An ASM Model for a Mutual Credit System

Specifying the Sardex Business Logic

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it

Case Study supplementing Ch.5 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de
Goal of the lecture

Illustrate the use of concurrent communicating ASMs to construct a ground model for a real-life commercial business process namely the business logic of the Sardex mutual credit system.


**Innovative system feature**: distribute to the circuit members the power to create zero-interest credit money (1 Sardex credit = 1 Euro), providing an alternative to credit money creation through bank loans.

- The development of a new decentralized transactional and ledger system architecture led to rigorously specifying also the underlying business logic, prior to its implementation, using communicating ASMs.
- Part of the work was done in the European Commission Project INTERLACE [https://www.interlaceproject.eu/](https://www.interlaceproject.eu/) (2017-2018).
Why INTERLACE used ASMs to specify the architecture

The ASM method allows one to achieve the following goals:

- to provide an *accurate requirements capture*
  – building a common understanding and a guide for the implementation

- to provide a *rigorous practical refinement process* down to code
  – so that the coding can be split efficiently into effectively controllable separate steps rather than achieving the full redesign in one step

- to provide an *efficient change management*
  – to handle the evolution of requirements, due to continuously emerging new or improved system functionalities

such that at any development level the model components are analysable both mathematically and experimentally—since they are precise and executable—and are easily extendable (via refinements)

- i.e. such that at each refinement stage the current implementation can be verified and validated against the more abstract version it refines, thus providing a reliable link to the original requirements.
Selection of characteristic Sardex service operations

Sardex members—companies, their employees, individual consumers, but also members of associated networks—interact with the Sardex network server by requesting services concerning Sardex accounts owned by members. Here we illustrate 4 core services (a bare minimum):

- handling credit or debit requests, the two central Business-to-Business (B2B) operations between companies to transfer the amount for a purchase from the buyer’s Sardex account—which can go negative up to some limit with a 0% interest rate—to the seller’s Sardex account

- handling Business-to-Consumer (B2C) operations where a customer purchase at a retail is paid in Euro (EUR) or Sardex (SRD) with Sardex acting as intermediary (honored by a retailer’s prepaid fee)

- Alerts about low/high balance, high sale volume, low prepayment, etc.

- debt record tracking, a refinement of credit/debit ops

NB. Selection covers a tiny but characteristic subset of all Sardex opns.
Focus of the illustrative **HandleSardexOpns** server model

- **functional requirements level of abstraction** to express the system’s business logic in user (not programming lg or implementation) terms
- proceeding by *stepwise refinements* of abstractions wherever possible
  - to efficiently support integration of new (or change of) requirements
    - here illustrated by adding debt record tracking to credit/debit opns
  - to *smoothly combine control and data* features of the system where behaviorally relevant (a crucial requirement for real-life BPs!)
- **component-based description** of Sardex services
  - defining for each service one component to **HandleSardexOpns**
  - further splitting each service component into subcomponents
- **separating the spec of the scheduler** from that of component behavior
  - using parallelism, limited by dependency constraints—imposing sequential execution—only where required by the business logic to leave the space open for efficient implementations of the model

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
Roadmap to construct the Sardex model

- Sardex server component HandleSardexOpns
  - parallel component structure of the model
  - actors and data involved in the services we model here:
    - member groups and their interaction constraints (group transfer type constraints)
    - account sets and their connectivity constraints
  - behavior of HandleSardexOpns (their semantical meaning)
    - Components of Credit op rule HandleCreditTransfer
    - Components of Debit op rule HandleDebitTransfer
    - Components B2C op rule HandleB2cTransfer
    - Components of Alerts
- MemberOpns component to Send/Receive input to/from the server
- Refinement example: integrate debt record tracking
  - CreateDebtEntry and UpdateDebtRecord
**HANDLESARDEXOPNS** structure for the selected service opns

**HANDLESARDEXOPNS** = -- parallel composition of components

**HANDLECREDITTRANSFER**  **HANDLEDEBITTRANSFER**

**HANDLEB2CTRANSFER**  **APERTS**

**HANDLECREDITTRANSFER** = -- 2-phase request/response pattern

**HANDLECREDITPREVIEWREQ**  **HANDLECREDITPERFORMREQ**

**HANDLEDEBITTRANSFER** = -- 3-phase request/response pattern

**HANDLEDEBITPREVIEWREQ**  **HANDLEDEBITPERFORMREQ**

**HANDLEDEBITACKCOMPLETION**

**HANDLEB2CTRANSFER** =

**HANDLEB2CEUR**  **HANDLEB2CSRDR**  **RECHARGEPREPAID**

**HANDLE ALERTS** =

**HIGHBALANCE ALERT, LOW PREPAY ALERT, ...**
Some typical actors which trigger **HandleSardexOpns**

To **HandleSardexOpns** is triggered by actors of user groups:

- **Company**
  -- actors which participate only in B2B opns
- **Retail**
  -- actors which participate only in B2C operations
- **Full**
  -- actors with both Company and Retail functionality
- **Consumer**
  -- individuals which participate only in B2CEUR opns
- **Consumer Verified**
  -- registered consumers with additional B2CSRD opn

\[ Mngr = \{ mngr \} \]

-- *mngr* acts for Sardex as a company

\[ Group = \{ Company, Retail, Full, Consumer, Consumer_{Verified}, Mmgr \} \]

- We skip for example *Employees*, not-yet-cleared (called *Welcome* members), suspended (called *On Hold*) users or users of other circuits.

