A COWS specification of the Finance case study (version 4.6.2)

Technical Report

Author: Francesco Tiezzi

Date: November 13, 2009
Institute: Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

1 Introduction

In this document we report a COWS [3] specification of a large case study from the financial domain which is currently investigated within the EU project SENSORIA. We start by providing in Section 2 an informal specification of the scenario (a more detailed UML4SOA [4] one is reported in [1]). Then, we introduce in Section 3 the syntax of COWS accepted by CMC [2] that has been used in in Section 4 to formally specify the case study.

2 The Finance case study

The considered service provides a customer company with the possibility to ask for a loan to a bank and then orchestrates the necessary steps for processing the credit request, which may involve an evaluation by either a clerk or a supervisor before a contract proposal is sent to the customer. Initially, the customer logins to the *credit request* service by providing his username and password, then uploads the necessary data for his request. More specifically, he firstly provides the credit data (e.g. the desired amount), then the securities of the loan and his balance. When the request is completely filled by the customer, the service calculates the rating of the customer request, by resorting on a (possibly) external service, and takes a decision on it. The decision can be either to immediately accept the request, if the rating value is "*aaa*", or to accept or decline it according to a clerk or a supervisor evaluation, if the rating value is "*bbb*" or "*ccc*", respectively. In case of a decline, the possibility to update the data and restart the request processing is given to the customer. At any moment the customer may require to abort the process. If this happens, the process terminates and, in case, the request data are deleted. This requires execution of *compensation activities* to semantically rollback the action of storing the request data performed by the involved services. This prevents such services from maintaining information of already aborted requests.

3 Syntax accepted by CMC

The syntax of COWS accepted by CMC is presented in Table 1. *Killer labels* (ranged over by `k`, `k'`, ...) start with lower case letters and can only be used as argument of kill activities. *Variables* (ranged over by `X`, `Y`, ...) start with capital letters. *Service identifiers* (ranged over by `A`, `A'`, ...) start with capital letters and each of them has a fixed non-negative arity. *Names* (ranged over by `n`, `m`,...,`p`,`p'`,...,`o`,`o'`, ...) start with lower case letters. *Values* (ranged over by `v`, `v'`, ...) are integer numbers, booleans, or names. *Identifiers* (ranged over by `u`, `u'`, ...) are variables or names. The arguments of a receive-guarded choice must be receive activities. The expression operators + and = are defined as follows: if both e_1 and e_2 are evaluated as integer numbers then the evaluation of $e_1 + e_2$ returns the integer number corresponding to their sum, otherwise it returns the name corresponding to their concatenation; if both e_1 and e_2 are evaluated as values then the evaluation of $e_1 = e_2$ returns the boolean `true` if these values are the same value, otherwise it returns the boolean `false`.

The let construct permits to define services in a modular style, thus facilitating re-use of the same 'service code'. `let` `A`(*fparams*) $=s$ `in` s' `end` behaves like s', where calls to `A` can occur; a service call `A`(*aparams*) occurring in s' behaves like the service obtained from s by replacing the formal parameters *fparams* with the corresponding actual parameters *aparams*.

The syntax of SocL accepted by CMC slightly differs from that presented in [2] mainly for what concern the notation to indicate correlation variables. In fact, given a variable `var`, its binding occurrence (i.e. var in SocL) is written `$var`, while its free occurrences are written `%var`. Moreover, the logical operators $\vee$ and $\neg$ are written `or` and `not`, respectively.

s	::=		(services)
		`nil`	(empty activity)
	|	`kill(k)`	(kill)
	|	`u.u'!` <*args*>	(invoke)
	|	`p.o?` <*params*> . s	(receive)
	|	$s_1 + \ldots + s_n$	(receive-guarded choice)
	|	$s_1 \mid s_2$	(parallel composition)
	|	$\{ s \}$	(protection)
	|	[n♯] s	(name delimitation)
	|	[k] s	(kill delimitation)
	|	[X] s	(variable delimitation)
	|	$* s$	(replication)
	|	A(*aparams*)	(call)
	|	`let` A(*fparams*) =s `in` s' `end`	(let construct)
e	::=	X | v | $e_1 + e_2$ | $e_1 = e_2$	(expressions)
args	::=	e | *args*, *args*	(invoke arguments)
params	::=	X | v	(receive parameters)
	|	*params*, *params*	
fparams	::=	X | n | k	(formal parameters)
	|	*fparams*, *fparams*	
aparams	::=	X | v | k	(actual parameters)
	|	*aparams*, *aparams*	

