId]
          auth : KeyAuth;
```

The optional [PurseId] type is used since a **Wallet** will only have these slots filled if it is being used for a transaction. The authentication mechanism will use the **KeyAuth** and **PurseKey** types.

The overall system state is represented by the state **World,** which keeps track of: the total amount of cash in the system; the bank of purses that have not yet been issued to customers; the purses that have been issued and allocated a **PurseId**; and the collection of Wallets that are used to achieve transfer of cash between purses. Purses in the bank all have an initial balance of £100.

```
state World of
         totalCash : nat
         bank : set of Purse
         issuedPurses: PursesMap
         wallets: set of Wallet
```

Authentication is modelled as a function, very simply (i.e. ignored here and assumed to be done elsewhere):

```
Authenticate: KeyAuth * PurseKey -> bool
Authenticate(-, -) == true;
```

For the purposes of our model, authentication takes a wallet key and a purse key and always returns true. This abstracts away from any specific authentication scheme.

Some initial data has been defined for the model, in order to initialise the state. This defines **Wallets** (wa1, wa2, wa3); **PurseKeys** (pk1, pk2, pk3, pk4, pk5); and **PurseIds** (Purse1, Purse2). The state **init** clause uses these to set an initial total cash value of £500, a bank of unissued purses each holding £100, an empty map of issued purses, and three wallets.

Finally, the state invariant uses a number of auxiliary functions to model constraints on the model. These functions have not been written yet, but include functions to check uniqueness of purseKeys, walletIds, and to ensure the security properties have been accounted for.

```
state World of
         totalCash : nat
         bank : set of Purse
         issuedPurses: PursesMap
         wallets: set of Wallet
inv mk_World(totalCash,bank,issuedPurses,wallets) ==
             UniqueWalletData(wallets) and
             UniquePurseKeys(bank union rng issuedPurses) and
             NoPurseInTwoPlaces(bank, issuedPurses)  and
             AllValueAccountedFor(bank,issuedPurses,totalCash)
init w == w = mk_World(500,
                   {mk_Purse(100,pk) | pk in set {pk1,pk2,pk3,pk4,pk5}},
                   {|->},
                   {wa1,wa2,wa3})
end
```

The model as discussed so far can be found on Blackboard in the file **mondexCSC2021.vdmsl.**

It includes a **Run()** operation which can be used to test the model. There are no explicit marks for testing, but you may find it helpful to extend the Run operation to help you test your model.

4 Questions

The first questions require you to complete the initial model. Further questions deal with extensions to the model. **For all questions, ensure that you consider any constraints (i.e. invariants, preconditions) that may be required in the definitions**.

Begin by downloading the **mondexCSC2021.zip** file (from Blackboard) and import it into your workspace in the usual way (check the tutorials for a reminder on how to do this).

The state invariant includes several functions to ensure a consistent state. The following questions define these functions.

1. No two Wallets can have the same identifier (**id: WalletId**) or the same authorisation (**auth: KeyAuth**). The function `UniqueWalletData` takes a set of wallets and returns **true** if all wallets in the set have different ids, and all have different auths. This function has been partially defined in the specification:

   ```
   UniqueWalletData: set of Wallet -> bool
   UniqueWalletData(wallets) == is not yet specified;
   ```

   Complete the function definition (replacing "is not yet specified" with a predicate which evaluates to **true** when the wallets have unique identifiers and unique authorisations). [2 marks]

2. No two purses can have the same key (**key**: **PurseKey**). In the state invariant, the function `UniquePurseKeys` is used to check the whole collection of purses, in both the bank and the range of the purses mapping:

   ```
   UniquePurseKeys(bank union rng issuedPurses)
   ```

   The function `UniquePurseKeys` has been partially defined in the specification:

   ```
   UniquePurseKeys: set of Purse -> bool
   UniquePurseKeys(purseSet) == is not yet specified;
   ```

   Complete the function definition to check that all elements of `purseSet` have a different key. [2 marks]

3. A purse can be in only one of two places: either unissued (in the bank) or issued (in the issuedPurses mapping). The function `NoPurseInTwoPlaces` should check that there is no overlap between the two locations. It has been partially defined in the specification: complete the function definition. [2 marks]

4. The first security policy requires that *all value is accounted for*. This means that the sum of all money in the system – across all the purses – is constant. The field **totalCash** records what this sum should be. The function `AllValueAccountedFor` has already been defined in the specification. It compares the total cash across all the purses with the totalCash value, returning **true** if they have the same value. It uses an auxiliary function `SumBalances` which takes a set of purses and returns the total of their balances.

Complete the definition of the SumBalances function. Hint: consider using a recursive function definition. [4 marks]
If writing a recursive function, you can ignore any "recursive function has no measure" warning.

The remaining questions concern issuing purses and transferring money between purses.

The operation IssuePurse (defined in the model) allocates a given **pId**:**PurseId** to an arbitrary purse in the bank. This uses a **let** expression to define a new local variable p – defined as any element of the set bank – and then do two things: take the purse p out of the bank, and add it to the map of issued purses with the given identified **pId.** The keyword **atomic** is used to suspend invariant checking until both statements have been executed.

```
IssuePurse: PurseId ==> ()
IssuePurse (pId) ==
   let p in set bank in atomic (
            bank := bank \ {p};
            issuedPurses := issuedPurses munion {pId |-> p})
```

5. A newly allocated purse must be given a **PurseId** which does not already exist in the issuedPurses mapping. Record this restriction using a *precondition* on the **IssuePurse** operation, and consider any other restrictions which may be needed for this operation. [2 marks]

6. Money is transferred between purses using a Wallet. The wallet used must be one of the set of wallets in the World. The operation **SetupTransfer** sets up a transfer between two purses, each identified by their **PurseId**, and a selected wallet.

```
SetupTransfer: Wallet * PurseId * PurseId * nat ==> ()
SetupTransfer(wallet, fromId, toId, sum)==
      issuedPurses := Transfer(issuedPurses,
                          mk_Wallet(wallet.id,fromId,toId,wallet.auth),
                          sum)
pre wallet in set wallets
```

This operation is also provided in the initial specification file. Your task is to write the function **Transfer**, with signature

```
Transfer: PursesMap * Wallet * nat -> PursesMap
Transfer(pmap, wallet, sum) == is not yet specified;
```

The function takes a **Wallet** (containing the identifiers of 2 purses) and transfers a sum of money from the purse referenced in the wallet's **fromSlot** to the purse referenced in the wallet's **toSlot.** The **PursesMap** (pmap) holds information about the purses, referenced by the purse identifiers. An updated instance of **PursesMap** is returned.

Complete the Transfer function, taking account of any preconditions that may be needed for **both Transfer and SetupTransfer** in order to satisfy the security properties.

You may find it helpful to use one or two additional auxiliary functions to increase/decrease a purse balance by a sum of money. [4 marks]