Revisiting ATMs*

A COMP case study

Răzvan Diaconescu and Ionuț Țuțu

Simion Stoilow Institute of Mathematics
of the Romanian Academy, Romania
razvan.diaconescu@imar.ro, ionut.tutu@imar.ro

1 Summary

This report documents the formal specification and analysis in COMP of a system of automated teller machines (ATMs) that accommodates multiple users, bank accounts, cash machines, and complex interactions between them. We cover bank accounts, deposit and withdrawal operations, cash machines (modelled in terms of their components, each with a specific function), and interconnected systems of accounts and ATMs. All are built in a hierarchical manner following the specification methodology of COMP. In the last section of the report, we show how non-interference properties can be formalized and checked using the COMP tool.

For a full formal account of the syntax, semantics, and specification & verification methodologies of the language – which we bring into use in this case study – see the COMP language-definition document. Instructions on how to obtain and run the tool, including a short tutorial and a glossary of commands, can be found in the COMP user guide. Both documents are available at the COMP homepage.

*This work was supported by a grant of the Romanian Ministry of Education and Research, CCCDI – UEFISCDI, project number PN-III-P2-2.1-PED-2019-0955, within PNCDI III.

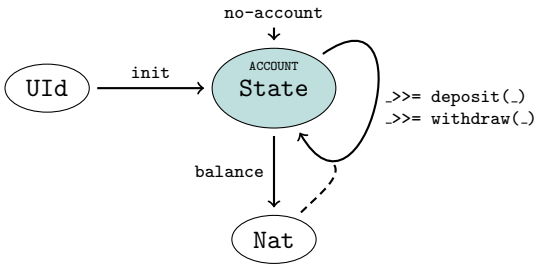
2 The bank-account system

We begin with the specification of the data type of user identifiers (or ids, for short). We identify users using natural numbers and also make provision for exceptional situations in the operation of an ATM where a user cannot be identified.

```
data USER-ID is
  protecting NAT .
  sort UId .
  op uid : Nat -> UId [ctor] .
  op unidentified-user : () -> UId [ctor] .
enddata
```

User accounts are modelled as behavioural objects without componets. We consider two initialization operations, `no-account` and `init`, that capture the absence of an account and the initial state of an account opened for a given user. In addition, we specify two actions, `_>>= deposit(_)` and `_>>= withdraw(_)`, for depositing and withdrawing funds from an account, respectively, which we declare in mixfix form in order to allow for chaining actions in the later part of the case study – as in `S >>= deposit(M) >>= withdraw(N)` for withdrawing `N` monetary units after depositing `M` monetary units to an account state `S`. The amount of money in (a given state of) a user's account is given by an observation called `balance`.

The following ADJ diagram gives a compact graphical representation of the signature of the `ACCOUNT` object. Data types, labelled in this case `UId` and `Nat`, are depicted using white elliptical shapes, while (the type of) object states, implicitly called `State`, are depicted using teal-blue elliptical shapes. Operations are represented using hyperarcs with multiple inputs depending on the arity of the operation (some of the lines are dashed in order to indicate data-type arguments).



In addition to the signature part discussed above, the object `ACCOUNT` also comprises five axioms (more precisely, strict equations, two of which are conditional) that define the initial value of an account and how the balance changes under deposit and withdraw actions. The full listing of the object module is as follows:

```
bobj ACCOUNT is
  protecting NAT/ORD .
  protecting NAT/ADDITION .
  protecting NAT/SUBTRACTION .
  protecting USER-ID .
  var S : State . var U : UId . var N : Nat .

  op no-account : () -> State .
  op init : UId -> State .
  act _>>=`deposit`(_`) : State Nat -> State .
  act _>>=`withdraw`(_`) : State Nat -> State .
  obs balance : State -> Nat .

  ax balance(no-ccount) = 0 .
  ax balance(init(U)) = 0 .
  ax balance(S >>= deposit(N)) = balance(S) + N .
  ax balance(S >>= withdraw(N)) = balance(S) - N if balance(S) >= N = true .
  ax balance(S >>= withdraw(N)) = balance(S) if balance(S) < N = true .
endbo
```