Table 1: Syntax of COWS accepted by CMC

4 CMC specification of the case study

The complete specification of the Finance case study (version 4.6.2) written in the syntax accepted by CMC is as follows.

```
let

Finalize(Id) =
  -- Notifies the portal that the process is terminated
  portal.goodbye!<Id>


Accept(Id,creditManagement,update,desired,Result,RatingData) =
  -- Requires the credit management service to generate the offer data
  creditManagement.generateOffer!<Id,RatingData>
  | [AgreementData]
    creditReq.generateOffer?<Id,AgreementData>.
      ( -- Forwards the offer data to the customer and asks for an acceptance
        portal.offerToClient!<Id,AgreementData>
        | [Accepted]
          creditReq.offerToClient?<Id,Accepted>.
            [if#][then#][end#]
            ( -- Check the customer acceptance
              if.then!<Accepted>
              | if.then?<false>.
                    -- The customer declines the offer:
                    -- skips to the finalize step
```

```
                update.desired!<false>
            + if.then?<true>.
                -- The customer accepts the offer:
                -- notifies the credit management service that the customer
                -- accepted the offer
                ( creditManagement.acceptOffer!<Id,Accepted>
                  | portal.acceptOffer?<Id>.
                      -- Goes to the finalize step
                      update.desired!<false>
                )
          )
    )


Decline(Id,creditManagement,update,desired,Result,RatingData) =
  -- Requires the credit management service to generate the decline data
  creditManagement.generateDecline!<Id,RatingData>
  | [DeclineData]
    creditReq.generateDecline?<Id,DeclineData>.
      ( -- Forwards the decline data to the customer and asks for an update
        portal.declineToClient!<Id,DeclineData>
        | [UpdateDesired]
          creditReq.declineToClient?<Id,UpdateDesired>.
            -- Sends the response from the customer concerning the update to the Main scope
            update.desired!<UpdateDesired>
      )


Approval(Id,update,desired,Result,RatingData,end,ManualAcceptance) =
  [if#][then#][X]
  ( -- Check the rating result
    if.then!<Result>
    | ( if.then?<bbb>.
          -- The rating result is BBB: requires clerk approval
          portal.requestClerkApproval!<Id,RatingData>
        + if.then?<X>.
          -- The rating result is different from BBB: : requires supervisor approval
          portal.requestSupervisorApproval!<Id,RatingData>
      )
    | -- Waits a response from either a clerk or a supervisor
      [ApprovalData]
      creditReq.approvalResult?<Id,ManualAcceptance,ApprovalData>.
        -- The approval phase terminates
        approval.end!<>
  )

Decision(Id,creditManagement,update,desired,Result,RatingData) =
  [if#][then#][end#][ManualAcceptance][X]
  ( -- Check the rating result
    if.then!<Result>
    | ( if.then?<aaa>.
          [var#][set#]
          ( -- The rating result is AAA: set ManualAcceptance to "undef"
            -- and skip the approval
            var.set!<undef>
            | var.set?<ManualAcceptance>.
                approval.end!<>
          )
        + if.then?<X>.
          -- The rating result is different from AAA
          Approval(Id,update,desired,Result,RatingData,end,ManualAcceptance)
      )
    | approval.end?<>.
        ( if.then!<Result,ManualAcceptance>
          | [X1][X2][X3][X4]
            ( -- If result=AAA or manualAcceptance==true : accept
              if.then?<aaa,X1>. Accept(Id,creditManagement,update,desired,Result,RatingData)
              + if.then?<X2,true>. Accept(Id,creditManagement,update,desired,Result,RatingData)
              -- Otherwise : decline
              + if.then?<X3,X4>. Decline(Id,creditManagement,update,desired,Result,RatingData)
            )
        )
```

```
  )

RatingCalculation(Id,creditManagement,update,desired,LoginName,FirstName,LastName) =
  ( -- Invokes the rating service
    rating.calculateRating!<Id,LoginName,FirstName,LastName>
    | [Result][RatingData]
      creditReq.calculateRating?<Id,Result,RatingData>.
        Decision(Id,creditManagement,update,desired,Result,RatingData)
  )


HandleBalanceAndSecurityData(Id,creditManagement,comp,update,desired,LoginName,FirstName,LastName) =
  [flow#][end#]
  ( -- Requires balance data
    ( portal.enterBalanceData!<Id>
      | [BalancePackage]
        creditReq.enterBalanceData?<Id,BalancePackage>.
        ( -- Updates balance rating
          balance.updateBalanceRating!<Id,LoginName,FirstName,LastName,BalancePackage>
          | creditReq.updateBalanceRating?<Id>. flow.end!<>
        )
    )
    |
    -- Requires security data
    ( portal.enterSecurityData!<Id>
      | [SecurityPackage]
        creditReq.enterSecurityData?<Id,SecurityPackage>.
        ( -- Updates security rating
          security.updateSecurityRating!<Id,LoginName,FirstName,LastName,SecurityPackage>
          | creditReq.updateSecurityRating?<Id>. flow.end!<>
        )
    )
    | flow.end?<>.
        flow.end?<>.
          ( RatingCalculation(Id,creditManagement,update,desired,LoginName,FirstName,LastName)
            | -- Install the compensation handler
              [End]
              { comp.handleBalanceAndSecurityData?<handleBalanceAndSecurityData,End>.
                  [completed#]
                  ( -- Clears balance and security data
                    -- (notably, b and s allows creditReq to distinguish the responses)
                    balance.clearData!<Id> | creditReq.clearData?<Id,b>. comp.completed!<>
                    | security.clearData!<Id> | creditReq.clearData?<Id,s>. comp.completed!<>
                    | comp.completed?<>. comp.completed?<>. comp.End!<>
                  )
              }
          )
  )


Creation(Id,creditManagement,comp,update,desired,LoginName,FirstName,LastName) =
  -- Gets credit data from the customer
  [CustomerId][CreditAmount][CreditType][MonthlyInstalment]
  crediReq.createNewCreditRequest?<Id,CustomerId,CreditAmount,CreditType,MonthlyInstalment>.
     ( -- Forward credit data to the credit management service
       creditManagement.initCreditData!<Id,CustomerId,CreditAmount,CreditType,MonthlyInstalment>
       | creditReq.initCreditData?<Id>.
         ( -- Replies to Portal to notify that the system is working on the request
           portal.createNewCreditRequest!<Id,working>
           | ( HandleBalanceAndSecurityData(Id,creditManagement,comp,update,desired,LoginName,
                                            FirstName,LastName)
               | -- Install the compensation handler CompensateCreditManagementInitialization
                 [End]
                 { comp.creation?<creation,End>.
                     ( -- Clears credit data
                       creditManagement.removeData!<Id>
                       | creditReq.removeData?<Id>. comp.End!<>
                     )
                 }
             )
         )
     )
```

```
Main(Id,creditManagement,LoginName,FirstName,LastName) =
  [raise#][comp#]
  ( [k]
    ( -- Scope body
      [repeat#][until#][update#][desired#]
      ( -- Repeat until loop
        repeat.until!<>
        | * repeat.until?<>.
            [k_loop]
            ( Creation(Id,creditManagement,comp,update,desired,LoginName,FirstName,LastName)
              | update.desired?<true>.
                   ( -- Removes the remaining activities of the previous iterative step
                     -- and restarts the loop
                     kill(k_loop) | { repeat.until!<> }
                   )
                + update.desired?<false>. Finalize(Id)
            )
      )
      | -- Event Handler: CANCEL service activity --
        creditReq.cancel?<Id>.
          -- Stops the normal execution of Main scope
          -- and raises a fault
          ( kill(k) | {raise.abort!<>} )
    )
    | -- Fault Handler: MAIN FAULT service activity --
      raise.abort?<>.
        ( -- CompensateAll
          [end#]
          ( comp.creation!<creation,end>
            | comp.end?<>.
              ( comp.handleBalanceAndSecurityData!<handleBalanceAndSecurityData,end>
                | comp.end?<>.
                   portal.abortProcess!<Id>
              )
          )
        )
    | -- The following activities provide an unlimited number of termination
      -- signals to allow the compensation chain to progress also if the
      -- compensation handlers have not been installed yet
      -- (see Compesate and CompensateAll semanitcs)
      * [X1][X2] comp.creation?<X1,X2>. comp.X2!<>
      | * [X3][X4] comp.handleBalanceAndSecurityData?<X3,X4>. comp.X4!<>
  )


CreditRequest(customerManagement,creditManagement) =
 *[k][raise#]
  ( -- INITIALIZE service activity --
    [Id][Name][Password]
    creditReq.initialize?<Id,Name,Password>.
      ( -- Invokes the customer management service
        -- to check user login data
        customerManagement.checkUser!<Id,Name,Password>
        | [UserOK]
          creditReq.checkUser?<Id,UserOK>.
            ( -- Notifies the result of the login check to Portal
              {portal.initialize!<Id,UserOK>}
              | [if#][then#]
                ( if.then!<UserOK>
                  | if.then?<false>.
                      -- If the login check fails, the process is aborted
                      ( kill(k)
                        | -- This fault is uncaught
                          {raise.abort!<>}
                      )
                    + if.then?<true>.
                       -- If the login check succeeds, gets customer data
                       -- from the customer management service
                       ( customerManagement.getCustomerData!<Id,Name,Password>
                         | [LoginName][FirstName][LastName]
```

```
                        crediReq.getCustomerData?<Id,LoginName,FirstName,LastName>.
                          Main(Id,creditManagement,LoginName,FirstName,LastName)
                    )
                )
            )
        )
    )


PortalUpdate(k,id) =
  -- Requires a new credit request
  crediReq.createNewCreditRequest!<id,customerId,amount2,mortgage2,instalment2>
  | portal.createNewCreditRequest?<id,working>.
    ( -- Provides (in parallel) balance and security data
      portal.enterBalanceData?<id>.
        creditReq.enterBalanceData!<id,balancePackage2>
      |
      portal.enterSecurityData?<id>.
        creditReq.enterSecurityData!<id,securityPackage2>
      |
      -- Waits for an offer or a decline
      [Agreement][DeclineData]
      ( portal.offerToClient?<id,Agreement>.
          [nonDet#][choice#]
          ( -- Non-deterministically either accepts the offer or not
            nonDet.choice!<>
            | nonDet.choice?<>.
                ( -- Disables the possibility to cancel
                  kill(k)
                  | { creditReq.offerToClient!<id,true>
                      | -- Receives the logout message and terminates
                        portal.goodbye?<id>. nil
                    }
                )
              + nonDet.choice?<>.
                ( -- Disables the possibility to cancel
                  kill(k)
                  | { creditReq.offerToClient!<id,false>
                      | -- Receives the logout message and terminates
                        portal.goodbye?<id>. nil
                    }
                )
          )
        +
        portal.declineToClient?<id,DeclineData>.
          ( -- Disables the possibility to cancel
            kill(k)
            | { -- Does not require a second update
                creditReq.declineToClient!<id,false>
                | -- Receives the logout message and terminates
                  portal.goodbye?<id>. nil
              }
          )
      )
    )



Portal =
  -- Portal acts as interface for customers
  --
  [id#]
  ( -- Portal performs a login required by the customer
    creditReq.initialize!<id,name,pwd>
    | [UserOK]
        portal.initialize?<id,UserOK>.
        [if#][then#]
        ( if.then!<UserOK>
            | if.then?<false>.
                -- Process aborted: the following receive
```

```
            -- is never executed (see Initialize service activity)
            portal.abortProcess?<id>. nil
        + if.then?<true>.
            [k]
            ( -- The login succeeded.
                -- From now on, the customer can require a cancellation
                [nonDet#][choice#]
                ( nonDet.choice!<>
                | nonDet.choice?<>.
                    ( kill(k)
                        | { creditReq.cancel!<id> | portal.abortProcess?<id>. nil }
                    )
                )
                |
                -- Requires a new credit request
                crediReq.createNewCreditRequest!<id,customerId,amount,mortgage,instalment>
                | portal.createNewCreditRequest?<id,working>.
                ( -- Provides (in parallel) balance and security data
                    portal.enterBalanceData?<id>.
                    creditReq.enterBalanceData!<id,balancePackage1>
                    |
                    portal.enterSecurityData?<id>.
                    creditReq.enterSecurityData!<id,securityPackage1>
                    |
                    -- Waits for an offer or a decline
                    [Agreement][DeclineData]
                    ( portal.offerToClient?<id,Agreement>.
                        [nonDet#][choice#]
                        ( -- Non-deterministically either accepts the offer or not
                            nonDet.choice!<>
                            | nonDet.choice?<>.
                                ( -- Disables the possibility to cancel
                                    kill(k)
                                    | { creditReq.offerToClient!<id,true>
                                        | -- Receives the logout message and terminates
                                        portal.goodbye?<id>. nil
                                    }
                                )
                            + nonDet.choice?<>.
                                ( -- Disables the possibility to cancel
                                    kill(k)
                                    | { creditReq.offerToClient!<id,false>
                                        | -- Receives the logout message and terminates
                                        portal.goodbye?<id>. nil
                                    }
                                )
                        )
                        +
                        portal.declineToClient?<id,DeclineData>.
                            [nonDet#][choice#]
                            ( -- Non-deterministically either requires an update or not
                                nonDet.choice!<>
                                | nonDet.choice?<>.
                                    ( -- Disables the possibility to cancel
                                        kill(k)
                                        | { creditReq.declineToClient!<id,false>
                                            | -- Receives the logout message and terminates
                                            portal.goodbye?<id>. nil
                                        }
                                    )
                                + nonDet.choice?<>.
                                    ( creditReq.declineToClient!<id,true>
                                        | -- UPDATE required
                                        PortalUpdate(k,id)
                                    )
                            )
                    )
                )
            )
    )
)
|
```

```
  -- Portal acts as interface for clerks and supervisors
  ( -- Receives an approval request for a clerk
    * [Id][RatingData]
      portal.requestClerkApproval?<Id,RatingData>.
        [nonDet#][choice#]
        ( nonDet.choice!<>
          | nonDet.choice?<>. creditReq.approvalResult!<Id,true,approvedDataByClerk>
            + nonDet.choice?<>. creditReq.approvalResult!<Id,false,nonApprovedDataByClerk>
        )
    |
    -- Receives an approval request for a supervisor
    * [Id][RatingData]
      portal.requestSupervisorApproval?<Id,RatingData>.
        [nonDet#][choice#]
        ( nonDet.choice!<>
          | nonDet.choice?<>. creditReq.approvalResult!<Id,true,approvedDataBySup>
            + nonDet.choice?<>. creditReq.approvalResult!<Id,false,nonApprovedDataBySup>
        )
  )


CustomerManagement(customerManagement) =
  * [Id][Name][Password]
    customerManagement.checkUser?<Id,Name,Password>.
      -- Non-deterministically decides the result
      -- of the login check
      [nonDet#][choice#]
      ( nonDet.choice!<>
        | nonDet.choice?<>.
              -- Login fails
              creditReq.checkUser!<Id,false>
          + nonDet.choice?<>.
              -- Login succeeds
              ( creditReq.checkUser!<Id,true>
                | -- Provides customer data
                  customerManagement.getCustomerData?<Id,Name,Password>.
                     crediReq.getCustomerData!<Id,loginName,firstName,lastName>
              )
      )


CreditManagement(creditManagement) =
  * [Id][CustomerId][CreditAmount][CreditType][MonthlyInstalment]
    creditManagement.initCreditData?<Id,CustomerId,CreditAmount,CreditType,MonthlyInstalment>.
    [k]
    ( -- Replies to the credit request service that the credit data have been stored
      creditReq.initCreditData!<Id>
      | [RatingData]
        ( -- Provides the calculation of the offer
          creditManagement.generateOffer?<Id,RatingData>.
                         ( creditReq.generateOffer!<Id,agreementData>
                           | [Accepted]
                             creditManagement.acceptOffer?<Id,Accepted>.
                                -- Stores the acceptance response and replies
                                portal.acceptOffer!<Id>
                         )
          +
          -- Provides the calculation of the decline
          creditManagement.generateDecline?<Id,RatingData>.
                         creditReq.generateDecline!<Id,declineData>
        )
      | [CustomerIdUpd][CreditAmountUpd][CreditTypeUpd][MonthlyInstalmentUpd]
        ( -- Provides a receive for accepting a compensation request
          creditManagement.removeData?<Id>.
            ( -- Stops the calculation service, removes the stored data and sends a reply
              kill(k) | { creditReq.removeData!<Id> }
            )
          +
          -- Provides a receive for updating the credit data
          -- (NB: this receive activity has higher priority than the starting receive)
          creditManagement.initCreditData?<Id,CustomerIdUpd,CreditAmountUpd,CreditTypeUpd,
                                           MonthlyInstalmentUpd>.
```

```
            ( -- Stops the calculation service and restart the service by using the new data
              kill(k)
              | { creditManagement.initCreditData!<Id,CustomerIdUpd,CreditAmountUpd,CreditTypeUpd,
                                                  MonthlyInstalmentUpd> }
            )
        )
    )



Calculator(calculator) =
  * [Id][BalanceRating][SecurityRating]
    calculator.performRatingCalculation?<Id,BalanceRating,SecurityRating>.
      [nonDet#][choice#]
      ( -- The rating result and the overall rating data are non-deterministically chosen
        nonDet.choice!<>
        | nonDet.choice?<>. rating.performRatingCalculation!<Id,aaa,overallatingDataAAA>
          + nonDet.choice?<>. rating.performRatingCalculation!<Id,bbb,overallatingDataBBB>
          + nonDet.choice?<>. rating.performRatingCalculation!<Id,ccc,overallatingDataCCC>
      )



Rating(calculator) =
  * [Id][LoginName][FirstName][LastName]
    rating.calculateRating?<Id,LoginName,FirstName,LastName>.
      [BalanceRating][SecurityRating][flow#][end#]
      ( -- Invokes (in parallel) balance and security validation service
        balance.calculateBalanceRating!<Id,LoginName,FirstName,LastName>
        | rating.calculateBalanceRating?<Id,BalanceRating>. flow.end!<>
        |
        security.calculateSecurityRating!<Id,LoginName,FirstName,LastName>
        | rating.calculateSecurityRating?<Id,SecurityRating>. flow.end!<>
        |
        flow.end?<>.
          flow.end?<>.
            ( -- Calculate the rating
              calculator.performRatingCalculation!<Id,BalanceRating,SecurityRating>
              | [Result][OverallRating]
                 rating.performRatingCalculation?<Id,Result,OverallRating>.
                   -- Sends the overall rating to the credit request service
                   creditReq.calculateRating!<Id,Result,OverallRating>
            )
      )


BalanceAnalysisProvider =
  * [Id][LoginName][FirstName][LastName][BalancePackage]
    balance.updateBalanceRating?<Id,LoginName,FirstName,LastName,BalancePackage>.
       [k]
       (  -- Replies to the credit request service that the balance data have been stored
          creditReq.updateBalanceRating!<Id>
          | -- Provides the balance rating calculation as a persistent service
            * balance.calculateBalanceRating?<Id,LoginName,FirstName,LastName>.
                rating.calculateBalanceRating!<Id,balanceRating>
          | [LoginNameUpd][FirstNameUpd][LastNameUpd][BalancePackageUpd]
            ( -- Provides a receive for accepting a compensation request
              balance.clearData?<Id>.
                ( -- Stops the calculation service and replies
                  kill(k) | { creditReq.clearData!<Id,b> }
                )
              +
              -- Provides a receive for updating the balance data
              -- (NB: this receive activity has higher priority than the starting receive)
              balance.updateBalanceRating?<Id,LoginNameUpd,FirstNameUpd,LastNameUpd,BalancePackageUpd>.
                ( -- Stops the calculation service and restart the service by using the new data
                  kill(k)
                  | { balance.updateBalanceRating!<Id,LoginNameUpd,FirstNameUpd,LastNameUpd,
                                                   BalancePackageUpd> }
                )
            )
```

```
      )


SecurityAnalysisProvider =
  * [Id][LoginName][FirstName][LastName][SecurityPackage]
    security.updateSecurityRating?<Id,LoginName,FirstName,LastName,SecurityPackage>.
       [k]
       (  -- Replies to the credit request service that the security data have been stored
          creditReq.updateSecurityRating!<Id>
          | -- Provides the security rating calculation as a persistent service
            * security.calculateSecurityRating?<Id,LoginName,FirstName,LastName>.
                rating.calculateSecurityRating!<Id,securityRating>
          | [LoginNameUpd][FirstNameUpd][LastNameUpd][SecurityPackageUpd]
            ( -- Provides a receive for accepting a compensation request
              security.clearData?<Id>.
                ( -- Stops the calculation service and replies
                  kill(k) | { creditReq.clearData!<Id,s> }
                )
              +
              -- Provides a receive for updating the security data
              -- (NB: this receive activity has higher priority than the starting receive)
              security.updateSecurityRating?<Id,LoginNameUpd,FirstNameUpd,LastNameUpd,SecurityPackageUpd>.
                ( -- Stops the calculation service and restart the service by using the new data
                  kill(k)
                  | { security.updateSecurityRating!<Id,LoginNameUpd,FirstNameUpd,LastNameUpd,
                                                     SecurityPackageUpd> }
                )
            )
       )

in

[customerManagement#][creditManagement#]
( CreditRequest(customerManagement,creditManagement)
  | CustomerManagement(customerManagement)
  | CreditManagement(creditManagement) )
|
[calculator#] (Rating(calculator) | Calculator(calculator) )
|
BalanceAnalysisProvider() | SecurityAnalysisProvider()
|
Portal()

end




Abstractions {
 Action createNewCreditRequest<$id,*,*,*,*> -> request(cr,$id)
 Action offerToClient<$id,*> -> response(cr,$id)
 Action declineToClient<$id,*> -> response(cr,$id)
 Action cancel<$id> -> cancel(cr,$id)
 Action calculateRating<$id,*,*,*> -> request(rating,$id)
 Action clearData<$id> -> undo(cr,$id)
 Action removeData<$id> -> undo(cr,$id)
 State  abort! -> raising_abort(cr)
 Action abortProcess -> fail(cr)
 State initialize? ->  accepting_request(login)
 State cancel?<$id> -> accepting_cancel(cr,$id)
}




-------------------
-- SocL formulae --
-------------------
--
-- Availability: AG(accepting_request(login))
```

```
--
-- Responsiveness: AG [request(cr,$id)]
--                    AF {response(cr,%id) or cancel(cr,%id)} true
--
-- Interruptibility: AG [request(cr,$id)]
--                       A[accepting_cancel(cr,%id) {true}
--                         U {cancel(cr,%id) or response(cr,%id)} true]
--
-- Compensability: AG [request(rating,$id)] EF {cancel(cr,%id)}
--                    AF {undo(cr,%id)} AF {undo(cr,%id)} AF {undo(cr,%id)} true
--
-- Fault handling: AF ((not raising_abort(cr)) or AF {fail(cr)} true)
--
```

References

[1] J. Elgner, S. Gnesi, N. Koch, and P. Mayer. *The SENSORIA book*, chapter Introduction to case studies and their application domains. Springer, 2010.

[2] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[3] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, DSI, Università di Firenze, 2008. `http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf`. An extended abstract appeared in *ESOP*, LNCS 4421, pages 33-47, Springer.

[4] P. Mayer, A. Schroeder, and N. Koch. Mdd4soa: Model-driven service orchestration. In *EDOC*, pages 203–212. IEEE, 2008.