Groups come with **group profile metadata** (their ‘state’) and **transfer type constraints** on the groups among which transfers are allowed.
Three characteristic accounts HandleSardexOpns manages

Each user has a set Account(user) of Sardex accounts each of which the user is the owner(acc) of, at most one account per account type. Here we illustrate 3 characteristic account types (sets of accounts):

\[ \text{AcctType} = \{ \text{CC, Income, Prepaid} \} \]

- **CC**: contains SRD-credit accounts cc(user) used to pay for purchases in SRD currency (for each user ∈ group of any group ∈ Group)
- The sets Income resp. Prepaid of Euro-accounts contain for each user ∈ Retail ∪ Full additionally two accounts:
  - income(user) recording B2CEUR payments made by consumers
  - prepaid(user) recording the Euro fee due by the user to the intermediary—the Sardex company—for each B2CEUR operation

Accounts come with account metadata (functions) and connectivity constraints on types of accounts between which an operation is allowed.
Sardex $CC$-accounts can be used by members to
- pay another member in SRD, up to a $creditLimit$, for a purchase
- be payed by another member in SRD, up to an $upperBalanceLimit$, for goods sold to that other member

These limits protect the circuit and its members:
- in case of no $upperBalanceLimit$ limit an extremely positive balance could make it difficult for the member to spend the credits by purchases, inviting not to sell any more (risk to become inactive in the circuit)
- in case of no $creditLimit$ an extremely negative balance could make it difficult for the member to proceed with further purchases (to avoid a too high debt) with the risk to become inactive in the circuit if no credits arrive by sales.
CC-account functions/predicates

balance : Real
-- may become negative

NB. This function has an exclusive access constraint (see below).

creditLimit : Nat
-- limiting how far account can go negative, to buy

availableBalance = balance + creditLimit ∈ Nat

LowBalanceAlert iff availableBalance < lowBalanceAlert
-- small amount of money left for further purchases in SRD

upperBalanceLimit ∈ Nat \ {0}
-- limiting how far account can go positive, to sell

HighBalanceAlert iff balance > 0 and

upperBalanceLimit − balance < highBalanceAlert
-- small space left for further sales in SRD
There are other already existing or planned CC-account functions, e.g.:

- `saleCapacity` indicating the maximum yearly SRD volume the member committed to be willing to sell with payment in SRD
  - used to calculate the annual Sardex fee
- `saleVolume` denoting the current total volume of sales already made and paid in SRD (per year, we notationally suppress the param)
  - NB. This function has an exclusive access constraint (see below).
- `availableSaleCapacity = saleCapacity - saleVolume`
  - usable to issue a `HighVolumeAlert` if `availableSaleCapacity` goes below a `highVolumeAlert` value
  - possibly resulting in an update of `saleCapacity` and annual fee

We skip various restrictions on account functions, e.g. that:

- `Prepaid` accounts are without credit (i.e. `creditLimit = 0`)
- `Domu` accounts have no `saleCapacity`
- brokers may update certain functions, e.g. setting `creditLimit := 0`
Typical group profile metadata (ASM functions/predicates)

- **creditPercent** denotes the percentage of SRD-payments accepted by a Full or Retail member for B2B-transactions of value $\geq 1000$ Euro
  
  – NB. In a previous system version this holds also for B2E-transactions

- **acceptanceRate** defines the percent rate of the total purchase value at which Full or Retail members accept SRD currency from a registered consumer (i.e. for a B2C-transaction)

- **rewardRate** of Full and Retail members indicates the percentage of reward the member offers in SRD for a B2CEUR purchase

- fee locations for Company, Retail and Full members
  
  – for Retail a **B2CEuroFee** fctn indicating the fee for B2CEUR sales
  
  ● the percentage the retailer pays to Sardex for each B2CEUR sale
  
  – for Company an **InterTradeEuroFee** fctn indicating the fee for inter-circuit sales in the non-Euro circuit currency (SRD, VTX, etc).
  
  – Full members have 2 functions **InterTradeEuroFee, B2CEuroFee**.
Group transfer type constraints on Credit/Debit transactions

The constraints depend on the operation, the currency and the group:

\[ TT : \{ \text{Credit, Debit} \} \times \{ \text{SRD, EUR} \} \times \text{Group} \rightarrow \{ G \mid G \subseteq \text{Group} \} \]

We split the transfer type function \( TT \) as follows into subfunctions

\[ TT_{\text{op,cur}} : \text{Group} \rightarrow \{ G \mid G \subseteq \text{Group} \} \]

For each operation, each currency and each fromGroup of buyers:

- \( TT_{\text{op,cur}}(\text{fromGroup}) \) defines the set toGroups of groups of sellers which are admitted for the transfer operation in the currency – namely of those groups whose members are allowed to receive a transfer, concerning the operation in the currency, to one of their accounts from an account of a fromGroup member

- under separately defined appropriate account connectivity constraints and further constraints on the amount of the transfer

NB. Separating \( TT \)-subfunctions and constraints on groups, accounts and transfer amounts supports modularity and simplicity concerns.
A Credit SRD-op can be started only by members of
- Company or Full or Mngr or ConsumerVerified

\( \text{MayTriggerCreditOp}(mbr) \) iff
\( \text{group}(mbr) \in \{ \text{Company, Full, Mngr, ConsumerVerified} \} \)

The target groups allowed for a Credit SRD-op are defined by:
- every Company ∪ Full ∪ Mngr member can trigger a Credit SRD-op to a member of Company ∪ Full ∪ Mngr
- ConsumerVerified members can trigger a Credit SRD-op to Retail ∪ Full-members (NB. We simplify these opns below)

Equationally this is expressed as follows:

\[
TT_{Credit,SRD}(G) =
\begin{cases}
\{ \text{Company, Full, Mngr} \} & \text{if } G \in \{ \text{Company, Full, Mngr} \} \\
\{ \text{Retail, Full} \} & \text{if } G = \text{ConsumerVerified}
\end{cases}
\]
For Debit SRD-ops the following source/target groups are allowed:

- Every Company ∪ Full ∪ Mngr member MayTriggerDebitOp (in SRD) that draws on a member of Company ∪ Full.
- Some other constraints (there are more) we do not use in this model:
  • Every member of Retail ∪ Full can start a Debit SRD-op that draws on any member of Consumer Verified
  • mngr can start a Debit SRD-op that draws on a Retail member

\[ TT_{Debit,SRD}(G) = \begin{cases} 
\{\text{Company, Full, Mngr}\} & \text{if } G \in \{\text{Company, Full}\} \\
\{\text{Retail, Full}\} & \text{if } G = \text{Consumer Verified} \\
\text{Mngr} & \text{if } G = \text{Retail} 
\end{cases} \]
Account connectivity constraints

Since for reasons of simplicity we model B2C-ops not as Credit/Debit ops, it suffices here to consider user-initiated \( CC \)-transactions:

- An SRD-\textit{Credit} transfer op is triggered by a debitor, starts at a \( \text{fromAcc} \in CC \) and has as allowed \( \text{toAcc} \) a \( CC \) account.
- An SRD-\textit{Debit} op is triggered by a creditor, starts from a debitor’s \( \text{fromAcc} \in CC \) and goes to the creditor’s \( \text{toAcc} \in CC \).

\[
\text{AccT}_{\text{op,cur}} : \text{AccountType} \rightarrow \{ \text{Acct} \mid \text{Acct} \subseteq \text{AccountType} \}
\]

\[
\text{AccT}_{\text{Credit,SRD}}(CC) = \text{AccT}_{\text{Debit,SRD}}(CC) = \{ CC \}
\]

\[
\text{MayStartCreditOpns}(acct) \iff acct \in CC
\]

\[
\text{MayStartDebitOpns}(acct) \iff acct \in CC
\]

- NB. Such definitions (of ASM functions, predicates, rules, etc.) by case distinction \textit{support componentwise extensions to new requirements} – e.g. to new account type arguments for \( \text{AccT}_{\text{Credit,SRD}} \) we do not use here, which are similar to the above clauses for \( \text{TT}_{\text{Debit,SRD}} \).
Credit resp. Debit are *push resp. pull ops*, triggered by a debitor resp. creditor, to transfer amounts from a debitor’s to a creditor’s account.

Credit/Debit ops follow a *request/response pattern* between the actors:
- members equipped with the MEMBEROPNS program
- the Sardex server which executes the HANDLESARDEXOPNS pgm

We *abstract from communication Channels*, namely a website, a mobile Phone or a standard point-of-sale (POS) terminal retailers use for EUR transactions or to route SRD transactions via an API.

- Reason: The business logic effect of all Credit/Debit op elements we define below is independent of the particular value of their *Channel* parameter and of the particular communication mechanism used.

Credit/Debit ops follow a *multiple-phase request/response pattern*:
- Preview, where only group/account-related access rights are checked
- Perform, where the critical transfer amount constraints are checked
HandleCreditTransfer request/response control flow

debitor  sardex  debitor

CreditPreviewReq  →  CreditTypeCheck

PermitPerformReq  ←  if ok  if fail  →  FailureMsg

CreditPerformReq  →  CreditTypeCheck

if ok ↓  if fail  →  FailureMsg

CreditAmountLimitsCheck

if ok ↓  if fail  →  FailureMsg

ConfirmationMsg  ←  CompleteTransaction

NB. Arrows indicate the sequential control and data flow.
2-phase structure of the **HANDLECREDITTRANSFER** service

The Sardex server, upon receiving a member’s (a debitor’s)

- *CreditPreviewRequest*: will **HANDLECREDITPREVIEWREQ**
- *CreditPerformRequest*: will **HANDLECREDITPERFORMREQ**

\[
\text{HANDLECREDITTRANSFER} = \begin{cases} 
\text{HANDLECREDITPREVIEWREQ} \\
\text{HANDLECREDITPERFORMREQ} 
\end{cases}
\]

- The server can execute at any moment any of these two rules whose execution depends only on the parameters with which they are called.
  - But the rules are defined in such a way that for a specific user request parameter, **HANDLECREDITPERFORMREQ** can be triggered only if **HANDLECREDITPREVIEWREQ** yields a positive check result.
- For the transactional part of **HANDLESARDEXOPNS** concerning accounts it is assumed for the model that access to the involved accounts is exclusive (see below), as usual for db-transactions.
How \texttt{HandleCreditPreviewReq} interacts with members

- \texttt{HandleCreditPreviewReq} triggers a \texttt{CreditTypeCheck} concerning the group/account related access rights wrt the parameters of the member's \texttt{CreditPreviewRequest}.
  - upon a positive check result the server will \texttt{PermitPerformReq}
  - otherwise it will send a constraint violation msg to the member.

\texttt{HandleCreditPreviewReq}((\texttt{from}, \texttt{to}, \texttt{amount}), \texttt{mbr}) =

\begin{verbatim}
  let t = (\texttt{credit}, \texttt{from}, \texttt{to}, \texttt{amount}) -- the transfer params

  if Received(\texttt{CreditPreviewReq}(t), \texttt{from mbr}) then
    \texttt{CreditTypeCheck}(t, \texttt{mbr}, \texttt{PermitPerformReq}(t))
    \texttt{Consume}(\texttt{CreditPreviewReq}(t))
  
  where

  \texttt{PermitPerformReq}(t) = -- NB. \texttt{mbr} = owner(\texttt{from})
    \texttt{Send}(YouMayTriggerPerformReq(t), \texttt{to mbr})
\end{verbatim}
How `HandleCreditPerformReq` interacts with members

`HandleCreditPerformReq` repeats `CreditTypeCheck`

- upon a positive outcome it will `TryToCompleteCreditOpn` by a `CreditAmountLimitsCheck`
  – to `CompleteTransaction` definitely, in the positive case

```plaintext
HandleCreditPerformReq((from, to, amount), mbr) =

let t = (credit, from, to, amount) -- the transfer params

if Received(CreditPerformReq(t), from mbr) then
    CreditTypeCheck(t, mbr, TryToCompleteCreditOpn(t))
    Consume(CreditPerformReq(t))

where TryToCompleteCreditOpn(t) =
    CreditAmountLimitsCheck(t, CompleteTransaction(t))
```

