Egon Börger (Pisa)

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

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

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.

The SARDEX S.p.A. company operates an electronic, B2B, zero-interest mutual credit system on the island of Sardinia since 2009.

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/` (2017-2018).

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

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.

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

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.

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

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

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

HandleSardexOpns structure for the selected service opns

HandleSardexOpns = -- parallel composition of components

HandleCreditTransfer HandleDebitTransfer

HandleB2cTransfer Alerts

HandleCreditTransfer = -- 2-phase request/response pattern

HandleCreditPreviewReq HandleCreditPerformReq

HandleDebitTransfer = -- 3-phase request/response pattern

HandleDebitPreviewReq HandleDebitPerformReq

HandleDebitAckCompletion

HandleB2cTransfer =

HandleB2cEur HandleB2cSrd RechargePrepaid

HandleAlerts =

HighBalanceAlert, LowPrepayAlert, ...

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

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

$Group =$

$$\{Company, Retail, Full, Consumer, Consumer_Verified, Mngr\}$$

- We skip for example $Employee$s, 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.

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

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):

$$AcctType = \{CC, Income, Prepaid\}$$

- CC: contains SRD-credit accounts $cc(user)$ *used to pay for purchases in SRD currency* (for each $user \in group$ of any $group \in Group$)
- The sets $Income$ resp. $Prepaid$ of Euro-accounts contain for each $user \in Retail \cup Full$ 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.

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

Typical CC-account metadata

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 extremly 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 extremly 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.

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

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

$availableBalance = balance + creditLimit \in Nat$

$LowBalanceAlert$ **iff** $availableBalance < lowBalanceAlert$

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

$upperBalanceLimit \in Nat \setminus \{0\}$

-- 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*

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

Other similar CC-account metadata

- 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 $creditLimit := 0$

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

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$>$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 $InterTradeEuroFee$, $B2CEuroFee$.

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

Group transfer type constraints on Credit/Debit transactions

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

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

We split the transfer type function TT as follows into subfunctions

$$TT_{op,cur} : Group \rightarrow \{G \mid G \subseteq Group\}$$

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

- $TT_{op,cur}(fromGroup)$ 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.

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

Group Transfer Type Constraints for $TT_{Credit,SRD}$

- A *Credit SRD*-op can be started only by members of
  – $Company$ or $Full$ or $Mngr$ or $Consumer_Verified$

  $MayTriggerCreditOp(mbr)$ **iff**

  $group(mbr) \in \{Company, Full, Mngr, Consumer_Verified\}$
- The target groups allowed for a *Credit SRD*-op are defined by:
  – every $Company \cup Full \cup Mngr$ member can trigger a *Credit SRD*-op to a member of $Company \cup Full \cup Mngr$
  – $Consumer_Verified$ members can trigger a *Credit SRD*-op to $Retail \cup Full$-members (NB. We simplify these opns below)

Equationally this is expressed as follows:

$$TT_{Credit,SRD}(G) = \begin{cases} \{Company, Full, Mngr\} & \textbf{if } G \in \{Company, Full, Mngr\} \\ \{Retail, Full\} & \textbf{if } G = Consumer_Verified \end{cases}$$

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

Group Transfer Type Constraints for $TT_{Debit,SRD}$

- For $Debit\ SRD$-ops the following source/target groups are allowed:
  - *Every* $Company \cup Full \cup Mngr$ *member* $MayTriggerDebitOp$ *(in* SRD*) that draws on a member of* $Company \cup Full$.
  - Some other constraints (there are more) we do not use in this model:
    - Every member of $Retail \cup Full$ 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

$TT_{Debit,SRD}(G) =$ -- we use only the first clause

$$\begin{cases} \{Company, Full, Mngr\} & \textbf{if } G \in \{Company, Full\} \\ \{Retail, Full\} & \textbf{if } G = Consumer_Verified \\ Mngr & \textbf{if } G = Retail \end{cases}$$

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

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-$Credit$ transfer op is triggered by a debitor, starts at a $fromAcc \in CC$ and has as allowed $toAcc$ a CC account.
- An SRD-$Debit$ op is triggered by a creditor, starts from a debitor's $fromAcc \in CC$ and goes to the creditor's $toAcc \in CC$.

