interlace_	**Interacting Decentralized Transactional and Ledger Architecture for Mutual Credit**

WP3

Iterative Demonstrator Implementation

Deliverable D3.1

First Demonstrator Implemenation

Horizon 2020	**Project funded by the European Commission** Information and Communication Technologies FET OPEN Launchpad Project Grant no. 754494

Contract Number: 754494

Project Acronym: INTERLACE

Deliverable No: D3.1

Due Date: 31/01/2018

Delivery Date: 27/07/2018

Author: Eduard Hirsch, Thomas Heistracher (SUAS), Paolo Dini (UH), Egon Börger (UNI PASSAU), Luca Carboni, Maria Luisa Mulas, and Giuseppe Littera (SARDEX)

Partners contributed: Chrystopher L. Nehaniv (UH)

Made available to: Public

Versioning

Version	Date	Name, organization
1	20/01/2018	Paolo Dini (UH)
2	30/04/2018	Paolo Dini (UH), Eduard Hirsch (SUAS)
3	30/06/2018	Eduard Hirsch (SUAS), Paolo Dini (UH), Egon Börger (UNI PASSAU)
4	31/07/2018	PAOLO DINI (UH), EDUARD HIRSCH (SUAS), MARIA LUISA MULAS (SARDEX), Egon Börger (UNI PASSAU)

Internal Reviewer: Paolo Dini (UH)

Abstract

This report describes the ASIM implementation defined by Deliverable D2.1: Requirements and Architecture Definition, as well as mechanisms used to ensure a stable and shared runtime environment and to guarantee testability and easy execution of the ASIM model of the business logic. The implementation is based on a refinement of the requirements that is detailed in this report, along with an updated formal specification, relative to the original ASIM definitions of D2.1, in the Appendix. Finally, a presentation of the runtime environment is given and discussed in the context of the future connection to the blockchain-based backend.

Table of Contents

Chapter 1

Introduction

Eduard Hirsch, Thomas Heistracher and Paolo Dini

1.1 Objectives and Motivation

The overarching objective of this report is to present the concepts and steps necessary for planning, implementing, and running a demonstrator of the distributed ledger specification for the Sardex system developed by the INTERLACE project and based on the Abstract State Interaction Machines (ASIMs) [4, 5]. The ASIMs are a generalization of the Abstract State Machines (ASMs) [2, 1] that extends their functionality to support communications between ASIMs running concurrently and asynchronously on different hosts. More specifically, this report provides a consistent view of the Interaction Computing Execution Framework (ICEF), of the CoreASIM executable modelling language, and of the implementation of the INTERLACE business logic in this environment. This first demonstrator implementation reflects the core architectural decisions made for a full-stack environment.

Before addressing the modelling and execution framework and the implementation-related work done, this report also presents a refinement of the requirements laid out in Deliverable D2.1 [3] for the purpose of developing a prototypical software implementation. In addition, these refined requirements are also expressed as a formal ASIM specification in the Appendix of this report. The Appendix should be seen as the next iteration of the specification provided in Chapter 3 of D2.1, based on the updated requirements description presented in Chapter 2 of the present report. The ASIM specification in the Appendix forms the basis for the ICEF-compatible notation used for the actual demonstrator implementation, discussed in Chapter 4.

The main motivations of the implementation effort are to demonstrate the viability of the underlying Interaction Computing approach and to act as a proof-of-concept for the INTERLACE payment specifications.

1.2 Scope and Organization

Within the scope of this report, the refinements of the business logic specification are derived and explained in Chapter 2 through a textual description together with illustrations, diagrams, and tables. Chapter 3 explains the configuration of the execution environment with all necessary software products and tools, the setup of the ICEF, and the CoreASIM language for implementing executable models. Chapter 4 discusses in detail the main implementation aspects and features in CoreASIM. The report concludes by giving guidelines and addressing expected challenges for further development such as testing and increasing stability of the ICEF framework. A more detailed discussion of software testing and of the achieved outcomes will be documented in Deliverable D3.2.

Chapter 2

Refinement of the INTERLACE Business Logic Specification

Paolo Dini, Luca Carboni, Giuseppe Littera, Egon Börger and Chrystopher Nehaniv

2.1 Context and Overview

The business logic specification provided in Deliverable D2.1 [3] concerns user-initiated transactions. Although this is a subset of all possible operations (initiated by the users or by the System, which we refer to as $SysAdmin$) that a mutual credit system platform must support, it is a viable starting point for Ftesting an initial executable CoreASIM model. D2.1 did not provide all the details of the transaction request operations, it left their specification at a fairly abstract level. This chapter provides the next step in the iterative refinement of the specification in the form of a detailed description of the Permissioning workflow, whose implementation as part of the CoreASIM model is then described in Chapter 4. The description of the Permissioning iterative refinement relies on graphical and tabular depictions of the variables and functions involved. As this reflects the process that was used to build a shared understanding within the INTERLACE team itself, it is hoped that it will also make it easier for newcomers to the INTERLACE open source community to understand the implementation and the logic behind it. Finally, the refined requirements presented in this chapter are also formalized completely in the Appendix to this report, which could be seen as an iteration on Chapter 3 of D2.1.

At the highest level the INTERLACE transactional platform is a dynamic information system that interacts with a database at the backend and with live users at the front-end. As already discussed in D2.1, the transaction engine relies on many categories of concepts that, together, constitute the domain model of the system. According to the Abstract State Machine (ASM) methodology [2, 1], these have been subdivided into Abstract State Interaction Machines (ASIMs) [4, 5] comprised of rules, programs, and various kinds of functions. Whereas concepts such as 'user' or 'transaction amount' are immediately clear, many more concepts and data structures are needed to specify and model the system. Before we get into the details, it helps to introduce three high-level perspectives through which the transaction engine can be characterized as an abstract machine and as a social construct: privacy (private-public dichotomy), dynamics (frequency domain), and transaction workflow already mentioned (time domain).

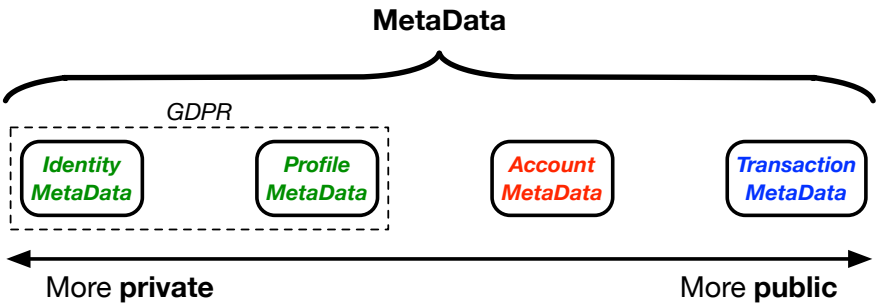

Fig. 2.1: **Relative privacy of different types of meta-data of the INTERLACE transaction engine**

The first principle that influences the architecture of the system at a global level is the need to comply with the GDPR directive. Figure 2.1 shows how from this point of view the Transaction MetaData can be regarded as more public than the Account MetaData since it only contains a memo describing the transaction. The figure also shows that sub-dividing the meta-data in this

manner makes it possible to limit the need for GDPR compliance[1] only to the Identity and Profile MetaData. Second, in a physics context the level of dynamicity of the variables could be described as the characteristic frequency (or its inverse, time-scale) at which different kinds of variables change as the transactional workflow is carried out. Figure 2.2 shows how this principle applies to the system's meta-data – and happens to match the same ordering as the privacy view.

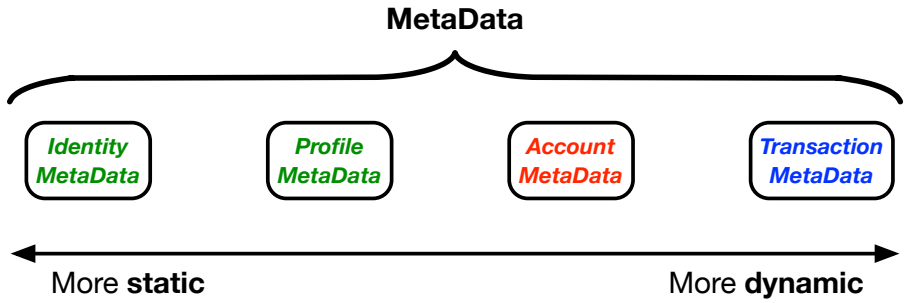

Fig. 2.2: **Relative dynamicity of different types of meta-data of the INTERLACE transaction engine**

Third, Figure 2.3 shows the high-level view of the transaction workflow, including how the PreviewRequest and PerformRequest rules specified in D2.1 map onto it. The process starts on the left of the figure, when a user or $SysAdmin$ initiates the request for a transaction. The request must pass the three tests shown: Transfer Types, Account Connectivity and, in Inter-Circuit operations, the euroFee is calculated and shown to the user. At this point the user is shown a preview screen that summarizes all the transaction data. When the user issues the command to proceed, the first two tests are repeated and a final test on the account limits (e.g. sufficient funds) is performed. If also this fourth test is passed, the transaction is executed.

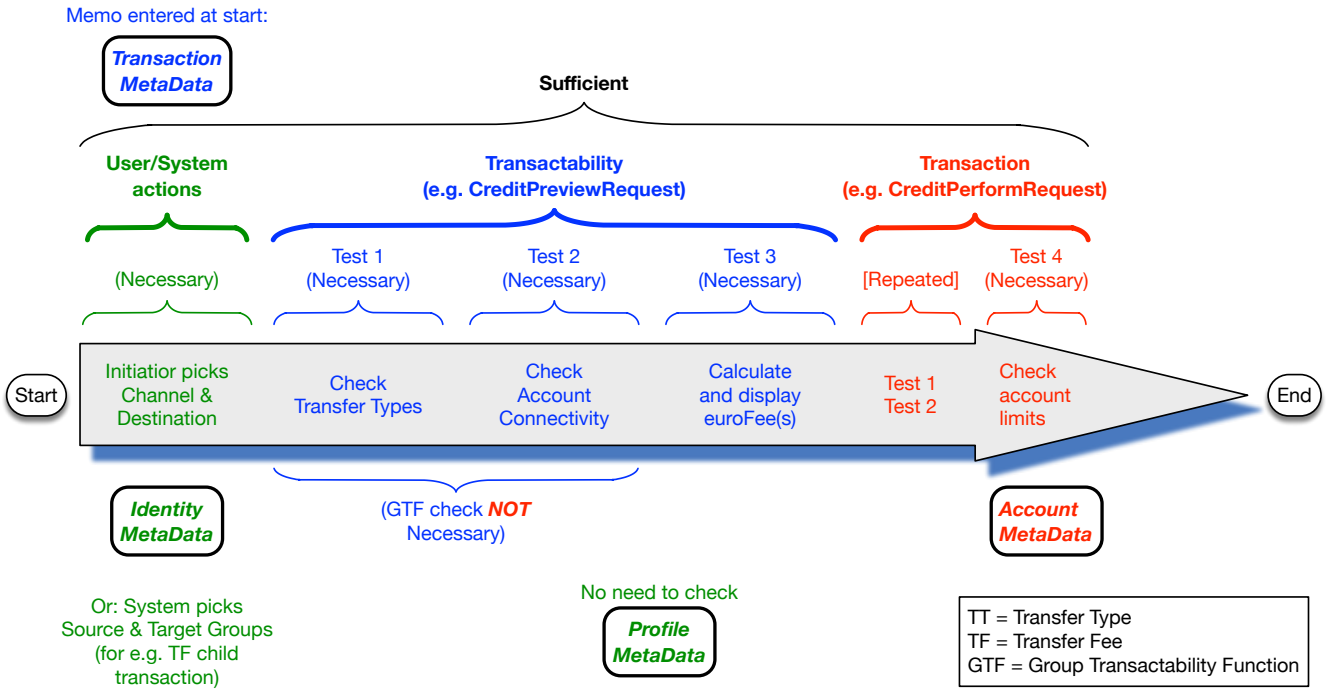

Fig. 2.3: **High-level workflow of the Permissioning process for user- or system-initiated transactions**

In the following, first we provide a taxonomy of all the terms and concepts used and needed by the model, and then proceed to explain each step of the transaction workflow.

[1] General Data Protection Regulation: https://www.eugdpr.org/

2.2 Permissioning Model Taxonomy

2.2.1 Users and Groups

Users are organized by user types called 'Groups'. Consistently with the Appendix, in mathematical notation a $Group$ is a set of $group$s, such that $group \in Group$. However, also an individual $group$ is a set, specifically it is a set of users of the same type. Therefore, the group $Retail$ is capitalized, whereas a single shop is a $retail$. In the original Sardex platform not much distinction is made between users and groups. However, in the current INTERLACE specification we do need to discriminate between them.

> For example, in a Credit transaction, $fromUser$, $fromGroup$, and $fromAccount$ are all associated with the Buyer, whereas the $toUser$, $toGroup$, and $toAccount$ are associated with the Seller, i.e. the party receiving the funds. In a Debit transaction, on the other hand, at the User level the Seller initiates the Debit transaction to draw funds from the Buyer's account. In this scenario, therefore, the Seller is the $fromUser$ and the Buyer is the $toUser$. However, at the Group level things are different and match the behaviour of the Accounts level. The $fromGroup$ and the $fromAccount$ are associated with the party that is paying, i.e. the Buyer; and the $toGroup$ and the $toAccount$ are associated with the Seller. See Section 2.2.6 for a detailed diagram.

The original Sardex system divided its users into 29 different groups. This generated a great deal of complexity that is drastically reduced in the INTERLACE version of the platform. The new user type taxonomy involves only 9 groups, as shown in Table 1.

Group	Description of Element of Group
Welcome	User who has joined and signed the contract, but has not yet been cleared to start trading
Retail	Retailer who can *only* participate in B2C operations (not B2E and not B2B)
Company	Company, which could be a retailer, that can use B2E and B2B but *not* B2C
Full	This group has all the functions of $Retail$ and of $Company$
Employee	Employee of a $company \in Company$ or of a $full \in Full$ (or, also, of $mngr \in Mngr$)
On_Hold	User whose privileges have been suspended (for whatever reason)
Consumer	Person not registered to the circuit who can only interact through B2C Use Case 2
Consumer_Verified	$consumer$ who has registered and can now also purchase with SRD (B2C Use Case 3)
MNGR	Manager of the circuit (e.g. Sardex S.p.A.) acting as a $company$ rather than $SysAdmin$

Table 1: **New user types (groups) for the INTERLACE platform**

$SysAdmin$ is not defined explicitly as a group, in this model, although technically it too is a user type. $SysAdmin$ has special privileges, some of which are discussed where relevant in what follows. We do not use lower-case for $MNGR$ and $SysAdmin$ because there is only one of each.

2.2.2 Currencies and Channels

As shown in Figure 2.4, the Sardex/INTERLACE platform supports transactions in both Sardex credits (SRD) and Euros (EUR) over two kinds of channels. 'Service' refers to transactions mediated either by a computer (web application) or a mobile phone (either a web application or a phone App). 'POS' means 'Point of Sale' and refers to the standard terminal used by retailers that accepts credit or debit cards, through which SRD transactions can be routed via an API. The figure also shows the four possible $\{currency, channel\}$ combinations that we need to support in the definition and implementation of the permissioning tests discussed in the next sections.

Fig. 2.4: **Currencies and channels supported by the platform**

2.2.3 Accounts

As shown in Table 2, the accounts implemented in the model reflect the current operations of the Sardex system and the needs of a wide range of business operations and interactions. Some of the account types, for example MIRROR, may be phased out as the high-level architecture of the family of Sardex circuits grows and different algorithms replace the current strict controls on the balance of payments between different circuits. Also, an account can be referred to as $fromAcct$ or $toAcct$ depending on its role in the transaction. See Section 2.2.6.

Account	Description
CC	Standard Sardex credits (SRD) account
$DOMU$	SRD account used for larger operations, such as for real estate or capital equipment
$MIRROR$	Account controlled by $MNGR$, used for inter-circuit purchases
$Income$	Statistical EUR account owned by $retail$ or $full$ that collects B2C payments
$Prepaid$	Statistical EUR account from which the 2% child B2C transaction fee is drawn
$Bisoo$	Statistical EUR account used by $consumer$ to pay into Income
$Topup$	Statistical EUR account used by $MNGR$ to recharge $retail$'s Prepaid account upon receipt of a EUR payment. It is recharged back to zero gradually with each B2C transaction.

Table 2: **New user types (groups) for the INTERLACE platform**

2.2.4 Account Limits

Table 3 shows the account limit parameters that apply to the SRD accounts CC, DOMU, and MIRROR. The credit limit is straightforward, it is the maximum negative value the account balance can reach. However, note that it is expressed as a *positive* number (actually, a *non-negative* number, since it can be 0) . This is important to keep in mind for the calculations that are based on the value of this parameter. The credit limit is set at the time a user signs the contract with Sardex and is reviewed at least every year after that. The upper limit is, similarly, the maximum value the balance of the account is allowed to reach. Whereas the credit limit can be considered to be a safety measure for the circuit, the upper limit is more a safety measure for the user, since if the balance becomes too large-positive it may be difficult for the user to find ways to spend the credits in a useful time (for the user). Capacity is the maximum total sale volume that the user commits to accepting in one year. The alerts are safety buffers set by the user to alert him/her when the account balance and/or the sale volume approach these limits.

Account Limit Parameter	Description
$creditLimit$	Maximum negative SRD balance allowed (*non-negative* number)
$upperLimit$	Maximum positive SRD balance allowed (positive number)
$capacity$	Maximum **sale** volume allowed in one year
$lowBalanceAlert$	Buffer set by user: alert if $(creditLimit + balance) < lowBalanceAlert$
$highBalanceAlert$	Buffer set by user: alert if $(upperLimit - balance) < highBalanceAlert$
$highVolumeAlert$	Buffer set by user: alert if $(capacity - saleVolume) < highVolumeAlert$

Table 3: **Account limit parameters**

Figure 2.5 shows a visualization of the account limits that apply to SRD accounts such as CC. The thick vertical green arrows highlight that the calculation of the sale volume is defined as *the sum of all the sales performed in one year*. In this example, the sum of all the vertical green arrows is 50,000. Figure 2.6 is a visualization of the $Prepaid$ EUR account, which is recharged or topped up once in a while (€400 in the figure) and slowly drawn down by B2C transaction fees (see B2C Use Case 2 in Figure 2.10 below).

2.2.5 Operations

As already specified in D2.1, transactions are effected through two operations: $credit$ and $debit$:

$$Operation = \{credit, debit\}. \quad (1)$$

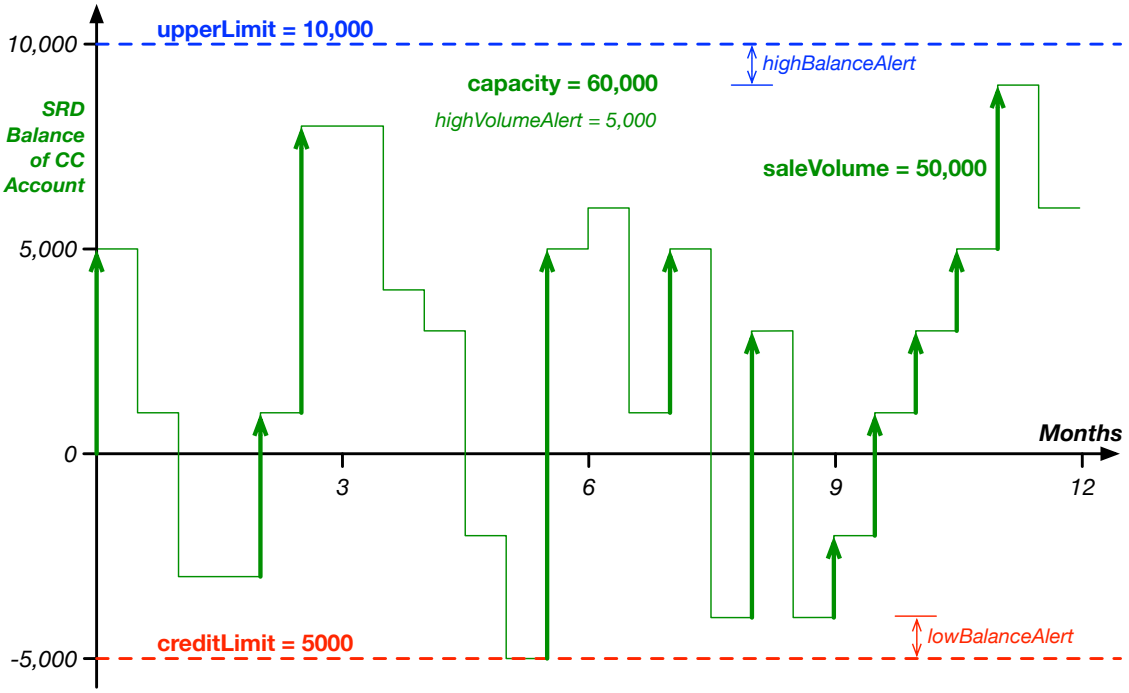

Fig. 2.5: **Visualization of the CC account limits**

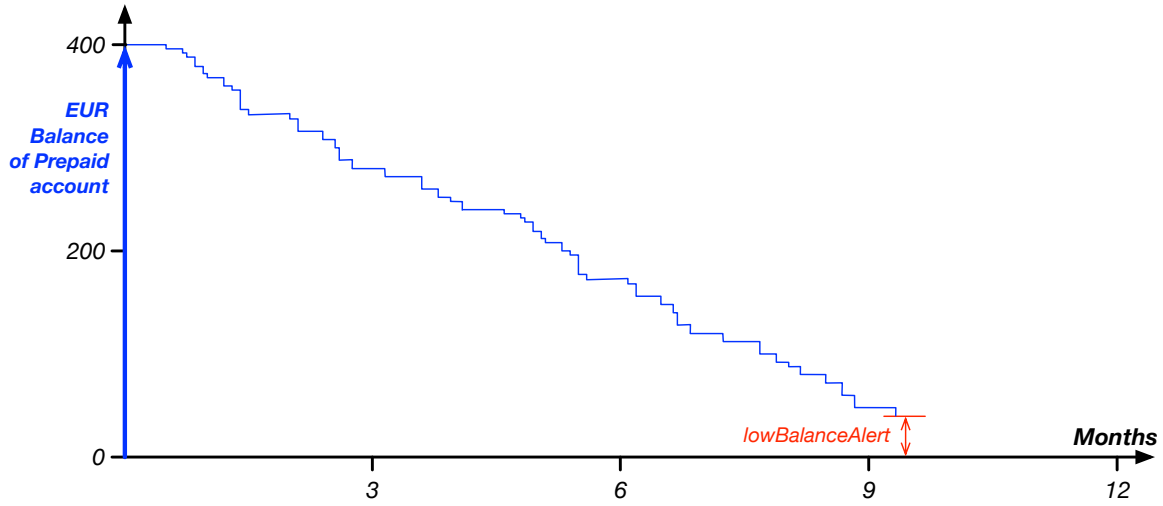

Fig. 2.6: **Visualization of the $Prepaid$ account limit**

2.2.6 Interaction Levels

The interactions between circuit participants can be described from different points of view that correlate loosely to a stack view of the system. As shown in Figure 2.7, it is helpful to identify qualitatively the different levels of such a stack, acknowledging that it is *more* than a networking communication stack in terms of scope but *less* than one in terms of precision. 'A' and 'B' refer to the Buyer and the Seller respectively. Figure 2.7 extends the two communication levels described in D2.1, but does so qualitatively. A formal desription will be provided in later sections and in the Appendix. For the purposes of the implementation, we need a smaller set of levels but a more precise and specific vocabulary to identify the end-points of the transaction.

Action	Level	Operation: Credit From	Operation: Credit To	Operation: Debit From	Operation: Debit To	Payload Example
Seller gives good to Buyer	0-Real Economy	**B** *(Seller)*	**A** *(Buyer)*	**B** *(Seller)*	**A** *(Buyer)*	*Goods & Services*
Buyer initiates credit transaction towards Seller or vice versa	1-User	**fromUser** *(Buyer)*	**toUser** *(Seller)*	**fromUser** *(Seller)*	**toUser** *(Buyer)*	*Transaction Request*
TT function is evaluated	2-Group	**fromGroup** *(Buyer)*	**toGroup** *(Seller)*	**fromGroup** *(Buyer)*	**toGroup** *(Seller)*	*Value of TT function*
Funds are moved from Buyer's account to Seller's account	3-Account	**fromAcct** *(Buyer)*	**toAcct** *(Seller)*	**fromAcct** *(Buyer)*	**toAcct** *(Seller)*	*SRD or EUR*
Client ASIM sends message to Server ASIM	4-ASIM	**fromASIM** *(Buyer)*	**toASIM** *(SysAdmin)*	**fromASIM** *(Seller)*	**toASIM** *(SysAdmin)*	*Message*

Start here!

The mapping this Transfer Type (TT) function models replicates the flow of currency in the "3-Account" layer just below.

Legend
- **Buyer**
- **Seller**
- **SysAdmin**

Fig. 2.7: **Stack view of the INTERLACE communication, economic, and financial interactions**
(Stack view of from where to where the different payloads are moved in interactions)

Figure 2.7 has taken a long time to arrive at, and was changed several times after long and difficult discussions. It was created, and should be interpreted, as some kind of Rosetta stone that brings together different views of the transactions: economic, user-centred, mathematical, database-centred, and system communications. In addition, it is also meant to maintain "conceptual" backward compatibility with the previous system, in which the Transfer Types are attributes of the transaction recipients (i.e. the Sellers), rather than mathematical functions whose *values* need to include the Sellers and whose input parameters are the Buyers. This second (mathematical) view is adopted here; it allows to express the condition about whether or not a Buyer and a Seller may transact based on how the function is defined. Section 2.3.3 describes this function in detail.

Order of Computation

Parameter/Actor	To/From Values	Values	Conditions	Phase
1-User	{fromUser, toUser}			Preview
Operation		{credit, debit}	chosen by User	
{Currency, Channel}		{SRD, EUR} x {service, POS}	chosen by User	
Identity MetaData				
Transaction MetaData				
2-Group	{fromGroup, toGroup}	{Welcome, Retail, Company, Full, Employee, On_Hold, Consumer, Consumer_Verified, MNGR}	Transfer Types	
3-AccountType	{fromAcct, toAcct}	{CC, DOMU, MIRROR, Income, Prepaid, Bisoo, Topup}	∃ connection	
Amount		$\mathbb{Z}$	Account Limits	Perform
Account MetaData				
4-ASIMs	{fromASIM, toASIM}	ASIM Identifier	∃	

Fig. 2.8: **Actors, parameters, levels, data structures, computational process, and conditions**

Figure 2.8 shows this additional information as concerns levels 1-4, which are labelled in the first column in the same way as in Figure 2.7, along with some more information. In particular, this figure could be seen to integrate aspects of Figures 2.7 and 2.3, with the purpose of facilitating the conceptual understanding of the specification. The different types of 'MetaData' relevant to transactions are shown in the approximate position where they are polled. For example, the % of SRD accepted by a given user over 1000-EUR transaction values is part of the Profile MetaData of the *Company* group but not the *Consumer_Verified* group's.