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
let $t = (\text{credit}, \text{from}, \text{to}, \text{amount})$  

--- the transfer params

if $mbr = \text{owner}(\text{from})$ and $\text{MayTriggerCreditOp}(mbr)$ then

--- check group transfer type constraints

if forsome $g \in TT_{Credit,SRD}(\text{group}(mbr)) \land \text{owner}(\text{to}) \in g$ then

if $\text{MayStartCreditOpns}(\text{from})$ then

--- check account connectivity constraints

if $\text{acctSet}(\text{to}) \in AccT_{Credit,SRD}(\text{acctSet}(\text{from}))$ then

\text{PROCEED}(t)  

--- in positive case $\text{PERMITPERFORMREQ}(t)$

else \text{SEND}(\text{ErrMsg}(\text{CreditTargetAcctError}(t)), \text{to mbr})

else \text{SEND}(\text{ErrMsg}(\text{CreditSourceAcctError}(t)), \text{to mbr})

else \text{SEND}(\text{ErrMsg}(\text{CreditTargetGroupError}(t)), \text{to mbr})

else \text{SEND}(\text{NotAcctOwnerOrCreditSourceGroupError}(t), \text{to mbr})
let \( t = (credit, from, to, amount) \) -- the transfer params

if \( \text{CanBeSpentBy}(\text{from}, \text{amount}) \) then

  if \( \text{CanBeCashedBy}(\text{to}, \text{amount}) \) then

    if \( \text{HasSellCapacityFor}(\text{amount}, \text{to}) \) then \text{PROCEED}(t)

      -- namely to \text{COMPLETE TRANSACTION}

    else \text{SEND}(\text{ErrMsg}(\text{CapacityViolation}(t)), \text{to owner}(\text{from}))

  else \text{SEND}(\text{ErrMsg}(\text{UpperBalanceLimitErr}(t)), \text{to owner}(\text{from}))

else \text{SEND}(\text{ErrMsg}(\text{AvailBalanceViolation}(t)), \text{to owner}(\text{from}))

where

\( \text{CanBeSpentBy}(\text{from}, a) \) iff \( \text{availableBalance}(\text{from}) \geq a \)

\( \text{CanBeCashedBy}(\text{to}, a) \) iff

\( \text{balance}(\text{to}) + a \leq \text{upperBalanceLimit}(\text{to}) \)

\( \text{HasSellCapacityFor}(a, \text{to}) \) iff \( a \leq \text{availableSaleCapacity}(\text{to}) \)
The abstract fctn \( \text{transact} \) encodes the transfer info that is appended to the \( \text{Ledger} \) and results in an update of the critical \( \text{balance} \) and \( \text{saleVolume} \) locations (constrained, as usual for db transactions, by an exclusive access condition in \( \text{TryToCompleteCreditOpn} \)).

\[
\text{CompleteTransaction}(\text{transfer}) =
\]

let \((\text{credit}, \text{from}, \text{to}, \text{amount}) = \text{transfer} \)

\[
\text{APPEND}(\text{transact}(\text{transfer}, \text{now}), \text{Ledger}) \quad \text{-- timestamp now}
\]

\[
\text{SEND}(\text{Confirmed}(\text{transfer}), \text{to owner}(\text{from}))
\]

where \( \text{APPEND}(\text{transact}(\text{transfer}, \text{now}), \text{Ledger}) \) includes

\[
\text{balance}(\text{from}) := \text{balance}(\text{from}) - \text{amount}
\]

\[
\text{balance}(\text{to}) := \text{balance}(\text{to}) + \text{amount}
\]

\[
\text{saleVolume}(\text{to}) := \text{saleVolume}(\text{to}) + \text{amount}
\]

NB. Here we abstract from the underlying concrete db-scheme.
3-phase structure of the `HANDLEDEBITTRANSFER` service

The Sardex server—note the symmetry of Credit/Debit ops—

- upon receiving a member’s (a creditor’s)
  - `DebitPreviewRequest`: will `HANDLEDEBITPREVIEWREQ`
  - `DebitPerformRequest`: will `HANDLEDEBITPERFORMREQ`

- upon a positive `DEBITAMOUNTLIMITSCHECK` will interact with the debitor to `REQUESTDEBITACKNOWLEDGEMENT` (except for small amounts) before it can `HANDLEDEBITACKCOMPLETION`

\[
HANDLEDEBITTRANSFER = \begin{cases} 
\text{HANDLEDEBITPREVIEWREQ} \\
\text{HANDLEDEBITPERFORMREQ} \\
\text{HANDLEDEBITACKCOMPLETION} 
\end{cases}
\]

`HANDLEDEBITACKCOMPLETION` splits into two suboperations:

- a `DEBITACKCOMPLETION` if the debitor’s ack arrives in time
- a `DEBITACKREQUESTEXPIRED` component, otherwise
**HANDLEDEBITTRANSFER request/response flow: phase 1-2**

**creditor**

`DebitPreviewReq → DebitTypeCheck`

`PermitPerformReq ← if ok else FailureMsg`

`DebitPerformReq → DebitTypeCheck`

  if ok ↓ else FailureMsg

`DebitAmountLimitsCheck`

  if ok ↓ else FailureMsg

`RequestDebitAck`

  if LargeAmount then → AckReq

**sardex**

**debitor**

`ConfirmationMsg ← else COMPLETE TRANSACTION`
HandleDebitTransfer request/response flow: phase 3

<table>
<thead>
<tr>
<th>creditor</th>
<th>sardex</th>
<th>debitor</th>
</tr>
</thead>
</table>

RejectMsg ← if ExpiredAckReq → ExpireMsg

DebitAckCompletion ← AnswerMsg

RejectMsg ← if RejectAnsw else ↓

FinalDebitAmtLimitsCheck

RejectMsg ← if CheckFails → FailureMsg else ↓

Confirmation ← COMPLETE TRANSACTION
How HandleDebitPreviewReq interacts with members