Next, we define the operations that users can perform on their accounts. These are either operations that may alter the state of a user's account (depositing or withdrawing funds to/from that account), or a special `nop` operation indicating that the state of the account should not be altered in any way.

```
data OPERATION is
  sort Operation .
  ops deposit, withdraw, nop : () -> Operation [ctor] .
enddata
```

Users, accounts, and account operations are all basic building blocks of an account-system object, which we call `BANK`. Intuitively, this is a database that maps each recorded user id to an account. We model it in COMP using an indexed composition.

```
bobj BANK is
  indexing ACCOUNT on USER-ID by UId .
  protecting OPERATION .
  var S : State . vars U, U' : UId . var N : Nat . var O : Operation .
```

We use a constant `init` to indicate a state in which the object `BANK` has no registered users – and thus no accounts either. We specify the later property by defining the `ACCOUNT` projection of `init` as `no-account` for every user id `U`.

```
  op init : () -> State .
  ax ACCOUNT/State(U, init) = no-account .
```

For BANK, we consider three actions. All of them are declared with mixfix syntax for convenience – to ease chaining – similarly to the actions of the object ACCOUNT. First, we consider an action that allows new users to be added to the bank database; for each such user, we consider an initial balance of their account.

```
  act _>>=`add`(_`,_`) : State UId Nat -> State .
  ax ACCOUNT/State(U, S >>= add(U, N))
   = init(U) >>= deposit(N) .
  ax ACCOUNT/State(U, S >>= add(U', N))
   = ACCOUNT/State(U, S) if not U = U' .
```

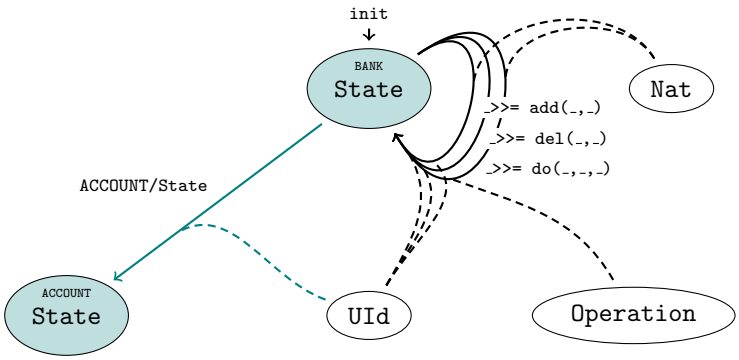
Second, we formalize (also using an action) the closing of an user's account.

```
  act _>>=`del`(_`) : State UId -> State .
  ax ACCOUNT/State(U, S >>= del(U))
   = no-account .
  ax ACCOUNT/State(U, S >>= del(U'))
   = ACCOUNT/State(U, S) if not U = U' .
```

The last BANK action we consider is do, which is used to apply an operation (deposit, withdraw, or nop) to a user's account. Notice that we no longer check the account's balance when withdrawing funds, because that functionality is already encapsulated in the ACCOUNT object. All we need to verify is that the user is correctly identified and that the operation is indeed meant to update their account.

```
  act _>>=`do`(_`,_`,_`) : State Operation UId Nat -> State .
  ax ACCOUNT/State(U, S >>= do(deposit, U, N))
   = ACCOUNT/State(U, S) >>= deposit(N)
   if not U = unidentified-user .
  ax ACCOUNT/State(U, S >>= do(withdraw, U, N))
   = ACCOUNT/State(U, S) >>= withdraw(N)
   if not U = unidentified-user .
  ax ACCOUNT/State(U, S >>= do(O, U', N))
   = ACCOUNT/State(U, S)
   if not U = U' or U' = unidentified-user or O = nop .
endbo
```

To summarize, the following ADJ diagram gives an overview of the algebraic signature of the `BANK` module. In order to keep the diagram readable, we omit the `ACCOUNT` actions and observations that the interpreter defines automatically for `BANK`. Those include, for example, `S >>= deposit(N) :: ACCOUNT(U)` and `S >>= withdraw(N) :: ACCOUNT(U)`, which are used to deposit/withdraw `N` monetary units to/from the account of a user `U` in a `BANK` state `S`. The only automatic behavioural operation we include (depicted in teal) is the `ACCOUNT` projection on `BANK` states, which captures the 'has component' relation between the objects `BANK` and `ACCOUNT`.