2.2.7 Visibility

Figure 2.9 shows which groups (*toGroup*) are visible to which groups (*fromGroup*) by a '1' at the intersection of the $(row, column)$ corresponding to a choice of $(fromGroup, toGroup)$. For example, *Company* is visible to *Employee*, meaning that *Employee* can do a search for *Company*, but not vice versa. In this case *Employee* is not visible due to privacy legislation.

fromGroup: \ toGroup:	Welcome	Retail	Company	Full	Employee	On_Hold	Consumer	Consumer_Verified	MNGR	Visibility (Look at the columns)
Welcome	0	1	1	1	0	-	-	0	1	{F}
Retail	0	1	1	1	0	-	-	0	1	{T, F}
Company	0	0	1	1	0	-	-	0	1	{T, F}
Full	0	0	1	1	0	-	-	0	1	{T, F}
Employee	0	1	1	1	0	-	-	0	0	{F}
On_Hold	-	-	-	-	-	-	-	-	-	{F}
Consumer	-	-	-	-	-	-	-	-	-	{F}
Consumer_Verified	0	1	0	1	0	-	-	0	0	{F}
MNGR	0	1	1	1	0	-	-	0	-	{T, F}

Fig. 2.9: **Mutual visibility of different groups**

In general, transactability correlates to visibility. However, there is not a strict 1-1 relationship between them. For the example of a *company* paying its own *employee*s as part of the B2E programme, *employee* remains invisible to a search, but *company* has the *username* of the *employee* and can perform a credit transaction to pay (part of) their salary.

On the right of Figure 2.9 we show that some groups are always invisible $\{F\}$ and others could be either visible or invisible $\{T, F\}$. For example, a company could be put in the shadow state if it has reached its maximum positive credit limit.

2.2.8 B2C Operations

This report extends the use cases covered by D2.1 by adding also the Business to Consumer (B2C) use cases. B2C was developed to increase the number and volume of transactions, i.e. the size of the Sardex economy, by extending the ability to transact in SRD to people not otherwise connected to the circuit. The principle involves offering the opportunity to *retail*ers to reward their EUR customers with an SRD rebate.

The amount of the reward is a percent, in SRD, of the amount in EUR paid by a *consumer* or a *consumer_verified* to a *retail*, where the percent is set by *retail* and it is an example of the *MetaData* for this group. The reward is stored on a smart card that is offered to *consumer* at the time of purchase. A *consumer_verified* already has a card, so their card is simply topped up. This is shown as a child transaction in B2C Use Case 2, Figure 2.10.

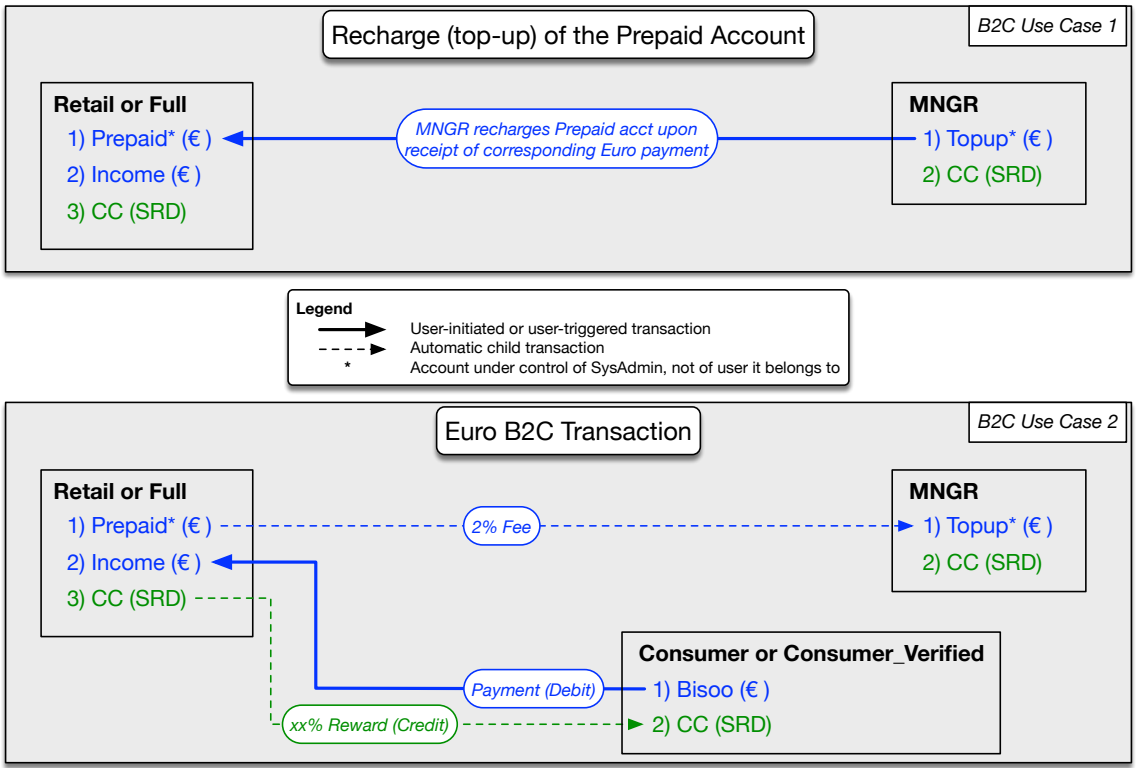

Fig. 2.10: **Recharging of *retail*'s Prepaid account and standard B2C EUR transaction**

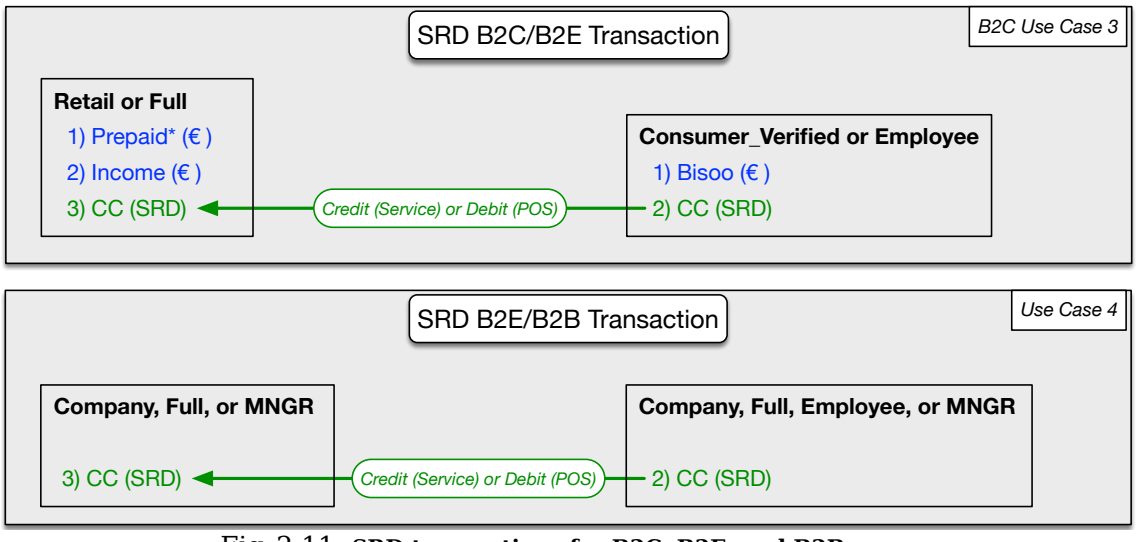

Fig. 2.11: **SRD transactions for B2C, B2E, and B2B users**

Use Case 2 also involves a second child transaction, a 2% fee, in EUR, paid from *retail*'s Prepaid account to *MNGR*'s Topup account. The asterisks next to these account names, in the figure, indicate that these two accounts are *owned* by these two users but are not *controlled* by them. They are controlled by *SysAdmin*. These EUR accounts are 'statistical' rather than 'real', meaning that they only keep track of actual EUR amounts but do not themselves hold Euros. Sardex S.p.A.

would need to be a bank for that to be possible. For Use Case 2 to be executable, for a given *retail*er, its Prepaid account needs to have sufficient funds. When it runs out of statistical Euros, *retail* can pay *MNGR* some amount of Euros using any of the standard payment systems, through a bank or a payment service provider like PayPal. This triggers Use Case 1, also shown in Figure 2.10.

Figure 2.11 shows the SRD transaction a *consumer* can perform, i.e. the spending of the SRD accumulated as rewards, once they have registered and have become *consumer_verified*. The figure also shows that Use Case 3 is relevant also to *employee*s (B2E) and that Use Case 4 is relevant to both B2E users and to non-*retail* company users (B2B).

2.2.9 Initial Account State

Table 4 shows the initial allocation of accounts to the different user groups. The allocation is 'initial' because depending on the history of a given user the number of its accounts could change. For legal reasons, as a general principle 'change' can only mean 'increase'. In other words, once a user has become the owner of an account it can never be taken away from them, even if, for example, the user's access to it is suspended due to misbehaviour. Another example is a *Retail* user who upgrades to the *Full* group and a year later changes its mind and goes back to *Retail*. It will retain its DOMU account even if it won't be able to use it anymore.

Note: Each user can have no more than one account of a given type.

Group	Initial Account Set
Welcome	$\{\}$
Retail	$\{CC, Prepaid^*, Income\}$
Company	$\{CC, DOMU, Prepaid^*\}$
Full	$\{CC, DOMU, Prepaid^*, Income\}$
Employee	$\{CC\}$
On_Hold	x, where $x \in Powerset(CC, DOMU, Prepaid^*, Income)$
Consumer	$\{CC, Bisoo^*\}$
Consumer_Verified	$\{CC, Bisoo^*\}$
MNGR	$\{CC, Topup^*, MIRROR\}$

Table 4: **Initial sets of accounts assigned to the groups**
(*Indicates an account under the control of *SysAdmin* (in the case of Prepaid) or of a *Retail* or *Full* member (in the case of Bisoo), not of the user it belongs to.)

Note: The Prepaid account shown for *Company* does not relate to B2C operations but to inter-circuit trade, which also involves a euroFee. A user cannot start as *On_Hold*, it will start as one of the other groups. For legal reasons an account cannot be taken away once it has been assigned to a user. Therefore, since any user can be suspended and become *On_Hold*, this group could have different combinations of accounts (with the exception of the MNGR account).

2.3 Transactability Workflow

2.3.1 Identity MetaData

Table 5 collects the identity meta-data for all the groups. *MemberID* is a unique identifier. *email* and *phone* are arrays to support multiple values of each. These values are set at registration and cannot be changed by the user.

The meta-data variables are not capitalized because they are assumed to be singletons: for a given member, e.g. a *company*, there is only one *memberID*. However, since there are many *memberID*s, one for each member, we could also say that $memberID \in MemberID$. Since there

are cases where a user may have more than one meta-data variable, for example a company with more than one phone number, this is indicated in the Type column as an array, i.e. 'String[ ]'.

Group	Identity MetaData	Type	Description
Welcome, Retail, Company, Full, Employee, On_Hold, Consumer_Verified, MNGR	**memberID**	Integer	Unique member identifier
	email	String[]	e-mail address
	phone	String[]	phone number(s)
Consumer	**memberID**	Integer	Unique member identifier

Table 5: **Identity MetaData for all the groups**

2.3.2 Profile MetaData

Tables 6 and 7 show the profile meta-data, some of which the user can inspect and edit. For example, the user may wish to include his/her personal name in addition to the company name.

Group	Profile MetaData	Type	Description
Welcome	*(Obligatory MetaData)*		
	entityName*	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	VAT*	String	Legal entity's VAT number
	capacity	Double	Commitment to maximum yearly SRD volume
	capacityDate	DateTime	Date capacity was set
	(Optional MetaData)		
	firstName*	String	First name
	surName*	String	Surname
Retail	*(Obligatory MetaData)*		
	entityName*	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	capacity	Double	Commitment to maximum yearly SRD volume
	capacityDate	DateTime	Date capacity was set
	rewardRate**	Double	% reward rate to consumer, in SRD
	euroFee	Double[]	% fee on B2C EUR sales, in EUR
	acceptanceRate**	Double	Rate of SRD acceptance in consumer purchases
	(Optional MetaData)		
	firstName *	String	First name
	surName*	String	Surname
Company	*(Obligatory MetaData)*		
	entityName*	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	VAT*	String	Legal entity's VAT number
	capacity	Double	Commitment to maximum yearly SRD volume
	capacityDate	DateTime	Date capacity was set
	creditPercent	Double	% SRD acceptance for transactions above 1000
	euroFee	Double[]	% fee on inter-circuit SRD sales, in EUR
	(Optional MetaData)		
	firstName *	String	First name
	surName*	String	Surname

Table 6: **Profile MetaData for the** *Welcome*, *Retail*, **and** *Company* **groups**
(*Indicates fields that can be modified by the user)
(**Indicates fields that can be modified by the user but that are updated only at a fixed time interval)

Group	Profile MetaData	Type	Description
	(Obligatory MetaData)		
Full	**entityName***	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	VAT*	String	Legal entity's VAT number
	capacity	Double	Commitment to maximum yearly SRD volume
	capacityDate	DateTime	Date capacity was set
	creditPercent	Double	% SRD acceptance for transactions above 1000
	rewardRate**	Double	% reward rate to consumer, in SRD
	euroFee	Double[]	% fees on B2C EUR sales, inter-circuit SRD sales, in EUR
	acceptanceRate**	Double	Rate of credits acceptance in consumer purchases
	(Optional MetaData)		
	firstName*	String	First name
	surName*	String	Surname
	(Obligatory MetaData)		
Employee	**firstName***	String	First name
	surName*	String	Surname
	employedBy	String	Name of legal entity employed by
	(Obligatory MetaData)		
On_Hold	**entityName***	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	VAT*	String	Legal entity's VAT number
	capacity	Double	Commitment to maximum yearly SRD volume
	capacityDate	DateTime	Date capacity was set
	creditPercent	Double	% SRD acceptance for transactions above 1000
	rewardRate**	Double	% reward rate to consumer, in SRD
	euroFee	Double[]	% fees on B2C EUR sales, inter-circuit SRD sales, in EUR
	acceptanceRate**	Double	Rate of credits acceptance in consumer purchases
	(Optional MetaData)		
	firstName*	String	First name
	surName*	String	Surname
Consumer			
	(Obligatory MetaData)		
Consumer_Verified	**firstName***	String	First name
	surName*	String	Surname
	(Obligatory MetaData)		
MNGR	**entityName***	String	Legal entity name
	entityAddress*	String	Legal entity's street address
	gps*	Double[]	Legal entity's GPS coordinates
	VAT*	String	Legal entity's VAT number
	creditPercent	Double	% SRD acceptance for transactions above 1000

Table 7: **Profile MetaData for the** *Full*, *Employee*, *On_Hold*, *Consumer*, *Consumer_Verified*, **and** *MNGR* **groups**
(*Indicates fields that can be modified by the user)
(**Indicates fields that can be modified by the user but that are updated only at a fixed time interval)

2.3.3 Transfer Types

As shown in Figure 2.3, the first test for transactability involves so-called Transfer Types. Transfer Types are mathematical functions of the **source** groups (*fromGroup*s) whose values are sets of **destination** groups (*toGroup*s). Referring to Figure 2.7, for the Group layer 'source' means the group where the money is coming from and 'destination' is the group where the money is going, regardless of whether the operation is a Credit or a Debit.

There is one different function for each combination of ordered pairs (x, y), where $x \in \{credit, debit\}$ and $y \in \{SRD, EUR\}$, leading to four different functions. However, it is simpler and also easier to implement to express them as a single function of 3 parameters. Formally,

$$TT\colon Operation \times Currency \times Group \rightarrow Powerset(Group), \tag{2}$$

where

$$Group = \{Welcome, Retail, Company, Full, Employee, On\_Hold, Consumer, Consumer\_Verified, MNGR\}. \quad (3)$$

With an abuse of notation and overloading the terminology we define for convenience the following sets of target groups ($toGroups$) as different "transfer types":

$$TT_1 = \{Retail\}, \quad TT_2 = \{Retail, Company\}, \quad TT_3 = \{Company, Employee\}$$
$$TT_4 = \{Company\}, \quad TT_5 = \{MNGR\}, \quad TT_6 = \{Full\}.$$

Through currying, Table 8 then shows the TT function as four sub-functions.

The Transfer Type test shown in Figure 2.3 involves checking whether the recipient (Seller) of a Credit or Debit transaction in a given currency is in the range of the corresponding TT function of the Buyer. In other words, the Buyer, or $fromGroup$ member, is the independent or input parameter to the function and appears in the left column in Table 8.

fromGroup	$\boldsymbol{TT^{Credit,SRD}}$	$\boldsymbol{TT^{Debit,SRD}}$	$\boldsymbol{TT^{Credit,EUR}}$	$\boldsymbol{TT^{Debit,EUR}}$
Welcome	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Retail	$\emptyset$	TT_5	$\emptyset$	$\emptyset$
Company	$TT_3 \cup TT_5 \cup TT_6$	$TT_4 \cup TT_5 \cup TT_6$	$\emptyset$	$\emptyset$
Full	$TT_3 \cup TT_5 \cup TT_6$	$TT_4 \cup TT_5 \cup TT_6$	$\emptyset$	$\emptyset$
Employee	$TT_2 \cup TT_6$	$TT_2 \cup TT_6$	$\emptyset$	$\emptyset$
On_Hold	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Consumer	$\emptyset$	$\emptyset$	$\emptyset$	$TT_1 \cup TT_6$
Consumer_Verified	$TT_1 \cup TT_6$	$TT_1 \cup TT_6$	$\emptyset$	$TT_1 \cup TT_6$
MNGR	$TT_3 \cup TT_5 \cup TT_6$	$TT_4 \cup TT_6$	$\emptyset$	$\emptyset$

Table 8: **The Transfer Types function expressed as 4 separate sub-functions through currying**

Note on Inter-Circuit Trade. Although inter-circuit operations are not formally modelled or specified in this report (or in the Appendix), Table 8 is consistent with them. Briefly, if User 1 (U1) in Circuit 1 (C1) and User 2 (U2) in Circuit 2 (C2) wish to trade, in the current model they need to obtain permission to do so from a broker. Assuming it is granted, U1 is the Buyer, and U2 is the Seller, the credits flow is

$$CC_{U1} \rightarrow MIRROR_{MNGR1} \quad \text{followed by} \quad MIRROR_{MNGR2} \rightarrow CC_{U2}. \quad (4)$$

Note that in this scenario $MIRROR_{MNGR2}$ could go negative. Since no credits are exchanged between the two MIRROR accounts, the difference in their balances reflects the balance of (CC) trade between the two circuits. With this context, TT_5 in the $(Credit, SRD)$ column, for the $MNGR$ row, is needed to support a special kind of inter-circuit operations, in which the $MNGR$ of a given circuit buys or sells products or services from the user in a *different* circuit. Assuming MNGR2 of C2 is buying a service from U1 in C1, for example, the credits flow is

$$CC_{MNGR2} \rightarrow MIRROR_{MNGR2} \quad \text{followed by} \quad MIRROR_{MNGR1} \rightarrow CC_{U1}. \quad (5)$$

In all cases $MNGR$-$MNGR$ transactions are implemented as Credit transactions, never as Debit.

2.3.4 Account Connectivity

Figure 2.12 shows the account connectivity function used for Test 2 in Figure 2.3. As for Visibility, a '1' indicates that the Source and Destination accounts are connected and funds can be transferred from one to the other.

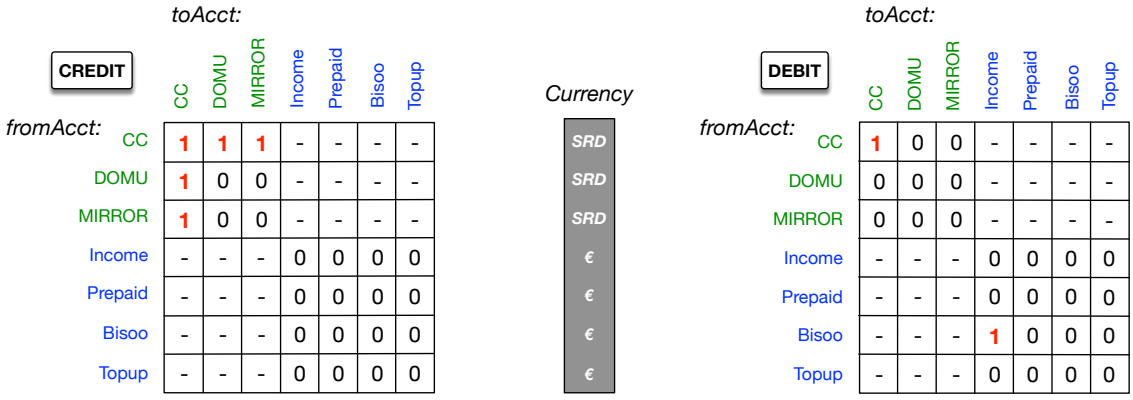

Fig. 2.12: **Account connectivity for user-initiated transactions: Test 2 in Figure 2.3**

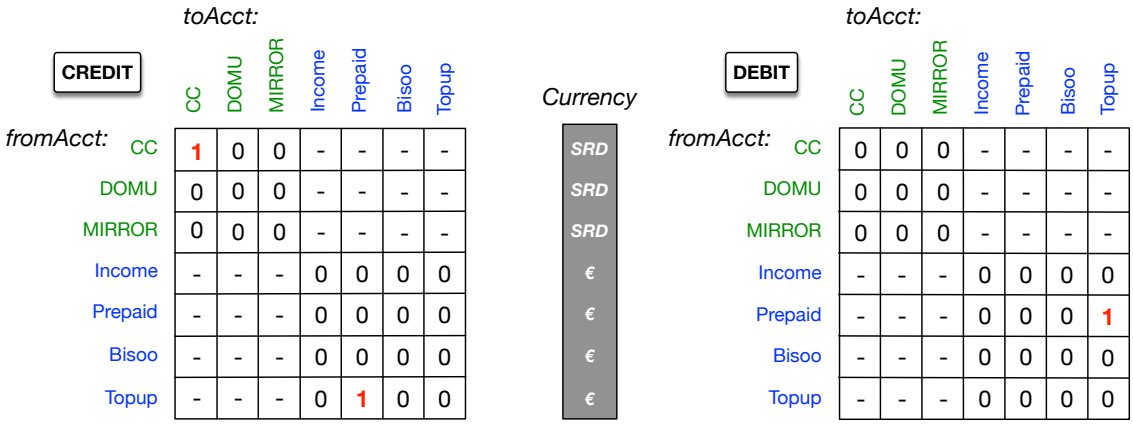

Fig. 2.13: **Account connectivity for system-initiated and child transactions: Not a test, hard-wired.**

Figure 2.13 shows a similar connectivity function for accounts that are controlled by $SysAdmin$. In this case, however, this function does not imply a transactability test since the permissions are hard-wired and $SysAdmin$ does not need to test for transactability.

2.3.5 User Transactability Functions

Note: this section has no bearing on the implementation, it is included to help interpret Figure 2.7.
Figures 2.14-2.17 show a visualization of the interactions relevant to the "1-User" layer of Figure 2.7, filtered by the Account Connectivity constraints.

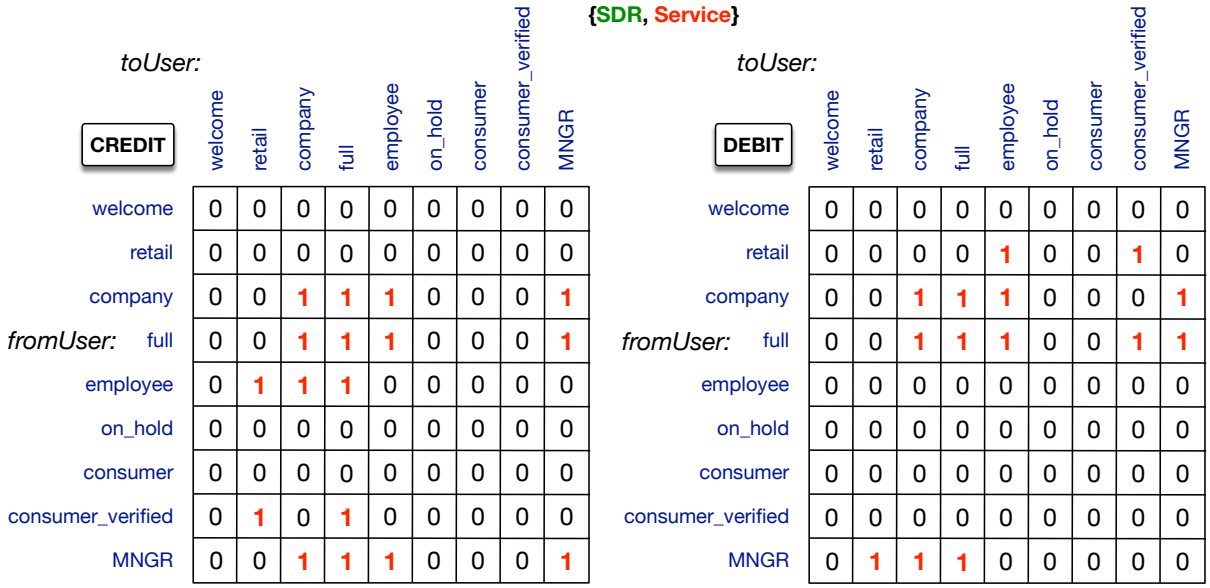

Fig. 2.14: **User Transactability Function 1**

{SDR, POS}

CREDIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

DEBIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	1	0	0	1	0
company	0	0	1	1	1	0	0	0	1
full	0	0	1	1	1	0	0	1	1
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	1	1	1	0	0	0	0	0

Fig. 2.15: **User Transactability Function 2**

{EUR, Service}

CREDIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	1*	1*	1*	0	0	0	0	0

DEBIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	1	1	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	1	1	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

* This transaction is initiated by *SysAdmin*, not within the scope of *MNGR*

Fig. 2.16: **User Transactability Function 3**

{EUR, POS}

CREDIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

DEBIT

fromUser: \ toUser:	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	1	1	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	1	1	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

Fig. 2.17: **User Transactability Function 4**

2.3.6 Group Transactability Functions

Note: this section has no bearing on the implementation, it is included to help interpret Figure 2.7.

Figures 2.18-2.21 show a visualization of the interactions relevant to "2-Group" layer of Figure 2.7, filtered by the Account Connectivity constraints. In other words, the effect of the Transfer Types test together with the Account Connectivity test results in what we call the Group Transactability Function (GTF). Since there are 4 combinations of currency and channel, there are 4 different GTFs. There is no need to use them as an additional test, since they do not add anything new to Tests 1 and 2 of Figure 2.3. However, it is still useful to include them here as another visualization of these first two tests of the Permissioning workflow. For the mathematically-minded reader, it may be interesting to note that the Debit adjacency matrices for the GTFs are the Transpose of the Debit adjacency matrices for the UTFs. Another potentially useful visualization of the UTF and GTF matrices would be as 16 directed graphs over the 9 nodes representing the groups.