- HandleDebitPreviewReq triggers a DebitTypeCheck concerning the group/account related access rights which correspond to the parameters of the creditor’s DebitPreviewRequest
  - upon a positive check result the server will PermitPerformReq
  - otherwise it will send a constraint violation msg to the creditor

```
HandleDebitPreviewReq((from, to, amount), creditor) =

  let t = (debit, from, to, amount) -- the transfer params

  if Received(DebitPreviewReq(t), from creditor) then
    DebitTypeCheck(t, creditor, PermitPerformReq(t))
    Consume(DebitPreviewReq(t))

  where

  PermitPerformReq(t) =  -- NB. creditor = owner(to)
    Send(YouMayTriggerPerformReq(t), to creditor)
```
DebitTypeCheck((from, to, amount), mbr, Proceed)

let t = (debit, from, to, amount)  let debitor = owner(from)
if mbr = owner(to) and MayTriggerDebitOp(creditor) then
    -- check group transfer type constraints for the creditor mbr
    if forsome g ∈ TT_{Debit, SRD}(group(debitor)) mbr ∈ g then
        if MayStartDebitOpns(to) then
            -- check account connectivity constraints
            if acctSet(from) ∈ AccT_{Debit, SRD}(acctSet(to)) then
                PROCEED(t)  -- in positive case PERMITPERFORMREQ(t)
            else SEND(ErrMsg(DebitTargetAcctError(t)), to mbr)
        else SEND(ErrMsg(DebitSourceAcctError(t)), to mbr)
    else SEND(ErrMsg(DebitTargetGroupError(t)), to mbr)
else SEND(NotAcctOwnerToReceiveDebit(t), to mbr)
How `HandleDebitPerformReq` interacts with members

`HandleDebitPerformReq` repeats `DebitTypeCheck` upon a positive outcome it will `TryToCompleteDebitOpn` by a `DebitAmountLimitsCheck` – to `RequestDebitAck`, from the debitor, upon a positive check outcome and then `HandleDebitAckCompletion`

```
HandleDebitPerformReq((from, to, amount), mbr) =
let t = (debit, from, to, amount) -- the transfer params
if Received(DebitPerformReq(t), from creditor) then
    DebitTypeCheck(t, creditor, TryToCompleteDebitOpn(t))
    Consume(DebitPerformReq(t))
where TryToCompleteDebitOpn(t) =
    DebitAmountLimitsCheck(t, RequestDebitAck(t))
```
DebitAmountLimitsCheck component

DebitAmountLimitsCheck((from, to, amount), Proceed) =

let t = (debit, from, to, amount)
let debitor = owner(from)
let creditor = owner(to)
if CanBeSpentBy(from, amount) then
  if CanBeCashedBy(to, amount) then
    if HasSellCapacityFor(amount, to) then Proceed(t)
      -- namely to RequestDebitAck
  else SEND(ErrMsg(SellCapacityViolation(t)), to creditor)
else SEND(ErrMsg(UpperBalanceLimitErr(t)), to creditor)
else SEND(ErrMsg(AvailBalanceViolation(t)), to debitor)
SEND(ErrMsg(DebitorHasSomeProblemWith(t)), to creditor)
RequestDebitAck behavior

- RequestDebitAck does CompleteTransaction without further ado if the amount is Small (less than 100).
- The definition of CompleteTransaction is extended to Debit transfers (the confirmation msg addressee changes):

\[
\text{CompleteTransaction}(\text{debit}, \text{from}, \text{to}, \text{amount}) = \\
\text{let transfer} = (\text{debit}, \text{from}, \text{to}, \text{amount}) \quad -- \text{symmetric to credit} \\
\text{APPEND(
}\text{transact(transfer, now), Ledger}) \\
\text{SEND(}
\text{Confirmed(transfer), to owner(to))} \\
\text{saleVolume(to)} := \text{saleVolume(to)} + \text{amount} \\
\]

DebitAmountLimitsCheck(t), if Small(amount) including CompleteTransaction(t), requires exclusive db-access.

- For other amounts it creates a OneTimePassword for an agreement request sent to the debitor and records the transaction as pending.

- HandleDebitAckCompletion can be executed when an answer msg for the pending Transaction arrives or when Expired(otp).
RequestDebitAck component

RequestDebitAck(transfer) =

let (debit, from, to, amount) = transfer
let debitor = owner(from)  creditor = owner(to)
if Small(amount)
    then COMPLETE_TRANSACTION(transfer)
else
    let otp = new (OTP)
    let pendgT = (otp, (transfer))
    birthTime(otp) := now
    INSERT(pendgT, PendingTransact)
    status(pendgT) := pending
    SEND(AgreementReq(pendgT), to debitor)
Debit transaction phase 3: **HANDLEDEBITACKCOMPLETION**

**HANDLEDEBITACKCOMPLETION** splits into two suboperations:
- a **DEBITACKCOMPLETION** if the debitor’s ack arrives in time
- a **DEBITACKREQUESTEXPIRED** otherwise

For **DEBITACKCOMPLETION** there are 3 disjoint cases to consider when an **AnswerMsg(pendingT)** arrives from the debitor:

1. no such pendingT exists (maybe any more) in **PendingTransact** or its otp is **Expired**. Such a msg is simply discarded.
2. **AnswerMsg(pendingT)** is a **DebitRejectMsg**. Then pendingT is moved from **PendingTransact** to **RejectedTransact** and its status updated to **rejected**.
3. **AnswerMsg(pendingT)** is a **DebitAcceptMsg**. Then, after a **FINALDEBITAMTLIMITSCHECK** with positive outcome, the transaction status can be updated from pending to performed to **COMPLETETRANSACTION**.
DebitAckCompletion component