$AccT_{op,cur} : AccountType \to \{Acct \mid Acct \subseteq AccountType\}$

$AccT_{Credit,SRD}(CC) = AccT_{Debit,SRD}(CC) = \{CC\}$

$MayStartCreditOpns(acct)$ **iff** $acct \in CC$

$MayStartDebitOpns(acct)$ **iff** $acct \in CC$

- NB. Such definitions (of ASM functions, predicates, rules, etc.) by case distinction *support componentwise extensions to new requirements*
  - e.g. to new account type arguments for $AccT_{Credit,SRD}$ we do not use here, which are similar to the above clauses for $TT_{Debit,SRD}$

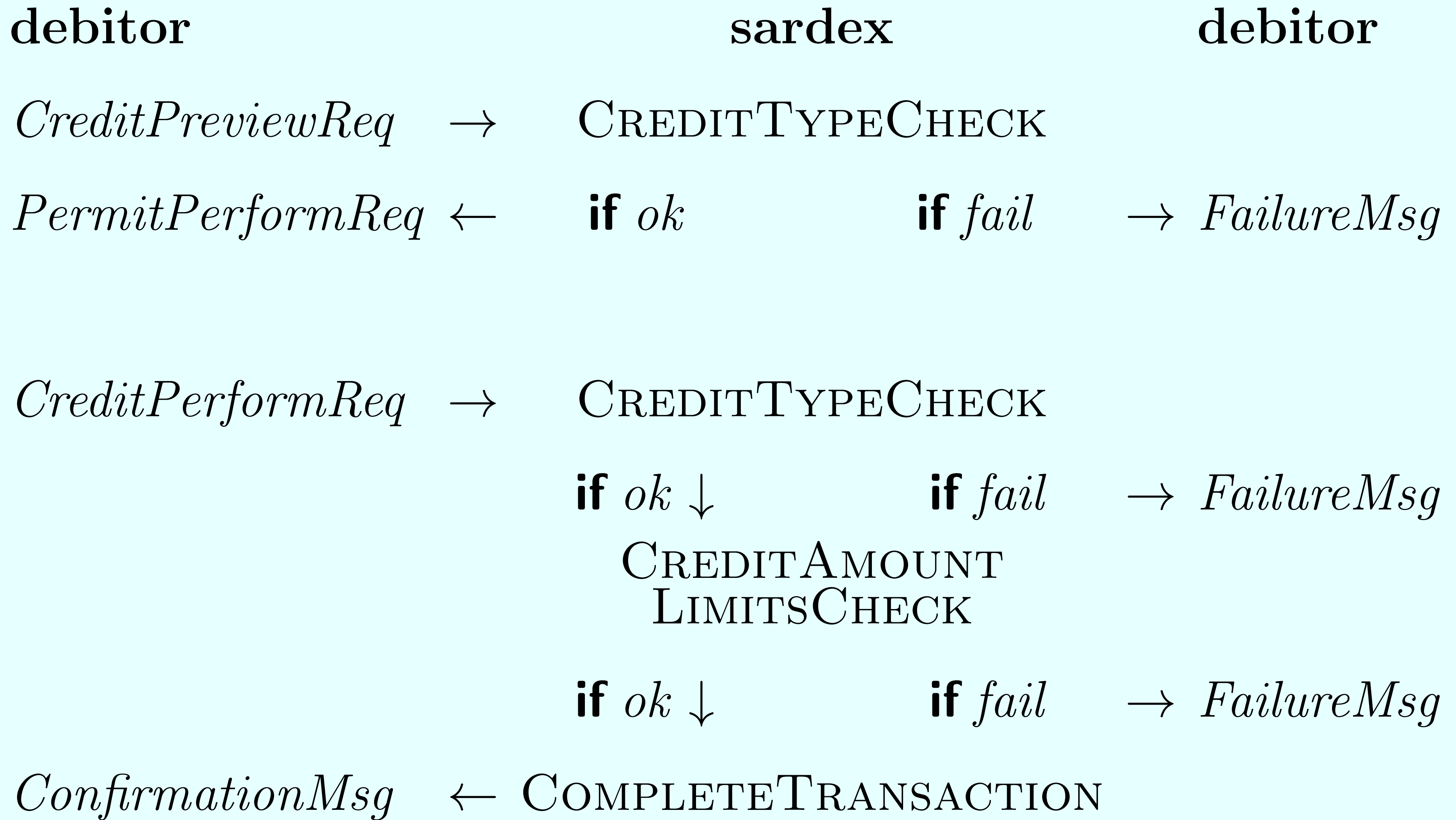
Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0

Request/Response pattern of Credit/Debit operations

- 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

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

HandleCreditTransfer request/response control flow



NB. Arrows indicate the sequential control and data flow.