{SDR, Service}

CREDIT (toGroup: / fromGroup:)	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	1	1	1	0	0	0	1
full	0	0	1	1	1	0	0	0	1
employee	0	1	1	1	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	1	0	1	0	0	0	0	0
MNGR	0	0	1	1	1	0	0	0	1

DEBIT (toGroup: / fromGroup:)	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	1
company	0	0	1	1	0	0	0	0	1
full	0	0	1	1	0	0	0	0	1
employee	0	1	1	1	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	1	0	1	0	0	0	0	0
MNGR	0	0	1	1	0	0	0	0	0

Fig. 2.18: **Group Transactability Function 1**

{SDR, POS}

CREDIT (toGroup: / fromGroup:)	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

DEBIT (toGroup: / fromGroup:)	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	1
company	0	0	1	1	0	0	0	0	1
full	0	0	1	1	0	0	0	0	1
employee	0	1	1	1	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	1	0	1	0	0	0	0	0
MNGR	0	0	1	1	0	0	0	0	0

Fig. 2.19: **Group Transactability Function 2**

{EUR, Service}

CREDIT

fromGroup: \ *toGroup:*	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	1*	1*	1*	0	0	0	0	0

DEBIT

fromGroup: \ *toGroup:*	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	1	0	1	0	0	0	0	0
consumer_verified	0	1	0	1	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

* This transaction is initiated by *SysAdmin*, not within the scope of *MNGR*

Fig. 2.20: **Group Transactability Function 3**

{EUR, POS}

CREDIT

fromGroup: \ *toGroup:*	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	0	0	0	0	0	0	0	0
consumer_verified	0	0	0	0	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

DEBIT

fromGroup: \ *toGroup:*	welcome	retail	company	full	employee	on_hold	consumer	consumer_verified	MNGR
welcome	0	0	0	0	0	0	0	0	0
retail	0	0	0	0	0	0	0	0	0
company	0	0	0	0	0	0	0	0	0
full	0	0	0	0	0	0	0	0	0
employee	0	0	0	0	0	0	0	0	0
on_hold	0	0	0	0	0	0	0	0	0
consumer	0	1	0	1	0	0	0	0	0
consumer_verified	0	1	0	1	0	0	0	0	0
MNGR	0	0	0	0	0	0	0	0	0

Fig. 2.21: **Group Transactability Function 4**

2.3.7 Transaction MetaData

The transaction meta-data consists of only 1 memo message that is to be implemented as part of the transaction. It is not obligatory, i.e. the initiator of the transaction can leave it blank. The length is to be determined at the time of implementation but the assumption is that it is “long enough” to support a reasonable description of the transaction.

2.3.8 Account MetaData

Tables 9 and 10 are also organized in terms of obligatory and optional meta-data, where the latter are modifiable by the user. At the implementation stage, for example, the modifiable fields can be included together with the profile meta-data. The fields that can be set by the user are indicated with an asterisk.

Account	Account MetaData	Type	Description
	(Obligatory MetaData)		
CC	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
	balance	Double	Account balance (positive or negative)
	creditLimit	Double	Credit limit (positive number)
	creditLimitDate	DateTime	Date at which credit limit was set
	availableBalance	Double	$balance + creditLimit$ (non-negative number)
	upperLimit	Double	Upper balance limit (positive number)
	availableCapacity	Double	$capacity - saleVolume$ (non-negative number)
	(Optional MetaData)		
	lowBalanceAlert*	String	Alert if $(creditLimit + balance) < lowBalanceAlert$ buffer
	highBalanceAlert*	String	Alert if $(upperLimit - balance) < highBalanceAlert$ buffer
	highVolumeAlert*	String	Alert if $(capacity - saleVolume) < highVolumeAlert$ buffer
	(Obligatory MetaData)		
DOMU	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
	balance	Double	Account balance (positive or negative)
	creditLimit	Double	Credit limit (positive number)
	creditLimitDate	DateTime	Date at which credit limit was set
	availableBalance	Double	$balance + creditLimit$ (non-negative number)
	(Obligatory MetaData)		
MIRROR	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
	balance	Double	Account balance (positive or negative)
	creditLimit	Double	Credit limit (positive number)
	creditLimitDate	DateTime	Date at which credit limit was set
	availableBalance	Double	$balance + creditLimit$ (non-negative number)
	upperLimit	Double	Upper balance limit (positive number)
	availableCapacity	Double	$capacity - saleVolume$ (non-negative number)
	(Optional MetaData)		
	lowBalanceAlert*	Double	Alert if $(creditLimit + balance) < lowBalanceAlert$ buffer
	highBalanceAlert*	Double	Alert if $(upperLimit - balance) < highBalanceAlert$ buffer
	highVolumeAlert*	String	Alert if $(capacity - saleVolume) < highVolumeAlert$ buffer
Income	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
	balance	Double	Account balance (positive or negative)

Table 9: **MetaData for** *CC*, *DOMU*, *MIRROR*, **and** *Income* **accounts**
(*Indicates fields that can be modified by the user)

Account	Account MetaData	Type	Description
	(Obligatory MetaData)		
Prepaid	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
	balance	Double	Account balance (positive or zero)
	creditLimit	Double	Credit limit **(Value can only be 0!)**
	(Optional MetaData)		
	lowBalanceAlert*	Double	Alert if $(creditLimit + balance) < lowBalanceAlert$ buffer
Bisoo	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency
Topup	**accountID**	Integer	Unique account identifier
	memberID	String	ProfileID of account owner
	unit	String	Account currency

Table 10: **MetaData for** *Prepaid*, *Bisoo*, **and** *Topup* **accounts**
(*Indicates fields that can be modified by the user)

2.4 Account Limit Tests

The account limit tests are specified abstractly in D2.1. In the context the description of this chapter, which is closer to the implementation, they are fairly obvious by looking at Figures 2.5 and 2.6. The important thing is that they need to be enforced *before* a transaction is completed. If one of the tests fails the transaction is not executed and the user must be alerted.

The alerts, on the other hand, can be issued *after* a transaction that came close to a limit but did not cross it, if the resulting balance or sale volume exceeds the buffer limits.

Chapter 3

Introduction to the CoreASIM Language, Interpreter, and ICEF

Eduard Hirsch

This chapter provides an overview of the technologies used for running the INTERLACE Specification. At the beginning a quick-start description is provided in order make it easier to jump right into the execution of the model. Additional details about how the executable models of Abstract State Interaction Machine (ASIM) specifications function in the environment are discussed, along with how this can happen in a simple and stable manner.

There are two base environments available – one based on Docker (henceforth 'docker') and one based on Vagrant (henceforth 'vagrant'). During project execution, the focus shifted from the vagrant environment, which can be downloaded from GitHub,[2] to a docker-based version which is explained in Section 3.1.2 and is also available on GitHub.[3] Nevertheless, the vagrant setup is still explained for those developers who prefer it.

3.1 Virtualization Environments

3.1.1 Quick-Start Vagrant

The vagrant definition provides a running environment for executing the INTERLACE ASIM definitions. During the provisioning process an Ubuntu vagrant box is set up. All the necessary components are installed in that box, which clones and builds the Interaction Computing Execution Framework (ICEF) framework[4] as well as the ASIM Specification[5] into the data directory where it is finally ready for use.

Prerequisites. Download and install the following software products:

- Virtual Box: https://www.virtualbox.org/
- git: https://git-scm.com/downloads
- Vagrant: https://www.vagrantup.com/

Clone Environment. To clone the ASIM vagrant environment from github into a directory git can be utilized:

```
git clone https://github.com/InterlaceProject/ASIMVagrantEnvironment.git
```

Execution. Once all software components are installed and the vagrant definitions are cloned it is possible to call

```
./execute.sh
```

[2] https://github.com/InterlaceProject/ASIMVagrantEnvironment
[3] https://github.com/InterlaceProject/ASIMDockerEnvironment
[4] https://github.com/biomics/icef
[5] https://github.com/InterlaceProject/ASIMSpec

from the main directory in order to run the INTERLACE specifications. Note: when using Windows it is necessary to start that command within git-bash which needs to run in elevated admin mode (right click → start as administrator). That is necessary to handle symbolic links in git-bash.

On the very first execution the script is provisioning a virtual machine based on Ubuntu by calling *vagrant up*, which may take some time. Subsequent calls will be much faster. A detailed description explaining the precise process is covered in Section 3.4.

Once the execution is started it will run until it is stopped by pressing **ctrl + c** or by calling

```
./stop.sh
```

from any other console window.

3.1.2 Quick-Start Docker

The docker project on GitHub[6] is also based on virtualization like the vagrant environment, but emphasizes *Operating System virtualization* instead of *Hardware virtualization* [6].

Prerequisites.

- install docker
- install git (including git bash for windows)

On Linux machines it is important to add the current user to the docker group in order to manage docker container and images. Otherwise all further explained commands need to be executed as root or with sudo.

For Windows machines use *git-bash* to execute the commands described in the following sections.

Before First Execution. In order to configure the environment it is necessary to call the following script:

```
./configure
```

This will generate a docker container image called *asim* where all the necessary frameworks are built and prepared for execution of the specifications. The ICEF framework as well as the ASIM model specifications are cloned outside of the container to simplify development.

Execute Specification. The container image *asim* created during the configuring step can be started by calling

```
./execute
```

A container started in this way is called *active_asim* and runs all the necessary steps, like starting an ICEF manager as well an ICEF brapper to run the ASIM specifications.

Like the Vagrant environment, a running execution may be stopped by pressing **ctrl + c**.

[6] https://github.com/InterlaceProject/ASIMDockerEnvironment

3.1.3 Container and Virtual Machine-Based Environments

Figure 3.1 shows two virtualization techniques. They have similar approaches to how to isolate and allocate resources. However, they function very differently because a container reaches virtualizsation over splitting an operating system resources and a virtual machine virtualizes hardware on which a whole operating system runs.

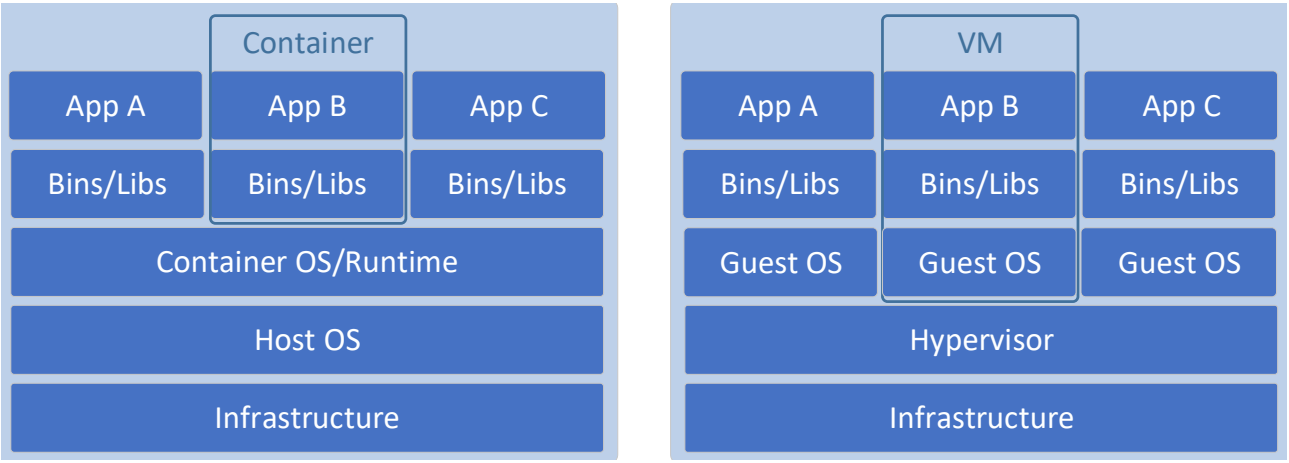

Fig. 3.1: **Container versus Virtual Machine**

As a consequence, containers can be much smaller, more portable, and in most of the cases also much faster because they are using the system resources more directly and not as wastefully as a virtual machine does.

To be more specific, **containers** package their libraries and binaries together with an application. Therefore, each container can run in parallel with other containers on the same machine, sharing the OS kernel and each running in an isolated process in user space as if it were a separate OS. Therefore containers have the ability to start instantly, because the operating system they are using is already up and running. Also, they are much smaller as they don't need to carry all the OS-specific software parts.

In contrast, **virtualized machines** run on an abstraction of the physical hardware which is present in the system. That means that a hypervisor doubles, triples, ... the hardware virtually and allows to install full versions of an operating system compatible with the current system. As each virtualization is a full copy of an OS, images are much larger and boot times most of the time are much slower. Nevertheless, a full operating system might have benefits in some cases over a containerized system.

Coming back to the INTERLACE environments those two approaches reflect the two approaches used for creating the environments. Vagrant is a controlling command line client for hypervisors (virtual machines) and docker is a command line client for a container-based virtualization system with the same name.

INTERLACE development efforts are drawn towards the docker environment because execution and starting of the ASIMSpecs is faster and the project is easier to maintain.

3.2 Execution Environment Stack

Regardless of the virtualization techniques used, a consistent base system is used. Therefore, both docker and vagrant are provisioning a Linux-based operating system. In this case the Ubuntu 16.04 LTS (Long Term Support) distribution is used. This consistent, stable, and reliable structure will be important later when considerations about provability as well as testability become relevant; namely, that design always provides the same preconditions and anybody executing or testing against the specification will obtain the same results.

3.2.1 Software Stack

The Ubuntu 16.04 LTS distribution needs to be enhanced and updated consistently with the needs of an ASIM executing machine as well as with the needs of developers working with this virtual system. More specifically, the following components are installed during the provisioning process:

- **curl** → Is a tool for querying REST resources and is used for downloading packages from various online repositories.
- **nodejs** → Is a JavaScript runtime environment built on Chrome's V8 engine and includes the package manager npm. This bundle is necessary for installing and running the Manager component of ICEF.
- **build-essential** → These packages are needed to compile a debian-based package and provide help for building (compiling, packaging, etc) the project sources.
- **maven** → Is a well-known Java build and packaging tool and is used for building the ICEF framework as well as the CoreASIM Eclipse plugin.
- **vim** → Well-known U/Linux editor which acts as helper for quick development or configuration issues inside the environment.
- **git** → Distributed Version-Control System which helps download source repositories from GitHub.
- **Java 8** → Programming Language used for CoreASIM's base system implementation; thus, also for running ASIM instances.

3.2.2 Provisioning Process

The provisioning process can be separated into 3 steps which are executed when *.configure* for docker or *vagrant up* for vagrant is called from the command line:

1. Download ASIMSpec and ICEF from GitHub
2. Install the software packages mentioned in Section 3.2.1
3. Build the ICEF framework and prepare a virtual machine for execution

For development purposes the ASIMSpec and the ICEF frameworks are cloned into directories which are available from the host machine and the virtual guest machine. This is necessary because then it is possible to directly edit the source or debug from outside and execute the code from within the container. Consequently it is not necessary to copy the code into the container when changes are carried out.

For **vagrant**, a shared folder is configured over the vagrant file shown here:

. . .

```
config.vm.synced_folder "./data", "/vagrant-data"
...
```

This directive refers to the fact that the folder *data* is used on the host system and a folder *vagrant-data* is mounted on the guest system. These folders are shared and thus contain the same content. Additionally, during provisioning a symbolic link in the home directory is created called project (*/home/ubuntu/project*), which links the mounted root folder /vagrant-data.

When the virtual machine is stopped the data directory on the host is kept and can still be manipulated or executed (of course only if the framework dependencies are installed on the host as well).

For **docker**, first all GitHub sources need to be cloned to the host machine, and only then can files be shared into the guest container by using the command line option "-v"

```
docker run -v "$1/ASIMSpec:/home/ASIMSpec" \
           -v "$1/icef:/home/icef" \
           --name active_asim -it asim /$2
```

This line is part of the script $scripts/runDocker.sh$ where "$1" denotes a bash variable and contains the first command-line parameter of this script. That parameter is used to declare the directory location of the environment. The second command-line parameter "$2" normally is defaulted to the main execution script ($executeASIMSpec.sh$) and started inside a running docker instance. It is important to note that the *ASIMSpec* folder of the host machine is mounted into[7] $/home/ASIMSpec$ of the docker container and the $icef$ directory is mounted into $/home/icef$.

3.2.3 Execution

The building block view in Figure 3.2 shows how an ICEF specification is submitted to the environment where it needs to be instantiated for it to be executed. The details will be covered in Section 3.3. In this part of the document we take a look at which services are started and how.

Due to limitations of the ICEF framework it is currently not possible to cleanly shut down a running ICEF simulation with several running ASIM instances. Therefore the whole environment has to be restarted including all services for every execution in order to guarantee a correct set-up. This is achieved and granted by the docker service which offers a base operating system image that is started for every single execution of the Interlace specification from scratch. That base OS-image is not changed and always starting from the same state when instantiated.

Start Services Processes. Both environments offer a script that needs to be called from the host system ($execute(.sh)$) and one that needs to be called from within the virtualized machine ($executeASIMSpec.sh/executeOnGuest.sh$). Whereas the script on the virtualized system only differs in directory references the outside scripts have to handle different things as one script deals with docker and the other with vagrant.

In more detail, docker can start the servers immediately and run the script inside the container, whereas vagrant needs to add an additional step if the virtual machine is not up and running yet. Thus, vagrant

1. tries to start the server processes and to submit the specifications;

[7] "Mounted into" refers to the fact that a local folder from a local host computer is available inside the virtualized docker system and accessible there as a mounted directory.

2. if the first step fails, the script checks if the virtual machine is running. If it isn't, the script tries to (re-)start it;
3. when the restart is successful the first step is retried.

If we focus on the first step, which is basically the same in both environments, we can discuss in detail how the process continues. Thus, on the guest system we are first running a so-called *CASIMA*, which is short for CoreASIM Manager 3.3. This manager takes care of ASIM states, scheduling, and also acts as messaging backbone.

Next, a second service process is started. The service is a wrapper for the CoreASIM implementation and its name is Brapper (BIOMICS wrapper). This Brapper service executes enhanced ASIM code named BSL that offers additional language primitives specifically designed during the BIOMICS project to include interaction features for decentralized and distributed computational systems inspired by and modelled on biochemical systems.

When a Brapper starts, it needs to register with a manager instance. As described in Section 3.3, it is possible to start multiple Brappers. Once a simulation is submitted to the manager, the manager distributes the different simulations, including their ASIMs, to different Brapper services to execute them there, while trying to spread the load equally.

For the sake of simplicity the environment starts only one Brapper by default. In the final implementation it will be possible to specify the number of Brappers to be used for execution.

Submit ICEF definition file. Finally, after the service processes are running it is possible to submit a specification file to the manager. This is done by executing a bash script which calls a nodejs client:

```
...
node loadICEF.js $project/ASIMSpec/run.icef localhost 9090
...
```

The above listing uses a *$project* variable containing the path of ASIMSpec to find the ICEF specification. By convention the ICEF definition file of the ASIM Specification is called *run.icef* to clearly identify the entry point for the environment. The last two parameters for the script are the host and the port where the manager service is running and waiting for requests, respectively.

3.2.4 Development

For developing new simulations it is important to have a proper development environment. There exist many different IDE/Tool/Debugger choices for Java and JavaScript which may be used for the Brapper, Manager, or coreAS(I)M plugin development. However, there are only limited Integrated Development Environment (IDE) choices for implementing AS(I)Ms.

For the well-known Eclipse IDE, developers of the ICEF framework have adopted the coreASM plugin for supporting the additional language primitives. So when implementing ASIMs using BSL it is highly recommended to use that plugin. It works with the new language elements (BSL) and is provided by the ICEF framework directly, but it needs to be built and imported **manually** into Eclipse. Thus, for working with ASIM implementations of INTERLACE you ideally DO NOT download the original CoreASM Plug-in package which is available at the marketplace of Eclipse.

Importing/Installation of the IDE Eclipse plugin for CoreASIM

Build Framework. To add the ASIM plugin to Eclipse the CoreASIM plugin needs to be built and installed manually first because it is not downloadable from the Eclipse marketplace and is only available as a source version delivered with the ICEF framework and as part of the CoreASIM engine. Building and installing can be done by calling

```
cd icef/coreASIM/org.coreasim.parent && mvn package install
cd icef/coreASIM/org.coreasim.eclipse && mvn package install
```

Note that the ICEF directory is placed in different folders in the two environments!

Install Plugin Development Environment in Eclipse. To install the ASIM plugin it is necessary to first add another plugin called Eclipse PDE (Plugin Development Environment) from the marketplace for the current Eclipse installation or get a distribution which already contains that plugin.

Import Plugin as Project. Next, the plugin needs to be imported to the workspace, which can be done in Eclipse using the import wizard. To reach the wizard go to "File" → "Import...", search for "Existing Projects into Workspace" and click to get to "Import Project" Window. Then click "Finish" to import the project.

Dry-Run Eclipse with CoreASIM Plugin. This optional step can be done to check if the plugin is working correctly. For that it is possible to right-click the imported Eclipse project and go to “Run As” → "Eclipse Application". Then a second Eclipse instance is started but this time a new tab should appear named “CoreASIM”. In addition, when opening a file with extension *casim* the ASIM definition should be syntax-highlighted.

Enable the Plugin. If the (optional) dry-run has been successful the export wizard will export the plugin into the running Eclipse installation by opening "File" → "Export" then choosing "Deployable plugins and fragments" in the new Dialog. The window which is opened next will offer to export fragment *org.coreasim.eclipse* by selecting the checkbox next to it. Before clicking "Finish", the combo box "Install into host" has to be chosen.

Once these steps have been completed, upon restart Eclipse will offer a new tab in the top menu bar called "CoreASIM".

Note on the installation: When using the *docker*-Environment you could skip the first **Build Framework** step, because all the necessary build steps are covered by the initial configuration script. Also a separate installation of the Eclipse plugin Development Environment (PDE) might be omitted because it usually comes in a bundle with most standard J2EE installations of Eclipse.

Notes on the Docker Environment

Normally the ASIM Specifications are started by calling *execute* from the main directory. This starts a script which will run the required services and then send the ICEF definition. When stopping the container it instantly goes back to its original state before execution. Only the shared directories ASIMSpec and ICEF will retain the changes.

If it is necessary or convenient, for example if the host is missing development packages (e.g. maven, nodejs, ...) which are available inside the container, it is possible to work from within the container by calling

```
./execute /bin/bash
```

which starts the docker container, runs a bash shell instead of the starting script, keeps the STDIN open, and connects a pseudo TTY. In this way it is possible to work with the container’s bash shell, which can be seen as analogous to connecting to a remote shell over SSH.

3.3 ICEF - The Interaction Computing Execution Framework

The interaction framework wraps the original coreASM framework in order to extend it and give it the capabilities to support concurrent and distributed computation. It was developed in a project called BIOMICS and financed by the European Commission.[8]

This wrapping took place on three levels:

First, the interpreter coreASM had to be extended supporting additional language primitives as well as communications features. Here BSL replaces ASM as a new language having a new interpreter CoreASIM.

Second, a space was created where the now so-called Abstract State Interaction Machines (ASIMs) take over and are able to execute in parallel. This environment is called Brapper (short for BIOMICS Wrapper).

Third, a central server called manager takes care of handling distributed Brapper instances, dealing with message and scheduling issues.

In terms of technology, the development of CoreASIM involved making changes to the Java coreASM implementation. Brappers are written in Java as well but the managers coordinate the Brappers using nodejs and are therefore written in JavaScript.

3.3.1 Framework Stack

We now describe in more detail the aspects of the ICEF framework stack that concern the INTERLACE implementation. Figure 3.2 shows the stack at different levels of granularity.

As a stable base for the INTERLACE execution environment stack we chose a Linux-based system, Ubuntu 16.04 LTS. A LTS (Long-Term Support) version is important to ensure that we have a reliable platform that is maintained by the distributor for a long time. Ubuntu publishes LTS versions every two years and promises a maintenance duration of five years.[9]

After installation of the software described in Section 3.2.1 and after building the ICEF components, the framework is ready. When the components are started it is possible to transmit a running definition, called ICEF JSON file.

Listing 3.1 shows what the specification for a simulation may look like. An *id* for the simulation is given, along with one scheduler and one ASIM. The *schedulers* section normally only hosts one scheduler which takes care of other active or suspended ASIM agents. The *asims* attribute in the JSON file defines the actual ASIM instances running the simulation.

Side note: A definition for the client here is not given because the client, which sends credit, debit or other requests, is spawned and destroyed on demand by the Scheduler ASIM defined in *casim/scheduler.casim*.

Detailed descriptions of parameters and options of the JSON ICEF specifications can be found in Deliverable D5.2 [5] of the BIOMICS project.

[8] http://biomicsproject.eu/
[9] https://www.ubuntu.com/info/release-end-of-life

```
{
    "id": "interlace",
    "schedulers": [{
        "file": "casim/scheduler.casim",
        "start": true
    }],
    "asims": [{
        "file": "casim/server.casim",
        "start": true
    }]
}
```

Listing 3.1: **Example ICEF JSON Specification**

The implementation idea here is to have a central server which takes over the various requests from the clients. A scheduler will take care of what exactly is running, and when. As a second task the scheduler will, as mentioned before, also spawn new clients which send or receive information/requests to other components, but mainly to the server.

This centralized client-server architecture is a reflection of the need to replicate some of the functionality of the current platform, which is based on a centralized relational database. Thus, when the centralized database is replaced with a blockchain backend the business logic should remain the same, at least at first, even though ASIMs work independently and a set of interacting ASIMS can be distributed over a network and be running at different locations.

The Submission Process of the *run.icef* works as follows:

Loading and parsing of the ICEF-File initializes the process. Using that specification file a node.js component transmits the structure to the manager server process which acts as the central managing and communication node.

Starting Simulation on the manger service process. During that process the service registers a new simulation and all components necessary for managing the different resources like messaging.

Distribute ASIMs. When simulations are initialized requests are sent to one or more Brappers in order to start up the actual running instances of the ASIM agent.

Run. Finally, when all ASIM instances are running the simulation is successfully executing till stopped from the outside or by a problem during execution.

Figure 3.2 illustrates on different levels of detail which components and services are active and necessary to get the processes described above to run an ASIM simulation and, in the case of INTERLACE, the model specification.

3.3.2 CoreASIM

The very core of the ICEF is an engine which was developed by different people at different universities[10] and implements a language called ASM. This language is based on Abstract State Machines [2, 1], which is used as a methodology for high-level system engineering, design, and analysis. This engine, called coreASM, was enhanced by the BIOMICS project in order to simulate computer interactions on a logic-based programming language.

The resulting engine is called CoreASIM and includes the interaction features that were missing in coreASM [4][5]. CoreASIM replaces the ASM language with the BIOMICS Specification Language (BSL). The changes from the original framework also comprise of:

[10] https://github.com/CoreASM/coreasm.core/wiki/About-CoreASM

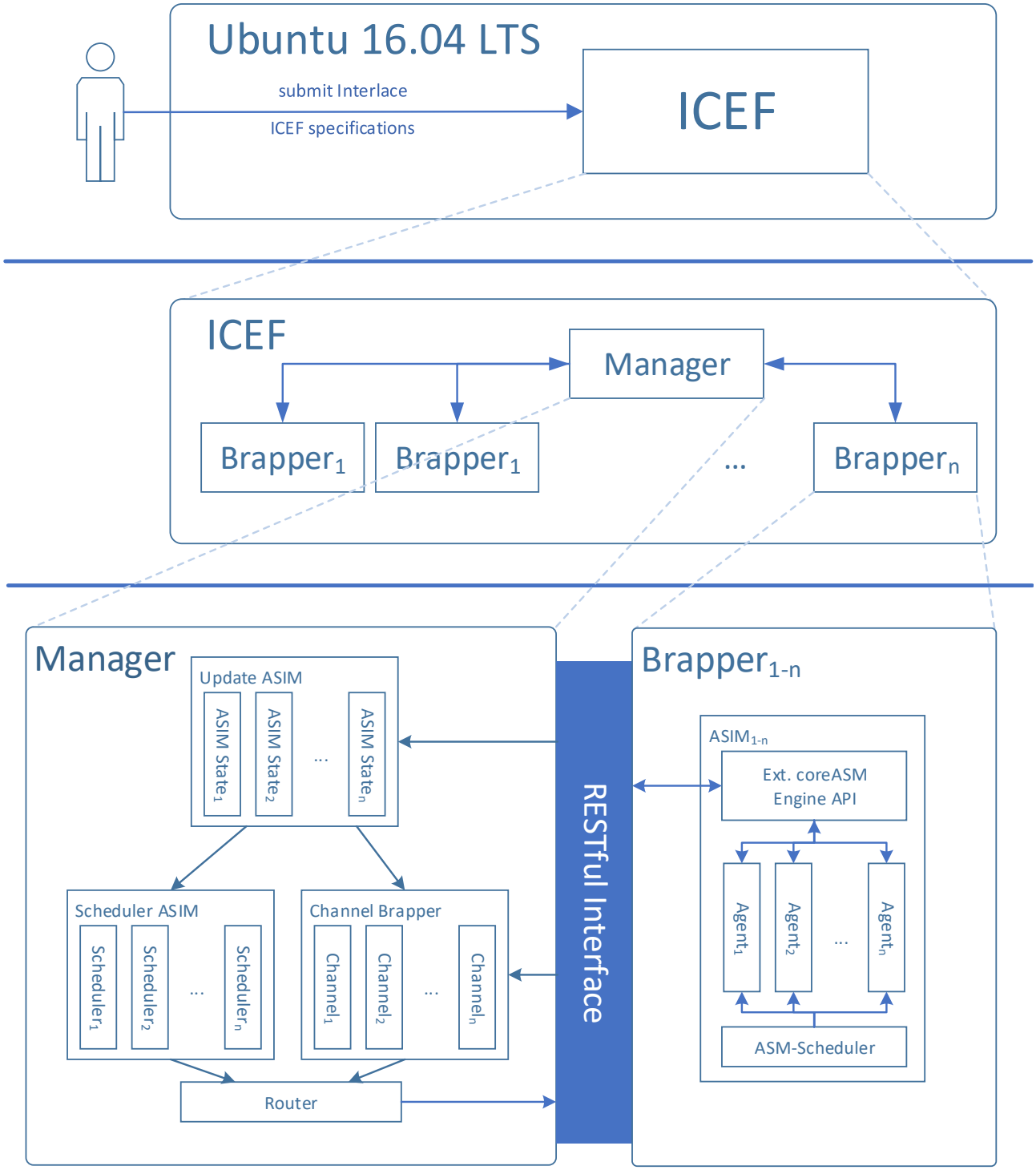

Fig. 3.2: **ASIM Execution Environment Overview**

- interaction possibilities
- abstract shared storage
- mailing system
- full scheduler
- interpreter enhancements

As this document provides an overview only on the CoreASIM features, just the most important implementations are described here; namely, the mailbox and the scheduling system.

The mailing system is easy to use and can be described as follows:

An ASIM $asim_1$ can send a message to another ASIM $asim_2$ by using the Send Message rule.

```
send Element to "asim_2" with subject "a subject"
```

Where the *Element* can be any of the element locations used in the ASIM language space – even, for example, *program(self)*.

The scheduler has the ability to coordinate different local agents to ensure that they act in a predictable and appropriate way. Like the coreASM scheduler, CoreASIM implements a controlling mechanism at a higher level. At the beginning of each running step, a scheduler takes the defined scheduling policy and uses it to determine a set of local agents which will be signalled to run their program. There are different BSL constructs to do so.

Here are two examples:

```
forall a in Agents do schedule a
```

Tells the scheduler to run all agents in *Agents*. If, by contrast, a single agent should be selected based on a condition $cond$ met by an agent a, the corresponding rule might look like this:

```
choose a in Agents with cond(a) do schedule a
```

3.4 Model Execution Environment Details

This section describes the docker environment in depth, how the scripts prepare/build the environment, and what is needed to finally execute the INTERLACE Model Specifications described by *run.icef* of the ASIMSpecs.

Starting from the directory structure in Figure 3.3, a detailed picture can be drawn. Based on performance and development considerationss docker was chosen in favour of the vagrant environment. Although scripts of the two environments are quite similar we focus on docker.

3.4.1 Environment Configuration

A crucial step for initializing a proper environment is the so-called provisioning of the docker Environment. The *configure* bash script handles this process. The script is prepared to be executed on Mac, Windows (using git bash), and Linux. It was tested with the docker community edition. It was not yet tested with the boot2docker environment and therefore might have problems during the provisioning process. Figure 3.4 shows the configuration process.

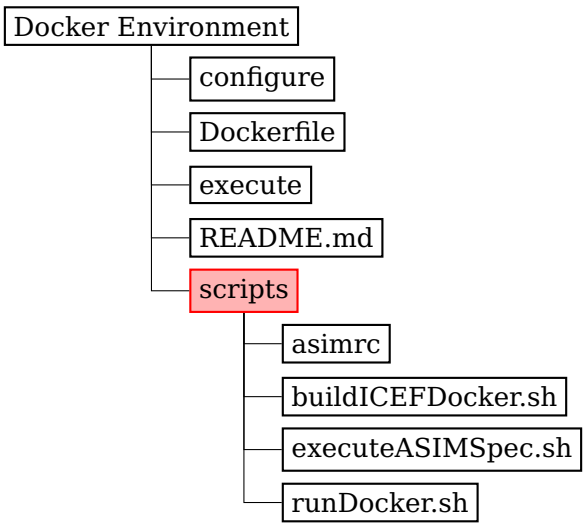

Fig. 3.3: **Docker Environment File Structure**

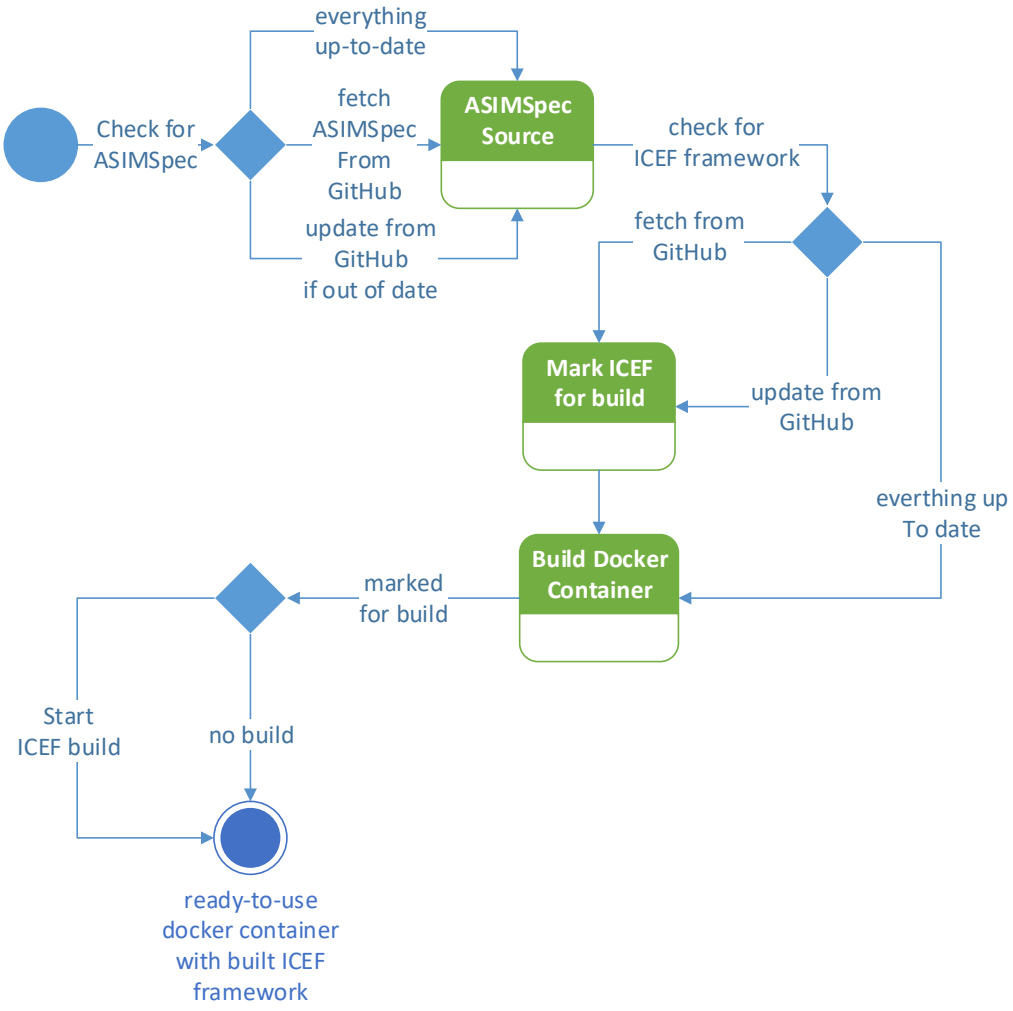

Fig. 3.4: **Docker Configuration Process**

Summarizing this process, the script tries to fetch the ASIMSpecs as well as ICEF from GitHub, builds the docker container and finally builds the ICEF framework if necessary. The ICEF framework build is done by the bash script in directory *script/buildICEFDocker.sh* and is executed inside the container.

Building the docker container is a core part of this process, in which the most important tools and packages are installed. These software packages are listed in Section 3.2.1 in detail. Docker uses a file called *Dockerfile*, listed in Figure 3.3, to know how the container is provisioned. The file contains several commands which are executed in a playbook-like manner to produce a deterministic system environment.

3.4.2 Execute the ASIM Specifications

After the environment has been configured the INTERLACE specifications are ready to be executed. This can be done by calling the *execute* script, which starts the prepared docker container and initializes a process illustrated in Figure 3.5.

This process tries to make development as well as instant executing easier by starting a manager, one brapper, and submitting the ICEF definitions using a single script.

Fig. 3.5: **Docker Execution Process**

Note on service locations: In directory *scripts* of Figure 3.3 a file called *asimrc* can be found which acts as a central setup file for configuration and execution and contains the main service locations in the containerized Ubuntu system.

Chapter 4

CoreASIM Implementation of the INTERLACE Business Logic

Eduard Hirsch, Maria Luisa Mulas and Paolo Dini[11]

This chapter describes the ASIM implementation of the INTERLACE business logic according to the specifications of Deliverable D2.1 [3], the requirements specification refinement presented in Chapter 2, and the formalization of the refinement provided in the Appendix.

While Chapter 3 discusses the code execution environment, this chapter describes how that environment was utilized. The following detailed discussion of the implementation design shows what issues were taken into account and what difficulties were overcome.

4.1 Introduction

The requirements specifications are the basis for the implementation of an executable ASIM model. That model can be found on GitHub,[12] and will act as a foundation and test/verification template for further business implementations.

This chapter focuses on the way the ASIM model was implemented and how the missing functional parts of the backend, which is mainly about simulating a simple ledger, were realized.

4.2 Agents

The implementation is based on several ASIM agents which are programmed to act independently and to communicate with each other through the messaging system provided by the ICEF infrastructure.

There are small differences between the available ASIM agents. They can be categorized into the following three groups, based on their purpose:

- full agents
- dynamic agents
- non-functional agents

Full agents are started right away when the ICEF simulation is launched. They process requests or handle other duties over the environment's lifetime. *Dynamic agents* are created during a specific phase of a test and destroyed after that test has been completed. *Non-functional agents* are never started directly and are structured to facilitate their integration into another agent. They host initialization code or helper functions in order to compensate for the missing modularization feature of the ASIM-BSL language.

[11] Eduard wrote this chapter and was the main implementer. Marylù supported the implementation effort. Paolo did no implementation but thoroughly edited this chapter to optimize clarity of expression.

[12] https://github.com/InterlaceProject/ASIMSpec

To get more familiar with the actual ASIM realization an extract of available agents is listed in Table 11. They may act as a template for extending the scenarios for additional use cases or tests. They cover the core payment functionality, thus credit as well as debit operations.

Agent	Function	Type
scheduler	Scheduling, creating and destroying dynamic agents	full
server	Server which talks to the dynamic clients in order to handle payment and ledger-specific requests	full
CreditRequestClient	Handles a test credit request	dynamic
DebitRequestClient	Initiates a test debit request	dynamic
DebitAcknoledgeClient	Confirms a test debit request	dynamic
initdata	Fake database and backend initalization code to be included into the *server* agent	non-functional

Table 11: **Agent list and their respective functionality**

4.3 Execution

As mentioned in Chapter 3, the specifications are executed using the ICEF framework. In Chapter 3 we covered how a JSON ICEF file can be loaded and started. Now details are given about how that process works in particular for the INTERLACE specification.

The process, shown in Listing 4.1, starts with the main ICEF definition file in the ASIMSpec directory called run.icef by definition. The listing illustrates how the different types of agents are included in the simulation. To explain further, all agents are located in the directory *casim* together with *casim/clients*. Agent file names are the same as described in Table 11 and include the suffix ".casim". The directory *casim* contains all full and their non-functional agents, which are joined later on. The directory *casim/clients* hosts all the relevant clients talking to a server. Currently there is one client for a credit request and two clients for a debit request.

```
1  {
2      "id": "interlace",
3      "schedulers": [{
4          "file": "casim/scheduler.casim",
5          "include": [
6              "casim/clients/CreditRequestClient.casim",
7              "casim/clients/DebitRequestClient.casim",
8              "casim/clients/DebitAcknowledgeClient.casim"
9          ],
10         "start": "true"
11     }],
12     "asims": [{
13         "file": "casim/server.casim",
14         "include": [
15             "casim/initdata.casim"
16         ],
17         "start": "true"
18     }]
19 }
```

Listing 4.1: **ICEF JSON Specification for INTERLACE**

When looking closer at the ICEF definition, one can see that there are a couple of agents defined inside of a JSON array with an attribute called *include*. All agents in that array will be added to the main file. That means that CreditRequestClient, DebitRequestClient and DebitAcknowledgeClient are added to the scheduler and initdata is added to the server agent.

Important note: The include syntax is used to append code from a different file to an agent but comes with strings attached. See Section 4.4 for further details.

Consequently, the loading module will assemble no more than two ASIMs: a scheduler and a server. The assembled JSON String is sent to the manager which initializes the simulation environment and distributes the clients over the available brappers.

4.3.1 Main Agent Tasks

As mentioned at the beginning of this chapter, two main agents, scheduler and server, are the core of the running environment. Once started, both instantly start working. The server listens for messages on the communication channel in order to process potential requests and the scheduler initializes the first test and starts clients which should be active during that particular test. When a client has been initialized, it will carry out its assigned duties which will be to assemble a request for the server and to handle the resulting query-response traffic. Of course the request types are of different types depending on the current test and the type of the client. When the client is done it sends a corresponding message to the scheduler, which terminates it.

The scheduler, as briefly introduced above, is responsible for spawning new clients which take over different tasks. To do so, the scheduler defines various things in advance.

First, the current tests need to be identified, which is done by creating a universe as well as two locations:

```
universe TEST_STATE = { START, TEST_CREDIT, TEST_DEBIT }
controlled currentTest: TEST_STATE
controlled nextTest: TEST_STATE -> TEST_STATE
```

The universe *TEST_STATE* defines the available test list. *currentTest* is an element of the universe *TEST_STATE* and defines the currently executed test. Finally, *nextTest* is a function which takes as parameter the current test as type *TEST_STATE* and gives you as result the next test in the queue which is of the same type. To initialize the predefined locations the following commands are executed:

```
currentTest := START
nextTest(START) := TEST_CREDIT
nextTest(TEST_CREDIT) := TEST_DEBIT
```

During the execution of the scheduler's main program it is possible to transition from one test to the next by

```
currentTest := nextTest(currentTest),
```

applying the current test to the nextTest function. This can be done until the return value of the function is *undef*, implying that no other test is left.
For each *current test*, a functional location is used which returns a list of client agents for a given test state. The functional location can be defined as follows:

```
controlled nextClient: TEST_STATE -> LIST
```

That *nextClient* function needs to be set up during the initialization phase of the agent as well:

```
nextClient(TEST_CREDIT) := [ CreditRequestClient ]
nextClient(TEST_DEBIT)  := [ DebitAcknowledgeClient, DebitRequestClient
  ↪ ],
```

showing that we have one client for a credit request test and two clients for a debit request. Also important to remember is that rules for a client agent are initially defined in a separate file but, as noted already, they are included in the scheduler later by just appending the code. Thus, each tested client is absolutely required to have unique names for the *initialization*, the *program* and the *policy* rules **over all** included files!
Next, the missing parts are added up to get the full picture on how the clients are started. In particular, how the base rules for a dynamic client are defined and how they are eventually instantiated and handled during runtime. Listing 4.2 illustrates the management of the different clients.

```
1  //definition of functional location for getting
2  //a function reference for a client name
3  controlled initBy: CLIENT -> FUNCTION
4  controlled withProg: CLIENT -> FUNCTION
5  controlled andPol: CLIENT -> FUNCTION
6
7  rule Start = {
8    ...
9    //Setup CreditRequestClient rules
10   initBy(CreditRequestClient) := InitCreditRequestClient
11   withProg(CreditRequestClient) := ProgramCreditRequestClient
12   andPol(CreditRequestClient) := SkipCreditRequestClient
13   ...
14 }
15
16 rule Program = {
17   ...
18   //Client rules are defined in clientTemplate script
19   if currentTest != undef then seq
20     forall createClient in nextClient(currentTest) do
21       createASIM createClient
22         initializedBy initBy(createClient)
23         withProgram withProg(createClient)
24         andPolicy andPol(createClient)
25         in activeList(createClient)
26
27     activeClients := | nextClient(currentTest) |
28   endseq
29   ...
30 }
```

Listing 4.2: **Scheduler core functionality**

In this listing we can identify three parts:

1. The definition of three functions that take a $CLIENT$ as parameter and return a $FUNCTION$ can be observed, namely $initBy$, $withProg$, and $andPol$. This $FUNCTION$ return type can be any rule we have defined inside of the scheduler or inside of the dynamic clients which are included into the scheduler.
2. For client $CreditRquestClient$ these three function values need to be set. So for example it is possible to define an $init$ rule called $InitCreditRequestClient$ for $CreditRequestClient$ like this:

   $$initBy(CreditRequestClient) := InitCreditRequestClient$$

and afterwards to call function *InitCreditRequestClient* by using:
initBy(*CreditRequestClient*)().

3. During the iterative execution of the *Program* rule the actual starting of the client is processed. If a valid test has been selected, *nextClient*(*currentTest*) provides a list of clients valid during that particular test. The *forall* loop walks through that list and uses the corresponding iteration to instantiate the chosen client. This instantiation is done by calling the *createASIM* command and passing the start-up rule (*initializedBy*), the main program rule (*withProgram*), and the scheduling policy (*andPolicy*).

In line 27 of Listing 4.2 the count of active clients is stored. That count is reduced by one when a "Done" message has been received by a dynamic client. Also the client is shut down by calling

```
destroyASIM clientName
```

After the *activeClients* counter has been reduced to 0 again, all clients have terminated and the next test can be covered. To conclude, it is possible to say that the *activeClients* counter is used as a structure similar to semaphores, which takes care that no new clients are issued as long as it has a value bigger than 0, and the scheduler is therefore in something similar to a paused state. Certainly, is it not really paused but in a “busy wait loop”.

The server agent acts as the main component that implements most of the requirements specified, which for the moment is credit and debit operations. The server is started right away when CASIMA has received and processed the JSON file for the simulation.

When the init-rule called *Start* is executed, the *Ledger* and *PendingTransaction* are initialized as empty maps, a one-time password (OTP) lifetime is set, and the *Logger* is set up. The logger offers several logging levels and details about it can be found in Section 4.6. Also a rule called *InitData*() is executed at start-up. It is important to know that the Eclipse Plug-in shows that this rule has a Problem/Error because it is not defined in the server.casim file and thus it is not recognizable by it. Nevertheless, this rule is placed correctly and will work fine, because it is defined in the non-functional agent *initdata*. Due to the *include* statement inside of the run.icef, the content of *initdata* is appended to the server.casim before it is sent to the CASIMA manager.

Rule *initData* in the *initdata* client consolidates the rest of the initialization inside. To be more specific, the following locations are prepared for later use when it is called:

- *sessionData* ... simulates session information
- *profileTable* ... contains user profiles
- *accountTable* ... user accounts
- *userGroupTable* ... specifies users’ group membership
- *TT* ... transfer type translation table
- *accountConnectivity* ... transferability between accounts

After the start-up phase has been completed, the server goes into listening mode, in which the main program *DispatchMessages* is called iteratively. In this mode, all relevant messages received by the server are taken care of. In Listing 4.3 the dispatch process of server can be viewed in detail. All inbox messages of the current engine tick are fetched using the *inboxOf* in addition to the *forall* statement. The message reference *m* as named in the loop is used to retrieve subject, message, and sender of the current post box entry *m*.

Messages that are not recognized as having an accurate message type are discarded. The message type is defined by the message subject and is compared to one of the predefined entries

in the universe definition $MESSAGE\_REQUESTS$. For real-world use it is essential to introduce some kind of additional message signing to verify by whom the message has been sent.

```
rule DispatchMessages = {
  forall m in inboxOf(self) do seq
    //fetch message information
    msubject := getMessageSubject(m)
    msgIn := getMessageContent(m)
    member := getMessageSender(m)

    //dispatch messages
    if msgIn != undef and member != undef then
      case msubject of
        toString(CreditPreviewReq): HandleCreditPreviewReq(msgIn, member)
        toString(CreditPerformReq): HandleCreditPerformReq(msgIn, member)
        toString(DebitPreviewReq): HandleDebitPreviewReq(msgIn, member)
        toString(DebitPerformReq): HandleDebitPerformReq(msgIn, member)
        toString(DebitAckCompletion): HandleDebitAckCompletion(msgIn, member)
      endcase
  endseq //end forall
}
```

Listing 4.3: **Server message dispatching**

4.4 Modularization and Include Syntax

For the INTERLACE specification the ICEF framework has been extended to support an "include" syntax inside of the ICEF JSON files in order to import sources to an agent. This has been added in order to compensate for the "Modularity" module of ASM which stops working in ASIM because of its distributed nature.

However, it is important to realize that the include statement needs to be used with caution, because one needs to be aware that it effects nothing more than the appending of the content of the included file to the main agent file. This appending raises the following issues:

- Line numbers are different to the original files
- For a compilation problem it might be necessary to take a look at all files, the main as well as included ones
- Naming needs to be consistent throughout all files. E.g. Eclipse will not notify a developer whether a name for a rule, location, etc has been used twice.

Nevertheless, it is a useful approach for handling code separation, in order to avoid a single and extremely long file which would be difficult to maintain and work with.

In order to be able to use **Eclipse** and the ASIM eclipse hinting/error detection provided by the plugin, another “quick-fix” has been introduced. For the case when a dynamic or non-function agent needs to be added, it will be appended at some point to a parent agent. Thus definitions like “$CoreASIM\ \ asimname$” would occur twice inside of that final agent. For the interpreter to work correctly it is necessary to have only one header defining the name of an agent and, therefore, to remove that header definition inside of the included files. However, if the header definition is not present at all in the file before it is included, Eclipse is not able to provide correct syntax highlighting, hinting or error detection.

```
/*includeskip begin*/
  //this part will be removed
  CoreASIM Company

  use Standard
  init dummy
  rule dummy = skip
  scheduling NoPolicy
  policy NoPolicy = skip
/*includeskip end*/

//this rule is included to main agent
rule somerule = {
  ...
}
```

Listing 4.4: **includeskip usage**

Thus, the new quick-fix is to give the programmer the possibility to mark a section which will be removed during the agent assembly. The beginning of such a section is marked with /*includeskip begin*/ and the end with /*includeskip end*/. When using these markers they need to be placed exactly as described – no additional white spaces (space, tab, return, ...) or different letter cases. Listing 4.4 shows a simple example.

Inside of the skipped section there may be many different things placed as shown in the example in order to work seamlessly with the Eclipse hinting. Place names of locations or universes can also used. The reason for putting them there is to avoid a warning by the Eclipse plugin that the variable/location has not been defined yet; in other words, in order to check correct spelling and avoid the problems caused by any such issues only becoming obvious at interpretation time.

4.5 Dynamic Clients

This section elaborates further on the dynamic client features and functionalities. It explains how they are used as well as how they process the information and create the various requests. Further, details about the message types are given in detail.

4.5.1 Communication and Message passing

As mentioned in Chapter 3, Section 3.3.2, communication in the ICEF framework is based on message passing. Thus each agent has the possibility to send messages to a named agent as well as receive messages from any other. The messages can be picked up at the agent-specific mailbox using the designated functions provided by the communication plugin.

All agents work independent, do not share any states, and only are aware of their own status. Consequently, they have to rely on the mailing system to share information. This is an important fact because in a real-world scenario clients are also working independently of each other. In order to support this independence ASIM agents apply the distributed design pattern.

Messages from one client to another are not encrypted. Thus it is important to realize that security needs to be taken into account separately. Security is currently not part of the implemented model but it is a very important part of the planning of a business-ready product.

4.5.2 Message Types

At the moment there are several message types that are part of debit and credit requests. Table 12 gives an overview of which messages are in use for handling transfer operations. The messages are listed in the order of occurrence. However, the precise overall communication sequence is shown in Figures 4.1 and 4.2. Finally, the message types listed in Table 13 are operation-agnostic (debit, credit) but vary based on their transfer parameters.

Message Name	Purpose	Attached Parameters
CreditPreviewReq	first check of **credit** request	CRP*
CreditPerformReq	request to actually perform a **credit** request	CRP*
DebitPreviewReq	first check of **debit** request	DRP**
DebitPerformReq	request to actually perform a **debit** request	DRP**
ConfirmationReq	to ask debitor for permission to perform the **debit** transfer	DRP**, OTP***
DebitAckMsg	debitor gives permission to perform the **debit** transfer	DRP**,[1], OTP***

Table 12: **Credit/Debit message types overview**
(*Credit Request Parameters: from-account, to-account, amount, meta-data, channel, member)
(**Debit Request Parameters: creditor, debtor, amount, meta-data, channel)
(***One Time Password, [1]optional)

Message Name	Purpose	Attached Parameters
Proceed	answer from server that the CreditPreviewReq has been successful	CRP*/DRP**
DoNotProceed	answer from server that the CreditPreviewReq has NOT been successful	error message, CRP*/DRP**
NotPermitted	answer from server that the CreditPerformReq has NOT been successful	error message, CRP*/DRP**
TransferPerformedSuccessful	answer from server that the CreditPerformReq has been successful and the transfer has been recorded and confirmed	success message, CRP*/DRP**
Done	Message to Scheduler that the client simulation is done and can be terminated	none

Table 13: **Operation-agnostic message types overview**
(*Credit Request Parameters: fromAccount, toAccount, amount, meta-data, channel, member)
(**Debit Request Parameters: creditor, debtor, amount, meta-data, channel)

Let's review the implementation of a credit preview request to a running agent called "server" in Listing 4.5. When taking a closer look at line 9 it can be observed that the *send* command is setting a location named *CREDITREQUEST* for the message content and one called *CreditPreviewReq* is used for defining the message subject.

CREDITREQUEST is of type of map that is used in the BSL/ASM language as a set of key/value-pairs. That map carries the information required for that message type, implying that for the credit transfer example in Listing 4.5 the parameters are chosen accordingly (see CRP in Table 12).

The actual message type is determined by the subject. As mentioned, the *CreditPreviewReq* content is used for defining the subject. At client level, unfortunately, it is not possible to predefine a universe containing the *CreditPreviewReq* like it is done in the head section of the

server. The reason is that the $createASIM$ command used in the scheduler for spawning dynamic clients does not allow to include a header section: just an init rule, the main program, and the scheduling policy. The consequence is that the message types used for clients are locations of type $STRING$, which are created in the init rule and set up as locations whose names are the same as their values. See line 1 in Listing 4.5.

```
1  CreditPreviewReq := "CreditPreviewReq"
2  CREDITREQUEST := {
3          "from" -> "accId1",
4          "to" -> "accId2",
5          "channel" -> "Service",
6          "amount"  -> 2000,
7          "metadata" -> {"message", "some transfer"}
8        }
9  send CREDITREQUEST  to "server" with subject CreditPreviewReq
```

Listing 4.5: **send message**

Concluding, the subject containing the message type is always converted to a $STRING$ representation in order to stay consistent. Also the server agent, which uses a universe definition, converts its enumerative representation of message types to $STRING$ by calling e.g. $toString(CreditPreviewReq)$. An example is shown in listing 4.5 for a $CreditPreviewReq$-Message.

The reason for naming locations exactly after their content is based on a programming principle which says that strings which are static and never change should be represented by an unchangeable variable name or, in ASM/BSL language terms, as a predefined location. This best practices principle has the following benefits:

- Typing errors can be immediately found because an IDE[13] usually shows them as unknown or undefined. Whereas a misspelled text might be only found at runtime.
- Not only typing errors but also not yet defined locations are marked by the ASIM-Eclipse Plug-in. Thus, in case of Interlace specifications, unrecognised message types can be found quickly and added to the message type space.
- Known variables/locations can be provided through syntax completion, thereby simplifying and speeding up the implementation process.
- Usage of specific types might be restricted under certain conditions.

4.5.3 Client Features and Functionalities

For INTERLACE, and of course for any other software project, it is important to test the correctness of the model and of the implementation. This document will not talk about these tests; rather, it covers details of how clients are prepared to support a testing environment. These clients will later be programmed to act like user devices performing a specific action or transfer.

Such clients are called “dynamic” due to their limited lifespan and as explained in Section 4.2 are not intended to be started right at the beginning of the INTERLACE simulation but, rather, on an on-demand basis. Thus, their lifespan is meant to be restricted to the duration of their specific task.

[13] Integrated Development Environment

Dynamic agents are used to simulate a user device where a user has been logged-in. In figure 4.1and 4.2 the dynamic agents are *CreditRequestClient* and *DebitRequestClient* respectively. To add this authentication mechanism an additional layer has been introduced which was not part of the requirements definitions. This layer, explained in detail in Section 4.7.3, adds a pseudo-login possibility and holds session-like information for each of those clients created by the scheduler. Thus, in terms of network topology the name of an ASIM is defined as the address of a device and the (member-)name of a user can be looked up inside a session table.

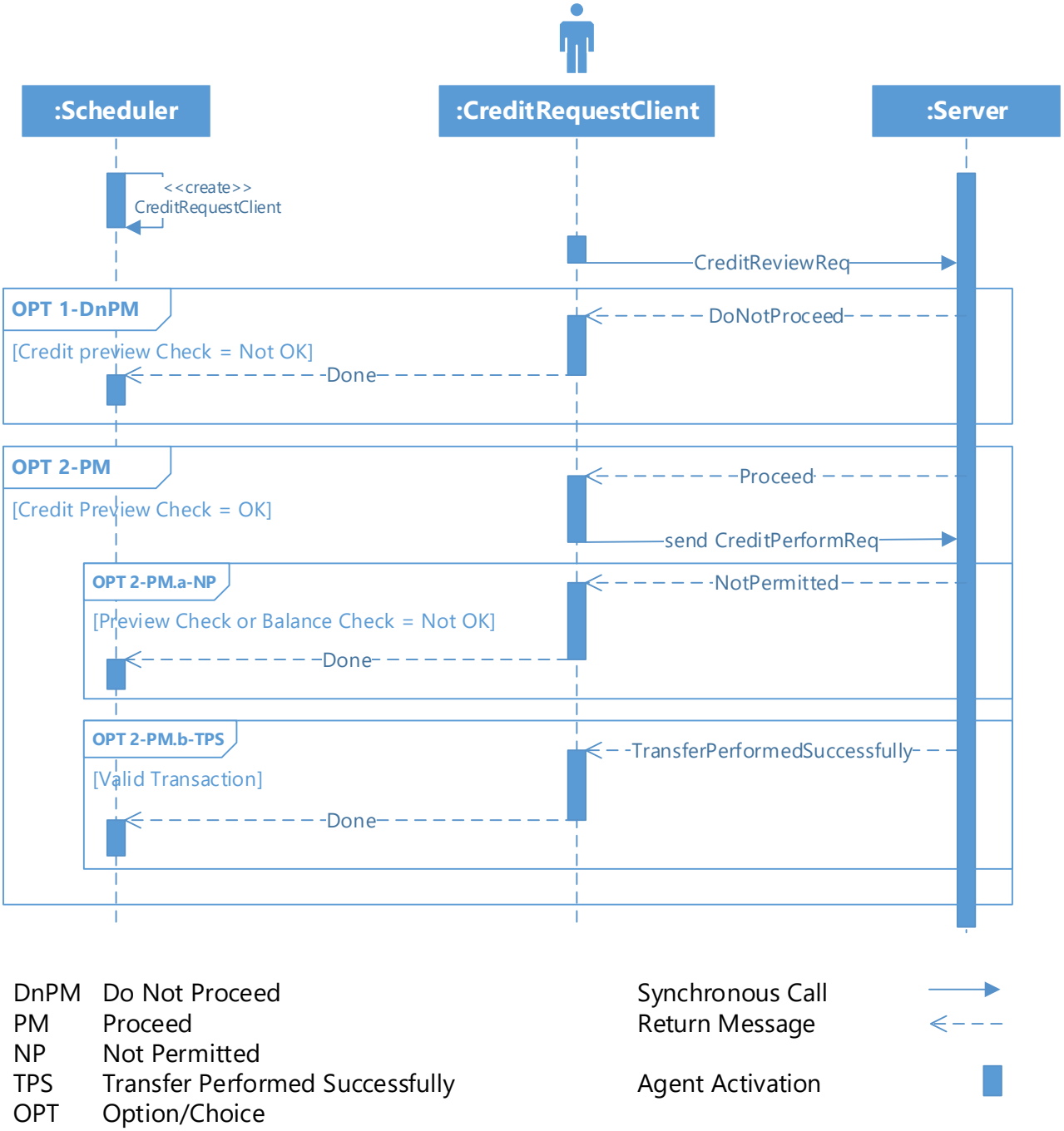

Fig. 4.1: **Credit request message protocol**

When following the sequence diagram in Figure 4.1, it is necessary to keep in mind that these messages are sent from a *CreditRequestClient* but initiated and owned by a member who is logged in and registered in the session's data lookup table. For the server agent process it is important to know which member is currently sending that transfer in order to perform various

checks and store valid transfers. Certainly, the same applies for the *DebitRequestClient* and the *DebitAcknowledgeClient* of Figure 4.2.

Dynamic Request Clients start off when they are created by the *scheduler* agent and do not send or receive any request beforehand. Consistently with a typical state machine, the clients manage their own states and act according to them. CRC[14] and DRC [15] use four main states that are explained in detail in Section 4.5.4.

The message sequence from client to server is specified in Section 4.1 for the CRC and in Section 4.2 for the DRC. In comparing those two figures, it is clear that the message sequences look very similar: only the names vary by their prefix (debit versus credit). However, what might not be obvious here is that the request parameters are quite different.

Another main difference between the credit and debit message sequences is, though, that a debit request needs *ConfirmationReq* as well as *DebitAckMsg*. This is necessary because an initiating creditor (Seller) needs to have the transaction confirmed by the debtor (Buyer) to be sure of having a valid transaction which credits the payment to the Seller's own account.

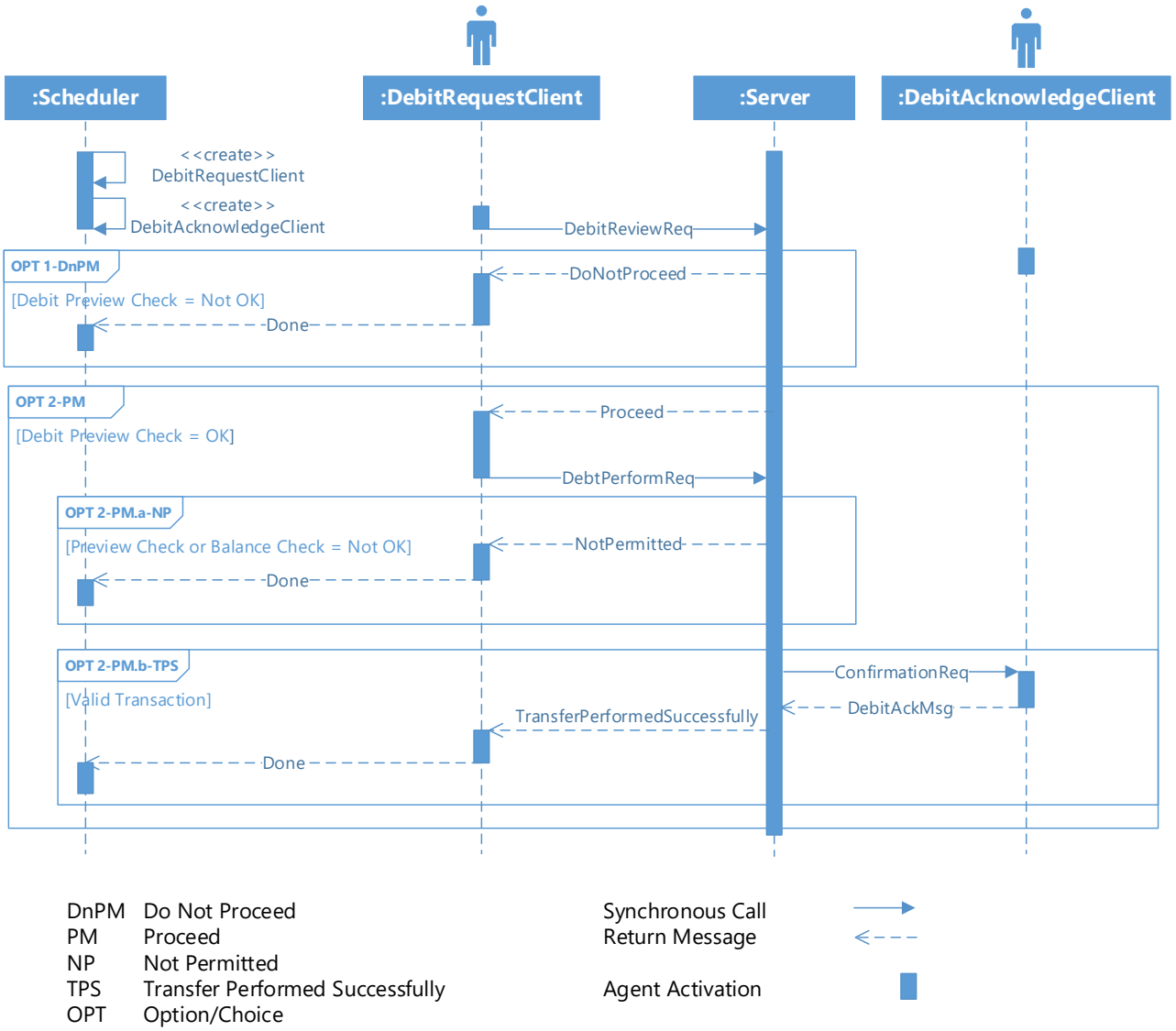

Fig. 4.2: **Debit request message protocol**

[14] Credit Request Client
[15] Debit Request Client

The sequence diagrams denote optional flows using option boxes. These boxes have a unique, numbered name. An option OPT at the first level has a number (choice 1 or 2) as well as an abbreviated name ($DnPM$, PM). Sub-options in the second level are alphabetically numbered and also followed by an abbreviated name. Square brackets underneath the option name indicate the current option choice.

ASIMs currently do not have the possibility to sleep for some time or until an event has occurred. Consequently, when waiting for a particular incident or circumstance they need to implement a so-called busy-wait. That means that the Program is still executed at each ASIM “tick” and not put to sleep by the OS-scheduler. Nevertheless, agents are able to reduce the executed commands in the current tick to a minimum, thereby achieving a similar result. For standard multi-threaded programming this would still be a huge issue and might even cause a whole system to be unusable. For ICEF, however, this is not the case because at each tick every program gets time to finish while others are pausing if they are done faster with their current execution step. Inside of the sequence diagrams, activated clients are shown by the vertical fully colored bars. Non-active phases are indicated by the vertical dashed lines.

The emulated users in the figures are pictured on top of the clients to show that requests are sent by particular members.

Scheduler and server are agents which can be seen as autonomous processes in the simulation. The scheduler mainly acts as an orchestration tool for testing purposes and will be removed for actually deployed applications, whereas the server agent needs to exist in a real-world scenario. Maybe not as a process running on a centralized server structure but when taking a real distributed scenario into account it will be implemented using some kind of smart contract running inside of a virtual machine in a blockchain environment.

4.5.4 State Management

States are managed over four predefined pseudo-static locations. Another location called $client\_state$ holds the current agent state which is one of those four states. A state change can be achieved by simply assigning a new state to $client\_state$ location like $client\_state := RECEIVE$. In the next program-iteration of the agent that state change will be processed. The four possible states to set are as follows:

- SEND: only when sending a message
- RECEIVE: receive and directly respond to messages
- TERMINATE: end execution; send “Done” message
- DONE: everything is done; ready to be terminated

$SEND$ might be used if a sequence of messages is sent where no reply message is needed, or just to submit an initial message at start-up before going right into the $RECEIVE$ mode.

$RECEIVE$ waits for the messages and acts accordingly. When a message has been received and it is necessary to send a response it is done right in $RECEIVE$ mode without changing to $SEND$ mode.

$SEND$ and $RECEIVE$ states could be set alternately, but when the $TERMINATE$ state is set to $client\_state$ the process of shutting down the agent is going to be started at the next iteration. When finishing the termination process agent state $DONE$ will be reached. Once the $DONE$ state is active, even if not stopped yet by the scheduler, the client won’t do anything useful anymore.

The DAC[16] uses the same states except for the *SEND* state because it only needs to receive and confirm a transfer by using the given one-time password. The DAC is an example of the fact that not all of the states need to be covered by a client, which may be relevant when creating new clients.

Server & Scheduler States are not technically a part of this section, since it is mainly about the dynamic clients. Nevertheless, a brief explanation of their states is given here to keep them together.

The server does not have any particular mode or state management yet. It just waits for messages and when it receives one it processes it directly.

The scheduler mainly has its test states, as mentioned in Section 4.3.1, and transitions from one test to another till there are no tests left. For each test the scheduler has two stages:

- **First**, in the starting phase all the clients of that particular test are started and counted. Clients which should be part of one test state need to be defined as dynamic and appended to the scheduler by using the *include* statement in the *run.icef* definition.
- Then, the scheduler in its **second** phase waits for as many *DONE* messages as the number of clients that have been instantiated. Finally, when the client count is zero, the scheduler goes to the next test and starts that two-phase process again.

4.6 Logging

The server agent needs to place many different status messages of different importance. In order to manage their occurrence, a simple logger has been added. The logger introduces five log levels:

1. FATAL
2. ERROR
3. WARN
4. INFO
5. DEBUG

These levels denote the different severities that can be recorded into the logger and necessarily communicate the maximum severity of a message presented to the executing user. To explain further, *FATAL* messages should be printed always, whereas *DEBUG* messages are only of interest for developers who need very detailed information of the current state.

Consequently, the maximum log level can be given at agent start-up inside of the *init* rule. The rule used to define the level is explained here

```
//rule header
rule initCustomLogger(set_log_level) = ...
...
//init logger call
initCustomLogger(DEBUG)
```

When initialized, the logger can be used at any place by calling the *do_log* rule, which takes two parameters. The first is the actual message printed to the output and the second parameter

[16] Debit Acknowledge Client

describes the importance of that message by using one of the five mentioned severity levels. The listing below shows first how to log a message with severity *INFO* and the second call illustrates a *FATAL* error message output.

```
// An "INFO"-level message
do_log("Transfer has been performed successfully", INFO)
...
// A "FATAL"-level error message
do_log("List 'Ledger' in unknown state", FATAL)
```

Currently, the logger implementation can only be utilized meaningfully inside of the server itself, because the implementation is also hosted by the server agent definition file *server.casim* directly. It would certainly be possible to put the logger into a different file, but the *include* statement of the *run.icef* is not supported by the Eclipse plugin. Consequently, if the logger implementation were separated out into an additional file, all occurrences of *do_log* calls would be marked as erroneous (even if they were not) and would then turn the coding-assistance tools inside of Eclipse into an unusable environment.

4.7 Test Scenario

This deliverable does not cover the test management for the ICEF implementation of INTERLACE. However, in order to check if a base workflow of a requirement is working in its most simple form a strategy has been followed which has been partly explained already by other sections of this chapter. The scope of this section will therefore be to put these parts together and extend the missing bits.

4.7.1 Separation of Concerns

When looking at the environment from a testing perspective there are three parts which can be distinguished:

1. Actual system requirements implementations
2. Testing routines
3. Simulation environment

The "actual system requirement implementations" cover parts that are needed for a possible real-world system. Without these, the system would not be able to function. Therefore, they can be seen as the core part of the application which is supposed to be tested.

Second, code parts have been implemented which are only there for performing and validating a test of a particular scenario use case. Currently these scenarios are credit and debit operation requests sent from virtual clients.

Last, parts of the system had to be implemented in a prototypical and most basic form in order to achieve a very simple executable version which can accommodate the test scenarios. These parts were not included in the requirements definitions or in the refinements. These parts comprise implementation details not relevant to the requirements or to the main ledger implementation.

In Section 4.3.1 the main agents are explained, *scheduler* and *server*. When assembled for execution, they are fully implemented. The *server* covers the actual system requirement

realizations as well as the simulation environment code. To be more specific, the *server* dispatches the message requests explained in Section 4.3 and performs read actions to the virtual profile, account, etc. information, together with pushing transactions to a primitive ledger discussed in Section 4.7.2.

The *scheduler* mostly contains testing routines but partly also routines that handle the necessary client implementations that are not covered by the requirements. The test routines are responsible for starting and destroying clients. As a short reminder, the *scheduler* is able to do so because it also contains all the client code. Further, Listing 4.5 shows the sending of a test credit request issued by the *CreditRequestClient*. This message is processed by a challenge-response sequence between the server and the client that imitates a transaction.

Concluding, a correct execution is currently only shown by a client which outputs a transfer-performed message when done. In case of an error the client would provide a message indicating the cause.

4.7.2 Simulation Environment

The simulation environment created for enabling the core server parts to be executed and later be validated has an active and a passive part. The so-called active part takes care of storing a transaction into the ledger, while the passive part contains various pre-prepared locations which are needed to satisfy the different constraint requirements.

Locations: Beginning with the passive part, the following locations have been defined in the non-functional client *initdata* shown in Listing 4.6.

```
1 controlled TT: OPERATION * UNIT * USER_TYPE_GROUP -> SET
2 controlled AccT: OPERATION * UNIT * ACCOUNT_TYPES -> SET
3
4 controlled profileTable: MEMBER_ID -> MAP
5 controlled accountTable: ACCOUNT_ID -> MAP
6 controlled userGroupTable: MEMBER_ID -> USER_TYPE_GROUP
7 controlled sessionData: MAP
```

Listing 4.6: **Simulation environment locations**

The two most important are the functional locations TT and $AccT$. They correspond to the transfer type functions described in Section 2.3.3 and the account connectivity conditions presented in Section 2.3.4, respectively. TT was predefined precisely in the requirement definitions and is described as a functional mapping

$$TT : Operation \times Currency \times Group \rightarrow \{G \mid G \subseteq Group\}.$$

So we can define that for a Credit operation in Sardex (SRD) an *Employee* is allowed to perform a transaction with a member of the circuit which is part of one of the groups *Company*, *Retail* or *Full*:

$$TT^{Credit,SRD}(Employee) = \{Company, Retail, Full\}.$$

When translating this discrete functional mapping to BSL, the outcome looks like this:

```
TT("credit", "SRD", "employee") := {"company", "retail", "full"}
```

The account connectivity function $AccT$ takes operation, currency, and account type as parameters and maps them to a group of account types:

$$AccT : Operation \times Currency \times AccountType \rightarrow \{Acct \mid Acct \subseteq AccountType\}.$$

As discussed in full detail in the Appendix, when replacing the parameters by actual values we obtain:

$$AccT^{Credit,SRD}(X) = \{CC, Domu, Mirror\} \textbf{ if } X \in CC,$$

which then can be transformed again into the "implemented" form and is represented like this inside of the *initdata* agent:

```
AccT("credit", "SRD", "CC") := {"CC", "DOMU", "MIRROR"}.
```

The other locations listed in Listing 4.6 accommodate important information for groups, members, and sessions as well as accounts. Except for the $userGroupTable$, which just maps a user to its current operational group, all functions implement a result of type MAP. A MAP is similar to a simple set with the difference that each entry of a MAP is a key-value pair.

```
1  //init profile data
2  profileTable("mbrId1") := {
3    "lowBalanceAlert" -> 700,
4    "highBalanceAlert" -> 1000,
5    "highVolumeAlert" -> 10000,
6    "capacity" -> 100000,
7    "saleVolume" -> 50000,
8    "accounts" -> ["accId1", "accId4"]
9  }
```

Listing 4.7: MAP **example showing a profile table entry**

Listing 4.7 shows a typical definition of such a map which may be compared to a hash map used in various different other programming languages. Those values are accessed based on different pre-defined rules which are explained in Section 4.8.

Ledger: The ledger but also the pending transactions are initialized as empty maps:

```
Ledger := {->}
PendingTransactions := {->},
```

and when a transaction matches the required constraints it is appended to that ledger map. For unconfirmed debit requests a second map called $PendingTransactions$ has been added which contains valid transactions which have not been confirmed by the debtor yet.

After the various checks – which vary a little based to the type of operation – have been passed successfully, the transaction is appended using the following command:

```
Append(Transaction(transfer, client, "debit", now), Ledger)
```

A pending transaction awaiting a correct OTP from a debtor is added to location $PendingTransactions$ as follows:

```
Insert(transfer, creditor, PendingTransactions) Ledger).
```

After a transaction has been confirmed by a valid OTP, it is moved, as mentioned, to the actual ledger. Inside the $PendingTransactions$ map the state of that transaction is changed to $TRANSACTION\_PERFORMED$ but the transaction is not deleted.

4.7.3 Additional Login Layer

The requirements specifications do not define how authentication is handled by the system. Agents in ICEF have predefined names which cannot be changed during runtime. In order to act flexibly and reuse agents, a login layer has been introduced. Thus agents who are actively communicating with the server need to be registered in the $sessionData$ location which is initialized in the $initdata$ function.

Agents in the current prototypical implementation do not login by themselves by providing a username and password but place their address together with a valid member ID in the $sessionData$ MAP as shown here:

```
sessionData := {
  "CreditRequestClient@CreditRequestClient" -> "mbrId1",
  "DebitAcknowledgeClient@DebitAcknowledgeClient" -> "mbrId4",
  "DebitRequestClient@DebitRequestClient" -> "mbrId2"
}
```

The address of an agent is defined by "$agent\_name$@$agent\_name$", thus by a concatenation of its own name with an "@" symbol. That address syntax is a mannerism of the ICEF framework and used during message processing. For example, when looking at the above $sessionData$ set-up, it is possible to determine that on agent $CreditRequestClient$'s user with id $mbrId1$ is logged in. Therefore, all message coming from agent $CreditRequestClient$ with address $CreditRequestClient$@$CreditRequestClient$ are considered as message sent by $mbrId1$ because of $sessionData$-MAP.

The member ID represents an alphanumerical string assigned after an initial registration. For the INTERLACE implementation this means that an entry also needs to exist in location $profileTable$, in addition to $userGroupTable$, at which point the user has been properly registered and is known to the system. If the user wishes to perform a transaction, for the transaction to be valid an entry in location $accountTable$ should be added for that user, in order to assign an actual account to that user and enable the system to work with it.

At runtime two derived functions are used to read the information of $sessionData$. As shown below, a member ID of an "active" user can be determined by calling $activeLogin$. Whether a member is currently logged-in may be found out by obtaining the result of $activeClient$.

```
//get member id by providing a client name using session information
derived activeLogin(client_address) = ...
//usage
login_mbrId := activeLogin("CreditRequestClient@CreditRequestClient")

//get client name by providing member id using session information
derived activeClient(login_mbrId) = ...
//usage
client_address := activeClient("mbrId1")
```

One restriction needs to be mentioned though: a client address can only be associated with one logged-in member at any one time, and vice versa.

4.8 Important Rules and Locations

Next, the emphasis is put on explaining rules and locations not discussed in detail or not mentioned at all in this document yet. All rules and locations in this section can be found in file *server.casim* or *initdata.casim*, which is added to the server part by means of the *include* statement in run.icef.

In the BSL language, the *choose* keyword is needed to read a value for a given key stored inside a *MAP* location. That syntax made the actual code difficult to read in some cases. To simplify this access, a derived function *v* has been introduced for working with *MAP*-type locations:

```
⇒ derived v(amap, key).
```

The parameter *amap* can be any *MAP*-type location and the *key* refers to the entry of interest. Consequently, the corresponding *value* can be read by passing an existing *key*, as illustrated in the following example:

```
mymap := {
  "thekey" => "assignedvalue",
  "another" => "somevalue"
}
valueofkey := v(mymap, "thekey")
```

Finally, if the map or the key is undefined, also the result will be.

The ICEF framework uses a plugin called “CommunicationPlugin” to establish a simple communication system. As mentioned in the previous section, a method called *send* is used to transfer a message between ASIMs. However, the server implementation usually does not call that method directly but, rather, it calls a rule with the same name starting with an upper-case letter. This *Send* rule takes as first parameter a message which contains a list of two strings. The two strings are the message content itself and the message subject. In order to abstract the message type from the user, a derived rule called *AssembleMessage* is provided for creating such a message. The second parameter is the recipient agent’s name, not the ID of a particular member:

```
⇒ derived AssembleMessage(msg, sub)
⇒ rule Send(msg, agent).
```

When passing a message before sending, other parts of the implementation may use the functions *messageContent* and *messageSubject* on the assembled message to get back the content and the subject, respectively. Also *Send* uses these two functions to read content and subject for the actual sending process. In case of error or success, corresponding response messages are generated. *IncorrectOtpFor* is one example which is used inside the *server* implementation to create an appropriate error message based on particular related arguments:

```
⇒ derived messageContent(msg)
⇒ derived messageSubject(msg)
⇒ derived IncorrectOtpFor(otp, amount, creditor).
```

Continuing further on OTPs, there are three important functions that should be mentioned in addition to the requirement definitions. *RandomHex* generates a random hexadecimal number. Next, *HexString* calls *RandomHex* *n* times, where *n* equals the first parameter *length* in the form of a hexadecimal string. This hexadecimal string generation is used for creating OTPs as well as transaction IDs for appending unique transactions to the ledger:

```
⇒ derived RandomHex
⇒ derived HexString(len)
⇒ derived OneTimePassword(strength)
```

Transactions are transferred using a MAP. A credit transfer is shown next:

```
⇒ CREDITREQUEST := {
    "from" -> "accId1",
    "to" -> "accId2",
    "channel" -> "Service",
    "amount"  -> 2000,
    "metadata" -> {"message", "some transfer"}
  }
```

For all attributes used inside these transfer-request maps, derived functions have been created. One function and its usage is shown here:

```
⇒ derived fromAcc(transfer)
// example
fA := fromAcc(CREDITREQUEST)
fA = "accId1" //is true
```

Next, we examine the ledger and the preliminary storage for pending transactions. Both are maps storing transactions. The difference is that the function $PendingTransactions$ is using an OTP key to map to a transaction, whereas a finally persisted transaction is mapped using an transaction ID key. OTPs are created by $OneTimePassword$ and transaction IDs are created by calling $NewTransactionID$.

```
⇒ controlled Ledger: MAP
⇒ controlled PendingTransactions: MAP
```

Rule $Insert$ adds a new transaction (which is awaiting confirmation) to a map (parameter $pendingTransactions$ of $Insert$). The $server$ stores all these unconfirmed requests in a map called $PendingTransactions$, as defined above.

After a valid and existing OTP has been sent for a transfer in $PendingTransactions$, the transfer needs to undergo some more checks; but, when approved, it is appended to the Ledger by executing the rule $Append$.

$Insert$ and $Append$ are listed here together with $Transactions$ and $NewTransactionID$:

```
⇒ rule Append(transfer, ledger)
⇒ rule Insert(transfer, creditor, pendingTransactions)
⇒ derived NewTransactionID(len)
⇒ derived Transaction(transfer, mbr, operation, date)
```

Finally, we described a rule that is part of the requirements but is used in a different way. The requirements call for three functions called $FinalDebitAccountLimitsCheck$, $DebitAccountLimitsCheck$, and $CreditAccountLimitsCheck$. These functions are similar and are integrated in the ICEF implementation. $PreviewCheck$ implements this check and needs the member ID mbr, the $from$ account $fromA$, the to account toA, a $from$ group $fromGroupId$, a $toGroupId$ and, as last parameter, the operation $operation$ as a String. The operation string can be either “credit” or “debit”. This code example shows the rule but also how it is used in the current setup:

```
  ⇒ rule PreviewCheck(mbr, fromA, toA, fromGrpId, toGrpId, operation)
  //example: do check the transfer if it is ok
  error <- PreviewCheck(
    activeLogin(client), //read mbr from message client-address
    fromAcc(transfer), //read from account of transfer
    toAcc(transfer), //read to-account of transfer
    groupOf(ownerOf(fromAcc(transfer))), //get group of from-account owner
    groupOf(ownerOf(toAcc(transfer))), //get group of to-account owner
    "credit" //Operation as string
```

The result of *PreviewCheck* is either *undef*, if all checks are successful, or it contains a textual message indicating the problem.

4.9 Implementation Challenges

When working with the ICEF framework, some things should be noted. Currently, the ICEF framework is an academic tool and not ready for industry use. Additional effort needs to be expended in order to achieve a fully integrated system for an industry-grade software design process or even a continuous integration process. We now outline some of the still-open issues.

4.9.1 Issues

During development the team has been confronted with different issues regarding the ICEF framework but also with the process of moving the specifications to a running ASIM INTERLACE Model:

Unknown line numbers Before transmitting a simulation to the ICEF manager, JavaScript code loads all specified .casim files, preparing them for being sent over the network in the form of a consolidated JSON message. This process, together with the added preliminary *include* syntax, exhibits a problem with respect to error messages. More precisely, if something goes wrong the error messages do not contain the relevant line number(s) of where the error occurred. Thus, in case of any failure it can be very difficult to locate the actual cause.

Error in block Errors inside a block cause the whole block not to be executed for unknown reasons. For example, if a block has n statements and the last one has a problem, all the other $n-1$ statements might not be executed either, even if a sequential block was used. Given that the error messages do not contain correct line numbers, finding a problem with debug output to the console also does not work appropriately in all cases.

Parallel blocks ASM code and consequently also BSL code allows to write so-called parallel blocks. These blocks start with *par*, end with *endpar*, and contain code which is executed in parallel – at least theoretically. The important thing to know here is that the order of execution is not guaranteed. This can lead – and actually led during development – to unpredictable behaviour and needs to be used with caution!

Bugs As the ICEF framework is still at the alpha stage, a couple of bugs occurred which could be fixed. However, further testing iterations are needed to make the framework stable.

Testing Various testing approaches still need to be investigated, because at the moment only the text output of a simulation can be checked to test if the system is running correctly. These testing approaches will be part of the upcoming research.[17]

[17] For example, we are likely to adopt Cucumber https://cucumber.io/ and Gherkin https://docs.cucumber.io/gherkin/, since they are used by the Hyperledger blockchain development community.

Modules The solution of how modules, or how other code snippets, are integrated comes with strings attached, like the inaccurate/missing line numbers mentioned above and duplicated name spaces. Thus, a re-factoring for later use is highly recommended.

From document-based to executable When a system behaviour has been written down in the form of an ASM/ASIM specification, it is necessary to note that a direct translation to ASM/BSL code is not always straightforward. One reason is that the functions and rules are not consistently available for CoreAS(I)M, and a lot of code for “getting it to run” needs to be worked out.

4.9.2 Current Status

The requirements for credit and debit operations have been implemented and are ready for more sophisticated tests. Also the base conditions for including further operations or requests handled by the server can be easily extended.

A test scenario has been built which may be used later for high-level testing, where particular use cases are played through using a predefined set of tests including agents with particular tests.

B2C-operations and also some of the other requests defined like $AccountHistReq$ still need to be taken care of and will be addressed in future work.

Chapter 5

Outlook and Next Steps

Paolo Dini and Eduard Hirsch

This report has provided an update on the business logic requirements of the next-generation mutual credit transactional platform, relative to currently available technology (Chapter 2). In the Appendix, it provides a formalization of the updated requirements. In Chapter 3 it provided a detailed explanation of the configuration and setup of the ICEF executable modelling framework for ASIMs and of its implementation language CoreASIM. Finally, in Chapter 4 it presented a detailed discussion of the CoreASIM implementation of the specification shown in the Appendix.

The business logic requirements of Chapter 2 represent an intermediate step between the current centralized system being used by SARDEX and based on a relational database and the blockchain-based platform INTERLACE is contributing to developing. Such an intermediate step is needed both from the point of view of engineering robustness, since migrating to a new platform is difficult and risky, as well as because the full functionality of the blockchain platform will depend to some extent on the architecture and protocol of the blockchain framework selected.

The implementation of the business logic as an executable CoreASIM model has also had the added benefit to bring into focus the need for a testing framework for the ICEF, which is currently missing. In the short term, this can be provided by manually composed testing scenarios written in Gherkin[18] and Cucumber,[19]since they are used by the Hyperledger blockchain development community. In the long term, it would be helpful to develop a "compiler" from the ASIM specification directly into Gherkin.

The next step for INTERLACE will be to specify, model (D2.2), and implement (D3.2) a blockchain platform that supports at least part of the business logic presented in this report. We have decided to use Hyperledger Fabric as the most suitable framework for the needs of INTERLACE and of the Sardex circuit. In other words, we will develop a permissioned blockchain with a limited number of nodes. However, the great flexibility and customizability of Hyperledger leaves open the possibility of coupling the permissioned blockchain to a public permissionless blockchain at some point in the future. Whereas this is not likely to be needed for mutual credit transactions within a single circuit, it could be useful for supporting scalability features for inter-circuit trade.

[18] https://docs.cucumber.io/gherkin/
[19] https://cucumber.io/

References

References

1. E Börger and A Raschke. *Modeling Companion for Software Practitioners*. New York: Springer-Verlag, 2018.
2. E Börger and R Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. New York: Springer-Verlag, 2003.
3. P Dini, E Börger, E Hirsch, T Heistracher, M Cireddu, L Carboni, and G Littera. *D2.1: Requirements and Architecture Definition*. INTERLACE Deliverable, European Commission, 2017. URL: https://www.interlaceproject.eu/.
4. E Rothstein and D Schreckling. *D4.2: Human-readable, Behaviour-based Interaction ComputingSpecification Language*. BIOMICS Deliverable, European Commission, 2015. URL: http://biomicsproject.eu/file-repository/category/11-public-files-deliverables.
5. E Rothstein Morris and D Schreckling. *D5.2: Execution Framework for Interaction Computing*. BIOMICS deliverable, European Commission, 2016. URL: http://www.biomicsproject.eu.
6. J Smith and R Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

Appendix: Complete Functional Requirements and Business Logic Model (2018)

Egon Börger, Paolo Dini and Luca Carboni

We specify here the refinement of the abstract model defined in Deliverable D2.1 [3] for the Sardex core payment operations Credit, Debit and B2C (in Euro or SRD).[20] The refinement moves towards what is needed for an executable version of the model, though it still stays at the functional requirements level of abstraction. It is based upon the additional information obtained in the meantime on details of various system components, on specific data concerning groups, accounts and transactions, and on the resulting intended definition of permission features. This information is taken from Chapter 2 of this report (D3.1).

Section A.1 describes the refinement of the signature elements that were introduced in D2.1. Section A.2 and Section A.3 refine the model for the basic credit and debit operations considered in D2.1, Section A.4 adds the new operations between companies and consumers, called B2C operations. Section A.5 describes the users' input operations. To simplify the model inspection, in a sub-appendix (Section A.6) we put all rules of the model together in a nutshell (without the explanatory text which is to be found in the preceding sections where the rules are introduced). Unexplained terminology and notation are used with the meanings explained in D2.1.

A.1 Signature Elements

In this section we describe the refined signature that is used for $Credit$, $Debit$ and B2C operations. The complex $transferTypeCheck$ function, defined in D2.1 in abstract terms to specify the $Credit$ and $Debit$ operations, is refined by splitting it into two independent checks (which may be thought of as executed in sequence, as will happen in the implementation). The first check concerns the involved source and target groups and is again called TransferTypeCheck (see Section A.1.1). The second check concerns the types of the source and destination accounts involved and is called AccountConnectivityCheck (see Section A.1.2).

This refinement is essentially a data refinement and concerns mainly

- user groups with their characteristic attributes (called *group profile metadata*) and the constraints on the groups among which transfers are allowed by the system (called *transfer type constraints*), and
- accounts with their charateristic attributes (called *account metadata*) and the constraints on types of accounts between which an operation is allowed by the system (called *account connectivity constraints*).

A.1.1 User groups: profile metadata and transfer type constraints

Groups. Out of the 29 types of users, which appear in the specification of the Sardex system as agents (actors) that interact with the system, following Chapter 2 the refined model considers the following 9 pairwise disjoint dynamic sets, called *groups*, which are characterized by the indicated specific attributes:

$Company$ // set of actors which participate only in B2E and B2B operations

[20] We skip the account history and balance operations explained in Section 3.2 of D2.1, because they seem not to be in the focus anymore.

$Mngr$ // (singleton set of) a distinguished element acting for the Sardex company
$Retail$ // set of actors which participate only in B2C operations
$Full$ // set of actors with both Company and Retail functionality

$Employee$ // set of actors working for a member of $Company$ or $Full$
$Consumer$ // set of individuals which participate only in B2CEur operations
$Consumer\_Verified$ // set of registered consumers with additional B2CSrd operation

$Welcome$ // set of users which have joined but are not yet cleared to start trading[21]
On_Hold // set of actors whose privileges have been suspended[22]

Notational convention. To simplify the exposition, we treat singleton sets, like $Mngr = \{mngr\}$, sometimes as set (here $Mngr$) and sometimes as element (here $mngr$), depending on the context, hoping that this slight abuse of language does not create any ambiguity. In general, we write x for elements of X where X is one of the above 9 groups. To prepare the step towards a natural implementation of the model, we treat $Welcome$ and On_Hold members as potential members of some of the other groups which have however a $Welcome$ or a On_Hold flag set, respectively.

For further reference we define $Group$ as the set of the above nine groups:

$$Group = \{Company, Mngr, Retail, Full, Employee, Consumer, Consumer\_Verified, Welcome, On\_Hold\}$$

Since by the disjointness constraint each user is assumed to belong to exactly one group, there is a function $group(user)$ which yields the group to which $user$ belongs.

Group Profile Metadata. Each group comes with a set of attributes (called metadata) that are characteristic for its members. They include the following five data types which are used in the refined model as 'profile metadata' (besides the usual 'identity metadata' described in Chapter 2, which provide the information on the ID of a group member, its e-mail address, phone numbers, also its legal name, address, gps, fiscal ID (VAT), etc.).

- $Company$, $Retail$ (and therefore also $Full$) and $Welcome$ members (but not the $Mngr$) come with a $capacity$, a location whose value represents the maximum yearly SRD volume the member committed to *selling*, with payment in SRD, when it stipulated its contract with the Sardex company. The date of the stipulation of $capacity$ is recorded in a location $capacityDate$.
  - For a given $account$ (typically in CC, see below Section A.1.2), a derived parametrized location introduced in Chapter 2 as belonging to AccountMetadata, $availableCapacity(account)$, is defined as follows:[23]

    $$availableCapacity(account) = capacity - saleVolume(account)$$

    where the dynamic function $saleVolume(account)$ indicates the current total volume of sales per year[24] using that $account$.

[21] To have joined means having signed the contract to become a member of $Company \cup Retail \cup Full$.
[22] Members of $Retail$, $Company$, $Full$, $Employee$, and $Consumer\_Verified$ can be suspended, not the $mngr$. For $Welcome$ and $Consumer$ the concept does not apply since they are not allowed to transact yet in any case.
[23] The defining equation for $availableCapacity$ holds only for non-$Welcome$ members, which – differently from $Welcome$ members (see Table 4) – have a defined $account$; it holds for $Welcome$ members only once they have been assigned an $account$.
[24] This implies that $availableCapacity(account)$ is implicitly parametrized by the year.

- *Company* (and therefore also *Full*) members and *Mngr* come with a *creditPercent* location whose value represents the percentage of payments accepted by the member in circuit currency (SRD, etc.) for transactions whose value is above 1000 Euro.
- *Company* and *Retail* (and therefore also *Full*) members come with a *euroFee* location.
  - For *Company* its value indicates an $InterTradeEuroFee$ for inter-circuit sales in the non-Euro currency of the circuit (SRD for Sardex, VTX for Venetex, etc).
  - For *Retail* its value indicates a $B2CEuroFee$ for B2C sales in EUR.
  - For *Full* its value is a two-element set $\{InterTradeEuroFee, B2CEuroFee\}$.

  A $B2CEuroFee$ is a function which yields the percentage of the total value of the B2C sale, to be paid by the selling *retailer* to the Sardex company.
  An $InterTradeEuroFee$ can be of two kinds. Either it is a function that simply yields 3% of the intertrade amount, regardless of the networks involved and their members. Or it is a dynamic function $fee(amount, network1, network2)$ which yields a percentage of the trade *amount* that may depend on the networks involved. In the current model, only the buyer has to pay the fee, though it is contemplated for a future extension that also the seller will have to pay a fee.
- Both *Full* and *Retail* members come with two locations: *rewardRate* and *acceptanceRate*. The *rewardRate* value is a function which yields the percentage of reward the member offers in SRD currency to consumers engaged in a B2CEur purchase with the member; similarly, the *acceptanceRate* defines the percent rate of the total value of a consumer purchase at which the member accepts SRD currency.

The *transferTypeCheck* function defined in D2.1 uses a *Match* predicate which expresses constraints on a) the groups of the account owners for the considered operation and on b) the type of the two accounts involved, in addition to c) constraints on the value of a set of meta-data, which in D2.1 were called 'custom fields'. The constraints on the groups involved are refined here by a function of the following type:

$$TT : Operation \times Currency \times Group \to \{G \mid G \subseteq Group\}$$

where $Operation = \{Credit, Debit\}$ and $Currency = \{SRD, EUR\}$. To simplify the exposition, we follow Figure 2.7 and define TT by a case distinction, considering the given pair of the first two arguments, say $op \in \{Credit, Debit\}$ and $cur \in \{SRD, EUR\}$. Formally,

$$TT(op, cur, group) = TT^{op,cur}(group).$$

For each group, say *fromGroup* of buyers, $TT^{op,cur}(fromGroup)$ defines the set *toGroups* of groups of sellers whose members are allowed to receive a transfer (to one of their accounts) from (an account of) a *fromGroup* member via the operation *op* in the currency *cur* (under appropriate constraints we specify below on the amount of the transfer and some metadata of the account involved).[25] We now define the requirements described for these functions.

Transfer Type Constraints for $TT^{Credit,SRD}$. A *Credit* operation in *SRD* currency can be started only by members of one of the following groups (called source group or *fromGroup* of the operation):

- *Company*, and therefore also *Full* and in particular *Mngr*,
- *Employee*,
- *Consumer_Verified*.

For SRD Credit operations the following groups are allowed as target group (read: group of the member receiving the SRD credit, also called *toGroup*[26]):

[25] In D2.1 the names *fromMemberGroup* and *toMemberGroup* were used, see the *Match* predicate definition there.
[26] See Figure 2.7.

- every $Company \cup Full \cup Mngr$ member can trigger an SRD Credit operation to a member of $Company \cup Full \cup Mngr$ or of $Employee$,
- every $Employee$ member can trigger an SRD Credit operation to a member of $Company \cup Retail \cup Full$,
- every $Consumer\_Verified$ member can trigger an SRD Credit operation to a member of $Retail \cup Full$.

This requirement is expressed in the refined ASM model by the following function definition:

$$\begin{aligned} &TT^{Credit,SRD}(Company) \\ &\quad = TT^{Credit,SRD}(Full) \\ &\quad = TT^{Credit,SRD}(Mngr) \\ &\quad = \{Company, Full, Mngr, Employee\} \\ &TT^{Credit,SRD}(Employee) = \{Retail, Company, Full\} \\ &TT^{Credit,SRD}(Consumer\_Verified) = \{Retail, Full\} \end{aligned}$$

Note that this definition, which will be used below for the transfer type check, allows no $Credit$ operation with target group On_Hold. But note that a $Retail$, $Company$ or $Full$ can have its $creditLimit$ set to 0 (by a broker operation we do not model here) which does not prevent that user from receiving credits for sales.

For ease of reference we say that a group member $MayAllowTransferForCreditOpns$ if it is an element of a group where $TT^{Credit,SRD}$ has a defined value:[27]

$$MayAllowTransferForCreditOpns(mbr) \iff mbr \in Company \cup Full \cup Mngr \cup Employee \cup Consumer\_Verified$$

Transfer Type Constraints for $TT^{Debit,SRD}$.

A $Debit$ operation in SRD currency can be started only by members of one of the following groups (called again source group or $fromGroup$ of the operation):

- $Retail$, $Company$, and therefore also $Full$ and in particular $Mngr$,
- $Employee$,
- $Consumer\_Verified$.

For SRD Debit operations the following groups are allowed as target groups (again called $toGroup$[28]):

- Every $Company \cup Full \cup Mngr$ can initiate an SRD Debit operation that draws on a member of $Company \cup Full$ (and the latter could initiate a corresponding credit transfer to the former, for the same effect).
- The $mngr$ can initiate an SRD Debit operation that draws on any member of $Retail$.
- Every member of $Retail \cup Company \cup Full$ can initiate an SRD Debit operation that draws on any $Employee$.

[27] Using the $group(mbr)$ function which denotes the unique group to which mbr belongs, the definition can be equivalently expressed by:

$$MayAllowTransferForCreditOpns(mbr) \iff group(mbr) \in \{Company, Full, Mngr, Employee, Consumer\_Verified\}$$

[28] See Figure 2.7.

- Every member of $Retail \cup Full$ can initiate an SRD Debit operation that draws on any member of $Consumer\_Verified$.
- Every member of $Company \cup Full$ can initiate an SRD Debit operation that draws on $mngr$.

This requirement is expressed in the refined ASM model by the following function definition:

$$
\begin{aligned}
&TT^{Debit,SRD}(Company)\\
&\quad = TT^{Debit,SRD}(Full)\\
&\quad = \{Company, Full, Mngr\}\\
&TT^{Debit,SRD}(Retail) = Mngr\\
&TT^{Debit,SRD}(Employee) = \{Retail, Company, Full\}\\
&TT^{Debit,SRD}(Consumer\_Verified) = \{Retail, Full\}\\
&TT^{Debit,SRD}(Mngr) = \{Company, Full\}
\end{aligned}
$$

Correspondingly we define the domain predicate of $TT^{Debit,SRD}$:

$$
\begin{aligned}
&MayAllowTransferForDebitOpns(mbr) \iff\\
&\quad mbr \in Retail \cup Company \cup Full \cup Employee \cup Consumer\_Verified \cup Mngr
\end{aligned}
$$

Transfer Type Constraints for operations in Euro. In the Sardex system, no $Credit$ operations are allowed in EUR currency, so that no function $TT^{Credit,EUR}$ is needed.[29] A EUR $Debit$ operation can be triggered only by a $Consumer$ or a $Consumer\_Verified$ member and must have as target group $Retail$ or $Full$. This requirement is expressed in the refined ASM model by the following definition:

$$
\begin{aligned}
&T^{Debit,EUR}(Consumer) = T^{Debit,EUR}(Consumer\_Verified)\\
&\quad = \{Retail, Full\}
\end{aligned}
$$

In the present formulation of the model we do not use the function $T^{Debit,EUR}$ because, to simplify the exposition in Section A.4, we describe the B2C operations there directly, not as instance of the DebitTransferReq rules.

Remark on the refinement. With the above definition of functions $TT^{op,cur}$, the transfer type part of the $Match$ predicate in D2.1 can be expressed as follows. By definition, the currency parameter of the $amount$ in question can be computed from the $account$ by the function $cur(account)$ (for accounts see Section A.1.2). Furthermore, given that each member belongs to exactly one group, from the account parameters $from/to$ we also obtain $group(owner(from/to))$ in the following refinement of the group type constraints, namely $owner(from) \in fromGroup(tt)$ and $owner(to) \in toGroup(tt)$:

$$
\begin{aligned}
&\textbf{let } transfer = (op, channel, from, to, amount, metaData)\\
&\quad TTMatch(transfer) \textbf{ iff}\\
&\qquad \textbf{forsome } g \in TT(op, cur(from), group(owner(from))) \quad owner(to) \in g
\end{aligned}
$$

Note that the transfer type check is independent of the $channel$ used for a transfer.[30]

[29] Table 8 includes the function but shows that it is completely **undef**ined.

[30] The model in D2.1 was based on the assumption that channels could play a role for the transfer type check, though the constraints had not been specified. Therefore an abstract channel condition appeared there in the $Match$ predicate in D2.1, which is not present anymore here.

A.1.2 Account types, account metadata and account connectivity constraints.

Account Types. Each $user$ may have a set $Account(user)$ of accounts each of which the $user$ is the $owner(acc)$ of (also denoted $MemberId(acc)$), but at most one account per account type. Each account is in one currency $cur(acc) \in Currency = \{SRD, EUR\}$ and of an $accountType(acc)$, defined for the group to which the $user$ belongs, among the following seven ones that are considered in the refined model. There are two SRD account types:

- CC: the set of standard Sardex credit accounts, in SRD
- Domu: the set of Sardex credit accounts for larger operations, in SRD

There are three types of EUR accounts, which are considered to be of statistical character:

- Income: the set of (statistical) accounts owned by $Retail$ or $Full$ users, which use such an account to record their collection of B2C payments in Euro.
- Prepaid: the set of (statistical) accounts from which the Euro transaction fee for a B2C operation is drawn by $SysAdmin$ on behalf of the $mngr$.[31] Prepaid is owned by $Retail$ or $Full$ members but controlled by $SysAdmin$.
- Bisoo: the set of (statistical) accounts in Euro which are used by $consumer$s to pay into an Income account. Bisoo is owned by $Consumer$ or $Consumer\_Verified$ members but controlled by $SysAdmin$.

There are two special $mngr$ account types:

- Mirror: type of account in circuit currency (SRD, VTX, etc.) used for inter-circuit purchases. There is one mirror account per circuit so that, formally, each Mirror account is implicitly parametrized by a circuit.
- Topup: a (statistical) account in Euro, which is $mngr$-owned and $SysAdmin$-controlled[32] and used to recharge a $retail$er's Prepaid account upon receipt of a (real) payment in Euro (through normal banking channels).

We treat each account type as the set of accounts of that type and say that $accountType(acc)$ is X if $acc \in X$, where X is one of the seven account types above.[33] We also write x for elements of X, where X is one of the above 7 account types. For the singleton set $Topup = \{topup\}$ we use the same notational convention as explained above for $Mngr = \{mngr\}$. For the set of the above 7 account types we write:

$$AccountType = \{CC, Domu, Income, Prepaid, Bisoo, Mirror, Topup\}$$

Initially, accounts are assigned to users satisfying the following constraints on $Account(x)$, depending on the user group X the user x belongs to:[34]

$Account(company) = \{cc(company), domu(company), prepaid(company)\}$[35]

[31] Differently from $Income$ accounts, any $Prepaid$ account is updated not by the $retail$er, for which it keeps track of its fee prepayments, but by the $mngr$. See Figure 2.10 (B2C Use Case 1/2) and Section A.4.2 for details.
[32] See Figure 2.10 (B2C Use Case 1/2).
[33] In D2.1 the names $creditAccount$, $domuAccount$, $feeAccount$ were used instead of CC, $Domu$, $Prepaid$.
[34] The accounts of type $Prepaid$, $Bisoo$ and $Topup$ are not controlled by the corresponding member for which they serve but by the $SysAdmin$istrator, see below.
[35] Please see note after Table 4 for the accounts of $company$.

$Account(retail) = \{cc(retail), prepaid(retail), income(retail)\}$
$Account(full) = \{cc(full), domu(full), prepaid(full), income(full)\}$

$Account(mngr) = \{cc(mngr), topup\} \cup \{mirror(circuit) \mid circuit \in Circuit\}$

$Account(employee) = \{cc(employee)\}$
$Account(consumer) = Account(consumer\_verified) = \{cc(consumer)\}$[36]

Remark. $Welcome$ members initially have no account yet (formally meaning $Account(welcome) = \emptyset$). The initial accounts of a $user \in On\_Hold$ are those accounts the $user$ has at the moment it is placed into On_Hold, but by being placed there the $user$ becomes unable to use these accounts.

Account Metadata. Account metadata are used to formulate the constraints for allowed transfer $amount$s, which involve various account attributes like the $balance$, the $creditLimit$, the $upperLimit$, the $capacity$, etc.

Every account has the locations $owner(acc)$ and $curr(acc)$ (which is also called $unit(acc)$) introduced above. For $Bisoo$ and $Topup$ accounts these are the only locations needed in the refined model, so for them there are no other metadata.

In addition, for each account in $CC \cup Domu \cup Mirror$ there are the following locations, classified as metadata:

- $balance$ (a Real number, modelled as 2-digit decimal)
- $creditLimit$ (a non-negative number)[37]
- $availableBalance$, a derived location required to be non-negative and defined by:

$$availableBalance = balance + creditLimit.$$

In addition, CC and $Mirror$ (but not $Domu$) accounts have the following location:

- $upperLimit$ denoting the upper balance limit, a positive number.

In connection with $balance$ and $saleVolume$, there are three predicates which trigger an alert when the monitored item reaches its (low or high) bound:

$LowBalanceAlert$ **iff** $creditLimit + balance < lowBalanceAlert$
// small amount of money left in the account to buy something with
$HighBalanceAlert$ **iff** $upperLimit - balance < highBalanceAlert$
// small amount of space (measured in money unit) left for further sales
$HighVolumeAlert$ **iff** $capacity - saleVolume < highVolumeAlert$
// almost reached yearly committed sale volume

$Income$ and $Prepaid$ accounts have, besides $owner(acc)$ and $curr(acc)$, also the $balance$ location. Note that a $Prepaid$ account has no $creditLimit$; otherwise stated, its credit limit is always 0.

[36] Contrary to Table 4, in this model no $Bisoo$ account is assigned since for simplicity we treat $consumer$-provided Euro payments directly as input, using a monitored $Received$ predicate. Then no $Bisoo$ account is needed. See Section A.4.1.

[37] Table 9 includes also a $creditLimitDate$ location, indicating the date at which the credit limit was set. We skip such locations because no operation has been specified which uses them, so that there is no rule in the model which involves them.

Account Connectivity Constraints for user-initiated operations. The account connectivity constraints serve to describe the account type conditions which are part of the $Match$ predicate defined in D2.1. To keep the model as simple as possible, we describe the intended effect of the new B2C operations in Section A.4.1 separately, so that we can limit ourselves here to formulate the account connectivity refinement only for user-initiated transactions.

The requirements on the types of the accounts that may be involved in an operation state which account types are allowed as $toAcc$ for an operation of a given $fromAcc$. They can be formalized using an account type check $AccT$ analogous to the transfer type check function TT above.

The essential requirement for user-initiated Credit or Debit operations is two-fold (see Figure 2.12):

- A $Credit$ operation in SRD that starts at a $fromAcc$ in $CC \cup Domu \cup Mirror$ has as allowed $toAcc$ a CC account; in the case of $fromAcc \in CC$, also a $Domu$ or $Mirror$ account is permitted as $toAcc$.
- A $Debit$ operation in SRD can only start from a $fromAcc \in CC$ and must have a $toAcc \in CC$.

One can formalize the above requirements by defining the function

$$AccT\colon Operation \times Currency \times AccountType \to \{Acct \mid Acct \subseteq AccountType\}$$

again by a case distinction $AccT(op, cur, accountType) = AccT^{op,cur}(accountType)$, where:

$$AccT^{Credit,SRD}(CC) = \{CC, Domu, Mirror\}$$
$$AccT^{Credit,SRD}(Domu) = AccT^{Credit,SRD}(Mirror) = \{CC\}$$
$$AccT^{Debit,SRD}(CC) = \{CC\}.$$

As above, for ease of reference we say that an account $MayStartCredit/DebitOpns$ if it is an element of an account type where $AccT^{Credit/Debit,SRD}$ has a defined value:

$MayStartCreditOpns(acct)$ **iff** $acct \in CC \cup Domu \cup Mirror$[38]
$MayStartDebitOpns(acct)$ **iff** $acct \in CC$.[39]

Remark on Bisoo. In Section 2.3.4 also $Debit$ operations in EUR from an $Bisoo$ to an $Income$ account are allowed. Formally this means that in the definition above one includes the clause

$$AccT^{Debit,EUR}(Bisoo) = \{Income\}.$$

Since in rule EurB2C below (Section A.4.1) we provide a direct formalization of the B2C operation, we can do without the $Bisoo$ account and without any $Debit$ operation in EUR from a $Bisoo$ to an $Income$ account.

Remark on system-initiated Credit/Debit operations. In Figure 2.13 the $AccT$ function is defined also for system-initiated operations. $AccT$ with first argument $Credit$ is defined by:

$$AccT^{Credit,SRD}(CC) = \{CC\}$$
$$AccT^{Credit,EUR}(Topup) = \{Prepaid\}$$

[38] For the reason explained in the next remark we exlude $Income$ accounts from this definition.
[39] Remember that we model here only Debit operations in SRD, not in EUR.

$AccT$ with first argument $Debit$ is defined by:

$$AccT^{Debit,EUR}(Prepaid) = \{Topup\}$$

Remark on the refinement. With the above definitions one can redefine the account connectivity constraint part of the $Match$ predicate of D2.1, namely $sourceType(tt) = accountType(from)$ and $destType(tt) = accountType(to)$, as follows:

let $transfer = (op, channel, from, to, amount, metaData)$
$AccTMatch(transfer)$ **iff** $accountType(to) \in AccT(op, cur(from), accountType(from))$

A.2 Credit Operation

Here we define the (refined version from D2.1 of) user-initiated Credit operations in SRD. Similarly to user-initiated Debit operations in SRD, they use either a $Service$ channel (website or mobile phone) or a $PointOfSale$ (POS), the set of standard terminals used by retailers for EUR transactions or to route SRD transactions via an API. This is a mere renaming w.r.t. the terminology used at the time of writing of D2.1:

$Channel = Service \cup POS$
$Service = \{website, mobilePhone\}$

In the refined model there is also a system-initiated Credit operation in Euro which involves $Topup$ and $Prepaid$ accounts. Since it does not involve the various transfer type checks, it can be modelled in a simpler way (see Section A.4).

As in D2.1, a CreditTransferReq initiated by a member mbr (which is permitted only in SRD currency) splits into a double exchange of messages between the member and the Sardex system, called preview and perform step.

CreditTransferReq$((channel, from, to, amount), mbr) =$
CreditPreviewReq$((channel, from, to, amount), mbr)$
CreditPerformReq$((channel, from, to, amount), mbr)$

A user-initiated Credit operation transfers an $amount$ of SRD via a specific $channel$ from one account $from$ to another to under a certain number of conditions:

- if the account owners belong to groups of permitted types, as specified by the transfer type check function $TT^{Credit,SRD}$,
- if the involved accounts $from, to$ are of permitted types, as specified by the account connectivity check function $AccT^{Credit,SRD}$,
- if the $amount$ satisfies the constraints on the various applicable limits, as specified by the $AccountLimitCheck$.

A.2.1 CreditPreviewReq **program**

The various constraints split again in a series of to-be-checked more detailed conditions. To reflect the transactional nature of the CreditTransferReq steps, we describe the execution of each of its two components as one atomic step. Nevertheless, to simplify the verification of the correctness and completeness of the rules, we formulate the entire check as successive

If-Then-Else checks of all its single conditions. As a byproduct we obtain a detailed analysis of the possibilities for error handling procedures one may wish to implement. Thus we define CreditPreviewReq and CreditPerfomReq using an instance of the IfThenElseCascade pattern (see Sub-Appendix A.7).

Furthermore, since it is required that the credit type checks in CreditPreviewReq be repeated for CreditPerformReq, we make the nested credit type check pattern explicit as a machine CreditTypeCheck which contains as parameter a final Completion step. The Completion component parameter can then be instantiated specifically for the two rules: for the preview step it comes up to inform the user that the CreditPerformReq can be triggered, whereas for the perform step it specifies the required AccountLimitCheck.

CreditTypeCheck($transfer, mbr,$ Completion) =
  if $mbr = owner(from)$ **and** $MayAllowTransferForCreditOpns(mbr)$ **then**
    if forsome $g \in TT^{Credit,SRD}(group(mbr))$ $owner(to) \in g$ **then** // check transfer type
      if $MayStartCreditOpns(from)$ **then** // check account connectivity
        if $accountType(to) \in AccT^{Credit,SRD}(accountType(from))$ **then**
          Completion($transfer$)
        else Send($ErrMsg(CreditTargetAccountViolation(transfer)), \mathbf{to}:\ mbr$)
      else Send($ErrMsg(CreditSourceAccountViolation(transfer)), \mathbf{to}:\ mbr$)
    else Send($ErrMsg(CreditTargetGroupViolation(transfer)), \mathbf{to}:\ mbr$)
  else Send($NotAccountOwnerOrCreditSourceGroupViolation(transfer), \mathbf{to}:\ mbr$)
where
  $transfer = (credit, channel, from, to, amount)$

Now we can define CreditPreviewReq as a CreditTypeCheck instance where the Completion parameter is instantiated to what is needed here, namely to PermitPerformReq by Sending a message that the $CreditPerformReq$ can be submitted successfully.

CreditPreviewReq($(channel, from, to, amount), mbr$) =
  let $transfer = (credit, channel, from, to, amount)$
  if $Received(CreditPreviewReq(transfer), \mathbf{from}:\ mbr)$[40] **then**
    CreditTypeCheck($transfer, mbr,$ PermitPerformReq)
    Consume($CreditPreviewReq(transfer)$)
where
  PermitPerformReq($transfer$) =
    Send($YouMayTriggerPerformReq(transfer), \mathbf{to}:\ owner(from)$)

A.2.2 CreditPerformReq **program**

Similarly, the CreditPerformReq rule uses an instance of CreditTypeCheck, where the parameter Completion is instantiated to a rule CreditAccountLimitsCheck. In other words, to execute CreditPerformReq, first the CreditTypeCheck is executed once more, but if it succeeds, to complete the operation, instead of Send($YouMayTriggerPerformReq(transfer), \mathbf{to}:\ owner(from)$), a rule CreditAccountLimitsCheck is called to TryToCompleteCreditOpn. That rule specifies the check of the various constraints on the $amount$ of the Credit operation, i.e. it refines the $balancecheck$ function of D2.1. Like CreditTypeCheck, it is an IfThenElseCascade pattern instance and also comes with a Completion parameter. For CreditPerformReq this parameter is instantiated by a CompleteTransaction component.

[40] Given the account $owner$ function, one could omit here the mbr parameter and write instead $owner(from)$.

CreditPerformReq$((channel, from, to, amount), mbr) =$
  let $transfer = (credit, channel, from, to, amount)$
  if $Received(CreditPerformReq(transfer), \mathbf{from}: mbr)$ **then**
    CreditTypeCheck$(transfer, mbr,$ TryToCompleteCreditOpn$)$
    Consume$(CreditPerformReq(transfer))$
where
  TryToCompleteCreditOpn$(transfer) =$
    CreditAccountLimitsCheck$(transfer,$ CompleteTransaction$(transfer))$

CreditAccountLimitsCheck checks the following data for the given $amount$:

- the $availableBalance$ of the source account $from$ (as already formulated in D2.1), where $from$ must be (and by the account connectivity check is known to be) a member of $CC \cup Domu \cup Mirror$,
- the $upperLimit$ of the target account to (as already formulated in D2.1), where by the account connectivity check to is known to be a member of $CC \cup Mirror \cup Domu$, but by the account metadata definition cannot be an element of $Domu$ (accounts without $upperLimit$ location),
- the $availableCapacity$ of the target account, where by the account metadata definition the target account (which by the account connectivity check is known to be an element of $CC \cup Mirror \cup Domu$) must not be a $Domu$ account (because $Domu$ accounts have no $availableCapacity$ location).

Remark on $creditPercent$**.** The main use of the $creditPercent$ location is for statistical purposes, namely to compare the average EUR volume moved by the SRD volume in a given year. However, this feature is not currently implemented and will not be implemented in the CoreASIM model either.

CreditAccountLimitsCheck$(transfer,$ CompletionStep$) =$
  if $CanBeSpentBy(from, amount)$ **then**
    if $CanBeCashedBy(to, amount)$ **then**
      if $HasSellCapacityFor(amount, to)$ **then**
        CompletionStep
      else Send$(ErrMsg(CapacityViolation(transfer)), \mathbf{to}: owner(from))$
    else Send$(ErrMsg(UpperLimitViolation(transfer)), \mathbf{to}: owner(from))$
  else Send$(ErrMsg(AvailBalanceViolation(transfer)), \mathbf{to}: owner(from))$
where
  $transfer = (credit, channel, from, to, amount)$
  $CanBeSpentBy(from, amount)$ **iff** $availableBalance(from) \geq amount$
  $CanBeCashedBy(to, amount)$ **iff** $to \notin Domu$ **and** $balance(to) + amount \leq upperLimit(to)$
  $HasSellCapacityFor(amount, to)$ **iff** $to \notin Domu$ **and** $amount \leq availableCapacity(to)$[41]

The CompleteTransaction component still remains rather abstract, as in D2.1, until we obtain more information on the $Ledger$ and the used $transaction$ function (which records the information on the $transfer$ that is appended to the $Ledger$, including a time stamp which we describe by a 0-ary system function now). However, by the knowledge of the transfer and account type functions $TT, accountType$ we can refine what in D2.1 was called the transfer type check result $ttResult$, namely the triple consisting of the group the $owner(to)$ belongs to and of the $accountType$ of the source and target accounts.

[41] Here we treat $availableCapacity$ as belonging to AccountMetaData, as shown in Table 9.

CompleteTransaction($transfer$) =
  let $(credit, channel, from, to, amount) = transfer$
  Append($transaction(transfer, ttResult, now), Ledger$)
  Send($Confirmed(transfer)$, **to**: $owner(from)$)
  $saleVolume(to) := saleVolume(to) + amount$[42]
where
  $ttResult = (group(owner(to)), accountType(from), accountType(to))$

Historical remark. At the time of writing the description of the model in D2.1, the understanding was that the (at the time otherwise not further specified) $custFlds$ parameter, which now would be renamed $metadata$, plays a role for the CreditPreviewReq rule. We now know that for the behaviour of this rule only the group and account type properties are relevant so that the parameter can be skipped. Similarly for CreditPerformReq.

Remark on Function Append. The conceptual heart of the Credit or Debit operation is the Append function shown above. Following common practice, we do not hold user balances in the ledger, we simply record the amount of the transactions along with the other essential parameters shown by *appending* this information to the ledger. If a user balance is required at any one point in time, it will need to be calculated in real time when the request is made. As a consequence, for a Credit operation performed by $userA$ as the payment of $amount$ to $userB$, the intuitively obvious double-entry book-keeping operation

$$balance(acct(userA)) = balance(acct(userA)) - amount$$
$$balance(acct(userB)) = balance(acct(userB)) + amount$$

does not appear at all, it is only implicit.

A.3 Debit Operation

As explained in D2.1, user-initiated Debit operations in SRD are executed in 3 phases: in addition to the DebitPreviewReq and the DebitPerformReq steps, where the system and the $creditor$ interact with each other similarly to the interaction for Credit operations, there is an interaction between the system and the $debitor$ where the system asks for an acknowledgement from the $debitor$ before performing the DebitAckReqAnswCompletion (or in case of failure a DebitLateAckReqAnswCompletion) step. Therefore we have:

DebitTransferReq =
  DebitPreviewReq
  DebitPerformReq
  DebitAckReqAnswCompletion
  DebitLateAckReqAnswCompletion

Remark on Debit operations in Euro. In the refined model there are also two Debit operations in Euro. One is system-initiated and involves $Topup$ and $Prepaid$ accounts, the other one is user-initiated and involves $Bisoo$ and $Income$ accounts. Since these two operations do not need the various transfer type checks, they can be modelled in a simpler way than by treating them as instances of DebitPreviewReq (see Section A.4).

[42] We treat the dynamic function $saleVolume$ as belonging to AccountMetaData, in accordance with Table 9.

A.3.1 DebitPreviewReq program (for SRD)

In this section we consider Debit operations in SRD. We treat the Debit operations in EUR separately in Section A.4.

As for Credit operations, the DebitPreviewReq step essentially performs a DebitTypeCheck on the *creditor* and *debitor* groups and on the permitted type of the accounts involved.[43] For a Debit operation TT is applied to $group(debitor)$ and yields the allowed creditor groups, to one of which the *creditor* must belong. By Figure 2.3, the account connectivity check for SRD-Debit operations checks whether the accounts $from, to$ to be used for the intended Debit operation are both of type CC. If the outcome of the two checks is positive, DebitPreviewReq calls as completion a PermitPerformReq component which enables the *creditor* to proceed to the DebitPerformReq phase.

DebitPreviewReq$((channel, from, to, amount), creditor) =$
  let $transfer = (debit, channel, from, to, amount)$
  if $Received(DebitPreviewReq(transfer), \mathbf{from}: \ creditor)$ **then**
    DebitTypeCheck$(transfer, creditor,$ PermitPerformReq$)$
    Consume$(DebitPreviewReq(transfer))$
where
  PermitPerformReq$(transfer) =$
    Send$(YouMayTriggerPerformReq(transfer)$[44]$, \mathbf{to}: \ owner(to)$[45]$)$

The DebitTypeCheck machine follows the same pattern as CreditTypeCheck.

DebitTypeCheck$(transfer, creditor,$ Completion$) =$
  let $(debit, channel, from, to, amount) = transfer$
  let $debitor = owner(from)$
  if $creditor = owner(to)$ **and** $MayAllowTransferForDebitOpns(debitor)$ **then**
    if forsome $g \in TT^{Debit,SRD}(group(debitor)) \ \ creditor \in g$ // check transfer type **then**
      if $MayStartDebitOpns(to)$ **then** // check accont connectivity $to \in CC$
        if $accountType(from) \in AccT^{Debit,SRD}(accountType(to))$ // i.e. $from \in CC$ **then**
          Completion$(transfer)$
        else Send$(ErrMsg(DebitTargetAccountViolation(transfer)), \mathbf{to}: \ creditor)$
      else Send$(ErrMsg(DebitSourceAccountViolation(transfer)), \mathbf{to}: \ creditor)$
    else Send$(ErrMsg(DebitTransferTypeViolation(transfer)), \mathbf{to}: \ creditor)$
  else Send$(ErrMsg(NotAccountOwnerToReceiveDebit(transfer)), \mathbf{to}: \ creditor)$

A.3.2 DebitPerformReq program

To define DebitPerformReq, we reuse the scheme applied for CreditPerformReq, calling once more the DebitTypeCheck component executed already by DebitPreviewReq, but with a new Completion parameter whose role is to trigger a DebitAccountLimitsCheck and – if that check succeeds – a RequestDebitAcknowledgement from the *debitor*.

DebitPerformReq$(transfer, creditor) =$
  if $Received(DebitPerformReq(transfer), \mathbf{from}: \ creditor)$ **then**

[43] Note: the argument *fromGroup* in $TT(fromGroup)$ is the debitor for *both* Credit and Debit operations.
[44] This message, by its parameter *debit*, differs from the message with same name *YouMayTriggerPerformReq* used in the rule CreditPreviewReq where the corresponding parameter is *credit*.
[45] *creditor*, the sender of the request, can be retrieved from *transfer* by the account *owner* function via $creditor = owner(to)$, which is the seller who will receive the transfer of SRD from the buyer $owner(from)$.

DebitTypeCheck($transfer$, $creditor$, TryToCompleteDebitOpn)
Consume($DebitPerformReq(transfer)$)
where
$transfer = (debit, channel, from, to, amount)$
TryToCompleteDebitOpn($transfer$) =
DebitAccountLimitsCheck($transfer$, RequestDebitAck($transfer$))

DebitAccountLimitsCheck is structurally similar to the CreditAccountLimitsCheck (and uses its definitions for the three check predicates), but it has a different parameter to be called in case the check is successful, namely to RequestDebitAcknowledgement from the $debitor$ (see below) before completing the transaction either successfully, by a DebitAckReqAnswCompletion, or in the failure case by a DebitLateAckReqAnswCompletion.

This leads to the following definition. To prevent confusion we use a new name NextStep for the parameter. Note that for privacy reasons, the definition of the content of an $ErrMsg(param)$ (which has to be defined separately) may have to hide some of the information the system knows in case of the given $param$eters.

DebitAccountLimitsCheck($transfer$, NextStep) =
let $(debit, channel, from, to, amount) = transfer$
let $debitor = owner(from), creditor = owner(to)$
if $CanBeSpentBy(cc(debitor), amount)$ **then**
if $CanBeCashedBy(cc(creditor), amount)$ **then**
if $HasSellCapacityFor(amount, cc(creditor))$ **then**
NextStep
else Send($ErrMsg(SellCapacityViolation(transfer))$, **to**: $creditor$
else Send($ErrMsg(UpperLimitViolation(transfer))$, **to**: $creditor$
else
Send($ErrMsg(AvailBalanceViolation(transfer))$, **to**: $debitor$)
Send($ErrMsg(DebitorHasSomeProblemWith(transfer))$, **to**: $creditor$)

The machine RequestDebitAck completes the transaction without further ado if the $amount$ is $Small$ (less than 100), namely by appending it to the $Ledger$ (using the system location for the current system time, denoted now). For every other $amount$, RequestAck creates a $OneTimePassword$ otp (using the current system time, denoted by now), records its birthtime (the beginning of its lifetime), records the otp with the transaction (including the computed transfer type) as a $PendingTransaction$, and sends the otp with an agreement request to the $debitor$. To execute DebitAckReqAnswCompletion a $DebitAckMsg$ must arrive; if such a message does not arrive within the lifetime of otp, DebitLateAckReqAnswCompletion will be executed.

RequestDebitAck($transfer$) =
let $(debit, channel, from, to, amount) = transfer$
let $debitor = owner(from), creditor = owner(to)$
if $Small(amount)$ // case where no acknowledgement from $debitor$ is requested **then**
CompleteTransaction($transfer$)
else
let otp =**new** $(OneTimePassword)$
let $pendingTransact = (otp, transfer)$
$birthTime(otp) := now$ // current system time
Insert($pendingTransact, PendingTransaction$)
$status(pendingTransact) := pending$

Send($ConfirmationReq(pendingTransact)$, **to**: $debitor$)[46]

where

$Small(amount)$ **iff** $amount < 100$

CompleteTransaction($transfer$)[47] =

Append($transaction(transfer, ttResult, now), Ledger$)

Send($Confirmed(transfer)$, **to**: $debitor$)

Send($Confirmed(transfer)$, **to**: $creditor$)

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

$ttResult = (group(creditor), CC, CC)$

A.3.3 DebitAck/RejectCompletion **programs**

Debit completion when debitor answers *ConfirmationReq*

- Case 0. When an $AnswerMsg(pendingTransact)$ arrives from the debitor, but the $pendingTransact$ (the acknowledgement request data together with the otp) does not exist in the receiver's database, an error handling procedure is called.
- Case 1. When, for a pending transaction t, the $ConfirmationReq(t)$ is answered by the $debitor$ too late, an error message informing that the otp expired is sent to the debitor and the creditor and the message is discarded. In this case, the DebitLateAckReqAnswCompletion rule below will delete the pending t together with its otp and update $status(t)$ to $rejected$.
- Case 2. The $ConfirmationReq(t)$ is answered by the $debitor$ in time, i.e. within the $lifetimeForOTPs$ foreseen for one-time passwords, but negatively by a $DebitRejectMsg$. In this case the pending t together with its otp is deleted and the pending transaction made $rejected$.
- Case 3. The $ConfirmationReq(t)$ is answered by the $debitor$ in time and positively. Then the system will CompleteTransaction and update the transaction status from $pending$ to $performed$, but only after a new FinalDebitAccountLimitsCheck has succeeded.

Refining the machine DebitAccountLimitsCheck in this way guarantees that, in case of failure, the Debit operation is rejected. Therefore, in every case the $status$ of the pending transaction is changed to either $performed$ or $rejected$ so that the one-time password can be deleted, preventing a later application of the DebitLateAckReqAnswCompletion rule (which will be applied in case of an $Expired(otp)$).

DebitAckReqAnswCompletion =

if $Received(AnswerMsg(pendingTransact)$, **from**: $debitor)$ **then**

if $pendingTransact \notin PendingTransaction$ // $otp(pendingTransact)$ does not exist **then**

HandleMissingOtpError($pendingTransact$)

else

let $(otp, (debit, channel, from, to, amount)) = pendingTransact$

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

if $Expired(pendingTransact)$ **then**

Send($ErrMsg(ExpiredOtpFor(DebitAck, amount, creditor))$, **to**: $debitor$)

Send($ErrMsg(ExpiredOtpFor(DebitAck, amount, creditor))$, **to**: $creditor$)

else if $IsDebitRejectMsg(AnswerMsg(pendingTransact))$ **then**

Delete($pendingTransact, PendingTransact$)

$status(pendingTransact) := rejected$

Send($RejectionMsg(Debit, amount, debitor)$, **to**: $creditor$)

[46] There is no need to keep the $channel$ parameter because the confirmation request can be sent through any channel, not necessarily the one through which the Debit request arrived, and also the acknowledgement can arrive via any channel.

[47] This machine is structurally the same but differs from the one with the same name used in CreditPerformReq by the debit parameter (instead of credit).

else if $status(pendingTransact) = pending$[48] **then**
FinalDebitAccountLimitsCheck
$(pendingTransact, \text{CompleteTransaction}(pendingTransact))$
Delete$(pendingTransact, PendingTransact)$ // otp 'expires when used'
Consume$(DebitAckMsg(pendingTransact))$
where
$Expired(pendingTransact)$ **iff** $now - birthtime(pendingTransact) > lifetimeForOTPs$
CompleteTransaction$(pendingTransact) =$
CompleteTransaction$(debit, channel, from, to, amount)$[49]
$status(pendingTransact) := performed.$

The FinalDebitAccountLimitsCheck refines the DebitAccountLimitsCheck by inserting into the failure cases a clause which makes the pending Debit transaction rejected and informs the creditor about the reason for rejection. Using the **let** clause in the definition relies on the assumption (which can be checked to be true for the model) that each time the submachine FinalDebitAccountLimitsCheck is called in the program, it is called with the expected correct parameters.

FinalDebitAccountLimitsCheck$(pendingTransact, \text{NextStep}) =$
let $(otp, transfer) = pendingTransact$
let $(debit, channel, from, to, amount) = transfer$
let $debitor = owner(from), creditor = owner(to)$
if $CanBeSpentBy(cc(debitor), amount)$ **then**
if $CanBeCashedBy(cc(creditor), amount)$ **then**
if $HasSellCapacityFor(amount, cc(creditor))$ **then**
NextStep
else RejectTransactionBecauseOf$(SellCapacityViolation, pendingTransact)$
else RejectTransactionBecauseOf$(UpperLimitViolation, pendingTransact)$
else RejectTransactionBecauseOf$(AProblemAtDebitor, pendingTransact)$
where
RejectTransactionBecauseOf$(reason, pendingTransact) =$
Send$(ErrMsg(reason(transfer)), \textbf{to}:\ creditor)$
$status(pendingTransact) := rejected$

Debit rejection upon missing debitor's answer to $ConfirmationReq$
In case the $debitor$ does not confirm the Debit request within the $lifetime$ foreseen for OTPs, the Sardex system will reject the DebitPerformReq (by changing the status of the pending transaction to $rejected$) and inform the $creditor$ about it.

DebitLateAckReqAnswCompletion $=$
if $t \in PendingTransaction$ **and** $Expired(t)$ **then**
let $t = (otp, transfer)$ // NB: otp is unique
let $(debit, channel, from, to, amount) = transfer$
Delete$(t, PendingTransaction)$
if $status(t) = pending$ **then**
$status(t) := rejected$

[48] Otherwise, following the requirements, the message is just discarded, nothing else happens to the pending transaction with its otp. In D2.1 the possibility to send out an error message was considered.

[49] Since the number of parameters in these two uses of CompleteTransaction is different, it may appear that we are using some form of overloading. However, there is only one definition of this function, a couple of pages back (although that definition is for $credit$ and here we are doing a $debit$). Therefore, the use here is just the same function call specified at two different levels of abstraction.

$\text{Send}(RejectMsg(Debit, amount, debitor), \mathbf{to}: creditor)$
$\text{Send}(OtpExpiredMsg(Debit, amount, creditor), \mathbf{to}: debitor)$
where $debitor = owner(from), creditor = owner(to)$

A.4 New B2C operations

The new account types *Income*, *Prepaid*, *Bisoo* and *Topup* serve for three new operations:

- An EurB2C operation triggered by a *consumer* or a *consumer_verified* and executed by a *Retail* (or *Full*) business member when either a *consumer* or a *consumer_verified* purchases some good and pays in Euro. The operation consists in issuing a reward (in SRD) to the costumer and paying a fee (in EUR) to the Sardex company.
- An SrdB2C operation initiated (in the real world) by a *Consumer_Verified* member and executed (electronically as a debit transfer) by a *retail*er (or *full* member). The operation consists in accepting that for a purchase the *Consumer_Verified* member pays the *retail*er (or *full* member) with rewards the *Consumer_Verified* member accumulated in SRD currency.
- A RechargePrepaid operation executed by the *mngr*, triggered by an input received from a *Retail* (or *Full*, or also *Company* due to inter-circuit trade fees) member and declared as *FeePrepayment* for fees to be paid to the Sardex company in future B2C (or inter-circuit) operations the member may perform with its customers.

All these operations concern exchange of money which we model as Send operations with appropriate parameters.

A.4.1 Retail B2C operations (EurB2C and SrdB2C)

In the refined model there are two new operations, of type B2C (Business to Consumer), which are in Euro and SRD currency, respectively. We do not discuss here inter-circuit trade as it is outside the scope of this model.

It seems that these rules are considered to be part of the Sardex system software and not of software which is executed locally on machines of the business member (in *Retail* or *Full*). Therefore we describe the rules as triggered by receiving corresponding messages.

Chapter 2 considers the operations as instances of the general Credit/Debit operations. However, these B2C operations do not involve the transaction and account type checks every Credit/Debit operation has to perform. Therefore we simplify the formulation of the rules as rules tailored to perform the necessary dedicated checks and updates, but also to avoid the other general types checks, which are unnecessary here.

In the current Sardex system, EurB2C operations are triggered by a *retail* or a *full* member as Debit operations from the consumer's Bisoo account to the Income account of the *retail* or *full* member, respectively. Since the pairing of the accounts is hard-wired, to simplify the description of the desired functionality in this model we have chosen to describe such operations directly, without the artificial detour via empty account type and account connectivity checks. Thus, an EurB2C operation is triggered by a *consumer* (whether in *Consumer* or in *Consumer_Verified*) who buys a product at a $retail \in Retail \cup Full$ and pays for it in Euro. The money is recorded in the $income(retail)$ account, a reward as SRD credit is issued to the *customer* and the Euro fee is paid. The consumer from which the *EuroAmount* is *Received* remains anonymous. This means that, until it becomes a *Customer_Verified* member, namely by a registration action that we do not model here, it remains known to the Sardex system only by the number of the *card* issued.

Note: A customer is either a card (‘*consumer*’) or a registered B2C user (‘*consumer_verified*’). There is no need to initialise $bisoo_{cust}$ or $reward_{cust}$. They have already been initialised upstream, before the cards are distributed to the *Retail* and *Full* members.

EurB2C =
  if $Received(EurB2CMsg(EuroAmount, \textbf{from}: \ customer), \textbf{from}: \ retail)$ **and**
      $customer \in Consumer \cup Consumer\_Verified$ **then**
    if $ThereIsEnoughPrepaidFeeFor(EuroAmount, retail)$
      and $ThereAreEnoughCreditsFor(EuroAmount, retail)$ **then**
      ManageEuroPayment($EuroAmount, retail, customer$)
      ManageRewardPayment($EuroAmount, retail, customer$)
      ManageFeePayment($EuroAmount, retail$)
    else
      IssueWarning($NotEnoughFundsInPrepaidOrCCAccounts$)
  Consume($EurB2CMsg(EuroAmount, \textbf{from}: \ customer)$) // consume input
where
  $ThereIsEnoughPrepaidFeeFor(amount, retail)$ **iff** $balance_{retail}(prepaid_{retail}) \geq euroFee_{retail}(amount)$
  $ThereAreEnoughCreditsFor(amount, retail)$ **iff** $availableBalance(cc_{retail}) \geq amount$
  ManageEuroPayment($amount, retail, cust$) =
    $balance(income_{retail}) := balance(income_{retail}) + amount$
    $balance(bisoo_{cust}) := balance(bisoo_{cust}) - amount$[50]
  ManageRewardPayment($amount, retail, cust$) =
    $balance(cc_{retail}) := balance(cc_{retail}) - rewardRate_{retail}(amount)$
    $balance(cc_{cust}) := balance(cc_{cust}) + rewardRate_{retail}(amount)$
  ManageFeePayment($amount, retail$) =
    if $retail \in Retail \cup Full$ **then**
      $balance(prepaid_{retail}) := balance(prepaid_{retail}) - fee(amount)$ // subtract *fee* from $prepaid_{retail}$
      $balance(topup) := balance(topup) + fee(amount)$ // add *fee* to *topup*
  $availableBalance(cc_{retail}) = balance_{retail}(cc_{retail}) + creditLimit(cc_{retail})$
  $euroFee_{retail}(amount) = 0.02 * amount$

NB. By registering, a *consumer* becomes a member of *Consumer_Verified* whereby its ‘identity’ changes from being a *card* to a *customer*, with name, surname, etc. And correspondingly its *acc*ounts turn out to be known now as $acc_{customer}$ together with the associated account access function.

Remark on Bisoo and Income. For simplicity of exposition, we have included the $bisoo_{consumer}$ action (to record, for statistical purposes, the Euro amount of the sale) into the IssueReward rule. In the use case description in Figure 2.10 this action is introduced as a “debit-like” action initiated by a member of *Retail* or *Full* who can access *Bisoo* accounts for payment purposes. The *Bisoo* account does not have a “spending limit”, it is only used for statistical purposes as a way to record what the *consumer* spends in Euros through B2C transactions. Although adding the amount spent each time would achieve this purpose as well, we prefer to subtract each time – resulting in an always-negative balance – out of a desire for consistency with the double-entry book-keeping method at the heart of mutual credit, whereby the *Bisoo* and the *Income* accounts are paired: what is *added to* the latter must be *subtracted from* the former. Note the direct update of $income_{retail}$ in ManageEuroPayment, which in Chapter 2 is introduced as a Debit operation involving $income_{retail}$ and $bisoo_{customer}$.

[50] This is a purely statistical account, see below.

A $retail \in Retail \cup Full$ can also execute a SrdB2C operation which can be triggered by a member of $Consumer\_Verified$. This happens when the $retail$er accepts a purchase the member pays in SRD via its accumulated rewards. It is assumed that when registering (an operation we do not model here), a consumer is turned from an element $card \in Consumer$ into an element of $Consumer\_Verified$, so that formally the account cc_{card} becomes $cc_{consumerVerified}$.

SrdB2C =
  if $Received(SrdB2CMsg(amount, \textbf{from}: \ consumer), \textbf{from}: \ retail)$ **and**
    $consumer \in Consumer\_Verified$ **then**
    PayWithReward$(amount, consumer, retail)$
    Consume$(SrdB2CMsg(amount, \textbf{from}: \ consumer))$
where
  PayWithReward$(amt, consumer, retail) =$
    $balance(cc_{consumer}) := balance(cc_{consumer}) - amt$
    $balance(cc_{retail}) := balance(cc_{retail}) + amt$

PayWithReward is described in Section 2.2.8 as an instance of the standard Credit (via Service channel) or Debit (via POS channel) operation.

A.4.2 Mngr/SysAdmin fee operations:
RechargePrepaid**,** AcceptFee**,** LowPrepaymentAlert

In the refined model there are three new $SysAdmin$ operations.

For each $retail \in Retail \cup Full$ user, $SysAdmin$ manages the account $prepaid_{retail} \in Account(retail)$ (which is owned by $retail$ but controlled by $SysAdmin$). This account serves a double purpose: a) to keep track of the fee prepayments made by the $retail$er, in Euro currency, b) to keep track of the fee 'consumed' each time the $retail$er performs a EurB2C transaction.

To control the fee prepayments, when a $retail$er pays an $amount$ of Euros as fee prepayment into the $mngr$'s bank account, using any of the standard payment systems, that $amount$ is added to the $retail$er's $prepaid_{retail}$ account. We describe this in the RechargePrepaid rule directly, avoiding the detour via a Credit operation performed by the $mngr$ from its auxiliary $topup$ account. This is further justified by the fact that the management of these accounts is not by $mngr$ but, rather, by $SysAdmin$.

For statistical purposes, the fee consumption for each B2C operation performed by a retailer is traced by $SysAdmin$ by making the $retail$er 'pay the fee into the $topup$ account'. To do this, $SysAdmin$ moves the amount of Euro which represents the fee from $prepaid_{retail}$ into $topup$. We describe this in the ManageFeePayment rule below directly, avoiding the detour via a Debit operation performed by $SysAdmin$ to $mngr$'s $topup$ account.

Correspondingly, upon the receipt of a fee prepayment by the $retail$er through normal banking channels, the prepaid amount is detracted from $topup$, consistently with double-entry book-keeping.

In addition, before $prepaid_{retail}$ reaches zero, a $lowBalanceAlert$ is sent to the $retail$er.

RechargePrepaid =
  if $Received(EuroFeePrepaymentMsg(amount, \textbf{from}: \ retail))$ **and** $retail \in Retail \cup Full$ **then**
    $balance(topup) := balance(topup) - amount$ // subtract $amount$ from $topup$
    $balance(prepaid_{retail}) := balance(prepaid_{retail}) + amount$

// add $amount$ to $prepaid_{retail}$
Consume$((amount, FeePrepayment), \textbf{from}: \ retail)$ // consume input

When a $retail$er has to pay the fee for a EurB2C operation it performs with a customer, $SysAdmin$ is triggered to ManageFeePayment when it receives the corresponding $B2CEuroFeeMsg$ from the $retail$er. Following the Euro fee handling scheme via the $topup$ account described above, $SysAdmin$ must add the received fee to $topup$ and subtract it from the $retail$er's $Prepaid$ account.

LowPrepaymentAlert =
forall $retail \in Retail \cup Full$
if $CloseToZero(balance(prepaid_{retail}))$ **then**
Send$(PrepaymentAlertMsg(lowPrepaidBalance), \textbf{to}: \ retail)$

A.5 User Operations

Users can Send requests which appear as input for the INTERLACE network server. Whether Send$(CreditPreviewReq(transfer))$ or Send$(DebitPreviewReq(transfer))$ is invoked depends only on the $transfer$ parameter, which the user supplies by filling in the corresponding fields on the screen. The same holds, *mutatis mutandis*, for Send$(AccountHistReq(histParams))$ and Send$(BalanceReq(acc))$. The functionality is clear so that we do not model further this editing process.

For Credit/Debit Perform requests, the only relevant additional constraint is that they can be sent only after an OK message for the corresponding Preview request has been received. We use a function $kind$ to extract from a $transfer$ parameter its $credit$ or $debit$ component, respectively.[51]

if $Received(YouMayTriggerPerformReq(transfer), \textbf{from}: \ sardex)$ **then**
if $kind(transfer) = credit$ **then**
Send$(CreditPerformReq(transfer), \textbf{to}: \ sardex)$
if $kind(transfer) = debit$ **then**
Send$(DebitPerformReq(transfer), \textbf{to}: \ sardex)$
Consume$(YouMayTriggerPerformReq(transfer))$

In case of a Debit operation a debitor has to confirm a received debit request by Sending a $DebitAckMsg$; otherwise a $DebitRejectMsg$ is sent to the INTERLACE network server.

if $Received(ConfirmationReq(otp, transfer), \textbf{from}: \ sardex)$ **then**
let $(debit, channel, from, to, amount) = transfer$
if $Agreed(amount, owner(to), otp)$ **then**
Send$(DebitAckMsg(otp, transfer), \textbf{to}: \ sardex)$
else Send$(DebitRejectMsg(otp, transfer), \textbf{to}: \ sardex)$
Consume$(ConfirmationReq(otp, transfer))$

A.6 Sub-Appendix 1: Sardex Business Logic in a Nutshell

We assume that both the Credit/Debit/B2C and the $mngr$ operations are executed by the Sardex system with the following program SardexModel. Obviously these rules could be split and assigned to different agents, e.g. the last two ones to the $SysAdmin$ and the first two to an independent agent (who by those rules reacts to triggers by the users, but the rules themselves are not under user control).

[51] In the following ASMs the keyword 'sardex' stands for 'INTERLACE network server'.

SardexOps =
  CreditTransferReq
  DebitTransferReq
  EurB2C
  SrdB2C
  MngrOps
where
  CreditTransferReq =
    CreditPreviewReq
    CreditPerformReq
  DebitTransferReq =
    DebitPreviewReq
    DebitPerformReq
    DebitAckReqAnswCompletion
    DebitLateAckReqAnswCompletion
  MngrOps =
    RechargePrepaid
    ManageFeePayment
    LowPaymentAlert

A.6.1 The Credit operation components

Both CreditPreviewReq and CreditPerformReq rules use the CreditTypeCheck component defined below.

CreditPreviewReq =
  let $transfer = (credit, channel, from, to, amount)$
    if $Received(CreditPreviewReq(transfer), \textbf{from}:\ mbr)$ **then**
      CreditTypeCheck$(transfer, mbr,$ PermitPerformReq$)$
      Consume$(CreditPreviewReq(transfer))$
where
  PermitPerformReq$(transfer) =$
    Send$(YouMayTriggerPerformReq(transfer), \textbf{to}:\ owner(from))$

CreditPerformReq =
  let $transfer = (credit, channel, from, to, amount)$
  if $Received(CreditPerformReq(transfer), \textbf{from}:\ mbr)$ **then**
    CreditTypeCheck$(transfer, mbr,$ TryToCompleteCreditOpn$)$
    Consume$(CreditPerformReq(transfer))$
where
  TryToCompleteCreditOpn$(transfer) =$
    CreditAccountLimitsCheck$(transfer,$ CompleteTransaction$(transfer))$

Credit check subcomponents for account types and account limits

CreditTypeCheck$(transfer, mbr,$ Completion$) =$
  if $mbr = owner(from)$ **and** $MayAllowTransferForCreditOpns(mbr)$ **then**
    if forsome $g \in TT^{Credit,SRD}(group(mbr))\ \ owner(to) \in g$ // check transfer type **then**
      if $MayStartCreditOpns(from)$ // check account connectivity **then**
        if $accountType(to) \in AccT^{Credit,SRD}(accountType(from))$ **then**
          Completion$(transfer)$
        else Send$(ErrMsg(CreditTargetAccountViolation(transfer)), \textbf{to}:\ mbr)$

else Send($ErrMsg(CreditSourceAccountViolation(transfer))$, **to**: mbr)
else Send($ErrMsg(CreditTargetGroupViolation(transfer))$, **to**: mbr)
else Send($NotAccountOwnerOrCreditSourceGroupViolation(transfer)$, **to**: mbr)
where
$transfer = (credit, channel, from, to, amount)$

CreditAccountLimitsCheck($transfer$, CompletionStep) =
if $CanBeSpentBy(from, amount)$ **then**
if $CanBeCashedBy(to, amount)$ **then**
if $HasSellCapacityFor(amount, to)$ **then**
CompletionStep
else Send($ErrMsg(CapacityViolation(transfer))$, **to**: $owner(from)$)
else Send($ErrMsg(UpperLimitViolation(transfer))$, **to**: $owner(from)$)
else Send($ErrMsg(AvailBalanceViolation(transfer))$, **to**: $owner(from)$)
where
$transfer = (credit, channel, from, to, amount)$
$CanBeSpentBy(from, amount)$ **iff** $availableBalance(from) \geq amount$
$CanBeCashedBy(to, amount)$ **iff** $to \notin Domu$ **and** $balance(to) + amount \leq upperLimit(to)$
$HasSellCapacityFor(amount, to)$ **iff** $to \notin Domu$ **and** $amount \leq availableCapacity(to)$

Credit transaction completion subcomponent

CompleteTransaction($transfer$) =
let $(credit, channel, from, to, amount) = transfer$
Append($transaction(transfer, ttResult, now), Ledger$)
Send($Confirmed(transfer)$, **to**: $owner(from)$)
$saleVolume(to) := saleVolume(to) + amount$[52]
where
$ttResult = (group(owner(to)), accountType(from), accountType(to))$

A.6.2 The Debit operation components

DebitPreviewReq =
let $transfer = (debit, channel, from, to, amount)$
if $Received(DebitPreviewReq(transfer)$, **from**: $creditor)$ **then**
DebitTypeCheck($transfer, creditor$, PermitPerformReq)
Consume($DebitPreviewReq(transfer)$)
where
PermitPerformReq($transfer$) =
Send($YouMayTriggerPerformReq(transfer)$, **to**: $owner(to)$)

DebitPerformReq =
let $transfer = (debit, channel, from, to, amount)$
if $Received(DebitPerformReq(transfer)$, **from**: $creditor)$ **then**
DebitTypeCheck($transfer, creditor$, TryToCompleteDebitOpn)
Consume($DebitPerformReq(transfer)$)
where
TryToCompleteDebitOpn($transfer$) =
DebitAccountLimitsCheck($transfer$, RequestDebitAck($transfer$))

Debit check subcomponents for account types and account limits

[52] We treat the dynamic function $saleVolume$ as belonging to AccountMetaData, as shown in Table 9.

DebitTypeCheck($transfer$, $creditor$, Completion) =
  let ($debit$, $channel$, $from$, to, $amount$) = $transfer$
  let $debitor$ = $owner(from)$
  if $creditor = owner(to)$ **and** $MayAllowTransferForDebitOpns(debitor)$ **then**
    if forsome $g \in TT^{Debit,SRD}(group(debitor))$ $creditor \in g$ **then** // check transfer type
      if $MayStartDebitOpns(to)$ // check accont connectivity $to \in CC$ **then**
        if $accountType(from) \in AccT^{Debit,SRD}(accountType(to))$ **then** // i.e. $from \in CC$
          Completion($transfer$)
        else Send($ErrMsg(DebitTargetAccountViolation(transfer))$, **to**: $creditor$)
      else Send($ErrMsg(DebitSourceAccountViolation(transfer))$, **to**: $creditor$)
    else Send($ErrMsg(DebitTransferTypeViolation(transfer))$, **to**: $creditor$)
  else Send($ErrMsg(NotAccountOwnerToReceiveDebit(transfer))$, **to**: $creditor$)

DebitAccountLimitsCheck($transfer$, NextStep) =
  let ($debit$, $channel$, $from$, to, $amount$) = $transfer$
  let $debitor = owner(from).creditor = owner(to)$
  if $CanBeSpentBy(cc(debitor), amount)$ **then**
    if $CanBeCashedBy(cc(creditor), amount)$ **then**
      if $HasSellCapacityFor(amount, cc(creditor))$ **then**
        NextStep
      else Send($ErrMsg(SellCapacityViolation(transfer))$, **to**: $creditor$
    else Send($ErrMsg(UpperLimitViolation(transfer))$, **to**: $creditor$
  else
    Send($ErrMsg(AvailBalanceViolation(transfer))$, **to**: $debitor$)
    Send($ErrMsg(DebitorHasSomeProblemWith(transfer))$, **to**: $creditor$)

FinalDebitAccountLimitsCheck($pendingTransact$, NextStep) =
  let (otp, $transfer$) = $pendingTransact$
  let ($debit$, $channel$, $from$, to, $amount$) = $transfer$
  let $debitor = owner(from)$, $creditor = owner(to)$
    if $CanBeSpentBy(cc(debitor), amount)$ **then**
      if $CanBeCashedBy(cc(creditor), amount)$ **then**
        if $HasSellCapacityFor(amount, cc(creditor))$ **then**
          NextStep
        else RejectTransactionBecauseOf($SellCapacityViolation$, $pendingTransact$)
      else RejectTransactionBecauseOf($UpperLimitViolation$, $pendingTransact$)
    else RejectTransactionBecauseOf($AProblemAtDebitor$, $pendingTransact$)
where
  RejectTransactionBecauseOf($reason$, $pendingTransact$) =
    Send($ErrMsg(reason(transfer))$, **to**: $creditor$)
    $status(pendingTransact) := rejected$

Debit acknowledgement subcomponent

RequestDebitAck($transfer$) =
  let ($debit$, $channel$, $from$, to, $amount$) = $transfer$
  let $debitor = owner(from)$, $creditor = owner(to)$
  if $Small(amount)$ **then** // case where no acknowledgement from $debitor$ is requested
    CompleteTransaction($transfer$)
  else
    let otp =**new** ($OneTimePassword$)
    let $pendingTransact = (otp, transfer)$
      $birthTime(otp) := now$ // current system time

Insert($pendingTransact, PendingTransaction$)
$status(pendingTransact) := pending$
Send($ConfirmationReq(pendingTransact)$, **to**: $debitor$)
where
$Small(amount)$ **iff** $amount < 100$
CompleteTransaction($transfer$) =
Append($transaction(transfer, ttResult, now), Ledger$)
Send($Confirmed(transfer)$, **to**: $debitor$)
Send($Confirmed(transfer)$, **to**: $creditor$)
$saleVolume(creditor) := saleVolume(creditor) + amount$
$ttResult = (group(creditor), CC, CC)$

Debit completion components (accept/reject transaction)

DebitAckReqAnswCompletion =
if $Received(AnswerMsg(pendingTransact)$, **from**: $debitor)$ **then**
if $pendingTransact \notin PendingTransaction$ **then** // $otp(pendingTransact)$ does not exist
HandleMissingOtpError($pendingTransact$)
else
let $(otp, (debit, channel, from, to, amount)) = pendingTransact$
let $debitor = owner(from), creditor = owner(to)$
if $Expired(pendingTransact)$ **then**
Send($ErrMsg(ExpiredOtpFor(DebitAck, amount, creditor))$, **to**: $debitor$)
Send($ErrMsg(ExpiredOtpFor(DebitAck, amount, creditor))$, **to**: $creditor$)
else if $IsDebitRejectMsg(AnswerMsg(pendingTransact))$ **then**
Delete($pendingTransact, PendingTransact$)
$status(pendingTransact) := rejected$
Send($RejectionMsg(Debit, amount, debitor)$, **to**: $creditor$)
else if $status(pendingTransact) = pending$[53] **then**
FinalDebitAccountLimitsCheck
($pendingTransact$, CompleteTransaction($pendingTransact$))
Delete($pendingTransact, PendingTransact$) // otp 'expires when used'
Consume($DebitAckMsg(pendingTransact)$)
where
$Expired(pendingTransact)$ **iff** $now - birthtime(pendingTransact) > lifetimeForOTPs$
CompleteTransaction($pendingTransact$) =
CompleteTransaction($debit, channel, from, to, amount$)
$status(pendingTransact) := performed$

DebitLateAckReqAnswCompletion =
if $t \in PendingTransaction$ **and** $Expired(t)$ **then**
let $t = (otp, transfer)$ // NB. otp is unique
let $(debit, channel, from, to, amount) = transfer$
Delete($t, PendingTransaction$)
if $status(t) = pending$ **then**
$status(t) := rejected$
Send($RejectMsg(Debit, amount, debitor)$, **to**: $creditor$)
Send($OtpExpiredMsg(Debit, amount, creditor)$, **to**: $debitor$)
where $debitor = owner(from), creditor = owner(to)$

[53] Otherwise, following the requirements, the message is just discarded, nothing else happens to the pending transaction with its otp. In D2.1 the possibility to send out an error message was considered.

A.6.3 The B2C operations

EurB2C =
  if $Received(EurB2CMsg(EuroAmount, \textbf{from}:\ customer), \textbf{from}:\ retail)$ **and**
      $customer \in Consumer \cup Consumer\_Verified$ **then**
    if $ThereIsEnoughPrepaidFeeFor(EuroAmount, retail)$
      and $ThereAreEnoughCreditsFor(EuroAmount, retail)$ **then**
      ManageEuroPayment($EuroAmount, retail, customer$)
      ManageRewardPayment($EuroAmount, retail, customer$)
      ManageFeePayment($EuroAmount, retail$)
    else
      IssueWarning($NotEnoughFundsInPrepaidOrCCAccounts$)
  Consume($EurB2CMsg(EuroAmount, \textbf{from}:\ customer)$) // consume input

SrdB2C =
  if $Received(SrdB2CMsg(amount, \textbf{from}:\ consumer), \textbf{from}:\ retail)$ **and**
      $consumer \in Consumer\_Verified$ **then**
    PayWithReward($amount, consumer, retail$)
    Consume($SrdB2CMsg(amount, \textbf{from}:\ consumer)$)
where
  PayWithReward($amt, consumer, retail$) =
    $balance(cc_{consumer}) := balance(cc_{consumer}) - amt$
    $balance(cc_{retail}) := balance(cc_{retail}) + amt$

The B2C operation macros

$ThereIsEnoughPrepaidFeeFor(amount, retail)$ **iff** $balance_{retail}(prepaid_{retail}) \geq euroFee_{retail}(amount)$
$ThereAreEnoughCreditsFor(amount, retail)$ **iff** $availableBalance(cc_{retail}) \geq amount$
ManageEuroPayment($amount, retail, cust$) =
  $balance(income_{retail}) := balance(income_{retail}) + amount$
  $balance(bisoo_{cust}) := balance(bisoo_{cust}) - amount$
ManageRewardPayment($amount, retail, cust$) =
  $balance(cc_{retail}) := balance(cc_{retail}) - rewardRate_{retail}(amount)$
  $balance(cc_{cust}) := balance(cc_{cust}) + rewardRate_{retail}(amount)$
ManageFeePayment($amount, retail$) =
  if $retail \in Retail \cup Full$ **then**
    $balance(prepaid_{retail}) := balance(prepaid_{retail}) - fee(amount)$ // subtract fee from $prepaid_{retail}$
    $balance(topup) := balance(topup) + fee(amount)$ // add fee to $topup$
$availableBalance(cc_{retail}) = balance_{retail}(cc_{retail}) + creditLimit(cc_{retail})$
$euroFee_{retail}(amount) = 0.02 * amount$

A.6.4 The Manager/SysAdmin Operation Components

RechargePrepaid =
  if $Received(EuroFeePrepaymentMsg(amount, \textbf{from}:\ retail))$ **and** $retail \in Retail \cup Full$ **then**
    $balance(topup) := balance(topup) - amount$ // subtract $amount$ from $topup$
    $balance(prepaid_{retail}) := balance(prepaid_{retail}) + amount$
      // add $amount$ to $prepaid_{retail}$
    Consume($(amount, FeePrepayment), \textbf{from}:\ retail$) // consume input

LowPrepaymentAlert =
  forall $retail \in Retail \cup Full$
    if $CloseToZero(balance(prepaid_{retail}))$ **then**
      Send($PrepaymentAlertMsg(lowPrepaidBalance), \textbf{to}:\ retail$)

A.7 Sub-Appendix 2: IfThenElseCascade Pattern

The pattern, used for CreditPreviewReq and CreditPerformReq, is obtained by an iteration of the following machine IfThenElse:

IfThenElse($Cond, M, N$) =
  if $Cond$ **then**
    M
  else N

This machine is applied to conditions $Cond_i$ and machines Yes_i and No_i where Yes_i is again an IfThenElse:

$$\text{Yes}_i = \text{IfThenElse}(Cond_{i+1}, \text{Yes}_{i+1}, \text{No}_{i+1})$$

Given a family $IFE = (IFE_i)_{1 \leq i \leq n+1}$ of conditions $Cond_i$ with ASMs Yes_i and No_i, the pattern machine IfThenElseCascade(IFE) can be defined recursively as follows, starting for $n = 1$ with IfThenElse($Cond_1, \text{Yes}_1, \text{No}_1$):

IfThenElseCascade($(IFE_i)_{1 \leq i \leq n+1}$) =
  IfThenElse($Cond_1$, IfThenElseCascade$(IFE_i)_{2 \leq i \leq n+1}, \text{No}_1$)