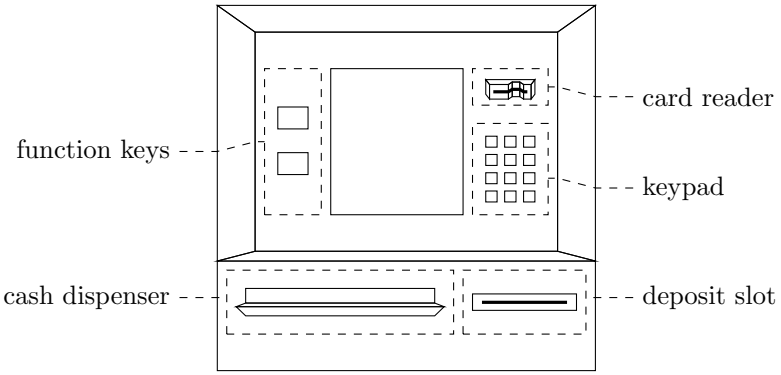


3 ATMs

Analogously to user accounts (because we aim to support multiple devices) we consider identifiers of ATMs: each machine has a unique natural-number identifier.

```
data ATM-ID is
  protecting NAT .
  sort AId .
  op aid : Nat -> AId [ctor] .
enddata
```

For the machines we analyse in this case study, it suffices to consider five physical components: a card reader, a set of function keys, a keypad, a cash dispenser, and a deposit slot. These are depicted in the image below.



We model each of the five physical components as a base object. For instance, for the card reader, we consider the following module with one initial-state constant, one action (capturing the insertion of a card into the reader), and one observation (capturing the reading of user-identification data from an inserted card).

```
bobj CARD-READER is
  protecting USER-ID .
  var S : State . var U : UId .

  op init : () -> State .
  act _>>=`insert-card`(_`) : State UId -> State .
  obs get-user : State -> UId .

  ax get-user(init) = unidentified-user .
  ax get-user(S >>= insert-card(U)) = U .
endbo
```

For the function keys, we consider two physical buttons utilized to select what kind of account operation the user intends to perform: i.e., deposit or withdraw. For each button, we define a corresponding action (with the same name as the operation). In addition, we use a `get-op` observation (with sort `Operation`) through which the ATM controller could determine which of the two function keys has been pressed.

```
bobj FUNCTION-KEYS is
  protecting OPERATION .
  var S : State .

  op init : () -> State .
  act _>>=`deposit : State -> State .
  act _>>=`withdraw : State -> State .
  obs get-op : State -> Operation .

  ax get-op(init) = nop .
  ax get-op(S >>= deposit) = deposit .
  ax get-op(S >>= withdraw) = withdraw .
endbo
```

The keypad is an array of buttons through which users can input numeric values. We model it as an object with one initial-state constant, one action (to indicate that a value has been entered into the device, which is usually done by pressing a special button on the keypad), and one observation (to retrieve the input data).

```
bobj KEYPAD is
  protecting NAT .
  var S : State . var N : Nat .

  op init : () -> State .
  act _>>=`input`(_`) : State Nat -> State .
  obs get-input : State -> Nat .

  ax get-input(init) = 0 .
  ax get-input(S >>= input(N)) = N .
endbo
```