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

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.

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

How HandleCreditPreviewReq interacts with members

- HandleCreditPreviewReq triggers a CreditTypeCheck concerning the group/account related access rights wrt the parameters of the member's *CreditPreviewReq*uest
  - upon a positive check result the server will PermitPerformReq
  - otherwise it will send a constraint violation msg to the member

HandleCreditPreviewReq$((from, to, amount), mbr) =$

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

if $Received(CreditPreviewReq(t),$ **from** $mbr)$ **then**

CreditTypeCheck$(t, mbr,$ PermitPerformReq$(t))$

Consume$(CreditPreviewReq(t))$

where

PermitPerformReq$(t) =$ -- NB. $mbr = owner(from)$

Send$(YouMayTriggerPerformReq(t),$ **to** $mbr)$

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

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

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

$\textsc{CreditTypeCheck}((from, to, amount), mbr, \textsc{Proceed})$

let $t = (credit, from, to, amount)$ -- the transfer params
if $mbr = owner(from)$ **and** $MayTriggerCreditOp(mbr)$ **then**
-- check group transfer type constraints
  if forsome $g \in TT_{Credit,SRD}(group(mbr))$ $owner(to) \in g$ **then**
    if $MayStartCreditOpns(from)$ **then**
-- check account connectivity constraints
      if $acctSet(to) \in AccT_{Credit,SRD}(acctSet(from))$ **then**
        $\textsc{Proceed}(t)$ -- in positive case $\textsc{PermitPerformReq}(t)$
        else $\textsc{Send}(ErrMsg(CreditTargetAcctError(t))$, **to** $mbr)$
    else $\textsc{Send}(ErrMsg(CreditSourceAcctError(t))$, **to** $mbr)$
  else $\textsc{Send}(ErrMsg(CreditTargetGroupError(t))$, **to** $mbr)$
else $\textsc{Send}(NotAcctOwnerOrCreditSourceGroupError(t)$, **to** $mbr)$

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

$\textsc{CreditAmountLimitsCheck}((from, to, amount), \textsc{Proceed})$

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

if $CanBeSpentBy(from, amount)$ **then**

  if $CanBeCashedBy(to, amount)$ **then**

    if $HasSellCapacityFor(amount, to)$ **then** $\textsc{Proceed}(t)$

-- namely to $\textsc{CompleteTransaction}$

      else $\textsc{Send}(ErrMsg(CapacityViolation(t)),$ **to** $owner(from))$

  else $\textsc{Send}(ErrMsg(UpperBalanceLimitErr(t)),$ **to** $owner(from))$

else $\textsc{Send}(ErrMsg(AvailBalanceViolation(t)),$ **to** $owner(from))$

where

  $CanBeSpentBy(from, a)$ **iff** $availableBalance(from) \geq a$

  $CanBeCashedBy(to, a)$ **iff**

    $balance(to) + a \leq upperBalanceLimit(to)$

  $HasSellCapacityFor(a, to)$ **iff** $a \leq availableSaleCapacity(to)$

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

CompleteTransaction **component**

The abstract fctn *transact* encodes the *transfer* info that is appended to the *Ledger* and results in an update of the critical *balance* and *saleVolume* locations (constrained, as usual for db transactions, by an exclusive access condition in TryToCompleteCreditOpn).

CompleteTransaction(*transfer*) =

let $(credit, from, to, amount) = transfer$

Append($transact(transfer, now), Ledger$) -- timestamp *now*

Send($Confirmed(transfer)$, **to** $owner(from)$)

where Append($transact(transfer, now), Ledger$) includes

$balance(from) := balance(from) - amount$

$balance(to) := balance(to) + amount$

$saleVolume(to) := saleVolume(to) + amount$

NB. Here we abstract from the underlying concrete db-scheme.