DebitAckCompletion =

if Received(AnswerMsg(pendgT), from debitor) then
  if pendgT ∈ PendingTransact and not Expired(pendgT) then
    let (otp, (debit, from, to, a)) = pendgT creditor = owner(to)
    if AnswerMsg(pendgT) = DebitRejectMsg(pendgT) then
      RecordRejected(pendgT)
      Send(RejectionInfo(Debit, a, debitor), to creditor)
    else RecordAccepted(pendgT)
    FinalDebitAmtLimitsCheck (pendgT, CompleteTransaction(pendgT))
  Consume(AnswerMsg(pendgT))
DebitAckCompletion macros

\textbf{Expired}(pendingT) \textbf{iff}
\[ \text{now} - \text{birthtime}(pendingT) > \text{OtpLifetime} \]

\textbf{RecordRejected}(t) =
\[ \text{status}(t) := \text{rejected} \]

\textbf{DELETE}(t, PendingTransact) \quad \textbf{INSERT}(t, RejectedTransact) \quad -- \text{analogous for Accepted}

\textbf{CompleteTransaction}(pendingT) =
\begin{align*}
\text{let} \ (otp, \text{transfer}) &= \text{pendingT} \\
\text{CompleteTransaction}(\text{transfer}) \\
\text{status}(\text{pendingT}) &= \text{performed}
\end{align*}

NB. The two versions of \textbf{CompleteTransaction} for pending transactions and for transfers differ by the number of parameters.
**FinalDebitAmtLimitsCheck** is a refinement of the above defined component **DebitAmountLimitsCheck**:

- into the failure cases an additional clause is inserted which makes the pending Debit transaction rejected and informs the creditor about the reason for rejection.

**FinalDebitAmtLimitsCheck**, in case of a positive check result including **CompleteTransaction**, is required to be transactional

- i.e. with exclusive access to the two critical functions *balance* and *salesVolume*.

  - For some examples of Mutex algorithms one could use to satisfy the exclusive access requirement see the lecture notes [http://modelingbook.informatik.uni-ulm.de/additionalmaterial#mutualexclusion](http://modelingbook.informatik.uni-ulm.de/additionalmaterial#mutualexclusion)
let (otp, (debit, from, to, a)) = pendgT
let debitor = owner(from), creditor = owner(to)
if CanBeSpentBy(cc(debitor), a) then
  if CanBeCashedBy(cc(creditor), a) then
    if HasSellCapacityFor(a, cc(creditor)) then Proceed
    else RejectTransDueTo(SellCapacityViolation, pendgT)
  else RejectTransDueTo(UpperBalanceLimitErr, pendgT)
else RejectTransDueTo(AProblemAtDebitor, pendgT)
  Send(ErrMsg(AvailBalanceViolation(debit, from, to, a)), to debitor)

where RejectTransDueTo(reason, pendgT) =
  Send(ErrMsg(reason(debit, from, to, a)), to creditor)
status(pendgT) := rejected
If the debitor’s AnswerMsg for the Debit request is not Received within OtpLifetime, the Server will reject to perform the transaction and inform creditor and debitor about it.

DebitAckRequestExpired =

if \( t \in \text{PendingTransact} \) and \( \text{Expired}(t) \) then

let \( t = (\text{otp}, (\text{debit}, \text{from}, \text{to}, \text{amount})) \)

DELETE(\( t, \text{PendingTransact} \))

INSERT(\( t, \text{RejectedTransact} \))

status(\( t \)) := rejected

SEND(\( \text{RejectMsg}(\text{Debit}, \text{amount}, \text{debitor}), \text{to creditor} \))

SEND(\( \text{OtpExpiredMsg}(\text{Debit}, \text{amount}, \text{creditor}), \text{to debitor} \))

where \( \text{debitor} = \text{owner}(\text{from}) \), \( \text{creditor} = \text{owner}(\text{to}) \)
Two typical retail/consumer B2C operations

A consumer $\in$ Consumer $\cup$ Consumer$_{\text{Verified}}$ purchase at a retail in Retail $\cup$ Full paid in Euro triggers Sardex to HANDLEB2cEur:

- the EuroAmount is recorded in the income(retail) account
- a reward is issued as SRD-credit to a consumer’s Sardex account
  - A consumer $\in$ Consumer remains anonymous (a card represents its account) until it becomes a Consumer$_{\text{Verified}}$ member
    - namely by a registration action that we do not model here
- the Euro fee is paid by retail to the Sardex company

A consumer who is registered as Consumer$_{\text{Verified}}$ member can pay also in SRD, thereby triggering a HANDLEB2cSrd operation.

NB. These 2 ops do not involve any group transfer or account type checks so that we model them directly, not as Credit/Debit op instances.
HandleB2cEur component

HandleB2cEur = -- typically triggered via POS

if Received(B2CEurMsg(amount, consumer), from retail)
and consumer ∈ Consumer ∪ Consumer_Verified
and retail ∈ Retail ∪ Full then
  if ThereIsEnoughPrepaidFeeFor(amount, retail)
    and ThereAreEnoughCreditsFor(amount, retail)
  then
    HandleEuroPayment(amount, retail)
    HandleRewardPayment(amount, retail, consumer)
    HandleFeePayment(amount, retail)
  else
    IssueWarning(NotEnoughFundsInPrepaidOrCCAccounts)
    Consume(Msg(amount, from consumer))
**HANDLEB2CEur macros**

**ThereIsEnoughPrepaidFeeFor**(amount, retail) **iff**

\[ \text{balance}_{\text{retail}}(\text{prepaid}_{\text{retail}}) \geq \text{B2CEuroFee}_{\text{retail}}(\text{amount}) \]

**ThereAreEnoughCreditsFor**(amount, retail) **iff**

\[ \text{availableBalance}_{\text{cc}_{\text{retail}}} \geq \text{rewardRate}_{\text{retail}}(\text{amount}) \]