The last two physical components of the ATM that we consider here are used for dealing with cash. One of them is suited to dispensing cash (as a result of a withdrawal), while the other is suited to depositing cash. Each of them has a corresponding action for dealing out or taking in a given amount, and an observation that reveals how many monetary units have been delivered to or obtained from the user. In addition, for the cash dispenser, we consider a separate `clear` action to model the physical withdrawal of banknotes from the dispenser slot.

```
bobj CASH-DISPENSER is
  protecting NAT .
  var S : State . var N : Nat .

  op init : () -> State .
  act _>>=`dispense`(_`) : State Nat -> State .
  act _>>=`clear : State -> State .
  obs get-amount : State -> Nat .

  ax get-amount(init) = 0 .
  ax get-amount(S >>= dispense(N)) = N .
  ax get-amount(S >>= clear) = 0 .
endbo
```

```
bobj DEPOSIT-SLOT is
  protecting NAT .
  var S : State . var N : Nat .

  op init : () -> State .
  act _>>=`pay`(_`) : State Nat -> State .
  obs get-amount : State -> Nat .

  ax get-amount(init) = 0 .
  ax get-amount(S >>= pay(N)) = N .
endbo
```

We are now ready to specify ATMs as compound objects combining card readers, function keys, cash dispensers, and so on. We consider a synchronized composition instead of a simpler parallel composition because, while using an ATM, none of the five components of the device can operate independently of the other four.

```
bobj ATM is
  syncing CARD-READER and FUNCTION-KEYS and KEYPAD
          and DEPOSIT-SLOT and CASH-DISPENSER .
  protecting ATM-ID .
  protecting NAT/ORD .
  var S : State . var N : Nat . var A : AId . var U : UId .
```

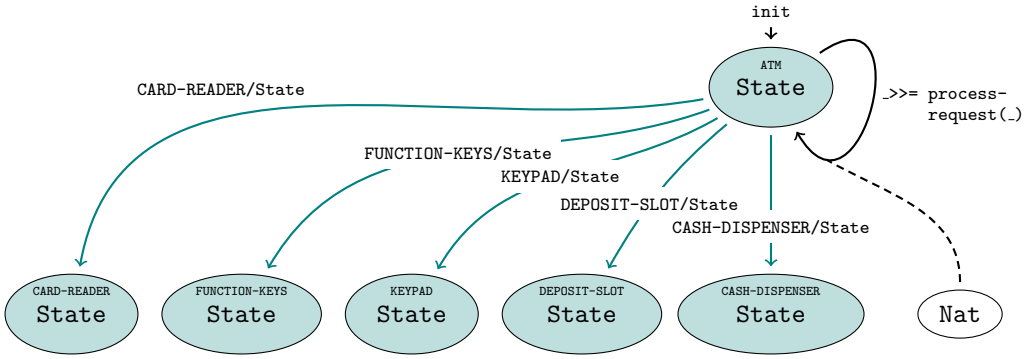
To model the initial state of the machine we consider a monadic operation `init` that defines an initial state `init(A)` for each ATM identifier `A`. According to the specification methodology of COMP, we need to specify what are the component projections of initial states – which is trivial in this case because we simply initialize each of the five components of the ATM.

```
  op init : AId -> State .
  ax CARD-READER/State(init(A)) = init .
  ax FUNCTION-KEYS/State(init(A)) = init .
  ax KEYPAD/State(init(A)) = init .
  ax DEPOSIT-SLOT/State(init(A)) = init .
  ax CASH-DISPENSER/State(init(A)) = init .
```

The synchronization action of ATM, called process-request, marks the completion of an operation in front of the machine. Its only data-type argument, a natural number, is an upper bound on how much cash a user may withdraw; we need to provide this argument because, unless we connect the ATM to a bank, there is no way of determining the user's account balance. Under this action, most of the ATM components are reset to their initial states in order to be ready for the next user interaction. Only the deposit slot and the cash dispenser are continuously updated to capture changes in the vault of the ATM as a result of deposit/withdraw operations.

```
  ax DEPOSIT-SLOT/State(S >>= process-request(N))
   = DEPOSIT-SLOT/State(S) .
  ax CASH-DISPENSER/State(S >>= process-request(N))
   = CASH-DISPENSER/State(S) >>= dispense(KEYPAD/get-input(S))
   if not CARD-READER/get-user(S) = unidentified-user
      and FUNCTION-KEYS/get-op(S) = withdraw
      and KEYPAD/get-input(S) <= N = true .
  ax CASH-DISPENSER/State(S >>= process-request(N))
   = CASH-DISPENSER/State(S)
   if CARD-READER/get-user(S) = unidentified-user
      or not FUNCTION-KEYS/get-op(S) = withdraw
      or KEYPAD/get-input(S) > N = true .
endbo
```

The ADJ diagram below gives an overview of the structure of ATM.



Similarly to BANK, we introduce a new object, called DEVICES, to model a database of ATMs. We specify it in COMP through an indexed composition mapping ATM ids to cash machines. We consider an initial state of the database, which contains no machines, and two actions: add for adding a new ATM (given by its id) to the database, and del for removing an ATM from the database.

```
bobj DEVICES is
  indexing ATM on ATM-ID by AId .
  var S : State . vars A, A' : AId .

  op init : () -> State .
  ax ATM/State(A, init) = no-atm .

  act _>>=`add`(_`) : State AId -> State .
  ax ATM/State(A, S >>= add(A)) = init(A) .
  ax ATM/State(A, S >>= add(A')) = ATM/State(A, S) if not A = A' .

  act _>>=`del`(_`) : State AId -> State .
  ax ATM/State(A, S >>= del(A)) = no-atm .
  ax ATM/State(A, S >>= del(A')) = ATM/State(A, S) if not A = A' .
endbo
```

4 Connecting ATMs to bank accounts

The final assembly step of the COMP specification of an ATM system consists in synchronizing ATMs with bank accounts so that any cash-machine operation is properly linked to a user-account update.

```
bobj ATM-SYSTEM is
  syncing BANK and DEVICES .
  var S : State . vars A, A' : AId . var U : UId . var N : Nat .
```

Initially, the system has no registered users and no ATMs are deployed.

```
  op init : () -> State .
  ax BANK/State(init) = init .
  ax DEVICES/State(init) = init .
```

We synchronize ATMs with bank accounts through the action `all-done`, which captures the completion of a transaction. For ATMs, this reduces to applying the action `process-request` (with the proper account balance this time, since the system has access to it); but for bank accounts we need to consider several cases depending on which function key (account operation) has been selected.

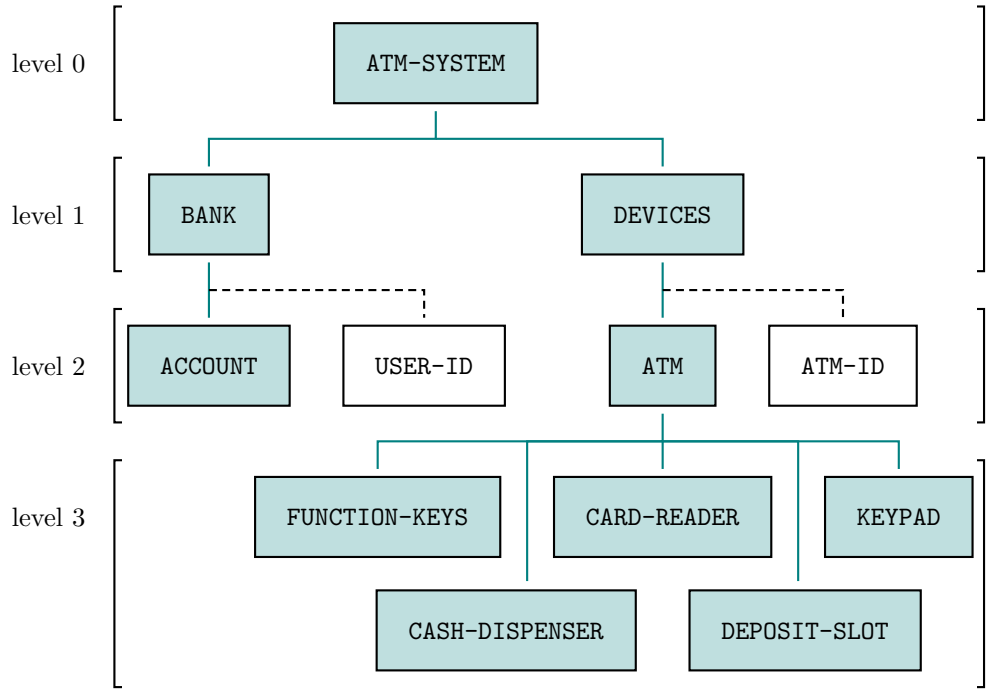
```
  act _>>=`all-done`(_`) : State AId -> State .
  ax BANK/State(S >>= all-done(A))
   = BANK/State(S) >>= do(deposit,
       DEVICES/ATM/CARD-READER/get-user(A, S),
       DEVICES/ATM/DEPOSIT-SLOT/get-amount(A, S))
   if DEVICES/ATM/FUNCTION-KEYS/get-op(A, S) = deposit .

  ax BANK/State(S >>= all-done(A))
   = BANK/State(S) >>= do(withdraw,
       DEVICES/ATM/CARD-READER/get-user(A, S),
       DEVICES/ATM/KEYPAD/get-input(A, S))
   if DEVICES/ATM/FUNCTION-KEYS/get-op(A, S) = withdraw .

  ax BANK/State(S >>= all-done(A))
   = BANK/State(S)
   if DEVICES/ATM/FUNCTION-KEYS/get-op(A, S) = nop .

  ax DEVICES/State(S >>= all-done(A))
   = DEVICES/State(S)
     >>= process-request(
           BANK/ACCOUNT/balance(DEVICES/ATM/CARD-READER/get-user(A, S), S))
     :: ATM(A) .
```

The resulting hierarchical structure of the ATM system consists of four levels, which we label from 0 (the root level) to 3. Each teal edge indicates a composition of objects, either synchronized (as it is the case for `ATM-SYSTEM` and `ATM`) or indexed, in which case we use dashed lines to link the corresponding indexing modules.



Lastly, we add two macro operations that encode the deposit or withdraw operations performed by a user at an ATM. For instance, to deposit `N` monetary units at an ATM `A`, a user `U` first inserts the card, then selects `deposit` using the function keys, inserts banknotes into the deposit slot, and confirms the operation.

```
  ax S >>= deposit(U, A, N)
   = S >>= insert-card(U) :: CARD-READER :: ATM(A) :: DEVICES
       >>= deposit :: FUNCTION-KEYS :: ATM(A) :: DEVICES
       >>= pay(N) :: DEPOSIT-SLOT :: ATM(A) :: DEVICES
       >>= all-done(A) .

  ax S >>= withdraw(U, A, N)
   = S >>= insert-card(U) :: CARD-READER :: ATM(A) :: DEVICES
       >>= withdraw :: FUNCTION-KEYS :: ATM(A) :: DEVICES
       >>= input(N) :: KEYPAD :: ATM(A) :: DEVICES
       >>= all-done(A)
       >>= clear :: CASH-DISPENSER :: ATM(A) :: DEVICES .
endbo
```

The following table gives an overview of the increasing complexity of the hierarchical specification of the ATM system. For each of its ten objects, we record the overall number of declarations/sorts/operations/axioms/etc. – including those imported from other modules – and we highlight in parentheses how many of them are automatically generated by COMP. As expected, only a handful of declarations are generated automatically for those objects in the lower part of the hierarchy such as `ACCOUNT`, `KEYPAD`, or `DEPOSIT-SLOT`; but the situation changes drastically for objects with a richer structure: more than half of the final specification of the ATM system is generated automatically by the COMP interpreter.

Object	Declarations	Sorts	Operations	Actions	Obs.	Axioms
`ACCOUNT`	53(1)	4(1)	21(0)	2(0)	1(0)	28(0)
`BANK`	75(11)	6(2)	32(4)	7(2)	2(1)	41(5)
`FUNCTION-KEYS`	12(1)	2(1)	7(0)	2(0)	1(0)	3(0)
`CARD-READER`	12(1)	3(1)	7(0)	1(0)	1(0)	2(0)
`KEYPAD`	10(1)	2(1)	5(0)	1(0)	1(0)	2(0)
`CASH-DISPENSER`	11(1)	2(1)	6(0)	2(0)	1(0)	3(0)
`DEPOSIT-SLOT`	10(1)	2(1)	5(0)	1(0)	1(0)	2(0)
`ATM`	147(63)	11(6)	55(17)	15(7)	10(5)	81(40)
`DEVICES`	191(99)	12(7)	72(31)	25(15)	15(10)	107(61)
`ATM-SYSTEM`	303(171)	15(10)	114(58)	48(32)	23(17)	174(103)

5 Non-interference properties

Using the theorem-proving capabilities of COMP, we show that ATM withdrawals are non-interfering. More precisely, we show that, for any two distinct users `u1` and `u2` and any system state where both `u1` and `u2` have accounts open, every withdrawal made by `u1`, using any ATM registered in the system, is independent of every withdrawal made by `u2`, once more, using any ATM in the system.

First, we formalize the assumptions and we introduce them using the `let` command:

```
open ATM-SYSTEM
  let op s : () -> State .
  let ops u, u1, u2 : () -> UId .
  let ops a, a1, a2 : () -> AId .
  let ops m, m1, m1', m2, m2', n1, n2 : () -> Nat .

  let op system-state : Nat Nat -> State .
  let ax system-state(M1:Nat, M2:Nat)
       = s >>= add(a)          :: DEVICES
           >>= add(a1)         :: DEVICES
           >>= add(a2)         :: DEVICES
           >>= add(u, m)       :: BANK
           >>= add(u1, M1:Nat) :: BANK
           >>= add(u2, M2:Nat) :: BANK .

  let ax not u = u1 .
  let ax not u = u2 .
  let ax not u1 = unidentified-user .
  let ax not u2 = unidentified-user .
  let ax not u1 = u2 .
  let ax not a = a1 .
  let ax not a = a2 .
```

We consider several cases, depending on whether the amount withdrawn by ui (for $i \in \{1, 2\}$, given by ni), is 0, less than or equal to the balance of ui's account (given by mi), or greater than the balance of ui's account (given by mi'). The following additional axioms – also introduced using `let` – provide support for case analysis.

```
  let ax not n1 = 0 .
  let ax n1 <= m1 = true .
  let ax n1 <= m1' = false .
  let ax not n2 = 0 .
  let ax n2 <= m2 = true .
  let ax n2 <= m2' = false .
```

With all the preparations in place, the final verification step is straightforward:

```
  check system-state(M1:Nat, M2:Nat)
        >>= withdraw(u1, A1:AId, N1:Nat)
        >>= withdraw(u2, A2:AId, N2:Nat)
      ~ system-state(M1:Nat, M2:Nat)
        >>= withdraw(u2, A2:AId, N2:Nat)
        >>= withdraw(u1, A1:AId, N1:Nat)
  forall (A1:AId = a1 and (A2:AId = a1 or A2:AId = a2)
          or A1:AId = a and (A2:AId = a or A2:AId = a2))
     and (N1:Nat = n1 and (M1:Nat = m1 or M1:Nat = m1')
          or N1:Nat = 0 and M1:Nat = m1)
     and (N2:Nat = n2 and (M2:Nat = m2 or M2:Nat = m2')
          or N2:Nat = 0 and M2:Nat = m2)
  given  (<BANK/State.ACCOUNT/State>-UI:UId = u
          or <BANK/State.ACCOUNT/State>-UI:UId = u1
          or <BANK/State.ACCOUNT/State>-UI:UId = u2)
     and <DEVICES/State.ATM/State>-AI:AId = a .
close
```

In a similar manner, we can show that any other combination of operations (two deposits, a deposit followed by a withdrawal, or vice versa) is non-interfering as well. Using the command `show check stats` (executed before `check`), we can also see how many equalities are examined by COMP in order to fully verify the property:

```
| Proved! The property holds.
|
| Equations examined for each bobj hierarchy level
| level 0: 108 behavioural and 0 strict
| level 1: 216 behavioural and 0 strict
| level 2: 216 behavioural and 0 strict
| level 3: 135 behavioural and 21 strict
|_______________________________________
```

The full specification of the ATM system, including proof scores of non-interference properties, is available in the COMP online repository of examples.