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

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

$$\text{HandleDebitTransfer} = \begin{cases} \text{HandleDebitPreviewReq} \\ \text{HandleDebitPerformReq} \\ \text{HandleDebitAckCompletion} \end{cases}$$

HandleDebitAckCompletion splits into two suboperations:

- a DebitAckCompletion if the debitor's ack arrives in time
- a DebitAckRequestExpired component, otherwise

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

HandleDebitTransfer request/response flow: phase 1-2

creditor		sardex			debitor
DebitPreviewReq	$\rightarrow$	DebitTypeCheck			
PermitPerformReq	$\leftarrow$	**if** *ok*	**else** *FailureMsg*		
DebitPerformReq	$\rightarrow$	DebitTypeCheck			
		if *ok* $\downarrow$	**else** *FailureMsg*		
		DebitAmountLimitsCheck			
		if *ok* $\downarrow$	**else** *FailureMsg*		
		RequestDebitAck			
		if *LargeAmount* **then**		$\rightarrow$	*AckReq*
ConfirmationMsg	$\leftarrow$	**else** CompleteTransaction			

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

HandleDebitTransfer request/response flow: phase 3

creditor		**sardex**		**debitor**
RejectMsg	$\leftarrow$	**if** *ExpiredAckReq*	$\rightarrow$	*ExpireMsg*
		DebitAckCompletion	$\leftarrow$	*AnswerMsg*
RejectMsg	$\leftarrow$	**if** *RejectAnsw* **else** $\downarrow$		
		FinalDebitAmtLimitsCheck		
RejectMsg	$\leftarrow$	**if** *CheckFails*	$\rightarrow$	*FailureMsg*
		else $\downarrow$		
Confirmation	$\leftarrow$	CompleteTransaction		

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

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 $DebitPreviewReq$uest
  - 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** $creditor)$

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

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 $g \in TT_{Debit,SRD}(group(debitor))$ $mbr \in g$ **then**
if $MayStartDebitOpns(to)$ **then**
-- check account connectivity constraints
if $acctSet(from) \in AccT_{Debit,SRD}(acctSet(to))$ **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)

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

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$(t))$

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

DebitAmountLimitsCheck component

$\textsc{DebitAmountLimitsCheck}((from, to, amount), \textsc{Proceed}) =$

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** $\textsc{Proceed}(t)$

-- namely to RequestDebitAck

    else $\textsc{Send}(ErrMsg(SellCapacityViolation(t))$, **to** $creditor)$

  else $\textsc{Send}(ErrMsg(UpperBalanceLimitErr(t))$, **to** $creditor)$

else $\textsc{Send}(ErrMsg(AvailBalanceViolation(t))$, **to** $debitor)$

  $\textsc{Send}(ErrMsg(DebitorHasSomeProblemWith(t))$, **to** $creditor)$

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

$\textsc{RequestDebitAck}$ **behavior**

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

    $\textsc{CompleteTransaction}(debit, from, to, amount) =$

    let $transfer = (debit, from, to, amount)$ -- symmetric to credit

    $\textsc{Append}(transact(transfer, now), Ledger)$

    $\textsc{Send}(Confirmed(transfer),$ **to** $owner(to))$

    $saleVolume(to) := saleVolume(to) + amount$

    $\textsc{DebitAmountLimitsCheck}(t)$, if $Small(amount)$ including $\textsc{CompleteTransaction}(t)$, requires exclusive db-access.
- For other $amount$s it creates a $OneTimePassword$ for an agreement request sent to the $debitor$ and records the transaction as $pending$.
- $\textsc{HandleDebitAckCompletion}$ can be executed when an answer msg for the $pendingT$ransaction arrives or when $Expired(otp)$.

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

REQUESTDEBITACK component

$\textsc{RequestDebitAck}(transfer) =$

let $(debit, from, to, amount) = transfer$

let $debitor = owner(from) \quad creditor = owner(to)$

if $Small(amount)$

then $\textsc{CompleteTransaction}(transfer)$

else

let $otp = \textbf{new}\ (OTP)$

let $pendgT = (otp, (transfer))$

$birthTime(otp) := now$

$\textsc{Insert}(pendgT, PendingTransact)$

$status(pendgT) := pending$

$\textsc{Send}(AgreementReq(pendgT),$ **to** $debitor)$

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

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.

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

DebitAckCompletion component

DebitAckCompletion =

if *Received*(*AnswerMsg*(*pendgT*), **from** *debitor*) **then**

if $pendgT \in PendingTransact$ **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*))

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

$\text{DebitAckCompletion}$ **macros**

$Expired(pendingT)$ **iff**

$\quad now - birthtime(pendingT) > OtpLifetime$

$\text{RecordRejected}(t) =$ -- analogous for Accepted

$\quad status(t) := rejected$

$\quad \text{Delete}(t, PendingTransact) \quad \text{Insert}(t, RejectedTransact)$

$\text{CompleteTransaction}(pendingT) =$

$\quad$ **let** $(otp, transfer) = pendingT$

$\qquad \text{CompleteTransaction}(transfer)$

$\qquad status(pendingT) := performed$

NB. The two versions of $\text{CompleteTransaction}$ for pending transactions and for transfers differ by the number of parameters.

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

FinalDebitAmtLimitsCheck

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

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

FinalDebitAmtLimitsCheck($pendgT$, Proceed)

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** $creditor$)

  $status(pendgT) := rejected$

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

DebitAckRequestExpired component

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.

$\textsc{DebitAckRequestExpired} =$

if $t \in \mathit{PendingTransact}$ **and** $\mathit{Expired}(t)$ **then**

  let $t = (otp, (debit, from, to, amount))$ -- NB: otp is unique

    $\textsc{Delete}(t, \mathit{PendingTransact})$

    $\textsc{Insert}(t, \mathit{RejectedTransact})$

    $status(t) := rejected$

    $\textsc{Send}(\mathit{RejectMsg}(\mathit{Debit}, amount, debitor),$ **to** $creditor)$

    $\textsc{Send}(\mathit{OtpExpiredMsg}(\mathit{Debit}, amount, creditor),$ **to** $debitor)$

      where $debitor = owner(from), creditor = owner(to)$

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

Two typical retail/consumer B2C operations

A $consumer \in Consumer \cup Consumer_Verified$ 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 $Consumer_Verified$ 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 $Consumer_Verified$ 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.

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

HandleB2cEur component

HandleB2cEur = -- typically triggered via POS

if *Received*(*B2CEurMsg*(*amount*, *consumer*), **from** *retail*)
  and $consumer \in Consumer \cup Consumer_Verified$
  and $retail \in Retail \cup Full$ **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*))

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

$\textsc{HandleB2cEur}$ **macros**

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

$\quad balance_{retail}(prepaid_{retail}) \geq B2CEuroFee_{retail}(amount)$

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

$\quad availableBalance(cc_{retail}) \geq rewardRate_{retail}(amount)$

$\textsc{HandleEuroPayment}(amount, retail) =$

$\quad balance(income_{retail}) := balance(income_{retail}) + amount$

$\textsc{HandleRewardPayment}(amount, retail, cons) =$

$\quad balance(cc_{retail}) := balance(cc_{retail}) - rewardRate_{retail}(amount)$

$\quad balance(cc_{cons}) := balance(cc_{cons}) + rewardRate_{retail}(amount)$

$\textsc{HandleFeePayment}(amount, retail) =$

$\quad balance(prepaid_{retail}) :=$

$\qquad balance(prepaid_{retail}) - B2CEuroFee_{retail}(amount)$

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

HandleB2cSrd

HandleB2cSrd is triggered by a member of $\mathit{Consumer_Verified}$ at a $\mathit{retail} \in \mathit{Retail} \cup \mathit{Full}$ when the retailer accepts a purchase the member pays partly by accumulated SRD rewards, the rest in Euro.

$\textsc{HandleB2cSrd} =$ -- amount is the total purchase value

$$
\begin{aligned}
&\textbf{if } \mathit{Received}(\mathit{B2CSrdMsg}(\mathit{amount}, \mathit{consumer}), \textbf{from } \mathit{retail}) \\
&\quad \textbf{and } \mathit{consumer} \in \mathit{Consumer_Verified} \\
&\quad \textbf{and } \mathit{retail} \in \mathit{Retail} \cup \mathit{Full} \textbf{ then} \\
&\qquad \textsc{PayWithReward}(\mathit{amount}, \mathit{consumer}, \mathit{retail}) \\
&\qquad \textsc{Consume}(\mathit{SrdB2CMsg}(\mathit{amount}, \mathit{consumer})) \\
&\textbf{where } \textsc{PayWithReward}(\mathit{amount}, \mathit{consumer}, \mathit{retail}) = \\
&\quad \mathit{balance}(cc_{\mathit{consumer}}) := \\
&\qquad \mathit{balance}(cc_{\mathit{consumer})}) - \mathit{acceptanceRate}(\mathit{amount}) \\
&\quad \mathit{balance}(cc_{\mathit{retail}}) := \mathit{balance}(cc_{\mathit{retail}}) + \mathit{acceptanceRate}(\mathit{amount})
\end{aligned}
$$

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 *retail*er 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 *retail*er.

RechargePrepaid =

if $Received(EuroFeePrepaymentMsg(amount,$ **from** $retail))$

and $retail \in Retail \cup Full$

then

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

Consume$(EuroFeePrepaymentMsg(amount,$ **from** $retail))$

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

Components to HANDLEALERTS

$\text{LowPrepayAlert} =$

forall $retail \in Retail \cup Full$

if $CloseToZero(balance(prepaid_{retail}))$ **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** $availableSaleCapacity < highVolumeAlert$

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

Recap HandleSardexOpns

HandleCreditPreviewReq HandleCreditPerformReq	-- HandleCreditTransfer
HandleDebitPreviewReq HandleDebitPerformReq HandleDebitAckCompletion	-- HandleDebitTransfer
HandleB2cEur HandleB2cSrd RechargePrepaid	-- HandleB2CTransfer

HandleAlerts

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

The SARDEX system of communicating agents

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 *systemAdmin*istrator
  - 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.

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

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 $Credit/DebitPreviewReq(t)$ and similar requests.

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

if $Received(YouMayTriggerPerformReq(t),$ **from** $sardex)$ **then**
  if $transferKind(t) = credit$ **then**
    $\text{SEND}(CreditPerformReq(t),$ **to** $sardex)$
  if $transferKind(t) = debit$ **then**
    $\text{SEND}(DebitPerformReq(t),$ **to** $sardex)$
  $\text{CONSUME}(YouMayTriggerPerformReq(t))$

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

MemberOpns (Cont'd)

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.

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.

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

A characteristic refinement example: debt record tracking

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).

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

How to integrate the debt record tracking requirement

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

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

$\textsc{CreateDebtEntry}(transfer)$

$\textsc{CreateDebtEntry}(opKind, from, to, amount) =$

if $balance(from) - amount < 0$ **then**

-- $balance(from)$ turns negative

let $debtEntry =$**new** $(DebtRecord)$

$birthTime(debtEntry) := now$

$transfer(debtEntry) := (opKind, from, to, amount)$

$restDebt(debtEntry) :=$

$$\begin{cases} \mid balance(from) - amount \mid & \textbf{if } balance(from) > 0 \\ amount & \textbf{else} \end{cases}$$

NB. Because of the time parameter, two calls of $\textsc{CreateDebtEntry}(t)$ 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.

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

UPDATEDEBTRECORD$(opKind, from, to, amount) =$

if $amount \neq 0$ **then** -- $balance(to)$ increases

let $OpenDebtEntries = \{e \in DebtRecord \mid restDebt(e) > 0\}$

UPDATE$(OpenDebtEntries, amount)$

where UPDATE$(E, amount) =$ -- a recursive ASM

if $E \neq \emptyset$ **then** -- recursion termination condition

let $entry = oldest(E)$ -- element with earliest $birthTime$

if $restDebt(entry) \geq amount$

then $restDebt(entry) := restDebt(entry) - amount$

else $restDebt(entry) := 0$

let $restAmount = amount - restDebt(entry)$

UPDATE$(E \setminus \{entry\}, restAmount)$ -- recursive call

For the definition of recursive ASMs see the References.

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

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 *acc*ount involved. This can be done by simply adding to ALERTS a corresponding rule in parallel.
  - NB. *DebtRecord* belongs to an *acc*ount 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 $entry \in DebtRecord_{acc}$ **with** $restDebt(entry) > 0$

$now - birthtime(entry)\ IsCloseTo\ payBackWindow$

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

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.

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

References

- P. Dini, L. Carboni, G. Littera, E. Börger, and C. Nehaniv: Refinement of the INTERLACE Business Logic Specification.
  - Ch.2 and Appendix of INTERLACE WP3 Deliverable D3.1 (2018). https://www.interlaceproject.eu/
- E. Hirsch and P. Dini: Additional Functional Requirements and Business Logic Model.
  - 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
- E. Börger and K.-D. Schewe: *A Behavioural Theory of Recursive Algorithms*. (2019, Submitted)

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

Copyright Notice

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

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