**HANDLEEUROPAYMENT**(amount, retail) =

\[ \text{balance}(\text{income}_{\text{retail}}) := \text{balance}(\text{income}_{\text{retail}}) + \text{amount} \]

**HANDLEREWARDPAYMENT**(amount, retail, cons) =

\[ \text{balance}(\text{cc}_{\text{retail}}) := \text{balance}(\text{cc}_{\text{retail}}) - \text{rewardRate}_{\text{retail}}(\text{amount}) \]

\[ \text{balance}(\text{cc}_{\text{cons}}) := \text{balance}(\text{cc}_{\text{cons}}) + \text{rewardRate}_{\text{retail}}(\text{amount}) \]

**HANDLEFEEPAYMENT**(amount, retail) =

\[ \text{balance}(\text{prepaid}_{\text{retail}}) := \]

\[ \text{balance}(\text{prepaid}_{\text{retail}}) - \text{B2CEuroFee}_{\text{retail}}(\text{amount}) \]
**HandleB2cSrd**

*HandleB2cSrd* is triggered by a member of *Consumer_Verified* at a *retail ∈ Retail ∪ Full* when the retailer accepts a purchase the member pays partly by accumulated SRD rewards, the rest in Euro.

```
HandleB2cSrd = -- amount is the total purchase value
    if Received(B2CSrdMsg(amount, consumer), from retail)
        and consumer ∈ Consumer_Verified
        and retail ∈ Retail ∪ Full then
            PayWithReward(amount, consumer, retail)
            Consume(SrdB2CMsg(amount, consumer))
    where PayWithReward(amount, consumer, retail) =
        balance(cc_consumer) :=
            balance(cc_consumer) − acceptanceRate(amount)
        balance(cc_retail) := balance(cc_retail) + acceptanceRate(amount)
```

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
MngrOps component RechargePrepaid

- Upon the receipt of a fee prepayment by a retailer through normal banking channels, the prepaid amount is added to $prepaid_{retail}$. – We skip the additional updates for double-entry bookkeeping.
- Before $prepaid_{retail}$ reaches zero, a LowPrepayAlert msg is sent to the retailer.

**RechargePrepaid** =

if $Received(EuroFeePrepaymentMsg(amount, \text{from retail}))$

and $retail \in Retail \cup Full$

then

$balance(prepaid_{retail}) := balance(prepaid_{retail}) + amount$

Consume($EuroFeePrepaymentMsg(amount, \text{from retail})$)
Components to handleAlerts

**LowPrepayAlert** =

\[
\text{forall } \text{retail} \in \text{Retail} \cup \text{Full} \quad \text{if } \text{CloseToZero(balance(prepaid}_{\text{retail}})) \text{ then }
\]

\[
\text{Send(PrepaymentAlertMsg(lowPrepaidBalance), to retail)}
\]

Analogously for other alerts concerning CC-accounts, e.g.

**LowBalanceAlert** iff availableBalance < lowBalanceAlert

--- small amount of money left for further purchases in SRD

**HighBalanceAlert** iff balance > 0 and

upperBalanceLimit − balance < highBalanceAlert

--- small space left for further sales in SRD

**HighVolumeAlert** iff availableSaleCapacity < highVolumeAlert
Recap **HandleSardexOpns**

- **HandleCreditPreviewReq**
- **HandleCreditPerformReq**

---

- **HandleDebitPreviewReq**
- **HandleDebitPerformReq**
- **HandleDebitAckCompletion**

---

- **HandleB2cEur**
- **HandleB2cSrd**

---

- **RechargePrepaid**
- **HandleAlerts**

---

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
What makes Sardex (not the company, but the mutual credit system) a system of communicating agents is the interaction of:

- instances of MemberOpns each circuit member is equipped with
  - those which trigger and support to HandleSardexOpns
- program HandleSardexOpns the network server sardex executes
  - in reality with many more ops than those shown here
- operations of other agents we did not model here, e.g. of
  - a software systemAdministrator
  - brokers (which survey and support the well functioning of the circuit)
  - agents belonging to associated circuits

Essentially, MemberOpns consists of rules which describe the Send and Receive actions of circuit members, here concerning the request/response msgs exchanged with HandleSardexOpns.
MemberOpns

- Members supply their various request and response msgs by filling in appropriate fields on a screen. The editing functionality is clear so that we skip rules which describe how to prepare and send a $\text{Credit/DebitPreviewReq}(t)$ and similar requests.

For $\text{Credit/Debit Perform requests}$, the only relevant additional constraint is that these requests are triggered by receiving an OK message, sent by $\text{sardex}$, for the corresponding Preview request.

\[
\text{if } \text{Received}(\text{YouMayTriggerPerformReq}(t), \text{from sardex}) \text{ then}
\]
\[
\text{if } \text{transferKind}(t) = \text{credit} \text{ then}
\]
\[
\text{SEND}(\text{CreditPerformReq}(t), \text{to sardex})
\]
\[
\text{if } \text{transferKind}(t) = \text{debit} \text{ then}
\]
\[
\text{SEND}(\text{DebitPerformReq}(t), \text{to sardex})
\]
\[
\text{CONSUME}(\text{YouMayTriggerPerformReq}(t))
\]
In case of a Debit operation the debitor is expected to confirm a received debit request by sending a *DebitAckMsg*; otherwise the debitor should send a *DebitRejectMsg* to the network server.

```plaintext
if Received(AgreementReq(otp, transfer), from sardex) then
    let (debit, from, to, amount) = transfer
    if Agreed(amount, owner(to), otp)
        then SEND(DebitAckMsg(otp, transfer), to sardex)
        else SEND(DebitRejectMsg(otp, transfer), to sardex)
    CONSUME(AgreementReq(otp, transfer))
```

We skip the obvious rules for receiving an *ErrMsg* or pure information msgs like *RejectionInfo*, *OtpExpiredMsg*, etc.
The requirement: The balance of an account can go negative, but the debt must be ‘paid back’ within 12 months of when it was incurred.

