Specification of a Bank Account in PSF

Indra Polak
June 30, 1994
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098SJ Amsterdam, The Netherlands.
E-mail: **ipolak@fwi.uva.nl**

Abstract

The behaviour of a bank account is specified in the Process Specification Formalism PSF. The following aspects will be specified: transfers, transactions, communications between accounts mutually and with a central cash and interest accounting at regular periods. We follow a report, [SS93].

1 Introduction

In this article a specification of a bank account is presented in the Process Specification Formalism PSF. We follow a report written by A. Stins and A. Schoneveld, [SS93]. The following aspects of a bank account will be specified :

- transfers and transactions,
- communication between accounts mutually and with a central cash,
- interest accounting.

The main goals of this article are to show that a bank account can be modelled as a communicating process, and to present an executable specification. A bank account communicates with an environment consisting of the following processes: a cash, other bank accounts and the owner(s) of the account. To all these other processes messages are sent concerning transactions, transfers and interest calculations. To keep balances in order, synchronisation of some communications is required. When we look at a bank account this way, we clearly see the process-like behaviour of a bank account.

We will start with a very simple model. After that, gradually more complexity is added to get a more realistic view. We suppose the reader is acquainted with process algebra, if not [BW91] forms a good source. Also PSF is used of course; see [Mau91] for an introduction.

We will use a different font to write actions or processes from the specification, as in `action`. Names of programs are written like *program*.

2 A very simple bank

In a very simple bank, only two processes exist. Firstly, there is the `cash` process. This is the bank's own account, keeping track of the bank's balance. Secondly, there is a customer account called `bank-account`. Only these two processes interact for the time being. Furthermore, only symbolic actions are possible. No data is transferred, no money flows: only the skeleton of possible actions of `bank-account` and `cash` is visible. The specification is as follows:

Specification 2.1

```
process module BAsimple
```

```
begin

 exports
  begin
   atoms
      message-inp-client
      message-out-client
      message-inp-intern
      message-out-intern
      req-deposit
      req-collect
      req-transfer
      identification
      pay-amount
      deposit-amount
      c

    processes
      req-transaction
      transfer
      test-amount
      transaction
      bank-account
      cash
      bank

    end

    sets of atoms
      H = { message-inp-intern,
           message-out-intern }

    communications

       message-inp-intern | message-out-intern = c

    definitions

      req-transaction =
        req-deposit +
        req-collect

      test-amount       =
           message-out-intern + skip

      transfer =

(
 (req-transfer + message-inp-client) .
     identification .
     test-amount
) +
  message-inp-intern
```

```
      transaction  =
        req-transaction .
        identification .
        message-out-intern

      cash       =

        message-inp-intern .
        (deposit-amount  + pay-amount) .
        cash

      bank-account =

        (transaction + transfer) .
        message-out-client .
        bank-account

      bank =  encaps(H, cash || bank-account)

end BAsimple
```

We will start with explaining the meaning of all atomic actions and their initiators.

message-inp-client A message from a client ordering the bank to transfer money. This message is sent by a client to the bank.

message-out-client A message to a client telling the client of the changed balance. This message is sent by the bank to a client.

message-inp-intern Receive a message from another Bankaccount concerning a transfer. If the Cash process receives this message, it is the result of a Transaction; otherwise it is the result of a Transfer.

message-out-intern Send a message to another Bankaccount or to the Cash. With a transaction, send it to the Cash; with a transfer send it to the target account.

req-deposit A request to deposit money. This request is done by the client.

req-collect A request to collect money. This request is done by the client.

req-transfer A request to transfer money between your own account and another one. Also done by the client.

identification An action that tries to make sure the identity of the person initiating the flow of money. This action is done by the bank.

pay-amount The cash pays out looking glum.

deposit-amount The cash greedily accepts your life savings.

c The communication between or the Cash and a Bankaccount or two Bankaccounts mutually.

Next we will show the message interface, visualised from the perspective of a Bank account A.

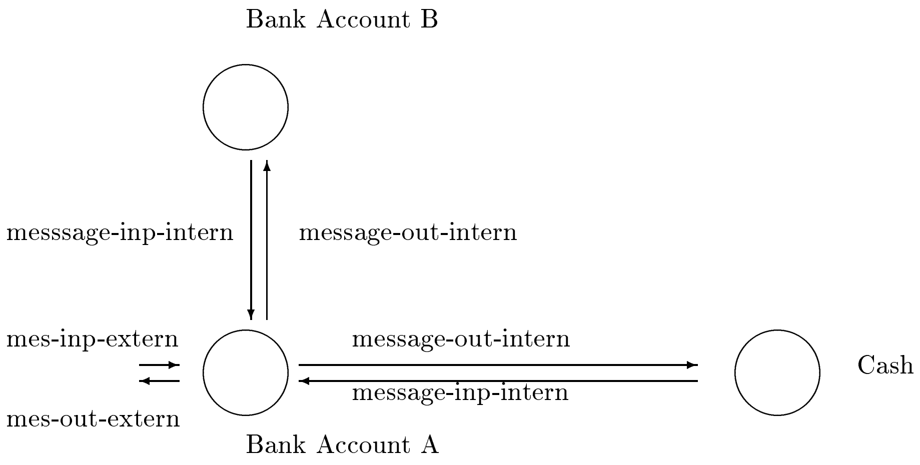

Figure 1 : The message interface

In our specification, a bank account has the following possibilities:

- a transaction;
- a transfer.

A transaction is accomplished by requesting to collect or to deposit, this is mostly done at the bank or an automaton.

A transfer means the transfer of money between two accounts, where the amount transferred must be positive. This test is done in the process `test-amount`. In such a transfer, one account is the active account, and one the passive. The one that initiates the transfer is active; the other passive. When 'doing' a passive transfer the bank account receives a message from another account, without having an obligation to identify itself, since only positive amounts of money may be transferred. A transfer can be made by sending a message or by requesting a transfer. The difference is the physical contact between customer and bank: `request-transfer` is done at the bank, `message-inp-client` is the result of a mailed request. In both cases an identification is performed; in the mailed case this means checking the signature, among other things. After each 'active' possibility an identification is done; then a message is sent to `cash` to pay or collect the money. Finally, the customer gets a message concerning his new balance, and the process starts anew. We see no data, no numbers or other misleading figures in the process definition. The purest form of a bank account appears in all its beauty.

2.1 Example trace

To execute a PSF specification there exists a PSF tool called *sim*. This tool simulates a given PSF specification after it has been compiled to *.til* code. See [Vel91] for more information concerning *sim* and *.til* code. When executing this specification with *sim*, we see the following sample traces, where `c` stands for communication:

```
atom req-transfer
atom identification
com. c
atom pay-amount
atom message-out-client
```

```
atom req-collect
atom identification
com. c
atom message-out-client
atom pay-amount

atom req-deposit
atom identification
com. c
atom message-out-client
atom req-collect
atom identification
atom deposit-amount
com. c
atom pay-amount
atom message-out-client
```

These are normal traces, but also the following traces are possible:

```
atom req-collect
atom identification
com. c
atom deposit-amount
atom message-out-client

atom req-deposit
atom identification
com. c
atom pay-amount
atom message-out-client
```

These should not happen of course, but we see now the result of omitting all data information: when `cash` must choose whether to add or subtract the money from its balance, it should know the sign of the amount in question. Since we omitted all data information, `cash` leaves both possibilities open. Later on we will see that with more information added, this trace cannot occur anymore.

More important, we see that in all traces, between two communications, at least one `pay-amount` or `deposit-amount` action must happen. This is important, because otherwise the bank's real balance at a certain moment can be different from the computed balance. Now at communication time, things are in order. So at any instant the bank's balance is correct. We can see this in the specification of `cash`: after a communication, there must occur a `pay-amount` or a `deposit-amount`. Only after one of these, a new communication is possible. So in the whole system, various bank accounts must wait for the cash to finish its actions between two communications. This is a crucial point.

Since in a big bank many accounts are concurrently in operation, this immediately implies an important role for the updating mechanism: it must be fast enough.

A strange thing is this: we must communicate to ask whether a certain transaction is possible, but there is no answer from `cash` telling us whether the transaction could be done. On page 4 and 5 of [SS93] this is resolved: we need data to specify it. So first we will introduce some data structures to help us to proceed.
In general, only two data structures are needed:

1. identifiers;
2. floating point numbers.

In [SS93] we see processes with parameters. These parameters are defined in PSF using abstract data types, defined with the Algebraic Specification Formalism ASF. See [BHK87] for details. Now in Process Algebra it is allowed to sum over infinite sets, so in PSF it is also allowed to sum over infinite data types. But our simulator *sim* cannot handle these infinite sums yet. This means we have to make some restrictions. We chose the following solutions: at any instant only a finite amount of transactions is possible. These transactions are defined as a finite data type. In this way we can still look at the different traces of the system and see whether the numbers are correctly added, subtracted and tested.

For the floating-point numbers we use a PSF library documented in [vW93]. We had to make some floating-point functions which did not appear in the library, and finally all processes are augmented with parameters that reflect their data. We follow the specification in [SS93] as closely as possible. However we have left out the Personal information parameter, because this parameter was never used. In a formal specification one should try to keep things as clear as possible for the reader. Adding information that is never used only makes the specification more difficult.

Specification 2.2

```
data module Clients
begin

        exports
         begin
          sorts
           CLIENT
          functions
           client-1     : -> CLIENT
           cash-c       : -> CLIENT
           eq           : CLIENT # CLIENT -> BOOLEAN

         end
        imports Booleans

        variables
         c1,c2 : -> CLIENT

        equations

         [C1]    eq(c1,c1) = true
         [C2]    eq(client-1,cash-c) = false
         [C3]    eq(cash-c,client-1) = false

end Clients

data module Transactions
begin
 exports
  begin
   sorts
    TRANSACTION
```

```
   functions
    tr1         : -> TRANSACTION
    tr2         : -> TRANSACTION
    tr3         : -> TRANSACTION
    tr4         : -> TRANSACTION
    amount      : TRANSACTION -> FP
  end
 imports
  Floating-Points-3-4
 equations
  [T1]  amount(tr1)     = fp(^1^0^0)
  [T2]  amount(tr2)     = fp(neg,^1^0^0)
  [T3]  amount(tr3)     = fp(^3^0^0)
  [T4]  amount(tr4)     = fp(neg, ^3^0^0)
end Transactions


data module Floating-Points-Funcs
begin
 exports
  begin
-- The functions we define are :
-- le : less or equal, <=
-- ge : greater or equal, >=
-- eq : equal, =
-- lt : less than, <
   functions
    le          : FP # FP -> BOOLEAN
    ge          : FP # FP -> BOOLEAN
    eq          : FP # FP -> BOOLEAN
    lt          : FP # FP -> BOOLEAN
  end
  imports
    Floating-Points-3-4, Booleans
  functions
   nf           : FP -> FP

-- nf means normal form, nf computes the normal form of a given FP.
-- This is necessary if one wants to define functions on normal forms
-- without knowing the rewrite strategy beforehand. When an innermost-
-- outermost strategy is performed always, then the nf function is not
-- necessary.

   le-nf        : FP # FP -> BOOLEAN
   ge-nf        : FP # FP -> BOOLEAN
   eq-nf        : FP # FP -> BOOLEAN
   lt-nf        : FP # FP -> BOOLEAN
  variables
    dig                : -> DIGIT
    dstr, dstr1, dstr2 : -> DIGIT-STRING
    s                  : -> SIGN
    exp                : -> INTEGER

   f1,f2        : -> FP
```

```
   d1,d2,d3,d4  : -> DIGIT-STRING
   i1,i2        : -> INTEGER
   s1,s2        : -> SIGN
  equations
   [']  nf(fp(s,dstr1^dig,dstr2,exp)) =
        nf(fp(s,cut-left-zeros(dstr1^dig),dstr2,exp))
        when eq(^first(dstr1),^0) = true
   [']  nf(fp(s,dstr1,dstr2,exp)) =
        nf(fp(s,^0,concat(dstr1,dstr2),exp + int(pos,dstr-length(dstr1))))
          when eq(^first(dstr1),^0) = false
   [']  nf(fp(s,^0,dstr,exp)) = nf(fp(s,^0,tail(dstr),exp - int(pos,^1)))
          when and(eq(^first(dstr),^0),not(eq(nat(dstr),nat(^0)))) = true
   [']  nf(fp(s,^0,dstr,exp)) = nf(fp(pos,^0,dstr,int(pos,^0)))
          when and(eq(^first(dstr),^0),not(eq(exp,int(pos,^0)))) = true
   [']  nf(fp(dstr)) = nf(fp(pos,dstr))
   [']  nf(fp(dstr,exp))        = nf(fp(pos,dstr,exp))
   [']  nf( fp(dstr1,dstr2)) = nf(fp(pos,dstr1,dstr2,int(pos,^0)))
   [']  nf(fp(dstr1,dstr2,exp)) = nf(fp(pos,dstr1,dstr2,exp))
   [']  nf(fp(s,dstr)) =
   nf(fp(s,^0,cut-left-zeros(dstr),int(pos,dstr-length(cut-left-zeros(dstr)))))
   [']  nf(fp(s,dstr,exp)) = nf(fp(s,^0,cut-left-zeros(dstr),exp +
int(pos,dstr-length(cut-left-zeros(dstr)))))
   [']  nf(fp(s,dstr1,dstr2)) = nf(fp(s,dstr1,dstr2,int(pos,^0)))
   [']  nf(fp(int(s,dstr),exp)) = nf(fp(s,dstr,exp))
   [']  nf(fp(int(s,dstr))) = nf(fp(s,dstr))
   [']  nf(fp(s,^0,dstr,exp)) = fp(s,^0,dstr,exp)
         when eq(^first(dstr),^0) = false
   [']  nf(fp(pos,^0,^0,int(pos,^0))) = fp(pos,^0,^0,int(pos,^0))

   [F1] le-nf(fp(neg,d1,d2,i1), fp(pos,d3,d4,i2))       = true
   [F2] le-nf(fp(pos,d1,d2,i1), fp(neg,d3,d4,i2))       = false
   [F3] le-nf(fp(pos,d1,d2,i1), fp(pos,d1,d3,i2))       =
          or(lt(i1,i2), and(eq(i1,i2), lte(nat(d2),nat(d3))))
   [F4] le-nf(fp(neg,d1,d2,i1), fp(neg,d1,d3,i2))  =
          or(lt(i1,i2), and(eq(i1,i2), lte(nat(d3),nat(d2))))
   [F5] ge-nf(f1,f2) = le(f2,f1)
   [F6] eq-nf(fp(s1,d1,d2,i1),fp(s2,d3,d4,i2)) =
         and(eq(s1,s2),and(eq(d1,d3), and(eq(d2,d4), eq(i1,i2))))
   [F7] lt-nf(f1,f2) = and(le(f1,f2), not(eq(f1,f2)))

   [']  le(f1,f2) = le-nf(nf(f1),nf(f2))
   [']  ge(f1,f2) = ge-nf(nf(f1),nf(f2))
   [']  eq(f1,f2) = eq-nf(nf(f1),nf(f2))
   [']  lt(f1,f2) = lt-nf(nf(f1),nf(f2))
end Floating-Points-Funcs

process module BA2
begin

 exports
  begin
   atoms
      message-inp-client : CLIENT # CLIENT # FP
      message-out-client : CLIENT #  FP
```

```
      message-inp-intern : CLIENT # FP
      message-out-intern : CLIENT # FP
      req-deposit        : CLIENT # FP
      req-collect        : CLIENT # FP
      req-transfer       : CLIENT # CLIENT # FP
      identification     : CLIENT
      pay-amount         : FP
      deposit-amount     : FP
      c                  : CLIENT # FP

    processes
      transfer           : CLIENT # CLIENT # TRANSACTION # FP # FP
      transaction        : CLIENT # TRANSACTION
      bank-account       : CLIENT # FP # FP
      mou                : CLIENT # TRANSACTION # FP # FP
      do-cash            : FP # TRANSACTION
      cash               : FP
      bank

   end

    imports Naturals, Integers, Floating-Points-3-4, Floating-Points-Funcs,
            Transactions, Clients

    sets of atoms
     H = { message-inp-intern(n,f), message-out-intern(n,f) |
           n in CLIENT, f in FP }

    communications

       message-inp-intern(n,f) | message-out-intern(n,f) = c(n,f)
        for n in CLIENT, f in FP

    variables
     n1,n2       : -> CLIENT
     f1,f2,f3    : -> FP
     ds1,ds2     : -> DIGIT-STRING
     i1          : -> INTEGER
     t1,t2       : -> TRANSACTION

    definitions

      transfer(n1,n2,t1,f2,f3) =
(
 ((req-transfer(n1,n2,amount(t1)) + message-inp-client(n1,n2,amount(t1))) .
     identification(n1) .
     ([le(amount(t1), fp(^0)) = true] ->
           message-out-intern(n2, fp(neg,^1) * amount(t1)) .
           mou(n1,t1,f2,f3))
+    [le(amount(t1), fp(^0)) = false] -> skip . bank-account(n1,f2,f3))
) +
message-inp-intern(n1,amount(t1)) . mou(n1,t1,f2,f3)

      transaction(n1,t1)  =
```

```
        ([le(amount(t1), fp(^0)) = true] -> req-collect(n1,amount(t1)) +
        [le(amount(t1), fp(^0)) = false] -> req-deposit(n1, amount(t1))) .
        identification(n1) .
        message-out-intern(cash-c, fp(neg,^1) * amount(t1))

      do-cash(f1,t1) =
         message-inp-intern(cash-c, amount(t1)) .
         ([le(amount(t1), fp(^0)) = true] -> deposit-amount(amount(t1)) +
         [le(amount(t1), fp(^0)) = false] -> pay-amount(amount(t1))) .
         cash(amount(t1) + f1)

      cash(f1)  =
        sum(t1 in TRANSACTION, do-cash(f1,t1))

-- mou stands for Message OUt
      mou(n1,t1,f1,f2) =
       message-out-client(n1, amount(t1) + f1) .
       bank-account(n1, amount(t1) + f1, f2)

-- bank-account(client,balance,limit)
      bank-account(n1,f1,f2) =
      sum(t1 in TRANSACTION, (transaction(n1,t1) . mou(n1,t1,f1,f2) +
      sum(n2 in CLIENT,
      transfer(n1,n2,t1,f1,f2))))


      bank =  encaps(H, cash(fp(^1^0^0^0)) ||
       bank-account(client-1,fp(^2^0^0),fp(neg,^5^0^0)))

end BA2
```

2.2 Example trace

```
bank = encaps(H, cash(fp(^1^0^0^0)) ||
       bank-account(client-1,fp(^2^0^0),fp(neg,^5^0^0)))


Trace :
       atom req-collect(client-1, fp(neg, (^ 0), (((^ 1) ^ 0) ^ 0),
                        int(pos, (^ 3))))
       atom identification(client-1)
       com. c(cash-c, fp(pos, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
       atom pay-amount(fp(pos, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
       atom message-out-client(client-1, fp(pos, (^ 0), (((^ 1) ^ 0) ^ 0),
                              int(pos, (^ 3))))

So 100 was subtracted from 200 yielding a correct 100 on the bank
account of client-1.
```

3 Reaching the limit

From a customers point of view, the bank as specified above might be called an ideal one. You can collect as much money as you like, without any penalty. To make the specification more realistic, we will add some tests to the specification. Now only money may be collected or transferred if the limit is not reached. If one tries to collect or transfer more money than one owns, then there is no communication.

Specification 3.1

```
process module BA3
begin

 exports
  begin
   atoms
      message-inp-client : CLIENT # CLIENT # FP
      message-out-client : CLIENT #  FP
      message-inp-intern : CLIENT # FP
      message-out-intern : CLIENT # FP
      req-deposit        : CLIENT # FP
      req-collect        : CLIENT # FP
      req-transfer       : CLIENT # CLIENT # FP
      identification     : CLIENT
      pay-amount         : FP
      deposit-amount     : FP
      c                  : CLIENT # FP

    processes
      transfer           : CLIENT # CLIENT # TRANSACTION # FP # FP

      transaction        : CLIENT # TRANSACTION # FP # FP
      bank-account       : CLIENT # FP # FP
      mou                : CLIENT # TRANSACTION # FP # FP
      do-cash            : FP # TRANSACTION
      cash               : FP
      bank

   end

    imports Naturals, Integers, Floating-Points-3-4, Floating-Points-Funcs,
            Transactions, Clients

    sets of atoms
     H = { message-inp-intern(n,f), message-out-intern(n,f) |
           n in CLIENT, f in FP }

    communications

       message-inp-intern(n,f) | message-out-intern(n,f) = c(n,f)
       for n in CLIENT, f in FP

    variables
     act-cl,pas-cl       : -> CLIENT
```

```
     amnt,balance,limit    : -> FP
     ds1,ds2     : -> DIGIT-STRING
     i1          : -> INTEGER
     t1,t2       : -> TRANSACTION

    definitions

      transfer(act-cl,pas-cl,t1,balance,limit) =
(
 ((req-transfer(act-cl,pas-cl,amount(t1)) +
   message-inp-client(act-cl,pas-cl,amount(t1))) .
     identification(act-cl) .
     ([and( le(amount(t1), fp(^0)), le(limit, amount(t1) + balance))  = true] ->
          message-out-intern(pas-cl, fp(neg,^1) * amount(t1)) .
          mou(act-cl,t1,balance,limit)
+      [and( le(amount(t1), fp(^0)), le(limit, amount(t1) + balance)) = false] ->
          skip .
          bank-account(act-cl,balance,limit)))
) +
message-inp-intern(act-cl,amount(t1)) . mou(act-cl,t1,balance,limit)

      transaction(act-cl,t1,balance,limit)  =
        ([le(amount(t1), fp(^0)) = true] -> req-collect(act-cl,amount(t1)) +
        [le(amount(t1), fp(^0)) = false] -> req-deposit(act-cl, amount(t1))) .
        identification(act-cl) .
        ([le(limit,amount(t1) + balance) = true] ->
        message-out-intern(cash-c, fp(neg,^1) * amount(t1)) +
        [le(limit,amount(t1) + balance) = false] -> skip )

      do-cash(balance,t1) =
         message-inp-intern(cash-c, amount(t1)) .
         ([le(amount(t1), fp(^0)) = true] -> deposit-amount(amount(t1)) +
         [le(amount(t1), fp(^0)) = false] -> pay-amount(amount(t1))) .
         cash(amount(t1) + balance)

      cash(balance)     =

        sum(t1 in TRANSACTION, do-cash(balance,t1))

      mou(act-cl,t1,balance,limit) =
       ([le(limit, amount(t1) + balance) = true] ->
                message-out-client(act-cl, amount(t1) + balance) .
                bank-account(act-cl, amount(t1) + balance,limit) +
        [le(limit, amount(t1) + balance) = false] ->
                message-out-client(act-cl, balance) .
                bank-account(act-cl, balance, limit))

      bank-account(act-cl,balance,limit) =
       sum(t1 in TRANSACTION, (transaction(act-cl,t1,balance,limit) .
       mou(act-cl,t1,balance,limit) +
       sum(pas-cl in CLIENT,
        ([eq(act-cl, pas-cl) = false] ->
        transfer(act-cl,pas-cl,t1,balance,limit)))))
```

```
    bank =  encaps(H, cash(fp(^1^0^0^0)) ||
     bank-account(client-1,fp(^2^0^0),fp(neg,^5^0^0)))

end BA3
```

3.1 Example trace

```
atom req-collect(client-1,
 fp(neg, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom identification(client-1)
com. c(cash-c, fp(pos, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom pay-amount(fp(pos, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom message-out-client(client-1,
 fp(neg, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
atom req-collect(client-1,
 fp(neg, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom identification(client-1)
com. c(cash-c, fp(pos, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom pay-amount(fp(pos, (^ 0), (((^ 3) ^ 0) ^ 0), int(pos, (^ 3))))
atom message-out-client(client-1,
 fp(neg, (^ 0), (((^ 4) ^ 0) ^ 0), int(pos, (^ 3))))
atom req-collect(client-1,
 fp(neg, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
atom identification(client-1)
com. c(cash-c, fp(pos, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
atom pay-amount(fp(pos, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
atom message-out-client(client-1,
 fp(neg, (^ 0), (((^ 5) ^ 0) ^ 0), int(pos, (^ 3))))
atom req-collect(client-1,
 fp(neg, (^ 0), (((^ 1) ^ 0) ^ 0), int(pos, (^ 3))))
atom identification(client-1)
     skip<1>
atom message-out-client(client-1,
 fp(neg, (^ 0), (((^ 5) ^ 0) ^ 0), int(pos, (^ 3))))
```

We see the skip action indicating that no communication occurred. The balance did not change. One must know ones limitations...

4 Interest calculations

Although our sample bank is now far more realistic then our first one, we still get the feeling something is missing. Why do people bring their money to the bank ? To calculate the interest over an amount of money at a certain rate, we need some notion of *time*. Interest calculations are not performed at every instant, and the time an amount of money must be at a bank to earn interest also fluctuates from bank to bank and between different accounts mutually. In order to specify exactly what happens, we will now analyse the timing of things.

4.1 Timing of interest

We firstly address the minimum observable interval of time. This defines the smallest period of time over which interest calculations are made. In the article of Stins and Schoneveld, [SS93], the length of this interval is one day. When you deposit 100 and collect 100 on the same day, this will not benefit you. Only when you change your balance between days this will have consequences for the interest. So to get interest you should deposit on 23:55 and collect at 0:05; now you will be rewarded by one day of interest. The second important time are the intervals between which the interests are computed. In [SS93] this is each month.

4.2 Process Algebra and time

In normal process algebra, actions are not associated with time. The normal approach to deal with time is to introduce a clock process that generates clock ticks. This is also partly done in [SS93]. All actions that are double underlined must take place in the current time slice. A time slice ends at a clock tick. When we model time this way, we see that time restricts our processes. When a certain action is timed out, i.e. a clock tick is received before we did the action, the action changes into a deadlock. In fact, the clock ticks function as a multi communication action: all actions communicate with time when a time slice has been reached. At that point, all actions are checked against the time slice in which they should occur. If this time slice has already passed, then these actions change into a deadlock in the current time slice. If we want to model this in a process algebra without time, we face the problem of multi communication: in standard process algebra this is not possible.

In our bank, we would like to perform interest calculations in a way that abstracts from the problem of multi communication. The simplest way is to let every bank account process be able to perform an interest calculation at any time at the start of the process. This takes the form of a communication with the cash: the current balance is taken to compute the interest. This interest is added to the balance and subtracted from the cash. Now a bank account needs two new parameters: the credit rate and the debit rate.

The only real problem we face is the obligation to specify all the possible transactions beforehand, including all possible interest calculations! As before, we can't take the sum over an infinite collection of data, so this quirk of the simulator restricts our possibilities. Since this problem is probably only temporal, we will continue by neglecting the impossibility to simulate. It must be noted, however, that all specifications are compiled to til-files, and should be executable after the simulator has been upgraded. So from now on we don't need our auxiliary sort `TRANSACTION` any more.

Specification 4.1

```
process module BA4
begin

 exports
  begin
   atoms
      message-inp-client : CLIENT # CLIENT # FP
      message-out-client : CLIENT #  FP
      message-inp-intern : CLIENT # FP
      message-out-intern : CLIENT # FP
      req-deposit        : CLIENT # FP
      req-collect        : CLIENT # FP
      req-transfer       : CLIENT # CLIENT # FP
      identification     : CLIENT
```

```
      pay-amount         : FP
      deposit-amount     : FP
      c                  : CLIENT # FP

    processes
      transfer           : CLIENT # CLIENT # FP # FP # FP # FP # FP
      transaction        : CLIENT # FP # FP # FP
      interest           : CLIENT # FP # FP # FP # FP
      bank-account       : CLIENT # FP # FP # FP # FP
      mou                : CLIENT # FP # FP # FP
      do-cash            : FP # FP
      cash               : FP
      bank

   end

    imports Naturals, Integers, Floating-Points-3-4, Floating-Points-Funcs,
            Clients

    sets of atoms
     H = { message-inp-intern(n,f), message-out-intern(n,f) |
           n in CLIENT, f in FP }

    communications

       message-inp-intern(n,f) | message-out-intern(n,f) = c(n,f)
       for n in CLIENT, f in FP

    variables
     act-cl,pas-cl             : -> CLIENT
     amnt,balance,limit        : -> FP
     ds1,ds2                   : -> DIGIT-STRING
     i1                        : -> INTEGER
     c-rate,d-rate             : -> FP

    definitions

      transfer(act-cl,pas-cl,amnt,balance,limit,c-rate,d-rate) =
(
 ((req-transfer(act-cl,pas-cl,amnt) +
   message-inp-client(act-cl,pas-cl,amnt)) .
     identification(act-cl) .
     ([and( le(amnt, fp(^0)), le(limit, amnt + balance))  = true] ->
          message-out-intern(pas-cl, fp(neg, ^1) * amnt) .
          mou(act-cl,amnt,balance,limit) .
          bank-account(act-cl,amnt + balance, limit, c-rate, d-rate)
+     [and( le(amnt, fp(^0)), le(limit, amnt + balance)) = false] ->
          skip .
          bank-account(act-cl,amnt,limit,c-rate,d-rate)))
) +
message-inp-intern(act-cl,amnt) .
mou(act-cl,amnt,balance,limit) .
bank-account(act-cl,amnt + balance, limit, c-rate, d-rate)
```

```
      transaction(act-cl,amnt,balance,limit)  =
        ([le(amnt, fp(^0)) = true] -> req-collect(act-cl,amnt) +
        [le(amnt, fp(^0)) = false] -> req-deposit(act-cl, amnt)) .
        identification(act-cl) .
        ([le(limit,amnt + balance) = true] ->
        message-out-intern(cash-c, fp(neg,^1) * amnt) +
        [le(limit,amnt + balance) = false] -> skip )

      do-cash(balance,amnt) =
         message-inp-intern(cash-c, amnt) .
         ([le(amnt, fp(^0)) = true] -> deposit-amount(amnt) +
         [le(amnt, fp(^0)) = false] -> pay-amount(amnt)) .
         cash(balance - amnt)

      cash(balance)     =

        sum(amnt in FP, do-cash(balance,amnt))

      mou(act-cl,amnt,balance,limit) =
       ([le(limit, amnt + balance) = true] ->
                message-out-client(act-cl, amnt + balance)
                 +
        [le(limit, amnt + balance) = false] ->
                message-out-client(act-cl, balance)
                )

      interest(act-cl,balance,limit,c-rate,d-rate) =
       ([le(balance, fp(^0)) = true] ->
                message-out-intern(cash-c, balance * d-rate) .
                message-out-client(act-cl, (balance + (balance * d-rate))).
                bank-account(act-cl, (balance + (balance * d-rate)), limit,
c-rate, d-rate)) +
        ([le(balance, fp(^0)) = false] ->
                message-out-intern(cash-c, balance * c-rate) .
                message-out-client(act-cl, (balance + (balance * c-rate))).
                bank-account(act-cl, (balance + (balance * c-rate)), limit,
c-rate, d-rate))


      bank-account(act-cl,balance,limit,c-rate,d-rate) =
  sum(amnt in FP, (transaction(act-cl,amnt,balance,limit) .
                  mou(act-cl,amnt,balance,limit) .
                  bank-account(act-cl,balance + amnt, limit,
c-rate,d-rate) +
  sum(pas-cl in CLIENT,
   ([eq(act-cl, pas-cl) = false] ->
  transfer(act-cl,pas-cl,amnt,balance,limit,c-rate,d-rate)))
  + interest(act-cl,balance,limit,c-rate,d-rate)))


      bank =  encaps(H, cash(fp(^1^0^0^0)) ||
bank-account(client-1,fp(^2^0^0),fp(neg,^5^0^0),
             fp(pos,^0,^5,int(neg,^1)),fp(pos,^0,^1,int(^0))))
```

```
-- So a bankaccount with 5% normal and 10% debet-interest

end BA4
```

This solution is the most simple one. For a more realistic one, we introduce a new process called `Clock`. For every bank account there is a clock process that generates interest ticks. Whenever a bank account communicates with such a clock, the interest process is started. We assume that all clocks are synchronized.

Specification 4.2

```
process module BA5
begin

 exports
  begin
   atoms
      message-inp-client : CLIENT # CLIENT # FP
      message-out-client : CLIENT #  FP
      message-inp-intern : CLIENT # FP
      message-out-intern : CLIENT # FP
      req-deposit        : CLIENT # FP
      req-collect        : CLIENT # FP
      req-transfer       : CLIENT # CLIENT # FP
      identification     : CLIENT
      pay-amount         : FP
      deposit-amount     : FP
      c                  : CLIENT # FP
      send-tick          : CLIENT
      read-tick          : CLIENT
      tick               : CLIENT

    processes
      clock              : CLIENT
      transfer           : CLIENT # CLIENT # FP # FP # FP # FP # FP
      transaction        : CLIENT # FP # FP # FP
      interest           : CLIENT # FP # FP # FP # FP
      bank-account       : CLIENT # FP # FP # FP # FP
      mou                : CLIENT # FP # FP # FP
      do-cash            : FP # FP
      cash               : FP
      bank

   end

    imports Naturals, Integers, Floating-Points-3-4, Floating-Points-Funcs,
            Clients

    sets of atoms
     H = { message-inp-intern(n,f), message-out-intern(n,f) |
           n in CLIENT, f in FP } +
         {send-tick(n), read-tick(n) | n in CLIENT}

    communications
```

```
       message-inp-intern(n,f) | message-out-intern(n,f) = c(n,f)
       for n in CLIENT, f in FP
       send-tick(n) | read-tick(n) = tick(n) for n in CLIENT

    variables
     act-cl,pas-cl             : -> CLIENT
     amnt,balance,limit        : -> FP
     ds1,ds2                   : -> DIGIT-STRING
     i1                        : -> INTEGER
     c-rate,d-rate             : -> FP

    definitions

      transfer(act-cl,pas-cl,amnt,balance,limit,c-rate,d-rate) =
(
 ((req-transfer(act-cl,pas-cl,amnt) +
   message-inp-client(act-cl,pas-cl,amnt)) .
     identification(act-cl) .
     ([and( le(amnt, fp(^0)), le(limit, amnt + balance))  = true] ->
          message-out-intern(pas-cl, fp(neg, ^1) * amnt) .
          mou(act-cl,amnt,balance,limit) .
          bank-account(act-cl,amnt + balance, limit, c-rate, d-rate)
+     [and( le(amnt, fp(^0)), le(limit, amnt + balance)) = false] ->
          skip .
          bank-account(act-cl,balance,limit,c-rate,d-rate)))
) +
message-inp-intern(act-cl,amnt) .
mou(act-cl,amnt,balance,limit) .
bank-account(act-cl,amnt + balance, limit, c-rate, d-rate)

      transaction(act-cl,amnt,balance,limit)  =
        ([le(amnt, fp(^0)) = true] -> req-collect(act-cl,amnt) +
        [le(amnt, fp(^0)) = false] -> req-deposit(act-cl, amnt)) .
        identification(act-cl) .
        ([le(limit,amnt + balance) = true] ->
        message-out-intern(cash-c, fp(neg,^1) * amnt) +
        [le(limit,amnt + balance) = false] -> skip )

      do-cash(balance,amnt) =
         message-inp-intern(cash-c, amnt) .
         ([le(amnt, fp(^0)) = true] -> deposit-amount(amnt) +
         [le(amnt, fp(^0)) = false] -> pay-amount(amnt)) .
         cash(balance - amnt)

      cash(balance)     =

        sum(amnt in FP, do-cash(balance,amnt))

      mou(act-cl,amnt,balance,limit) =
       ([le(limit, amnt + balance) = true] ->
               message-out-client(act-cl, amnt + balance) .
        [le(limit, amnt + balance) = false] ->
               message-out-client(act-cl, balance))
```

```
interest(act-cl,balance,limit,c-rate,d-rate) =
 ([le(balance, fp(^0)) = true] ->
         message-out-intern(cash-c, balance * d-rate) .
         message-out-client(act-cl, (balance + (balance * d-rate))).
         bank-account(act-cl, (balance + (balance * d-rate)), limit,
                       c-rate, d-rate)) +
  ([le(balance, fp(^0)) = false] ->
         message-out-intern(cash-c, balance * c-rate) .
         message-out-client(act-cl, (balance + (balance * c-rate))).
         bank-account(act-cl, (balance + (balance * c-rate)), limit,
                      c-rate, d-rate))


bank-account(act-cl,balance,limit,c-rate,d-rate) =
      sum(amnt in FP, (transaction(act-cl,amnt,balance,limit) .
      mou(act-cl,amnt,balance,limit) .
      bank-account(act-cl,balance + amnt, limit, c-rate, d-rate) +
      sum(pas-cl in CLIENT,
          ([eq(act-cl, pas-cl) = false] ->
              transfer(act-cl,pas-cl,amnt,balance,limit,c-rate,d-rate)))
                +
      read-tick(act-cl) . interest(act-cl,balance,limit,c-rate,d-rate)))


clock(act-cl) = send-tick(act-cl) . clock(act-cl)
bank =  encaps(H, cash(fp(^1^0^0^0)) || clock(client-1) ||
                bank-account(client-1,fp(^2^0^0),fp(neg,^5^0^0),
                fp(pos,^0,^5,int(neg,^1)),fp(pos,^0,^1,int(^0))))

-- a bankaccount with 5% normal and 10% debit-interest

end BA5
```

4.3 Adding the Database

The last refinement we will undertake in this article is to specify the database. The new specification differs on more than one point from the old one. Therefore we will say more about it.

In general, the bank account remains the same: transfers and transactions can be done at any instant. These changes are saved in the database, but no message is sent to the owner yet.

Only when the `Clock` process communicates with the `bank-account` process, the interest will be computed from the database. To this end, the record of a transaction has a (simple) time stamp denoted by a natural number. Whenever a new record is added, it will get a time stamp one more than the last added record. Also the interest transactions are saved. In this way, there is always an interest transaction with largest time stamp. As to compute the new interest amount, the database first looks for this interest transaction with greatest time stamp. All transactions with greater time stamps are inspected then to compute the new interest.

After this process, the `Send-message` process is started. In [SS93] this process runs parallel with the `Interest` process. This leads to undesired behaviour. In that way it is possible that the owner receives two interest transactions in one message or none at all, depending on the order in which the processes finish. This seems rather strange to us. Therefore we chose to do these things sequentially. The `Send-message` process sends all messages that have not been sent yet to the owner. To keep track of which transactions were mailed already, the records in the database have a boolean telling whether the transaction had been processed. After the sending of messages the booleans are updated.

Each bank account has its own *Clock* process. We assume that all the `Clock` processes run synchronously.

The database we use is a very simple one. We have made no considerations with regard to speed or efficiency. The only reason why we did specify it was that maybe in the future the simulator can be used to simulate the specification. In general, a record in the database is an n-tuple (client,time,balance,amount,type,sent) having the following format:

Abstract Data Type	Meaning
CLIENT	Client's identifier
NATURAL	Time stamp
FP	Balance after current transaction
FP	Amount of change
TYPE	Which type of transaction
BOOLEAN	Sent bit

Where TYPE $\in$ {TRANSACTION, TRANSFER, CREDITINTEREST, DEBITINTEREST}

We split the specification in submodules to keep things orderly. We see here a great advantage of the used formalism: things that belong together can be specified together, private functions can be hidden and data can be specified apart from the processes that use them. We have put the specification of the Database datatype in an appendix. Only four functions of that datatype are used. These are :

1. `calc-dinterest :  CLIENT # FP # NATURAL # DB -> FP`

2. `calc-cinterest :  CLIENT # FP # NATURAL # DB -> FP`

3. `col-rec :  DB # CLIENT -> REC-LIST`

4. `update-records :  CLIENT # DB -> DB`

The `calc` functions calculate the interest from a certain client. `col-rec` collects the records from a certain client that have not been mailed yet. `update-records` updates the records when a mailing has been done.

By grouping together all communications it will be easier to modify the specification if the communications become more time critical in future extensions. All communications are between a bank account process and another process.

Specification 4.3

```
-- This module specifies the clock-process.

process module Clock
begin
 exports
  begin
   atoms
    tick-out            : CLIENT # NATURAL
   processes
    Clock               : CLIENT # NATURAL
```

```
  end

  imports
   Naturals, Clients

  variables
   c            : -> CLIENT
   n1           : -> NATURAL

  definitions
   Clock(c,n1)  =
     tick-out(c,n1) . Clock(c,n1 + nat(^1))

end Clock

-- This module specifies all communications.
process module Communications
begin
 exports
  begin
   atoms
-- First the Clock-communications

    tick-inp            : CLIENT # NATURAL
    c-tick              : CLIENT # NATURAL

-- The Database-communications

    send-interest-request      : CLIENT # NATURAL # FP # TYPE
    read-interest-request      : CLIENT # NATURAL # FP # TYPE
    c-interest-request         : CLIENT # NATURAL # FP # TYPE
    send-interest              : CLIENT # FP
    read-interest              : CLIENT # FP
    c-interest                 : CLIENT # FP
    add-out                    : CLIENT # FP # FP # TYPE
    add-inp                    : CLIENT # FP # FP # TYPE
    c-add                      : CLIENT # FP # FP # TYPE
    collect-records-inp        : CLIENT
    collect-records-out        : CLIENT
    c-records-request          : CLIENT
    send-records               : CLIENT # REC-LIST
    read-records               : CLIENT # REC-LIST
    c-records                  : CLIENT # REC-LIST

-- The Cash-communications

    message-intern-inp         : CLIENT # FP
    message-intern-out         : CLIENT # FP
    c                          : CLIENT # FP

  end

  imports
```

```
   Clients, Naturals, Floating-Points-3-4, Type, Rec-list, Clock

  communications
   send-interest-request(c,n,f,t) | read-interest-request(c,n,f,t) =
    c-interest-request(c,n,f,t) for c in CLIENT, n in NATURAL, f in FP,
    t in TYPE
   send-interest(c,f) | read-interest(c,f) = c-interest(c,f)
    for c in CLIENT, f in FP
   add-out(c,a,b,t) | add-inp(c,a,b,t) = c-add(c,a,b,t)
    for c in CLIENT, a in FP, b in FP, t in TYPE
   collect-records-inp(c) | collect-records-out(c) =
   c-records-request(c) for c in CLIENT
   send-records(c,rl) | read-records(c,rl) = c-records(c,rl)
    for c in CLIENT, rl in REC-LIST
   message-intern-inp(c,a) | message-intern-out(c,a) = c(c,a)
    for c in CLIENT, a in FP

end Communications


process module Database
begin
 exports
  begin
   atoms
    return-out          : FP
    return-out          : REC-LIST
    tick-db-inp

   processes
    Database            : DB # NATURAL
  end

  imports
   Clients, Naturals, Floating-Points-3-4, Db-rec, Rec-list, Db,
   Communications

   variables
    c1          : -> CLIENT
    n1,n2       : -> NATURAL
    f1,f2       : -> FP
    db          : -> DB
    t1          : -> TYPE
    rl          : -> REC-LIST

   definitions
    Database(db,n1)     =
     sum(c1 in CLIENT, sum(f1 in FP, sum(f2 in FP, sum(t1 in TYPE,
      add-inp(c1,f1,f2,t1) .
      Database(insert(db,cons-rec(c1,n1,f1,f2,t1,false),n1), n1 + nat(^1)))))) +

     sum( n2 in NATURAL, sum(c1 in CLIENT, sum(dr in FP,
      read-interest-request(c1,n2,dr,DEBITINTEREST)
     . return-out(calc-dinterest(c1,dr,n2,db))))) .
```

```
     Database(db, n1 ) +

     sum(n2 in NATURAL, sum(c1 in CLIENT, sum(cr in FP,
     read-interest-request(c1,n2,cr,CREDITINTEREST) .
     return-out(calc-cinterest(c1,cr,n2,db)))))
     . Database(db, n1) +

     sum(c1 in CLIENT, collect-records-inp(c1) .
     send-records(c1, col-rec(db,c1)) . Database(update-records(c1,db),n1))

end Database

-- This module specifies the new Bank-account process.
process module Bank-account
begin
 exports
  begin
   atoms
    request                    : FP
    request                    : CLIENT # CLIENT # FP
    identification
    message-client-inp         : CLIENT # CLIENT # FP
    message-client-out         : CLIENT # REC-LIST
    pay-amount                 : FP
    deposit-amount             : FP

   processes
    Cash                       : FP
    Transaction                : CLIENT # FP # FP # FP # FP
    Transfer                   : CLIENT # CLIENT # FP # FP # FP # FP
    Interest                   : CLIENT # FP # FP # FP # FP
    Sendmessages               : CLIENT # FP # FP # FP # FP
    Ba                         : CLIENT # FP # FP # FP # FP
    System
  end

 imports
  Clients, Naturals, Db, Floating-Points-3-4, Type, Clock, Communications,
  Database

 sets of atoms
  H =
      { tick-inp(c1,n1), tick-out(c1,n1) | c1 in CLIENT, n1 in NATURAL} +

      { send-interest-request(c1,n1,f1,t1),
read-interest-request(c1,n1,f1,t1) | c1 in CLIENT, n1 in NATURAL, f1 in FP,
t1 in TYPE} +

      { send-interest(c1,f1), read-interest(c1,f1) | c1 in CLIENT, f1 in
FP} +

      { add-out(c1,f1,f2,t1), add-inp(c1,f1,f2,t1) | c1 in CLIENT, f1 in
FP, f2 in FP, t1 in TYPE} +
```

```
      {collect-records-inp(c1), collect-records-out(c1) | c1 in
CLIENT} +

      { send-records(c1,rl), read-records(c1,rl) | c1 in CLIENT, rl in
REC-LIST} +

      { message-intern-inp(c1,f1), message-intern-out(c1,f1) | c1 in CLIENT,
f1 in FP}


 variables
  a,a1,b,l,cr,dr                : -> FP
-- amount, balance, limit, credit-rate, debit-rate

  c1,c2                 : -> CLIENT
  n1,st                 : -> NATURAL

 definitions
  Cash(b)               =
   sum(a in FP,
    message-intern-inp(cash-c, a) .
    ([le(a,fp(^0)) = true] -> pay-amount(a) +
     [le(a,fp(^0)) = false] -> deposit-amount(a)) .
    Cash(b+a))

  Ba(c1,b,l,cr,dr)      =
   Transaction(c1,b,l,cr,dr)    +
   sum(c2 in CLIENT, Transfer(c1,c2,b,l,cr,dr)) +
   Interest(c1,b,l,cr,dr)


-- The interest process is triggered by the clock

  Interest(c1,b,l,cr,dr)        =
   sum(n1 in NATURAL, tick-inp(c1,n1) .
   send-interest-request(c1,n1,dr, DEBITINTEREST) .
   sum(a1 in FP, read-interest(c1,a1) .
   add-out(c1,b+a1,a1,DEBITINTEREST) .
   send-interest-request(c1,n1,dr, CREDITINTEREST) .
   sum(a2 in FP, read-interest(c1,a2) .
   add-out(c1,b+a2,a2,CREDITINTEREST) .
   Sendmessages(c1,b+a1+a2,l,cr,dr))))

  Transaction(c1,b,l,cr,dr)     =
   sum(a in FP, request(a) . identification .
    (([le(l,a+b) = true] ->
      message-intern-out(cash-c, fp(neg, ^1) * a) .
      add-out(c1,b+a,a, TRANSACTION) .
      Ba(c1, b+a, l, cr,dr)) +
    ([le(l,a+b) = false] ->
      skip . Ba(c1, b, l, cr, dr))))

   Transfer(c1,c2,b,l,cr,dr)    =
    sum(a in FP,
```

```
     (
      (request(c1,c2,a) + message-client-inp(c1,c2,a)) .
      identification .
      ([and(ge(b+a,l), lt(a,fp(^0))) = true] ->
        message-intern-out(c2,fp(neg, ^1) * a) .
        add-out(c1, b+a, a, TRANSFER) . Ba(c1,b+a,l,cr,dr)) +
      ([and(ge(b+a,l), lt(a,fp(^0))) = false] ->
         skip . Ba(c1,b,l,cr,dr))
-- So balance b did not change in this case

     ) +
     (
      message-intern-inp(c1,a) . add-out(c1,b+a,a,TRANSFER)
      . Ba(c1,b+a,l,cr,dr)
     ))

-- The sendmessage process is triggered by the Interest-process
-- so these two processes are now sequential. It is difficult to interleave
-- them because one of them modifies the database while the other one reads
-- it. I chose to do first the
-- updating of interest and afterwards sending of messages because this is
-- more natural in my view. In the article these processes are synchronized
-- by the discrete time, but nothing is said about the possible conflict of
-- sending no interest message because the following can happen :
-- suppose between two interest ticks no balance-change was made.
-- The last message the client received was a message without interest.
-- This can happen if the message process finished before the interest
-- process. Now the next interest-tick the client receives two interest
-- messages : one that remained to be sent, and one from the new period.
-- Now the user may ask three things :
-- 1. Why did I not receive interest the first time?
-- 2. Why do I receive two interest updates while my balance did not
-- change?
-- 3. Why couldn't they add the two interest amounts directly?

    Sendmessages(c1,b,l,cr,dr) =
      collect-records-out(c1) .
      sum(rl in REC-LIST, read-records(c1,rl) .
      message-client-out(c1,rl)) .
      Ba(c1,b,l,cr,dr)

    System = encaps(H, sum(b in FP, Cash(b)) ||
                    merge(c in CLIENT,
                     sum(b in FP,
                      sum(l in FP,
                       sum(cr in FP,
                        sum(dr in FP, Ba(c,b,l,cr,dr)))))) ||
                     sum(db in DB,
                      sum(n in NATURAL, Database(db,n))) ||
                     merge(c in CLIENT, sum(n in NATURAL, Clock(c,n))))

end Bank-account
```

5 Conclusions and further work

In this article we succeeded in specifying a bank account in PSF, an executable process specification formalism. We followed as close as possible [SS93]. Up to the solvency check our specification remained executable, we used a PSF tool *sim* to simulate the specifications.

After that we only specified and compiled the specifications. When dealing with discrete time, we had to make some concessions to be able to specify. We chose to let every `bank-account` process have an interest clock that triggers the interest calculations and the sending of messages. We used another PSF tool *trs* to test our database specification.

Many aspects of a bank account remained outside the scope of the article, for instance opening and closing an account, and foreign currencies. During our work we encountered limitations of the current PSF toolkit, like the inability to simulate alternative compositions when the number of alternatives is infinite. We think this needs attention too.

Acknowledgements

We would like to thank the following people (in alphabetical order) for proofreading the manuscript and giving useful comments: Jan Bergstra, Peter H. Polak, Huib Verweij and Jos van Wamel. Joris Hillebrand created the front page and helped with the preparation.

Appendix : Database specification

```
-- This module specifies the sort DB-REC, used as a record in the TABLE.
-- cons-rec(client,save-tick,balance,amount,type,sent) means the balance after
-- change by amount at time save-tick with reason type.
-- sent is a boolean indicating whether the user received the message
-- telling the change.
-- An equality function is defined as should be for each sort one defines.

data module Db-rec
begin
 exports
  begin

   sorts
    DB-REC

   functions
    cons-rec    : CLIENT # NATURAL #  FP # FP # TYPE # BOOLEAN -> DB-REC
-- cons-rec means construct-record
    empty-rec   : -> DB-REC
    eq          : DB-REC # DB-REC -> BOOLEAN

  end

 imports
   Booleans, Naturals, Clients, Type, Floating-Points-Funcs, Floating-Points-3-4

 variables
  c1,c2         : -> CLIENT
  st1,st2       : -> NATURAL
  b1,b2         : -> FP
  a1,a2         : -> FP
  t1,t2         : -> TYPE
  fl1,fl2       : -> BOOLEAN

 equations

 [D1]   eq(cons-rec(c1,st1,b1,a1,t1,fl1), cons-rec(c2,st2,b2,a2,t2,fl2)) =
        and(and(and(and(and(eq(c1,c2),
                        eq(st1,st2)),
                        eq(b1,b2)),
                        eq(a1,a2)),
                        eq(t1,t2)),
                        eq(fl1,fl2))

end Db-rec

data module Rec-list
begin
 exports
  begin
   functions
    empty-list          : -> REC-LIST
```

```
  end

 imports
  Strings {String-Parameters bound by [ITEM -> DB-REC] to Db-rec renamed by
[STRING -> REC-LIST]}
end Rec-list

-- This module specifies the data-type TYPE.

data module Type
begin
 exports
  begin

   sorts
    TYPE

   functions
     TRANSACTION        : -> TYPE
     TRANSFER           : -> TYPE
     DEBITINTEREST      : -> TYPE
     CREDITINTEREST     : -> TYPE
     eq                 : TYPE # TYPE -> BOOLEAN

  end

 imports
  Booleans

 variables
  t1,t2        : -> TYPE

 equations

 [T1]   eq(t1,t1)       = true
 [T2]   eq(TRANSACTION, TRANSFER) = false
 [T3]   eq(TRANSACTION,DEBITINTEREST) = false
 [T4]   eq(TRANSACTION,CREDITINTEREST) = false
 [T5]   eq(TRANSFER,DEBITINTEREST) = false
 [T6]   eq(TRANSFER,CREDITINTEREST) = false
 [T7]   eq(DEBITINTEREST,CREDITINTEREST) = false
 [T8]   eq(TRANSFER,TRANSACTION) = false
 [T9]   eq(DEBITINTEREST,TRANSACTION) = false
 [T10]  eq(CREDITINTEREST,TRANSACTION) = false
 [T11]  eq(DEBITINTEREST,TRANSFER) = false
 [T12]  eq(CREDITINTEREST,TRANSFER) = false
 [T13]  eq(CREDITINTEREST,DEBITINTEREST) = false

end Type

-- This module specifies the Database. It uses the TABLE from [vW93].
-- The KEY is a NATURAL; the ITEM is a DB-REC.

data module Db
```

```
begin
 exports
  begin
   functions
    calc-dinterest      : CLIENT # FP # NATURAL # DB -> FP
    calc-cinterest      : CLIENT # FP # NATURAL # DB -> FP
    col-rec             : DB # CLIENT -> REC-LIST
    update-records      : CLIENT # DB -> DB
  end

imports
  Tables {Keys bound by [KEY -> NATURAL, eq -> eq] to Naturals
  Items bound by [ITEM -> DB-REC, table-nil -> empty-rec] to Db-rec
  renamed by [TABLE -> DB]},
  Naturals, Clients, Floating-Point-Repr,Floating-Points-3-4, Rec-list

 functions
  find-last-cinterest   : CLIENT # DB -> NATURAL
  find-last-dinterest   : CLIENT # DB -> NATURAL
  comp-cinterest        : CLIENT # FP # NATURAL # DB -> FP
  comp-dinterest        : CLIENT # FP # NATURAL # DB -> FP

  -- These functions find the latest interest record of client CLIENT
  -- and compute the actual new interest starting from that point.

 variables
  f1,f2,f3      : -> FP
  c1,c2         : -> CLIENT
  n1,n2,n3      : -> NATURAL
  db            : -> DB
  fl1           : -> BOOLEAN
  t1            : -> TYPE
  t             : -> DB

 equations
  [D1]  find-last-cinterest(c1,empty-table) = nat(^0)
  [D2]  find-last-cinterest(c1,entry(n1,
         cons-rec(c2,n2,f1,f2,t1,fl1))~t) = find-last-cinterest(c1,t) when
         or(not(eq(c1,c2)),not(eq(t1,CREDITINTEREST))) = true
  [D3]  find-last-cinterest(c1,entry(n1,
           cons-rec(c1,n2,f1,f2,CREDITINTEREST,fl1))~t) =
           max(n2, find-last-cinterest(c1,t))

  [D4]  find-last-dinterest(c1,empty-table) = nat(^0)
  [D5]  find-last-dinterest(c1,entry(n1,
          cons-rec(c2,n2,f1,f2,t1,fl1))~t) = find-last-dinterest(c1,t) when
          or(not(eq(c1,c2)),not(eq(t1,DEBITINTEREST))) = true
  [D6]  find-last-dinterest(c1,entry(n1,
          cons-rec(c1,n2,f1,f2,DEBITINTEREST,fl1))~t) =
          max(n2, find-last-dinterest(c1,t))

  [D7]  calc-dinterest(c1,f1,n1,db) =
          comp-dinterest(c1,f1,find-last-dinterest(c1,db),db)
  [D8]  calc-cinterest(c1,f1,n1,db) =
```

```
        comp-cinterest(c1,f1,find-last-cinterest(c1,db),db)

  [D9] comp-dinterest(c1,f1,n1,empty-table) = fp(^0)
  [D10] comp-cinterest(c1,f1,n1,empty-table) = fp(^0)

-- First the case that there has not been an interest yet

  [D11] comp-dinterest(c1,f1,nat(^0),entry(n2,
        cons-rec(c2,n3,f2,f3,t1,fl1))~t) = (f1 * f2) +
        comp-dinterest(c1,f1,nat(^0),t) when
        lt(f2,fp(int(^0))) = true

  [D11a] comp-dinterest(c1,f1,nat(^0),entry(n2,
        cons-rec(c2,n3,f2,f3,t1,fl1))~t) = comp-dinterest(c1,f1,nat(^0),t)
        when lt(f2,fp(int(^0))) = false

  [D12] comp-cinterest(c1,f1,nat(^0),entry(n2,
         cons-rec(c2,n3,f2,f3,t1,fl1))~t) = (f1 * f2) +
         comp-cinterest(c1,f1,nat(^0),t) when
         ge(f2,fp(int(^0))) = true

  [D12a] comp-cinterest(c1,f1,nat(^0),entry(n2,
         cons-rec(c2,n3,f2,f3,t1,fl1))~t) =
          comp-cinterest(c1,f1,nat(^0),t) when
          ge(f2,fp(int(^0))) = false
-- Then the normal case

  [D13] comp-dinterest(c1,f1,n1,entry(n2,
         cons-rec(c2,n3,f2,f3,t1,fl1))~t) = (f1 * f2) +
         comp-dinterest(c1,f1,n1,t) when
          and(gt(n3,n1), lt(f2, fp(int(^0)))) = true

  [D14] comp-dinterest(c1,f1,n1,entry(n2,
          cons-rec(c2,n3,f2,f3,t1,fl1))~t) = comp-dinterest(c1,f1,n1,t) when
          or(and(gt(n3,n1), lt(f2, fp(int(^0)))), eq(n1,nat(^0))) = false

  [D15] comp-cinterest(c1,f1,n1,entry(n2,
           cons-rec(c2,n3,f2,f3,t1,fl1))~t) = (f1 * f2) +
           comp-cinterest(c1,f1,n1,t) when
             and(gt(n3,n1), ge(f2, fp(int(^0)))) = true

  [D16] comp-cinterest(c1,f1,n1,entry(n2,
             cons-rec(c2,n3,f2,f3,t1,fl1))~t) = comp-cinterest(c1,f1,n1,t)
             when or(and(gt(n3,n1), ge(f2, fp(int(^0)))), eq(n1,nat(^0))) = false

  [D17] col-rec(empty-table,c1) = empty-list

  [D20] col-rec(entry(n2, cons-rec(c1,n1,f1,f2,t1,fl1))~t,c2) =
        col-rec(t,c1)^cons-rec(c1,n1,f1,f2,t1,fl1) when
        and(not(fl1), eq(c1,c2)) = true

  [D21] col-rec(entry(n2,cons-rec(c1,n1,f1,f2,t1,fl1))~t,c2) =
        col-rec(t,c2) when and(not(fl1), eq(c1,c2)) = false
```

```
-- update-records updates the sent-flag for all records of client c1

  [B22] update-records(c1,empty-table) = empty-table
  [B23] update-records(c1,entry(n2,cons-rec(c2,n1,f1,f2,t1,fl1))~t) =
         entry(n2,cons-rec(c2,n1,f1,f2,t1,true))~update-records(c1,t) when
         and(eq(c1,c2),not(fl1)) = true
  [B24] update-records(c1,entry(n2,cons-rec(c2,n1,f1,f2,t1,fl1))~t) =
         entry(n2,cons-rec(c2,n1,f1,f2,t1,fl1))~update-records(c1,t)
         when and(eq(c1,c2),not(fl1)) = false

end Db
```

Testing Db.psf

We did some simple testing to check the correct functioning of our database. We used another PSF tool *trs* to rewrite some test terms. We found some bugs in the specification and corrected them. The test results follow:

```
calc-cinterest(client-1,
fp(pos,(^ 0),(^ 1),int((^ 0))),nat((^6)),
insert(
 insert(
  insert(
   insert(
    insert(empty-table,
cons-rec(
 client-1,
 nat((^1)),
 fp((^ 0)),
 fp((^0)),
 TRANSACTION,
 false),
             nat((^1))),
cons-rec(
 cash-c,
 nat((^ 1)),
 fp((^ 0)),
 fp((^0)),
 TRANSACTION,
 false),
             nat((^2))),
cons-rec(
 client-1,
 nat((^ 2)),
 fp(((^ 1) ^0)),
 fp(((^ 1) ^ 0)),
 TRANSACTION,
 false),
             nat((^ 3))),
cons-rec(
 client-1,
 nat((^2)),
 fp((^ 1)),
 fp(((^ 1) ^ 1)),
```

```
 CREDITINTEREST,
 false),
              nat((^4))),
cons-rec(
 client-1,
 nat((^ 3)),
 fp(((^ 2) ^ 0)),
 fp(((^ 3) ^1)),
 TRANSACTION,
 false),
              nat((^ 5))))

    = fp(pos,(^ 0),(((^ 2) ^ 0) ^0),int(pos,(^ 1)))
```

Since we used this instantiation of the database in every testcase, we will abbreviate it to db in all following test results.

```
calc-dinterest(client-1,
fp(pos,(^ 0),(^ 1),int((^ 0))),nat((^6)),db)
    = fp(pos,(^ 0),(^ 0),int(pos,(^ 0)))

col-rec(db,client-1) =

 ((empty-list ^
    cons-rec(
     client-1,
     nat((^ 3)),
     fp(pos,(^ 0),((^ 2) ^ 0),int(pos,(^2))),
     fp(pos,(^ 0),((^ 3) ^ 1),int(pos,(^ 2))),
     TRANSACTION,
     false)) ^
    cons-rec(
     client-1,
     nat((^ 2)),
     fp(pos,(^ 0),(^ 1),int(pos,(^ 1))),
     fp(pos,(^0),((^ 1) ^ 1),int(pos,(^ 2))),
     CREDITINTEREST,
     false)) ^
    cons-rec(
     client-1,
     nat((^ 1)),
     fp(pos,(^ 0),(^ 0),int(pos,(^ 0))),
     fp(pos,(^0),(^ 0),int(pos,(^ 0))),
     TRANSACTION,
     false)

update-records(client-1,db) =
entry(nat((^ 1)),
 cons-rec(
  client-1,
  nat((^ 1)),
  fp(pos,(^ 0),(^ 0),int(pos,(^0))),
  fp(pos,(^ 0),(^ 0),int(pos,(^ 0))),
  TRANSACTION,
```

```
  true)) ~
(entry(nat((^2)),
 cons-rec(
  cash-c,
  nat((^ 1)),
  fp(pos,(^ 0),(^ 0),int(pos,(^ 0))),
  fp(pos,(^0),(^ 0),int(pos,(^ 0))),
  TRANSACTION,
  false)) ~
(entry(nat((^3)),
 cons-rec(
  client-1,
  nat((^ 2)),
  fp(pos,(^ 0),((^ 1) ^ 0),int(pos,(^2))),
  fp(pos,(^ 0),((^ 1) ^ 0),int(pos,(^ 2))),
  TRANSACTION,
  true)) ~
(entry(nat((^ 4)),
  cons-rec(
   client-1,
   nat((^ 2)),
   fp(pos,(^ 0),(^1),int(pos,(^ 1))),
   fp(pos,(^ 0),((^ 1) ^ 1),int(pos,(^2))),
   CREDITINTEREST,
   true)) ~
(entry(nat((^ 5)),
 cons-rec(
  client-1,
  nat((^3)),
  fp(pos,(^ 0),((^ 2) ^ 0),int(pos,(^ 2))),
  fp(pos,(^ 0),((^ 3) ^1),int(pos,(^ 2))),
  TRANSACTION,
  true)) ~ empty-table))))
```

References

[BHK87] J. A. Bergstra, J. Heering, and P. Klint. Asf-an algebraic specification formalism. Technical report, Report CS-R8705, Programming Research Group, University of Amsterdam, 1987.

[BW91] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1991.

[Mau91] Sjouke Mauw. *PSF A Process Specification Formalism*. PhD thesis, University of Amsterdam, 1991.

[SS93] Arjen Schoneveld and Arjen Stins. Specification of a bankaccount with process algebra. Technical report, Report P9307, Programming Research Group, University of Amsterdam, 1993.

[Vel91] G. J. Veltink. The psf toolkit. Technical report, Report P9107, Programming Research Group, University of Amsterdam, 1991.

[vW93] J. J. van Wamel. A library for psf. Technical report, Report P9301, Programming Research Group, University of Amsterdam, 1993.