- The debt is not towards Sardex S.p.A. and it is not bilateral towards a single other member. Rather, the debt is towards the circuit as a whole.
- Therefore, the indebted circuit member can ‘pay back’ its debt by selling its products and services to other members using Sardex accounts for the payment.

More precisely, every single debt incurred by a transaction is required

- to be completely recovered, possibly in portions, within a payBackWindow (currently of 12 months)
- to remain recorded (for legal reasons) with its current rest debt value (initially the debt incurred, eventually 0).
Algorithmic refinement idea.

- For any new debt create a debtEntry, inserted into a DebtRecord and consisting of: its birthTime, the debt triggering transfer (opKind, from, to, amount) and the restDebt (initially debt).

- When an account receives a positive credit amount transfer, call UpdateDebtRecord to pay back the corresponding portion of still not-completely-repayed debts in DebtRecord, proceeding sequentially, starting from the oldest unpaid one.

Then the refinement (in fact a conservative refinement) is to simply add CreateDebtEntry and UpdateDebtRecord in parallel to the rules of the following components:

- CompleteTransaction
- HandleRewardPayment in HandleB2cEur
- PayWithReward in HandleB2cSrd
**CreateDebtEntry**\((\text{transfer})\)

\[
\text{CreateDebtEntry}(\text{opKind}, \text{from}, \text{to}, \text{amount}) =
\]

\[
\begin{align*}
\text{if } \text{balance}(\text{from}) - \text{amount} < 0 \text{ then} & \\
\quad & \text{-- balance}(\text{from}) \text{ turns negative}
\end{align*}
\]

\[
\text{let } \text{debtEntry} = \text{new } (\text{DebtRecord}) \\
\text{birthTime}(\text{debtEntry}) := \text{now} \\
\text{transfer}(\text{debtEntry}) := (\text{opKind}, \text{from}, \text{to}, \text{amount}) \\
\text{restDebt}(\text{debtEntry}) :=
\]

\[
\begin{cases} \\
| \text{balance}(\text{from}) - \text{amount} | & \text{if } \text{balance}(\text{from}) > 0 \\
\text{amount} & \text{else}
\end{cases}
\]

NB. Because of the time parameter, two calls of 
**CreateDebtEntry\((t)\)** with the same \(t\) create different debt entries.

- A coarser time scheme implies the need to keep the transfer parameters distinct to identify the corresponding debt entries.
**UpdateDebtRecord**

\[
\text{UpdateDebtRecord}(\text{opKind}, \text{from}, \text{to}, \text{amount}) =
\]

\[
\text{if } \text{amount} \neq 0 \text{ then} \quad \text{-- balance(to) increases}
\]

\[
\text{let } \text{OpenDebtEntries} = \{e \in \text{DebtRecord} \mid \text{restDebt}(e) > 0\}
\]

\[
\text{Update}(\text{OpenDebtEntries}, \text{amount})
\]

\[
\text{where } \text{Update}(E, \text{amount}) = \quad \text{-- a recursive ASM}
\]

\[
\text{if } E \neq \emptyset \text{ then} \quad \text{-- recursion termination condition}
\]

\[
\text{let } \text{entry} = \text{oldest}(E) \quad \text{-- element with earliest birthTime}
\]

\[
\text{if } \text{restDebt(entry)} \geq \text{amount}
\]

\[
\text{then } \text{restDebt(entry)} := \text{restDebt(entry)} - \text{amount}
\]

\[
\text{else } \text{restDebt(entry)} := 0
\]

\[
\text{let } \text{restAmount} = \text{amount} - \text{restDebt(entry)}
\]

\[
\text{Update}(E \setminus \{\text{entry}\}, \text{restAmount}) \quad \text{-- recursive call}
\]

For the definition of recursive ASMs see the References.
Extending Alerts by an UnpaidDebtAlert

The requirement that ‘the debt must be ‘paid back’ within 12 months of when it was incurred’ triggers the question how to enforce the ‘must’.

The question motivates the introduction of an UnpaidDebtAlert for each account involved. This can be done by simply adding to Alerts a corresponding rule in parallel.

– NB. DebtRecord belongs to an account by which it is implicitly parameterized. When we want to make this parameter explicit we write DebtRecord_{acc} or DebtRecord(acc).

UnpaidDebtAlert(acc) iff

forsome entry ∈ DebtRecord_{acc} with restDebt(entry) > 0
now – birthtime(entry) IsCloseTo payBackWindow
Concluding remarks

- The **Sardex** ASM describes a tiny subset of Sardex core services.
  - But the model is *effectively and easily extendable*, due to its component structure and its formulation by proceeding from abstract to more and more detailed descriptions by stepwise refinement.
  - In the INTERLACE project the stepwise refinement technique, which we illustrated here for adding the debt record tracking, has been used throughout to develop the model from scratch.

- The model is a **ground model**, i.e. tailored to express the services of the Sardex mutual credit system at the business logic level of abstraction, *avoiding any only code-related programming terms*.
  - The pseudocode character of ASMs permits to directly describe the computational rules by which a process step changes the underlying abstract state, without need to encode the involved state elements.
  - The pseudocode form of ASMs supports the use of ground models as *guide for the implementation*, as done in the INTERLACE project.
References

  – Ch.2 and Appendix of INTERLACE WP3 Deliverable D3.1 (2018).  
  https://www.interlaceproject.eu/

  – Appendix in INTERLACE Deliverable D2.3 (Final Architecture) of WP2 (Iterative Architecture Requirements and Definition) (2018)

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018  
  http://modelingbook.informatik.uni-ulm.de

It is permitted to (re-) use these slides under the CC-BY-NC-SA licence

[https://creativecommons.org/licenses/by-nc-sa/4.0/](https://creativecommons.org/licenses/by-nc-sa/4.0/)

i.e. in particular under the condition that

- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial