Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 235 (2009) 71–105

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Specification and Analysis of SOC Systems Using COWS: A Finance Case Study[1]

Federico Banti[2] Alessandro Lapadula[3] Rosario Pugliese[4]
Francesco Tiezzi[5]

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
50134 Firenze, Italy

Abstract

Service-oriented computing, an emerging paradigm for distributed computing based on the use of services, is calling for the development of tools and techniques to build safe and trustworthy systems, and to analyse their behaviour. Therefore many researchers have proposed to use process calculi, a cornerstone of current foundational research on specification and analysis of concurrent and distributed systems.
We illustrate this approach by focussing on COWS, a process calculus expressly designed for specifying and combining services, while modelling their dynamic behaviour. We present the calculus and one of the analysis techniques it enables, that is based on the temporal logic SocL and the associated model checker CMC. We demonstrate applicability of our tools by means of a large case study, from the financial domain, which is first specified in COWS, and then analysed by using SocL to express many significant properties and CMC to verify them.

Keywords: Service-oriented computing, service orchestration, process calculi, logics and model checking.

1 Introduction

In recent years, the increasing success of e-business, e-learning, e-government, and other similar emerging models, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for *service-oriented computing* (SOC) supporting automated use. SOC is a modern attempt, based on minimizing the dependencies among interacting components, to cope with old problems related to information exchange and software integration. This new, evolutionary, paradigm advocates the use of 'services', namely a sort of loosely coupled,

[1] This work has been supported by the EU project SENSORIA, IST-2 005-016004.
[2] Email: fbanti@gmail.com
[3] Email: lapadula@dsi.unifi.it
[4] Email: pugliese@dsi.unifi.it
[5] Email: tiezzi@dsi.unifi.it

doi:10.1016/j.entcs.2009.03.006

platform-independent, reusable software components, as the basic blocks for building interoperable and collaborative applications.

SOC systems deliver application functionalities to either end-user applications or other services. This general paradigm has many possible instantiations, Web Services (WSs) being probably the most successful and illustrative one. WSs are software components deployed on the World Wide Web, that can be discovered and exploited both by human clients and other services. A key factor for the success of WSs is the fact that their underlying architecture is the Web, that is nowadays a widespread and extensively used platform suitable to connect different companies and customers. Indeed, independently developed applications can be exposed as services and can be interconnected by exploiting the Web infrastructure with related standards, e.g. HTTP, XML, SOAP, WSDL and UDDI. This way, proprietary interfaces and data formats are replaced with a standard Web-messaging infrastructure based on XML technologies, thus facilitating automated integration of newly built and legacy applications, both within and across enterprise boundaries. Several international companies have invested a considerable amount of resources on the web service approach, and the SOC paradigm in general, which are nowadays supported by a plethora of new languages and technologies.

The new paradigm is calling for the development of tools and techniques to build safe and trustworthy SOC systems, to analyse their behaviour, and to demonstrate their conformance to given specifications. In the concurrency theory community, many researchers believe that *process calculi*, a cornerstone of current research on specification and analysis of concurrent and distributed systems, might play a central role in laying rigorous methodological foundations for specification and validation of SOC applications. Indeed, due to their algebraic nature, process calculi convey in a distilled form the compositional programming style of SOC. Thus, for example, many well-known problems related to services composition (e.g., messages not received, race conditions, deadlocks) could be investigated through an adequate and sufficiently expressive process calculus. Furthermore, process calculi enjoy a rich repertoire of elegant meta-theories, proof techniques and analytical tools that can be likely tailored to the needs of SOC. In fact, it has been already argued that observational semantics, type systems, and modal and temporal logics provide adequate tools to address topics relevant to SOC (see e.g. [19,25]). This 'proof technology' can eventually pave the way for the development of automatic property validation tools.

In this paper we illustrate this approach by focussing on COWS (*Calculus for Orchestration of Web Services*) [16], one of the many process calculi for SOC that have then been proposed in the literature (among which we want to mention [8,7,15,13,9,14,5,6,26]). COWS is a linguistic formalism for specifying and combining service-oriented systems, while modelling their dynamic behaviour. Although all the above mentioned formalisms are inspired to well-known process calculi, the design of COWS is also inspired to WS-BPEL [21], the OASIS standard language for *orchestration* of web services. This peculiarity allows to naturally define translations from COWS specifications into WS-BPEL code and vice versa (see e.g.

[18]), opening the possibility to develop tools for (semi-)automatic WS-BPEL code generation and (semi-)automatic program verification by COWS based analytical tools. We will introduce the calculus in Section 3 by first presenting its syntax and then informally presenting its operational semantics. In Section 4, we demonstrate specification style and expressiveness of COWS by means of the specification of a significant case study, namely a financial scenario studied within the EU project Sensoria [24] and presented in Section 2.

Since the definition of the calculus, a number of methods and tools have been devised to analyse COWS specifications, such as the type system to check confidentiality properties of [17], the stochastic extension to enable quantitative reasoning on service behaviours of [22], the static analysis to establish properties of the flow of information between services of [3], the bisimulation-based observational semantics to check interchangeability of services and conformance against service specifications of [23], and the logic and model checker to express and check functional properties of services of [11]. We illustrate this latter tools in Section 5, where we present the branching-time temporal logic SocL, specifically designed to capture peculiar aspects of services, and show many significant properties of the finance case study specified with the logic and verified by means of CMC [1], the on-the-fly model checker for SocL.

Section 6 concludes the paper by also reviewing related work and touching upon directions for future works. The Appendix reports the complete specification of the scenario, together with the SocL formulation of the properties we have checked, written in the syntax of CMC.

2 A finance case study

Hereafter we present a large case study from the financial domain which is currently investigated within the EU project Sensoria [24] on software engineering techniques for service-oriented applications. We start by providing an informal specification of the scenario, then a more detailed UML-based one (Section 2.1) and finally a formal COWS specification (Section 4). We also point out (Section 2.2) some desirable properties of the application that are later on formalized by SocL and checked against the COWS specification with the support of CMC (Section 4).

The considered service is a credit (web) portal that provides the customer companies with the possibility to ask for a loan to a bank, and then orchestrates the necessary steps for processing the credit request, involving a preliminary evaluation by an employee, and subsequent evaluation by a supervisor before a contract proposal is sent to the customer.

Initially, the customer logins to the portal by providing his username and password, then he selects service Credit Request. In the next step, the customer uploads the necessary data for his request. More specifically, he firstly provides the desired credit amount, then the securities of the loan and his balance. The service checks the balance by resorting on a validation service and, in case the balance is not validated, it asks the user to provide it again.

When the request is completely filled by the customer, the service puts it in the list of tasks that the bank employees must accomplish. Then, an employee withdraws the request from the task list and fills his evaluation about it. The evaluation has a private part (only available for the bank purposes) and a public one which is available to the customer. The private evaluation consists of the rating of the customer company and some additional information. The public evaluation consists of the decision about the request and, in case, the bank offer or the motivation for the rejection. The decision can be to reject the request, to accept it or to ask the customer for updating the request. According to the decision, the request processing may proceed in three different ways.

- If the request is rejected, the customer receives a message containing the response and its motivations, and then the process terminates.
- If an update is asked, a message is sent to the customer with the update request and its motivations. The customer may then decide to update the securities and/or the credit amount or refuse to update. In the latter case, the process terminates, while in the former one the updated request is processed again as from above.
- If the request is accepted, the service queues the contract (i.e. the request and the evaluation) in the list of tasks that the bank supervisors must accomplish. Then, a supervisor withdraws the contract from the task list and may update the public evaluation with its own decision. Again, the decision may be to ask for an update, to reject the request, or to accept it. The first two cases are processed as above, regarding the last one, the customer receives the offer and may answer positively or negatively. In both cases the process terminates. If the answer is positive, the process terminates positively and the contract is sent to an external service dealing with contracts for which customer and the bank have found an agreement.

At any moment the customer may require to abort the process. If this happens, the process terminates and, in case, the request is removed from task lists. As we will see later on, this last property requires execution of *compensation activities* to semantically rollback the action of queueing the request in the task list. This prevents an employee or a supervisor from examining an already aborted request.

2.1 An UML specification

We report here the UML specification of the scenario and its workflow. We relay, for what regards the activity diagrams, on a service-oriented profile of UML, i.e. UML4SOA [2]. Within UML4SOA, interaction activities between services plays a central role. The specific actions for service interaction are: send a message, receive a message and send&receive (a synchronous communication where a message is sent and then the service awaits for a reply). Actions have associated *pins*: the *link* pin specifies which is the partner of the interaction, the *input* and *output* pins specify the exchanged messages for send (only output pins), receive (only input pins), and send&receive (both input and output pins) actions. For instance, in Fig. 1, a

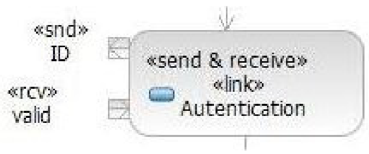

Fig. 1. An example of action in the profile UML4SOA

send&receive action is represented. The link pin, labelled by ≪link≫, specifies that service Authentication is the partner of the communication, the output pin, labelled by ≪snd≫, specifies that a message ID is sent and the input pin, labelled by ≪rcv≫, specifies that a message valid is received as an answer. The most important novelty in UML4SOA is the possibility to install compensations of executed activities that are executed in case of failure as discussed in the following.

Firstly, we illustrate the various services of the scenario, their orchestration and the kind of exchanged message. The customer initially logins to the Portal by sending his ID, i.e. his username and password. The customer identity is confirmed by an Authentication. For each successful login, Portal generates a sessionID, i.e. a datum univocally identifying a session. The value sessionID is used by the various actors for exchanging messages after the customer logins. Each message has, in fact, a body argument containing the exchanged data, and the sessionID as further argument identifying the corresponding session. This guarantees that messages referring to different requests are not erroneously mixed together. Portal then sends the requestID, i.e. the couple of sessionID and customer username, to service Information Upload, that starts a conversation with the customer in order to fill the request. The customer balance is validated by a Validation service. The filled request is then sent to Request Processing. Request Processing relies on employee and supervisor Task List services (shortened into empTaskList and supTaskList, respectively) for storing the request that is successively retrieved by, respectively, an employee and a supervisor, each of whom fills an Evaluation and forwards it to Request Processing. An Evaluation has two parts, i.e. Public Evaluation and Private Evaluation. The former is a tuple containing the strings Offer (the offer made to the customer), Motivation (the motivations of the rejection or of the request for an update) and Decision, which is equal to one of the values Accept, Reject or AskToUpdate. The latter is a tuple containing the strings rating and AdditionalInfo. Together, a Request and its Evaluation form a Contract. If either the employee or the supervisor asks to update the request, the related Contract is sent to service Information Update, that asks to the customer whether he wants to update the request and, in case, sends the updated request back to Request Processing . Finally, when an agreement between the bank and the customer is established, the related Contract is forwarded to a Contract Processing service.

We now specify the behavior of the involved services, i.e. Portal, Information Upload, Information Update and Request Processing, whose internal behavior is fundamental for a correct specification and implementation of the whole workflow.

The diagram of Fig. 2 relates to the interaction between the customer and service Portal. The customer ID is sent to the portal that starts a login scope.

Portal synchronously exchanges messages with service Authentication, sending the customer ID and receiving back the boolean valid. If valid=No, the service sends back to the customer a message signaling the failure of the login and then raises the exception failedLogin that terminates the process. If valid=Yes, the service generates (by means of action ≪create≫) a new sessionID and sends it back to the customer. Portal receives the customer choice about the desired service (here we only consider service Credit Request) and invokes service Information Upload (shortened into InfoUpload) sending to it a message with the requestID. From then on, the customer communicates with InfoUpload.

After the login, service InfoUpload (see Fig. 3) starts a conversation with the customer whose purpose is to produce a Request. The service workflow immediately forks in two parallel branches, one responsible for collecting the data of the Request, while the other one awaits for a message cancel from the customer, meaning that the customer wants to abort the process. In the last case, an exception abort is raised and the process terminates. The branch responsible of collecting the data of the Request first receives the desired amount from the customer. After that, the customer may choose to send first either his balance or the securities (shortened sec), hence the workflow forks in two parallel branches awaiting to receive the messages with this two data. Moreover, the branch responsible of receiving the balance, sends it to service Validation, that replies with a message containing the boolean valid. If valid=Yes, the workflow proceeds, otherwise, it sends a message to the customer, asking to resend the balance, and then cycles and awaits to receive a new message. After both the branches are completed, service InfoUpload terminates by invoking Request Processing (shortened into reqProcessing) and sending the Request to it.

Service Information Update (shortened into InfoUpdate) is similar to the previous service (see Fig. 4) but, unlike InfoUpload, it starts already receiving a Contract containing the existing Request and the Motivation for the request of an update. The Motivation and the request of an update are sent to the customer and the service awaits to receive an Answer. If Answer=No the process terminates, otherwise the workflow forks in two parallel branches. In one branch, the service asks the customer if he wants to update the securities: if the answer is positive, the service awaits to receive the new securities and then reaches a join point with the other branch, otherwise it immediately reaches the join point. The other branch does the same activities but with the amount in place of the securities. After both the branches have reached the join point, the service terminates by sending the updated Request back to service reqProcessing. In parallel with the described branch, the service starts another branch awaiting for a cancel request from the customer, exactly as in the case of InfoUpload described above.

As above specified, both InfoUpload and InfoUpdate send a request to reqProcessing (see Fig. 5). As for InfoUpload and InfoUpdate, the workflow initially forks in two branches, one responsible of the main interactions, while the other awaits to receive a cancel message from the customer and, in case, triggers an abort exception. Regarding the main branch, the received request is sent to service empTaskList that queues it. It is possible that the customer decides to cancel

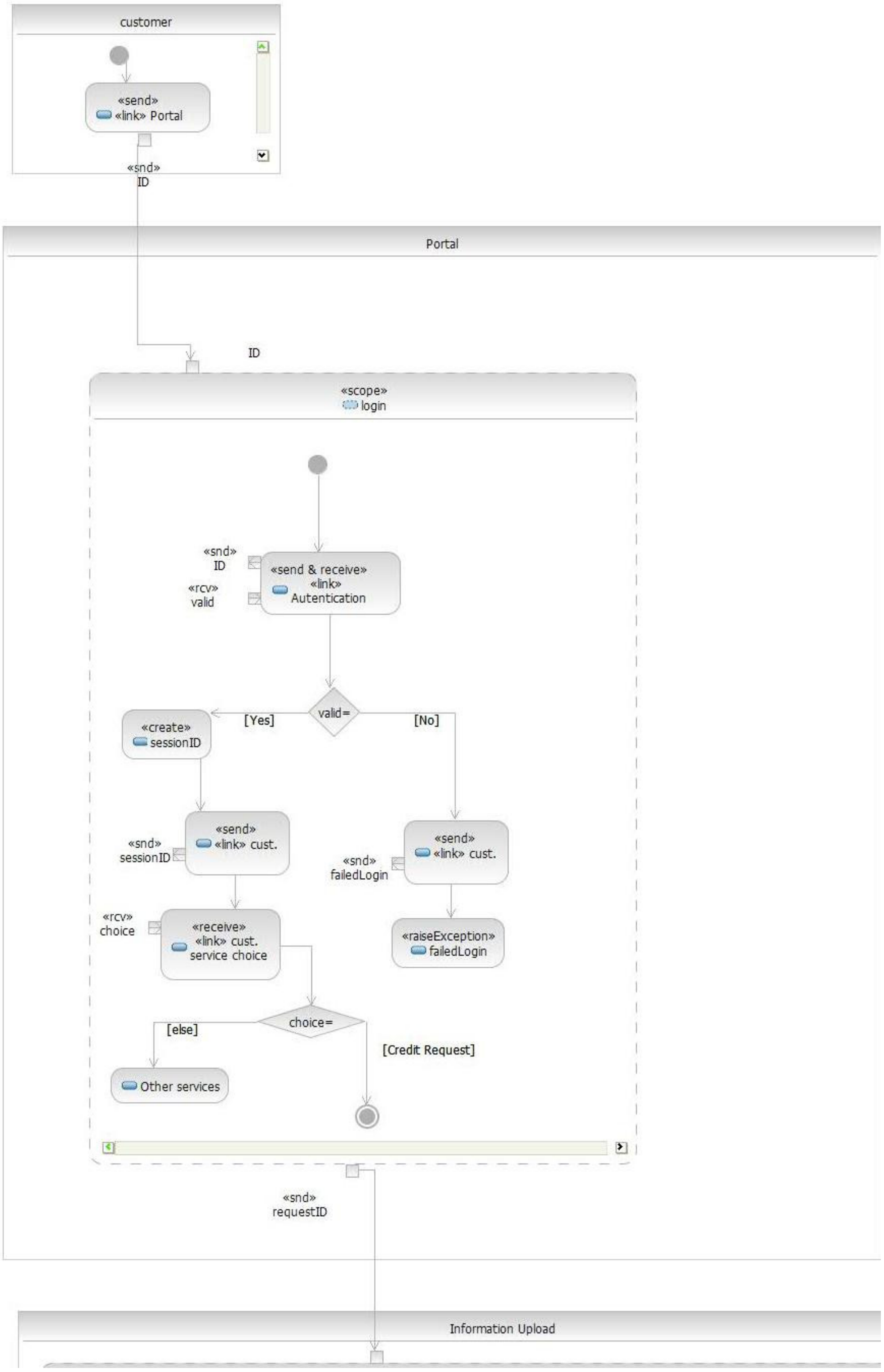

Fig. 2. Service Portal activity diagram

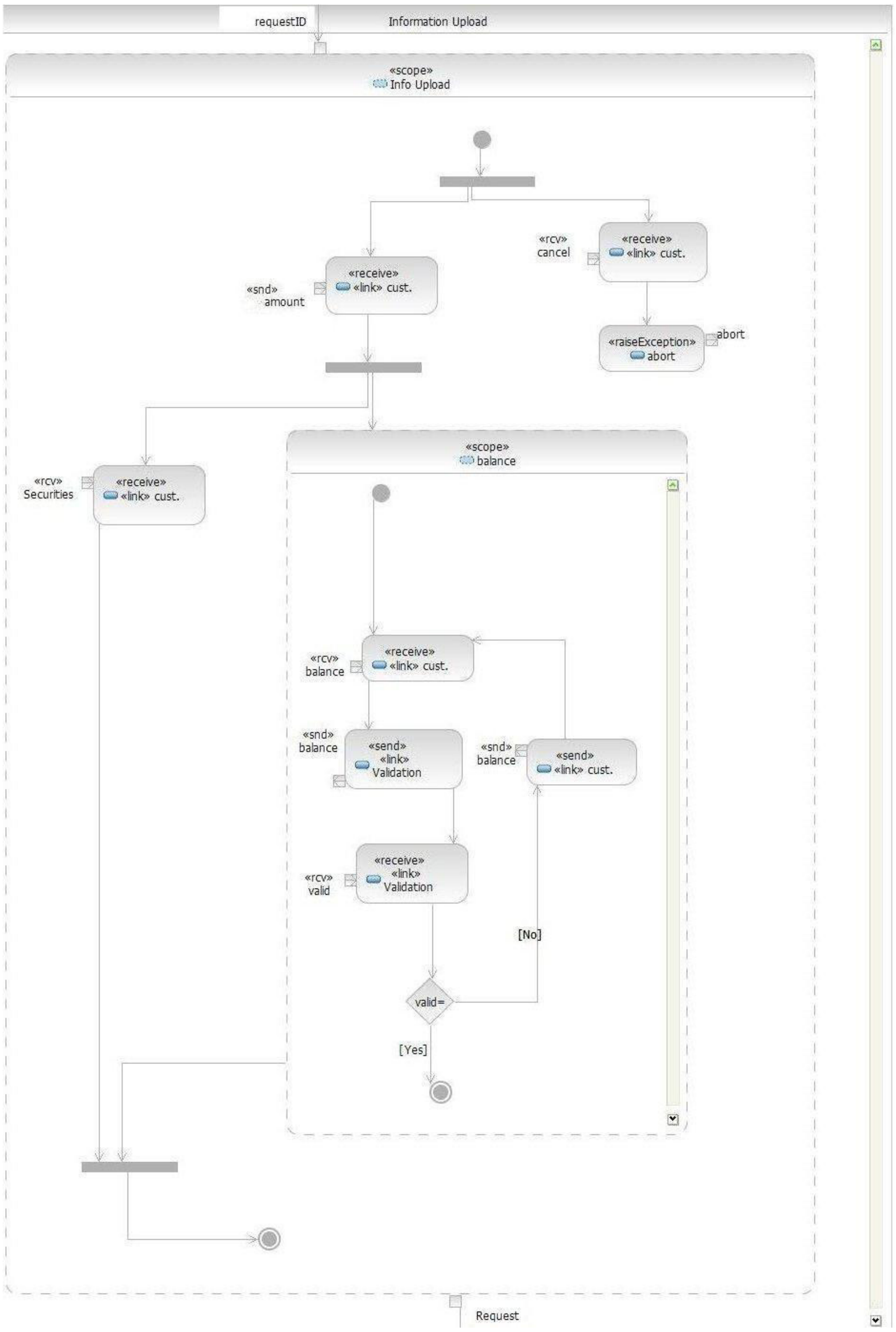

Fig. 3. Service InfoUpload activity diagram

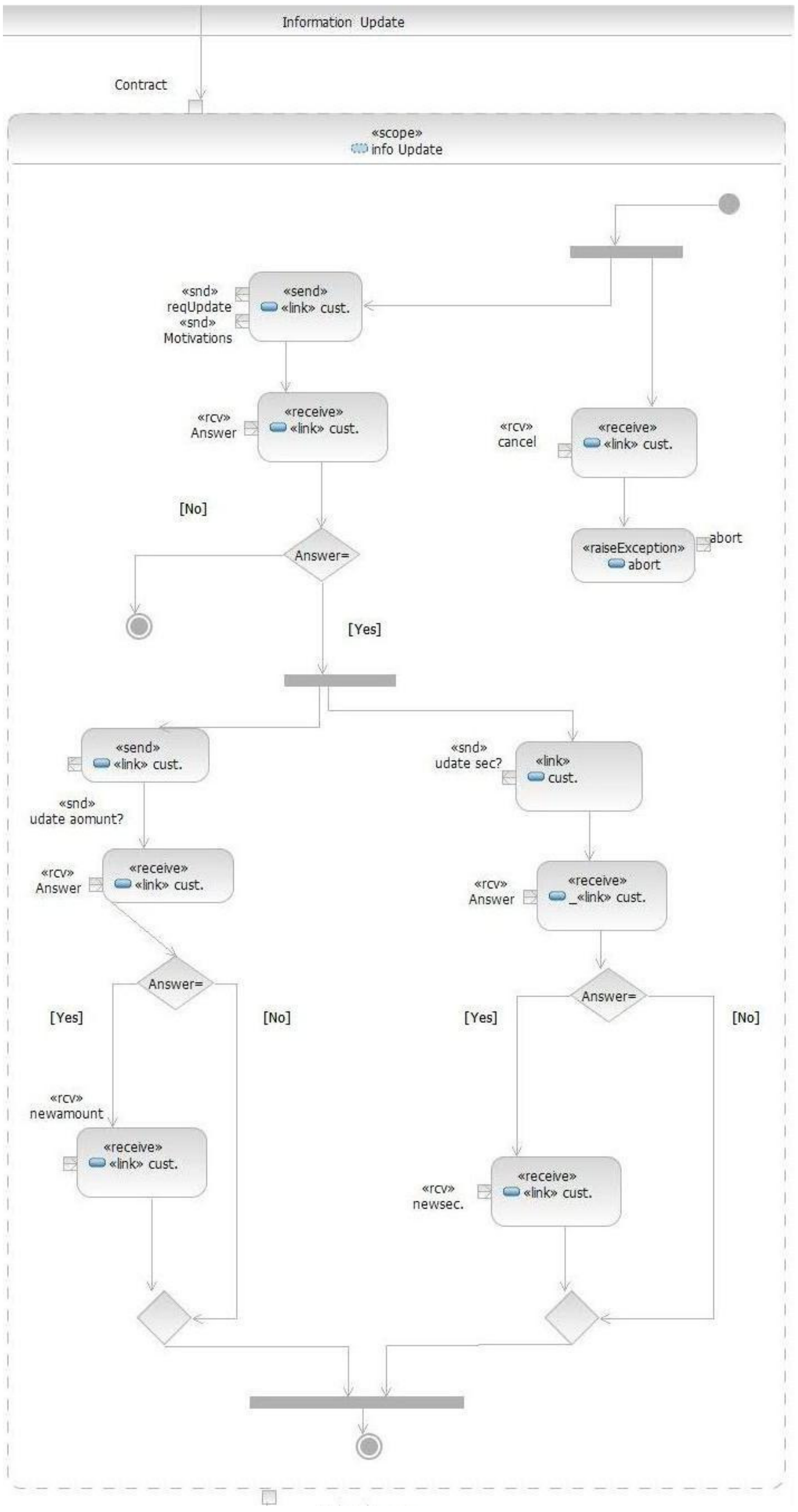

Fig. 4. Service InfoUpdate activity diagram

the request after this step has been performed. In this case, the request must be deleted from the task list, in order to prevent an employee to examine an already aborted request. Hence, the action of sending the request must be compensated with a delete action removing it from the task list. For this purpose, the service installs a 'compensation handler' consisting of an action sending the message Delete to service empTaskList. This message asks empTaskList to delete Request. Note that reqProcessing may not directly delete a request from a task list, since task lists are managed by services empTaskList and supTasklist that are autonomous from reqProcessing.

After sending the Request, reqProcessing awaits for the related Evaluation from an employee. The workflow then follows three alternative lines, according to the value of the argument Decision of Evaluation. If Decision=Reject, the service sends the Public Evaluation (shortened pubEvaluation), containing the decision and its motivation, to the customer and then terminates. If Decision=AskToUpdate, the service terminates by sending the Contract, containing the decision, its motivation and the request to InfoUpdate illustrated above. Finally, if Decision=Accept the second step of evaluation, similar to the described one, starts. The Contract is sent to supTaskList and the related compensation, asking for the deletion of the Contract from the supervisor task list, is installed. The service then awaits for the pubEvaluation by a supervisor. If Decision=Reject or Decision=AskToUpdate the service performs the same actions described above. If Decision=Accept, the service sends the pubEvaluation with the Decision and the bank Offer to the customer and awaits for his Answer. If Answer=No the process terminates, if Answer=Yes the service sends the Contract to a Contract Processing service and the process successfully terminates.

It still remains to examine the case when a customer asks to cancel the request while reqProcessing is running. As for services InfoUpload and InfoUpdate, the cancel message is received by a secondary branch of the process running in parallel with the main branch described above; then an exception abort is raised that eventually leads to process termination. However, before ending the process, some compensation activities may be required. The action ≪Compensate All≫ is executed; the meaning of this action is to execute all the installed compensations. Hence, if no compensation has yet been installed, the compensation activity is empty. If only the request of deletion for the employee task list was installed, that compensation is executed. If both the requests of deletion for the employee and the supervisor task lists were installed, the latter is first executed followed by the former.

2.2 Some expected properties

There are several requirements and properties concerning to liveness, correctness, and security that an implementation of the described scenario is expected to fulfill. Among these, in the following we will focus on:

Availability: The credit portal is always capable to accept a credit request.

Responsiveness: Whenever the customer uploads a credit request he always gets

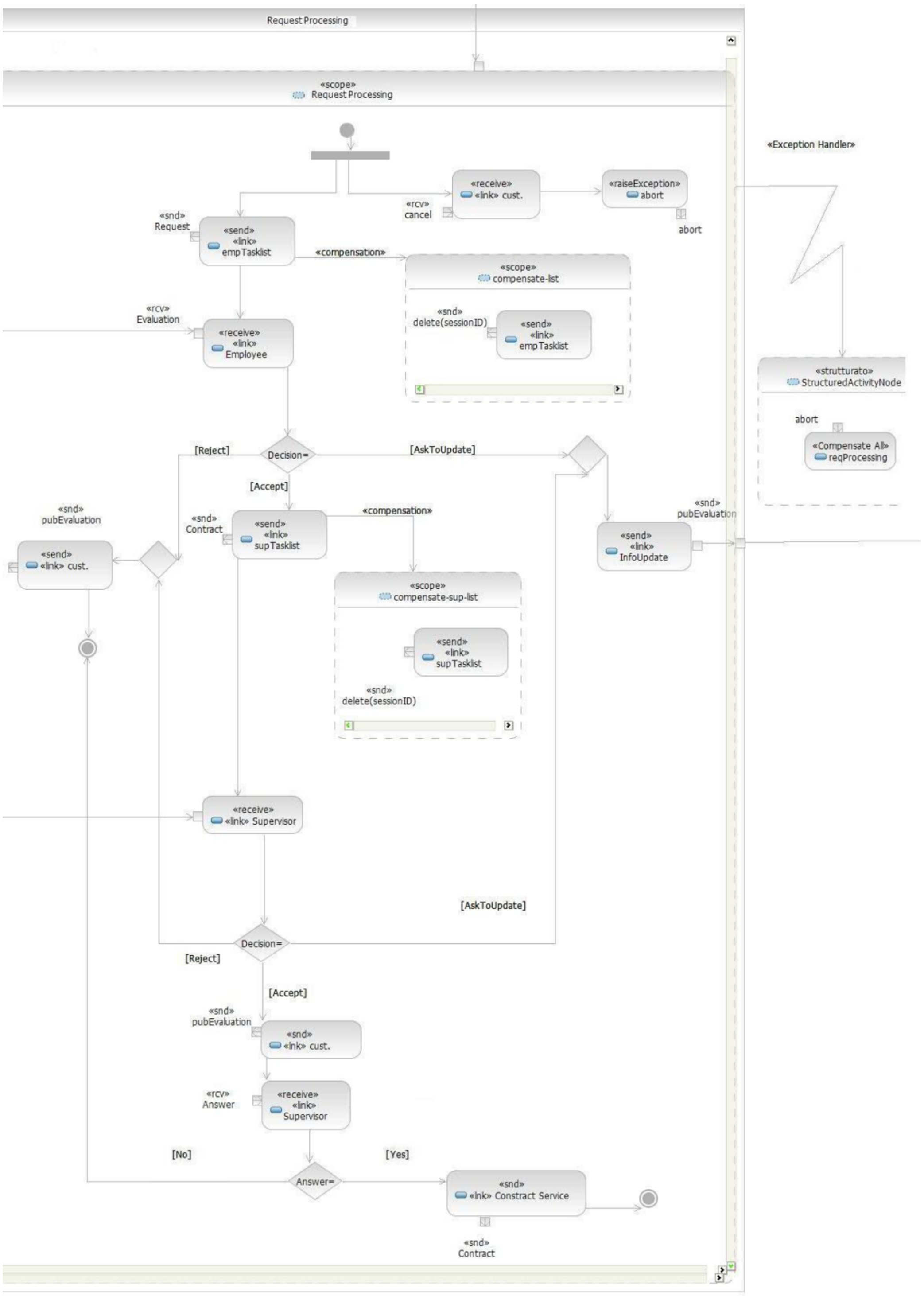

Fig. 5. Service reqProcessing activity diagram

an answer, unless he cancels his own request.

Correlation soundness: The customer always receives an answer which is relative to his credit request. Thus, it never occurs that the service sends him an evaluation related to another credit request or that it mixes data related to different credit requests.

Interruptibility: The customer may require the abort of the process after that he has selected the credit request service.

We will also examine some behavioral properties more specific for the case study like the following ones.

(i) The customer can receive an offer only after its credit request has been successfully evaluated by a supervisor.

(ii) The customer can receive a negative response only after its credit request has been negatively evaluated by an employee or a supervisor, or the given balance has deemed not to be valid.

(iii) If a credit request is accepted to be evaluated and the customer requires the cancellation, then compensation must be activated.

(iv) If a credit request is demanded to be updated, the customer will be notified or a cancellation will be invoked.

(v) Before processing a credit request, the customer must insert securities and balance data.

(vi) A credit request can always succeed.

(vii) A supervisor can always be involved for evaluating a credit request.

3 COWS: a Calculus for Orchestration of Web Services

In this section, we introduce the syntax of COWS and a glimpse of its semantics, and refer the interested reader to [16] for a formal presentation, for examples illustrating peculiarities and expressiveness of the language, and for comparisons with other process-based and orchestration formalisms. To get accustomed to using the language one can also use CMC, that, other than the model checking of SocL formulae, also supports the automated derivation of all computations originating from a COWS term.

The syntax of COWS is presented in Table 1. It is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x, y, ...). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, o, p, ..., mainly used to represent partners and operations. The language is also parameterized by a set of *expressions*, ranged over by ϵ, whose exact syntax is deliberately omitted. We just assume that expressions contain, at least, values and variables, but do not include killer labels (that, hence, are *not* communicable values). Partner names and operation names can be combined to designate *endpoints*, written $p \bullet o$, and can be

s	$::=$	$\mathbf{kill}(k) \mid u \bullet u'!\bar{\epsilon}$	(kill, invoke)
	$\mid$	$\sum_{i=0}^{l} p_i \bullet o_i?\bar{w}_i.s_i \mid s \mid s$	(receive-guarded choice, parallel)
	$\mid$	$\{\|s\|\} \mid [e]\, s \mid * \, s$	(protection, delimitation, replication)

Table 1
COWS syntax

communicated, but dynamically received names can only be used for service invocation. Indeed, endpoints of receive activities are identified statically because their syntax only allows using names and not variables.

We use w to range over values and variables, u to range over names and variables, and e to range over *elements*, namely killer labels, names and variables. Notation $\bar{\cdot}$ stands for tuples (i.e. ordered sequences) of homogeneous elements, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). We assume that variables in the same tuple are pairwise distinct. All notations shall extend to tuples component-wise. We adopt the following conventions about operators precedence: monadic operators bind more tightly than parallel composition, and prefixing more tightly than choice. In the sequel, we shall use $\mathbf{0}$ to denote empty choice and $+$ to abbreviate binary choice. We will omit trailing occurrences of $\mathbf{0}$, writing e.g. $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[e_1, \ldots, e_n]\, s$ in place of $[e_1] \ldots [e_n]\, s$. We will write $I \triangleq s$ to assign a name I to the term s.

Invoke and *receive* are the basic communication activities provided by COWS. An invoke can proceed as soon as evaluation of the expressions in its argument returns the corresponding values. A receive offers an invocable operation along a given partner name and its execution permits to take a decision among alternative behaviours. Besides input parameters and sent values, both activities indicate an *endpoint*, i.e. a pair composed of a partner name p and of an operation name o, through which communication should occur. An endpoint $p \bullet o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p.

The naming mechanism used to identify endpoints is very flexible. For example, it allows the same service to be identified by means of different logic names (i.e. to play more than one partner role as in WS-BPEL). Thus, the term $p_{slow} \bullet o?\bar{w}.s_{slow} + p_{fast} \bullet o?\bar{w}.s_{fast}$ accepts requests for the same operation o (with parameters $\bar{w}$) through different partners with distinct access modalities: the continuation s_{slow} implements a 'slower service' provided when the request is processed through the partner p_{slow}, while s_{fast} implements a 'faster service' provided when the request arrives through p_{fast}. Additionally, the naming mechanism allows the names composing an endpoint to be dealt with separately, as in a request-response interaction, where usually the service provider knows the name of the response operation, but not the partner name of the service it has to reply to. For example, the term $ping \bullet req?\langle x \rangle.\, x \bullet res!\langle \text{“}I\ live\text{”} \rangle$ behaves as a sort of 'ping' service that will know at run-time the partner name for the reply activity, i.e. the service which will be bound to x. In fact, partner and operation names are dealt with as values and, as such, can be exchanged in communication (although dynamically received names cannot form the endpoints used to receive further invocations). This enables easily modelling many service interaction and reconfiguration patterns.

An inter-service communication takes place when the arguments of a receive and of a concurrent invoke along the same endpoint do match, and causes substitution of the variables arguments of the receive with the corresponding values arguments of the invoke (within the scope of variables declarations). The substitution for a variable is applied only when the whole scope of the variable is determined and to the term resulting from removing the delimitation. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables, which is different from most process calculi. The range of application of the substitution generated by a communication is then regulated by the *delimitation* operator (namely, $[e]\, s$ binds e in the scope s), that is the *only* binder of the calculus. Delimitation is also used to generate fresh names (as the restriction operator of the π-calculus [20]) and to delimit the field of action of kill activities.

Execution of a *kill* activity $\mathbf{kill}(k)$ causes termination of all parallel terms inside the enclosing $[k]$, that stops the killing effect. However, critical code can be protected from the effect of a forced termination by using the *protection* operator. Notably, the scope of names and variables can be extended (by using 'structural laws' similar to those dealing with restricted names in the π-calculus), while that of killer labels cannot (in fact, they are not communicable values). The *replication* operator permits to spawn in parallel as many copies of its argument term as necessary thus, for example, enabling creation of concurrent service instances.

Execution of *parallel* terms is interleaved, but when a communication or a kill activity can be performed. Indeed, COWS parallel operator is equipped with a priority mechanism which allows some actions to take precedence over others.

For example, when a message arrives, the problem arises of rightly handling race conditions among those service instances and the corresponding service definition which are able to receive the message. This requires being able to determine if the message must be delivered to an already existing instance or if it must produce a new instance (i.e. it has to be delivered to the service definition). Receive activities are then assigned priority values which depend on the messages available so that, in presence of concurrent matching receives, only a receive using a more defined pattern (i.e. having greater priority) can proceed. This way, service definitions and service instances are represented as processes running concurrently, but service instances take precedence over the corresponding service definition when both can process the same message, thus preventing creation of wrong new instances.

Notably, receives would have *dynamically* assigned priority values since these values depend on the matching ability of their argument pattern. Indeed, while computation proceeds, some of the variables used in the argument pattern of a receive can be assigned values, because of execution of concurrent threads sharing these variables. This, on the one hand, restricts the set of messages matching the pattern but, on the other hand, increases the priority of the receive in comparison with other receives matching a same message. Furthermore, pre-emption is *local* since receives with greater priority can only pre-empt receives that are competing for the same message.

Moreover, COWS's priority mechanism assigns greatest priority to kill activities so that they pre-empt all other activities inside the enclosing killer label's delimitation. In other words, kill activities are executed *eagerly*, this way ensuring that, when a fault arises in a scope, (some of) the remaining activities of the enclosing scope are terminated before starting execution of the relative fault handler. In fact, activities forcing immediate termination of other concurrent activities are usually used for modelling fault handling. The same mechanism, of course, can also be used for exception and compensation handling.

4 A COWS specification of the finance case study

In this section, we present a relevant part of the COWS specification modelling the finance case study (the complete specification, written in CMC 'machine readable' syntax, is reported in the Appendix).

The COWS term representing the overall scenario is

$$\begin{array}{l}[\mathit{key}]\,(\,\mathit{Customer} \mid \mathit{CreditInstitute}\,)\\ \mid \mathit{Validation} \mid \mathit{Employee}_1 \mid \ldots \mid \mathit{Employee}_n \mid \mathit{Supervisor}_1 \mid \ldots \mid \mathit{Supervisor}_m\end{array}$$

The services above are composed by using the parallel composition operator $\_ \mid \_$ that allows the different components to be concurrently executed and to interact with each other. The delimitation operator $[\_]\,\_$ is used here to declare that *key* is a shared element known to *Customer* and *CreditInstitute*, and only to them.

CreditInstitute is defined as follows

$$\begin{array}{l}[\mathit{createInst}, \mathit{reqProcessing}, \mathit{reqUpdate}, \mathit{contractProcessing}]\\ (\ [\mathit{authentication}, \mathit{notAuthorized}, \mathit{authorized}]\,(\,\mathit{Portal} \mid \mathit{Authentication}\,)\\ \quad \mid \mathit{InformationUpload} \mid \mathit{InformationUpdate} \mid \mathit{RequestProcessing}\\ \quad \mid \mathit{ContractProcessing} \mid \mathit{EmployeeTaskList} \mid \mathit{SupervisorTaskList}\ )\end{array}$$

The term is the parallel composition of the (considered) subservices of the credit institute: the behaviour of *Portal*, *InformationUpload*, *InformationUpdate* and *RequestProcessing* is graphically represented by the UML activity diagrams depicted in Fig. 2, 3, 4 and 5; *Authentication* and *ContractProcessing* appear as external services in Fig. 2 and 5, respectively.

The delimitation operator ensures that operations *authentication*, *notAuthorized* and *authorized* are used to communicate only by *Portal* and *Authentication*, while operations *createInst*, *reqProcessing*, *reqUpdate* and *contractProcessing* can also be used by the other subservices. This guarantees that external services cannot interfere with the credit portal during the login and instantiation phases.

Service *Portal* is publicly invocable and can interact with customers other than with the 'internal' services of the credit institute. *Portal* is defined as follows:

$$
\begin{array}{l}
* \ [x_{user}, x_{pwd}, x_{cust}] \\
\quad portal \bullet login?\langle x_{user}, x_{pwd}, x_{cust}\rangle . \\
\quad (\, portal \bullet authentication!\langle x_{user}, x_{pwd}\rangle \\
\quad\quad \mid \ portal \bullet notAuthorized?\langle x_{user}\rangle .\ x_{cust} \bullet failedLogin!\langle key\rangle \\
\quad\quad\quad + \ portal \bullet authorized?\langle x_{user}\rangle . \\
\quad\quad\quad\quad [sessionID]\ (\, x_{cust} \bullet logged!\langle key, sessionID\rangle \\
\quad\quad\quad\quad\quad\quad\quad \mid \ portal \bullet creditRequest?\langle sessionID\rangle . \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad portal \bullet createInst!\langle sessionID\rangle \\
\quad\quad\quad\quad\quad\quad\quad + \ portal \bullet bankTransferRequest?\langle sessionID\rangle . \ \ldots \\
\quad\quad\quad\quad\quad\quad\quad + \ \ldots \text{other services provided by the credit portal} \ldots \ )\ )
\end{array}
$$

The replication operator $*\,\_$, that spawns in parallel as many copies of its argument term as necessary, is exploited to model the fact that *Portal* can create multiple instances to serve several customer requests simultaneously. Each interaction with the portal starts with a receive activity of the form $portal \bullet login?\langle x_{user}, x_{pwd}, x_{cust}\rangle$ corresponding to reception of a request emitted by a customer. The receive activity initializes the variables x_{user}, x_{pwd} and x_{cust}, declared local to *Portal* by the delimitation operator, with data provided by a customer. Whenever prompted by a customer request, *Portal* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each instance forwards the request to *Authentication*, by invoking the 'internal' operation *authentication* through the invoke activity $portal \bullet authentication!\langle x_{user}, x_{pwd}\rangle$, and waits for a reply on one of the other two internal operations *notAuthorized* and *authorized*, by exploiting the receive-guarded choice operator. In case of a positive answer, by means of the delimitation operator, a fresh session identifier *sessionID* is generated, and a reply is sent back to the customer by means of an invoke activity using the partner name of the customer stored in the variable x_{cust}. Moreover, by using the choice operator again, *Portal* allows the customer to choose among several services (however, only the credit request service is actually modelled). Whenever the customer selects this service, *InformationUpload* is instantiated through invocation of the private operation *createInst*. Notably, the identifier *sessionID* is passed both to *Customer* and to the created instance of *InformationUpload*, to allow them to safely communicate. In fact, in each interaction between *Customer* and the instances of the credit institute subservices, the identifier is used as a correlation datum, i.e. it appears within each message. Pattern-matching permits locating such datum in the messages and, therefore, delivering them to the instances identified by the same datum.

InformationUpload is defined as follows:

```
* $[x_{id}]\ portal \bullet createInst?\langle x_{id}\rangle.$
        $[k, fault, abort]$ (
          $[k_{abortFault}]$ (
              $[x_{custData}, x_{secData}, x_{amount}, x_{cust}]$
              $portal \bullet getCreditRequest?\langle x_{id}, x_{custData}, x_{amount}, x_{cust}\rangle.$
                ( $portal \bullet securities?\langle x_{id}, x_{secData}\rangle$
                 | **repeat**
                     $[x_{balance}]\ portal \bullet balance?\langle x_{id}, x_{balance}\rangle.$
                        ( $validation \bullet validateBalance!\langle x_{id}, portal, x_{balance}\rangle$
                         | $portal \bullet validateBalance?\langle x_{id}, no\rangle.$
                               $x_{cust} \bullet balanceNotValid!\langle x_{id}\rangle$
                          $+\ portal \bullet validateBalance?\langle x_{id}, yes\rangle$ )
                   **until** $(x_{balance}\ is\ valid)$
                ) ; ($\mathbf{kill}(k)$ | $\{\!|portal \bullet reqProcessing!\langle x_{id}, x_{custData}, x_{secData},$
                                                  $x_{balance}, x_{amount}, x_{cust}\rangle|\!\}$)
              | $portal \bullet cancel?\langle x_{id}\rangle.$ ($\mathbf{kill}(k_{abortFault})$ | $\{\!|fault \bullet abort!\langle\rangle|\!\}$) )
          | $fault \bullet abort?\langle\rangle.\,\mathbf{0}$ )
```

Each instance of *InformationUpload* is created to serve a customer request identified by a specific session identifier. Once created, an instance is actually activated by *Customer* by invoking operation *getCreditRequest* and transmitting the credit request application data (i.e. name, address, desired credit amount, ...). Then, two activities are concurrently executed: (1) additional security data are received, and (2) balance information are received and forwarded to *Validation* for checking consistency and validation; if the verification was negative (i.e. the second argument of operation *validateBalance* is *no*), then *Customer* is informed and (2) is repeated. Notice that, for the sake of readability, in the COWS term we have used repeat loop and sequential constructs. In fact, they can be easily expressed in COWS, as shown in the CMC specification reported in the Appendix. When (1) and (2) terminate successfully, *RequestProcessing* is instantiated, by invoking the (private) operation *reqProcessing* and initialising the created instance with all the request data (i.e. customer data, amount, security data and balance information). Notably, at any time after the login *Customer* can require the cancellation of the credit request processing by invoking operation *cancel* using the identifier. This causes the forced termination of all unprotected parallel activities, through the execution of the activity $\mathbf{kill}(k_{abort})$, and the emission of an (internal) fault signal $fault \bullet abort!\langle\rangle$. To deal with such faults, each instance has a specific fault handler, that catches a fault and does nothing. If an instance completes successfully, then its fault handler is removed by executing the activity $\mathbf{kill}(k)$.

RequestProcessing is defined as follows:

$$
\begin{array}{l}
* \; [x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{amount}, x_{cust}] \\
\quad portal \bullet reqProcessing?\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{amount}, x_{cust}\rangle. \\
\quad [k, fault, abort, undo]\,(\\
\qquad [k_{abortFault}]\,(\\
\qquad\qquad portal \bullet addToETL!\langle x_{id}, x_{secData}, x_{balance}, x_{amount}\rangle \\
\qquad\qquad \mid portal \bullet taskAddedToETL?\langle x_{id}\rangle. \\
\qquad\qquad\qquad (\,\{\!|portal \bullet undo?\langle empTaskList\rangle.\, portal \bullet removeTaskETL!\langle x_{id}\rangle|\!\} \\
\qquad\qquad\qquad\quad \mid\; EmployeeEval\,) \\
\qquad\qquad \mid\; portal \bullet cancel?\langle x_{id}\rangle.\,(\mathbf{kill}(k_{abortFault}) \mid \{\!|fault \bullet abort!\langle\rangle|\!\})\;) \\
\qquad \mid\; fault \bullet abort?\langle\rangle. \\
\qquad\qquad (portal \bullet undo!\langle empTaskList\rangle \mid portal \bullet undo!\langle supTaskList\rangle)\;)
\end{array}
$$

When an instance of *RequestProcessing* is created, all the data relevant for computing the rating are inserted in the *EmployeeTaskList*, through invocation of operation *addToETL*. When an acknowledgment from *EmployeeTaskList* is received, a compensation handler for undoing the insertion activity is installed, and the instance is blocked waiting for the employee evaluation (term *EmployeeEval*). The compensation handler is a protected term waiting for a compensation request, i.e. a signal *empTaskList* along $portal \bullet undo$. When this signal is received, the compensation handler becomes active and, to compensate the insertion activity, invokes operation *removeTaskETL* provided by *EmployeeTaskList*. Compensation is activated by the body of the fault handler, that sends the two compensation signals *empTaskList* and *supTaskList* (corresponding to action ≪Compensate All≫ of Fig. 5).

The term *EmployeeEval* is

$$
\begin{array}{l}
[x_{rating}, x_{info}, x_{decision}] \\
portal \bullet empEvaluation?\langle x_{id}, x_{rating}, x_{info}, x_{decision}\rangle. \\
[cond, choice]\,(cond \bullet choice!\langle x_{decision}\rangle \\
\qquad \mid\; cond \bullet choice?\langle no\rangle.\,(\mathbf{kill}(k) \mid \{\!|x_{cust} \bullet negativeResp!\langle x_{id}, x_{info}\rangle|\!\}) \\
\qquad\quad + cond \bullet choice?\langle update\rangle. \\
\qquad\qquad (\mathbf{kill}(k) \mid \{\!|portal \bullet reqUpdate!\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_{amount}, x_{cust}, x_{info}\rangle|\!\}) \\
\qquad + cond \bullet choice?\langle yes\rangle. \\
\qquad\qquad (\,portal \bullet addToSTL!\langle x_{id}, x_{secData}, x_{balance}, x_{amount}, x_{info}\rangle \\
\qquad\qquad\;\; \mid\; portal \bullet taskAddedToSTL?\langle x_{id}\rangle. \\
\qquad\qquad\qquad (\,\{\!|portal \bullet undo?\langle supTaskList\rangle.\, portal \bullet removeTaskSTL!\langle x_{id}\rangle|\!\} \\
\qquad\qquad\qquad\quad \mid\; SupervisorEval\,)\;)
\end{array}
$$

It receives the employee evaluation and performs a choice on the basis of the value stored in the variable $x_{decision}$, that can be either *no* (i.e. the credit request is rejected), or *update* (i.e. the customer is asked to update the desired amount and/or the security data), or *yes* (i.e. the employee accepts the request). Conditional choice is modelled in a natural way by a choice among three receives along the private endpoint *cond* • *choice*, and by relying on pattern-matching. In case of *no* and *update*, the instance is halted (by means of a kill activity), while in case of acceptance the request data are inserted in the supervisor task list, the corresponding compensation handler (i.e. the protected term) is installed, and the instance is blocked waiting for the supervisor evaluation (term *SupervisorEval*).

Finally, *SupervisorEval* is

$$
\begin{array}{l}
[x_{offer}, x_{motivation}, x_{supDecision}] \\
portal \bullet supEvaluation?\langle x_{id}, x_{offer}, x_{motivation}, x_{supDecision}\rangle . \\
[cond, choice]\ (cond \bullet choice!\langle x_{supDecision}\rangle \\
\quad \mid cond \bullet choice?\langle no\rangle .\ (\mathbf{kill}(k) \mid \{\!|x_{cust} \bullet negativeResp!\langle x_{id}, x_{motivation}\rangle |\!\}) \\
\qquad + cond \bullet choice?\langle update\rangle . \\
\qquad\quad (\mathbf{kill}(k) \mid \{\!|portal \bullet reqUpdate!\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, \\
\qquad + cond \bullet choice?\langle yes\rangle . \hfill x_{amount}, x_{cust}, x_{motivation}\rangle |\!\}) \\
\qquad\quad (\ x_{cust} \bullet offer!\langle x_{id}, x_{offer}, x_{motivation}\rangle \\
\qquad\quad\ \mid portal \bullet answer?\langle x_{id}, yes\rangle . \\
\qquad\qquad (\ \mathbf{kill}(k) \mid \{\!|portal \bullet contractProcessing!\langle x_{id}, x_{custData}, x_{secData}, \\
\qquad\qquad\qquad x_{balance}, x_{amount}, x_{cust}, x_{rating}, x_{info}, x_{offer}, x_{motivation}\rangle |\!\}) \\
\qquad\qquad + portal \bullet answer?\langle x_{id}, no\rangle .\mathbf{kill}(k)\ )\ )
\end{array}
$$

The above term behaves similarly to *EmployeeEval* except for the case of positive evaluation, for which it sends an offer to the customer and, in case of acceptance, forwards all the information to *ContractProcessing*.

The remaining terms composing the scenario are reported in the Appendix. In particular, the task list services *EmployeeTaskList* and *SupervisorTaskList* could be modelled in different ways, according to the underlying data structures and the properties that they enjoy. For simplicity sake, in the specification reported in the Appendix, task lists do not preserve the arrival order of requests that are then withdrawn in a non deterministic way.

5 Analysis of the finance case study

The analysis of the case study is carried out by exploiting the logical verification environment introduced in [11] for checking functional properties of services. This environment provides:

- SocL: an action- and state-based, branching time, temporal logic expressly designed to formalise in a convenient way peculiar aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses;
- CMC: the on-the-fly model checker for SocL formulae over their interpretation domain, namely Doubly Labelled Transition Systems (L^{2}TSs, [10]).

This approach takes an abstract point of view: services are thought of as software entities which may have an internal state and can interact with each other.

Actions that services can do are characterised by a *type*, e.g. accept a request, provide a response, etc., and belong to an *interaction* started when a client (possibly another service) firstly invokes one of the operations exposed by the service. To univocally identify an action, since multiple instances of a same interaction can be simultaneously active because service operations can be independently invoked by several clients, *correlation data* are used as a third attribute of service actions.

Correspondingly, the actions of the logic are characterised by three attributes: type, interaction name, and correlation data. They may also contain variables, called *correlation variables*, to enable capturing correlation data used to link together actions executed as part of the same interaction. For a given correlation variable var, its binding occurrence is denoted by $\underline{var}$; all remaining occurrences, that are called *free*, are denoted by var. Formally, SocL actions have the form $t(i, c)$, where t is the type of the action, i is the name of the interaction which the action is part of, and c is a tuple of correlation values and variables identifying the interaction (i and c can be omitted whenever do not play any role). We use $\underline{\alpha}$ as a generic action (notation $\underline{\cdot}$ emphasises the fact that the action may contain variable binders), and α as a generic action without variable binders.

For example, action $request(cr, 1234)$ could stand for a *request* action for starting an (instance of the) interaction cr which will be identified through the correlation tuple $\langle 1234 \rangle$. If some correlation value is unknown at design time, a (binder for a) correlation variable id can be used instead, as in the action $request(charge, \underline{id})$. This way, during the formula verification process, id will capture the corresponding value that can be then used to correlate subsequent actions performed as part of the same interaction. Thus, for instance, the *successful response* action for the specific request above could be written as $response(cr, id)$. Similarly, actions like $cancel(cr, id)$ and $fail(cr, id)$ could indicate *cancellation* and *failure* notification for the same request.

The syntax of SocL formulae is defined as follows:

$$\begin{array}{rrcl}
\textit{(action formulae)} & \gamma & ::= & \underline{\alpha} \mid \chi \qquad\qquad \chi \ ::= \ tt \mid \alpha \mid \tau \mid \neg\chi \mid \chi \wedge \chi \\
\textit{(state formulae)} & \phi & ::= & true \mid \pi \mid \neg\phi \mid \phi \wedge \phi' \mid E\Psi \mid A\Psi \\
\textit{(path formulae)} & \Psi & ::= & X_\gamma \phi \mid \phi \,{}_\chi U_\gamma\, \phi' \mid \phi \,{}_\chi W_\gamma\, \phi'
\end{array}$$

We comment on salient points. Action formulae are simply boolean compositions of actions, where tt is the action formula always satisfied, τ denotes unobservable actions, $\neg$ and $\wedge$ are the standard logical operator for negation and conjunction, respectively. As usual, we will use ff to abbreviate $\neg tt$ and $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$. π denotes an *atomic proposition*, that is a property that can be true over the states of services. Atomic propositions have the form $p(i, c)$, where p is the name, i is an interaction name, and c is a tuple of correlation values and variables identifying i (as before, i and c can be omitted whenever do not play any role). A and E are universal and existential (resp.) *path quantifiers*. X, U and W are the *next*, *until* and *weak until* operators. The next operator says that in the next state of the path, reached by an action satisfying γ, the formula ϕ holds. The until operator U says that ϕ' holds at some future state of the path reached by a last action satisfying γ, while ϕ holds from the current state until that state is reached and all the actions executed in the meanwhile along the path satisfy χ or τ (i.e. are unobservable). The weak until operator W (also called *unless*) holds on a path either if the corresponding strong until operator holds or if for all the states of the path the formula ϕ holds and all the actions of the path satisfy χ or τ.

Other useful logic operators can be derived as usual. In particular, the ones that we use in the sequel are:

- $[\gamma]\,\phi$ stands for $\neg\, EX_\gamma \neg\, \phi$; formula $[\gamma]\,\phi$ holds in a state q if, and only if, in the next state of any path starting from q, reached by an action satisfying the action formula γ, the formula ϕ holds.
- $EF\phi$ stands for $\phi \vee E(\mathit{true}\ {}_{tt}U_{tt}\phi)$; formula $EF\phi$ holds in a state if, and only if, ϕ holds at the starting state or at a future state.
- $AG\phi$ stands for $\neg\, EF \neg\, \phi$; formula $AG\phi$ holds in a state q if, and only if, ϕ holds in q and in all the states reachable along any path starting from q.
- $EF_\gamma\, \mathit{true}$ (resp. $AF_\gamma\, \mathit{true}$) stands for $E(\mathit{true}\, {}_{tt}U_\gamma\, \mathit{true})$ (resp. $A(\mathit{true}\, {}_{tt}U_\gamma\, \mathit{true})$); formula $EF_\gamma\, \mathit{true}$ (resp. $AF_\gamma\, \mathit{true}$) holds in a state q if, and only if, there exists a state reachable by a last action satisfying γ along a (resp. any) path starting from q.

We refer the interested reader to [11] for a formal account of the semantics of SocL formulae.

5.1 Properties of the case study specified with SocL

In this section we express as SocL formulae the properties that the case study is expected to fulfill and that we informally introduced in Section 2.2.

Availability: $AG(\mathit{accepting\_request}(\mathit{login}))$

This formula means that the service *CreditInstitute* is *available*, i.e. it is always capable to accept a credit request. Indeed, atomic proposition $\mathit{accepting\_request}(\mathit{login})$ means that a state is able to accept a credit request for interaction *login*.

Responsiveness and correlation soundness: Both properties are expressed by the following SocL formula:

$$AG\,[\mathit{request}(cr, \underline{id})]\, AF_{\mathit{response}(cr,id) \vee \mathit{fail}(cr,id) \vee \mathit{cancel}(cr,id)}\ \mathit{true}$$

This formula means that *CreditInstitute* is *responsive*, i.e. it always guarantees an answer (i.e. an offer or a negative response, sent by means of actions $\mathit{response}(cr, id)$ or $\mathit{fail}(cr, id)$, respectively) to each received credit request, unless the customer cancels his own request (by means of action $\mathit{cancel}(cr, id)$). The answers from *CreditInstitute* and the request of cancellation from *Customer* belong to the same interaction *cr* of the credit request and are properly *correlated* by variable *id*.

Interruptibility: $AG\,[\mathit{request}(cr, \underline{id})]\, EF_{\mathit{cancel}(cr,id)}\ \mathit{true}$

The customer may require the cancellation of a credit request, only after that he has selected the credit request service.

The specific behavioral properties of the case study are expressed in SocL as follows.

(i) $AG\,[request(cr,\underline{id})]\,\neg E(true_{\neg\, response(seval,id)} U_{response(cr,id)}\, true)$
The customer can receive an offer (action $response(cr,id)$) only after its credit request has been successfully evaluated by a supervisor (action $response(seval,id)$).

(ii) $AG\,[request(cr,\underline{id})]\,\neg E(true_{\neg\,(fail(eeval,id)\vee fail(seval,id)\vee fail(beval,id))} U_{fail(cr,id)}\, true)$
The customer can receive a negative response only after its credit request has been negatively evaluated by an employee or a supervisor (as indicated by the failure of the interactions $eeval$ and $seval$, respectively) or the given balance has deemed not to be valid (as indicated by the failure of the interaction $beval$).

(iii) $AG\,[request(eval,\underline{id})]\,EF\,[cancel(cr,id)]AF_{cancel(eval,id)}\ true$
If a credit request is accepted to be evaluated (i.e. it is added to some task list as indicated by the interaction $eval$) and the customer requires the cancellation (action $cancel(cr,id)$), then compensation must be activated, i.e. the task must be removed from the list (action $cancel(eval,id)$).

(iv) $AG\,[request(upd,\underline{id})]\,AF_{response(upd,id)\vee cancel(cr,id)}\ true$
If a credit request is demanded to be updated (interaction upd), the customer will be notified or a cancellation will be invoked.

(v) $AG\,[request(cr,\underline{id})]\,\neg E(true_{\neg\,(request(sec,id)\vee request(bal,id))} U_{request(rproc,id)}\, true)$
Before processing a credit request (interaction $rproc$), the customer must insert securities (interaction sec) and balance data (interaction bal).

(vi) $AG\,[request(cr,\underline{id})]\,AF_{\neg cancel(cr,id)\vee response(cr,id)}\ true$
A credit request can always succeed.

(vii) $AG\,[request(cr,\underline{id})]\,AF_{\neg cancel(cr,id)\vee request(tostl,id)}\ true$
A supervisor can always be involved for evaluating a credit request (interaction $tostl$ starts when a request is added to the supervisor tasks list).

5.2 Model checking SocL formulae

Although the logical verification environment directly handles L^{2}TSs, we have specified our case study by using COWS because it is a more convenient notation. Now, to check if our COWS specification enjoys the properties expressed as SocL formulae in Section 5.1, some steps must be performed. Firstly, the LTS representing the semantics of the COWS specification must be transformed into a *concrete* L^{2}TS by labelling each state with the set of actions the COWS specification is able to perform immediately from that state. Of course, the transformation preserves the structure of the original COWS LTS. Both in the original LTS and in the L^{2}TS obtained as explained before, transitions are labelled by ‘concrete’ actions, i.e. those actions occurring in the COWS term. For example, (an excerpt of) the concrete L^{2}TS obtained by applying this transformation is shown in Fig. 6. Notably, arcs

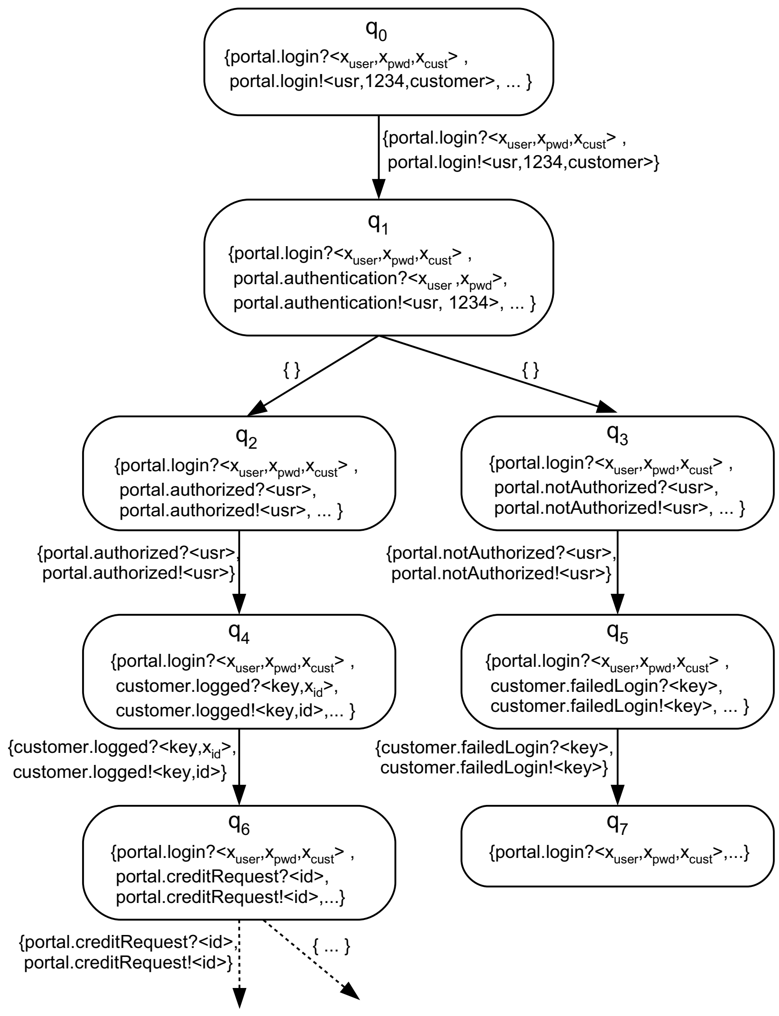
q0
{portal.login?<x_user,x_pwd,x_cust> , portal.login!<usr,1234,customer>, ... }
{portal.login?<x_user,x_pwd,x_cust> , portal.login!<usr,1234,customer>}
q1
{portal.login?<x_user,x_pwd,x_cust> , portal.authentication?<x_user ,x_pwd>, portal.authentication!<usr, 1234>, ... }
{ }
{ }
q2
{portal.login?<x_user,x_pwd,x_cust> , portal.authorized?<usr>, portal.authorized!<usr>, ... }
q3
{portal.login?<x_user,x_pwd,x_cust> , portal.notAuthorized?<usr>, portal.notAuthorized!<usr>, ... }
{portal.authorized?<usr>, portal.authorized!<usr>}
{portal.notAuthorized?<usr>, portal.notAuthorized!<usr>}
q4
{portal.login?<x_user,x_pwd,x_cust> , customer.logged?<key,x_id>, customer.logged!<key,id>,... }
q5
{portal.login?<x_user,x_pwd,x_cust> , customer.failedLogin?<key>, customer.failedLogin!<key>, ... }
{customer.logged?<key,x_id>, customer.logged!<key,id>}
{customer.failedLogin?<key>, customer.failedLogin!<key>}
q6
{portal.login?<x_user,x_pwd,x_cust> , portal.creditRequest?<id>, portal.creditRequest!<id>,...}
q7
{portal.login?<x_user,x_pwd,x_cust>,...}
{portal.creditRequest?<id>, portal.creditRequest!<id>}
{ ... }

Fig. 6. Excerpt of the concrete L^{2}TS for the finance case study

have attached labels corresponding to communications, retaining all information contained in the two synchronising invoke and receive activities.

Then, since we are interested in verifying the abstract properties of services shown in Section 5.1, we need to abstract away from unnecessary details by transforming concrete actions into 'abstract' ones. In general, this is done by applying a set of application-dependent abstraction rules that transform the concrete L^{2}TS into a more abstract one. Such transformation only involves the concrete actions we want to observe; the concrete actions that are not replaced by their abstract counterparts may not be observed. For example, the abstract L^{2}TS shown in Fig. 7 is obtained by applying to the concrete L^{2}TS of Fig. 6 the following rules:

$$\begin{array}{lll} Action: & creditRequest\langle \$1 \rangle & \rightarrow \quad request(cr, \$1) \\ Action: & offer\langle \$1, *, * \rangle & \rightarrow \quad response(cr, \$1) \end{array}$$

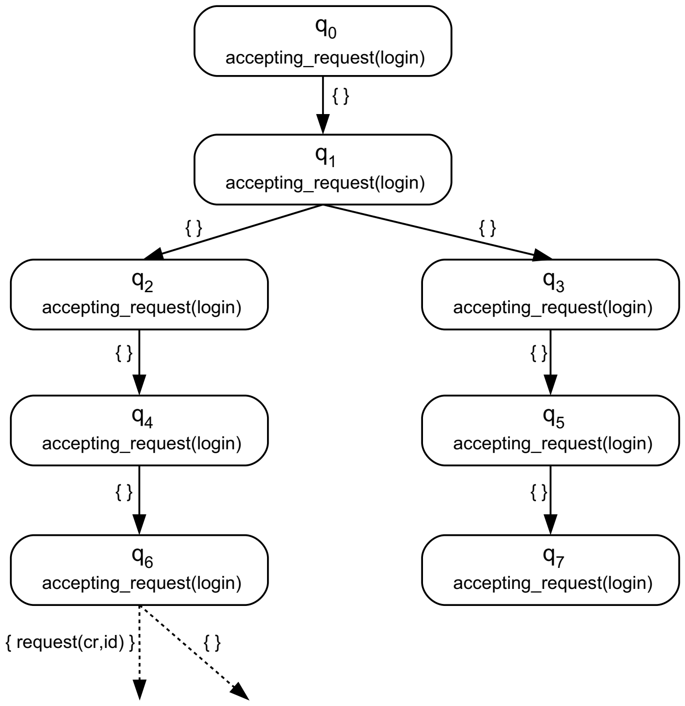

Fig. 7. Excerpt of the abstract L^2TS for the finance case study

$$\begin{array}{rl}
Action: & update\langle \$1, * \rangle \rightarrow fail(cr, \$1)\\
Action: & negativeResp\langle \$1, * \rangle \rightarrow fail(cr, \$1)\\
Action: & cancel\langle \$1 \rangle \rightarrow cancel(cr, \$1)\\
Action: & balanceNotValid\langle \$1 \rangle \rightarrow fail(cr, \$1)\\
Action: & empEvaluation\langle \$1, *, *, yes \rangle \rightarrow response(eeval, \$1)\\
Action: & empEvaluation\langle \$1, *, *, no \rangle \rightarrow fail(eeval, \$1)\\
Action: & supEvaluation\langle \$1, *, *, yes \rangle \rightarrow response(seval, \$1)\\
Action: & supEvaluation\langle \$1, *, *, no \rangle \rightarrow fail(seval, \$1)\\
Action: & validateBalance\langle \$1, yes \rangle \rightarrow response(beval, \$1)\\
Action: & validateBalance\langle \$1, no \rangle \rightarrow fail(beval, \$1)\\
Action: & taskAddedToETL\langle \$1 \rangle \rightarrow request(eval, \$1)\\
Action: & taskAddedToSTL\langle \$1 \rangle \rightarrow request(eval, \$1)\\
Action: & taskAddedToSTL\langle \$1 \rangle \rightarrow request(tostl, \$1)\\
Action: & removeTaskSTL\langle \$1 \rangle \rightarrow cancel(eval, \$1)\\
Action: & removeTaskETL\langle \$1 \rangle \rightarrow cancel(eval, \$1)\\
Action: & reqUpdate\langle \$1, *, *, *, *, *, * \rangle \rightarrow request(upd, \$1)\\
Action: & update\langle \$1, * \rangle \rightarrow response(upd, \$1)\\
Action: & securities\langle \$1, * \rangle \rightarrow request(sec, \$1)\\
Action: & balance\langle \$1, * \rangle \rightarrow request(bal, \$1)\\
Action: & reqProcessing\langle \$1, *, *, *, * \rangle \rightarrow request(rproc, \$1)\\
State: & login \rightarrow accepting\_request(login)
\end{array}$$

where variable "\$1" is used to define parametric abstraction rules and, similarly, the wildcard " $*$ " is used for increasing flexibility.

We comment on some of the rules, the remaining ones are interpreted similarly. The first rule prescribes that whenever an action over the endpoint $\mathit{portal}{\bullet}\mathit{creditRequest}$, with any sent data $\langle \mathit{id} \rangle$ matching $\langle \$1 \rangle$, occurs in the label of a transition, then it is replaced by the abstract action $\mathit{request}(\mathit{cr}, \mathit{id})$. Similarly, the second rule prescribes that whenever an action over the endpoint $\mathit{cust}{\bullet}\mathit{offer}$, with any sent data $\langle \mathit{id}, \mathit{offer}, \mathit{motivation} \rangle$ matching $\langle \$1, *, * \rangle$, occurs in the label of a transition, then it is replaced by the abstract action $\mathit{response}(\mathit{cr}, \mathit{id})$. This way, data *offer* and *motivation* are discharged in the 'abstraction process', while session identifier *id* is used to correlate responses from the contacted *Portal* service. To correlate cancellations to the corresponding credit requests, the fifth rule permits replacing actions involving operation *cancel*, with any sent data $\langle \mathit{id} \rangle$ matching $\langle \$1 \rangle$, by abstract action $\mathit{cancel}(\mathit{cr}, \mathit{id})$. The last rule works similarly, but it applies to labels of states rather than to labels of transitions.

Finally, the SocL formulae are checked over the abstract $\mathrm{L}^2\mathrm{TS}$ obtained by applying the previous rules. The overall verification process is supported by CMC. One can thus verify that, as expected, all the abstract properties we presented before do hold for the COWS specification of the finance case study, except for the last two properties, because, e.g., a supervisor can evaluate a credit request negatively. In case a property does not hold, CMC produces a clear and detailed explanation of the returned results, i.e. a so called *counterexample*. For example, Table 2 shows an excerpt of the output returned by CMC when checking the second-last property.

6 Concluding remarks

We have presented an approach to the specification and analysis of SOC systems based on the process calculus COWS and the temporal logic SocL. The design of COWS is inspired to π-calculus, and other similar calculi, and to the web service orchestration language WS-BPEL. In particular, activities *invoke* and *receive* allow communication with other services in a way similar to the homonymous WS-BPEL activities, while activity $\mathbf{kill}(k)$ for forcing termination supports more primitive and flexible forms of fault and compensation handling with respect to WS-BPEL. The case study we have considered, rather than being just a toy example, describes a realistic service-oriented scenario from the financial domain currently under investigation within the EU project SENSORIA. The system and its workflow have been described both informally and by resorting on UML activity diagrams. When moving on the COWS specification, the complexity of the system has demanded full exploitation of features and expressive capabilities of COWS, like delimitation, pattern-matching, service instances, fault handling and compensation. The COWS specification has been used as a model for analyzing and checking behavioral properties of the service system. The expected properties have been formalized by using SocL and then automatically checked by using the model checker CMC.

As noted in the Introduction, COWS is part of a large family of service-oriented

```
The Formula: "AG [request(cr,$id)] AF {not cancel(cr,%id) or response(cr,%id)}true"
is: FALSE
(states generated= 255, computations fragments generated= 403)
cmc> ------------------------------------------
The formula:  AG  [ request(cr,$id) ] AF {not cancel(cr,%id) or response(cr,%id)} true
   is FOUND_FALSE in State C1
 ...
  C9  -->  C10 { portal.creditRequest!<sessionID#1#>,
                 portal.creditRequest?<sessionID#1#> }
               {{  {{ request(cr,sessionID#1#) }} }}
 and the formula:  AF {not cancel(cr,sessionID#1#) or response(cr,sessionID#1#)} true
   is FOUND_FALSE in State C10
because
  C10  -->  C12 { portal.getCreditRequest!<sessionID#1#,data,15000,customer>,
                  portal.getCreditRequest?<sessionID#1#,CUST_DATA,AMOUNT,CUST> }
                  {{  {{  }} }}
  C12  -->  C14 { portal.securities!<sessionID#1#,secValues>,
                  portal.securities?<sessionID#1#,SEC_DATA> }
                  {{  {{ request(sec,sessionID#1#) }} }}
  C14  -->  C17 { portal.balance!<sessionID#1#,balance>,
                  portal.balance?<sessionID#1#,BALANCE> }
                  {{  {{ request(bal,sessionID#1#) }} }}
  C17  -->  C23 { validation.validateBalance!<sessionID#1#,portal,balance>,
                  validation.validateBalance?<ID,BANK,BALANCE> }
                  {{  {{  }} }}
  C23  -->  C28 { portal.validateBalance!<sessionID#1#,yes>,
                  portal.validateBalance?<sessionID#1#,yes> }
                  {{  {{ response(beval,sessionID#1#) }} }}
  C31  -->  C112 { portal.reqProcessing#1#!<sessionID#1#,data,secValues,balance,15000,customer>,
                   portal.reqProcessing#1#?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST> }
                   {{  {{  }} }}
  C112  -->  C244 { portal.empEvaluation!<sessionID#1#,rating,additionalInfo,no>,
                    portal.empEvaluation?<sessionID#1#,RATING,ADDITIONAL_INFO,DECISION> }
                    {{  {{ fail(eeval,sessionID#1#) }} }}
  C249  -->  C251 { customer.negativeResp!<sessionID#1#,additionalInfo>,
                    customer.negativeResp?<sessionID#1#,MOTIVATIONS> }
                    {{  {{ fail(cr,sessionID#1#) }} }}
------------------------------------------
```

Table 2
CMC: a counterexample

process calculi, like the ones developed in [8,7,15,13,9,14,5,6,26]. A peculiar feature of COWS with respect to most of the other calculi is its proximity to WS-BPEL. Despite of this fact, it is still an open question which are the relative (if not absolute) advantages of the various formalisms and which calculi are more suitable for specification and analysis of service-oriented systems. A promising way of comparing these calculi is to analyze how expressive they are and how naturally they can model SOC systems. The present work is a contribution in this direction.

The presented case study has also been considered in [12], where it has been used to illustrate a new workflow-based approach to business process modelling, and in [4], where it has been used to illustrate use of a behavioural equivalence over Petri nets for service replaceability. The selected scenarios, however, were simpler and therefore less challenging to model and analyse than the one presented herein. Also, our model-checking analysis of behavioral properties of a service system is, to the best of our knowledge, the first of its kind.

The case study we considered has been initially modelled by UML4SOA; then, we have elaborated an handmade, ad hoc translation in COWS. We leave for future work the definition of systematic and (semi-)automatic translations of UML diagrams into COWS. To the best of our knowledge, only the core of UML 2.0 activity diagrams is provided with a semantics (see, e.g., [27]) and, in particular, no semantics is yet defined for UML4SOA profile and UML compensation activities. We aim, firstly, at conservatively extending the semantics of UML4SOA activity diagrams, and, then, at providing a general translation technique from UML4SOA activity diagrams into COWS which is sound with respect to the semantics.

On the opposite direction, another line of research we want to explore is to move from COWS terms to *implementations*. A possibility could be to capitalize on the affinity that COWS shares with WS-BPEL by defining a translation of COWS terms into WS-BPEL programs. An alternative possibility could be to develop a (likely Java-based) running environment for supporting execution of COWS terms.

Acknowledgement

We would like to thank Lucia Acciai for discussions that contributed to the development of the finance case study.

References

[1] CMC: an on-the-fly model checker and interpreter for COWS. Available at: http://fmt.isti.cnr.it/cmc/.

[2] Uml4soa. Available at http://www.poplarsoftwaredesign.com/Poplar/UML4SOA.html.

[3] J. Bauer, F. Nielson, H.R. Nielson, and H. Pilegaard. Relational analysis of correlation. In *SAS*, volume 5079 of *LNCS*, pages 245–256. Springer, 2008.

[4] F. Bonchi, A. Brogi, S. Corfini, and F. Gadducci. Compositional specification of web services via behavioural equivalence of nets: A case study. In Kees M. van Hee and Rüdiger Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2008.

[5] M. Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F.S. de Boer, editors, *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[6] R. Bruni, I. Lanese, H.C. Melgratti, and E. Tuosto. Multiparty sessions in soc. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION2008*, volume 5052 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[7] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *LNCS*, pages 63–81. Springer, 2006.

[8] M.J. Butler, C.A.R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 133–150. Springer, 2005.

[9] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[10] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[11] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, *LNCS*. Springer, 2008. To appear.

[12] S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, Policies and Workflows. In *Proc. 3rd Int. Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition*, 2007.

[13] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[14] I. Lanese et al. Disciplining orchestration and conversation in service-oriented computing. In *SEFM'07*, pages 305–314. IEEE Computer Society, 2007.

[15] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[16] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, DSI, Univ. Firenze, 2007. Available at http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf. An extended abstract appeared in *Proc. of ESOP'07*, LNCS 4421, pages 33-47, Springer.

[17] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.

[18] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. Technical report, DSI, Univ. Firenze, 2008. Available at http://rap.dsi.unifi.it/cows/papers/blite_full.pdf. An extended abstract appeared in *Proc. of COORDINATION'08*, LNCS 5052, pages 199-215, Springer.

[19] L.G. Meredith and S. Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[21] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0., April 2007. Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[22] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.

[23] R. Pugliese, F. Tiezzi, and N. Yoshida. On observing dynamic prioritised actions in SOC. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2008. Available at http://rap.dsi.unifi.it/cows/bis4cows-full.pdf.

[24] Sensoria. Software engineering for service-oriented overlay computers. http://www.sensoria-ist.eu/.

[25] F. van Breugel and M. Koshkina. Models and verification of BPEL. Technical report, Department of Computer Science and Engineering, York University, 2006. Available at: http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

[26] H.T. Vieira, L. Caires, and J. Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *ESOP2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[27] V. Vitolins and A. Kalnins. Semantics of uml 2.0 activity diagram for business modeling by means of virtual machine. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, pages 181–194. IEEE Computer Society, 2005.

Appendix

We report here the 'machine readable' syntax of CMC and the complete specification of the finance case study, together with the SocL formulation of the properties we have checked, written using such syntax.

CMC syntax

The syntax accepted by CMC is presented in Table 3. *Killer labels* (ranged over by `k`, `k'`, ...) start with lower case letters, and can only be used as argument of kill activities; *variables* (ranged over by `X`, `Y`, ...) start with capital letters; *service identifiers* (ranged over by `A`, `A'`, ...) start with capital letters and each of them has a fixed non-negative arity; *names* (ranged over by `n`, `m`,...,`p`,`p'`,...,`o`,`o'`, ...) start with lower case letters; *values* (ranged over by `v`, `v'`, ...) are either integer numbers, booleans, or names; *identifiers* (ranged over by `u`, `u'`, ...) are either variables or names. The arguments of a receive-guarded choice must be receive activities. The expression operators `+` and `=` are defined as follows: if both e_1 and e_2 are evaluated as integer numbers then the evaluation of $e_1 + e_2$ returns the integer number corresponding to their sum, otherwise it returns the name corresponding to their concatenation; if both e_1 and e_2 are evaluated as values then the evaluation of $e_1 = e_2$ returns the boolean `true` if these values are the same value, otherwise it returns the boolean `false`.

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

CMC specification of the case study

The complete specification of the finance case study written in the syntax of CMC is as follows.

```
let

  Portal(key,authentication,notAuthorized,authorized,createInst) =
    * [USER] [PWD] [CUST] portal.login?<USER,PWD,CUST>.
      (portal.authentication!<USER,PWD>
         | portal.notAuthorized?<USER>. CUST.failedLogin!<key>
         +
         portal.authorized?<USER>.
         [sessionID#] (CUST.logged!<key,sessionID>
```

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

Table 3
CMC syntax

```
          | portal.creditRequest?<sessionID>.
            portal.createInst!<sessionID>
            + portal.bankTransferRequest?<sessionID>. nil
--          + ...other services provided by the credit portal...
         )
       )


 Authentication(authentication,notAuthorized,authorized) =
    * [USER] [PWD] portal.authentication?<USER,PWD>.
        [nonDet#] [choice#](nonDet.choice!<>
          | nonDet.choice?<>. portal.notAuthorized!<USER>
          + nonDet.choice?<>. portal.authorized!<USER>
        )

 Customer(key,username,password,amount,amountRevised)  =
    [k] (portal.login!<username,password,customer>
         | [ID] (customer.failedLogin?<key>.nil
            + customer.logged?<key,ID>.
```

```
               ( portal.creditRequest!<ID>
                 |
                -- at any time the customer could require
                -- the cancellation of the credit request processing
                   [exit#] (customer.exit!<> | customer.exit?<>.
                     ({portal.cancel!<ID>} | kill(k)))
                    |
                    portal.getCreditRequest!<ID,customerData,amount,customer>
                    | portal.securities!<ID,securityValues>
                    | portal.balance!<ID,balance>
                    | customer.balanceNotValid?<ID>.
                    -- balance not valid (1st time)
                    (portal.balance!<ID,balanceRevised>
                      | customer.balanceNotValid?<ID>.
                      -- balance not valid (2nd time): terminates
                      kill(k)
                    )
                    |
                   [OFFER] [MOTIVATIONS]
                     ( -- receives a negative response
                       customer.negativeResp?<ID,MOTIVATIONS>. kill(k)
                       +
                       -- receives an offer
                       customer.offer?<ID,OFFER,MOTIVATIONS>.
                         [nonDet#] [choice#](nonDet.choice!<>
                           | nonDet.choice?<>. (kill(k) | {portal.answer!<ID,yes>})
                           + nonDet.choice?<>. (kill(k) | {portal.answer!<ID,no>})
                         )
                       +
                       -- updating required
                       customer.update?<ID,MOTIVATIONS>.
                         (portal.updAnswer!<ID,yes>
                         | portal.updAmount!<ID,yes>
                             | portal.newAmount!<ID,amountRevised>
                         | portal.updSecurities!<ID,no>
                         | [OFFER] [MOTIVATIONS]
                          (customer.negativeResp?<ID,MOTIVATIONS>. kill(k)
                           +
                           customer.offer?<ID,OFFER,MOTIVATIONS>.
                             [nonDet#] [choice#](nonDet.choice!<>
                               | nonDet.choice?<>. (kill(k)
                                                      | {portal.answer!<ID,yes>})
                               + nonDet.choice?<>. (kill(k)
                                                      | {portal.answer!<ID,no>})
                             )
                             +
                             customer.update?<ID,MOTIVATIONS>. kill(k)
                           )
                         )
                       )
                     )
                   )
                )


InformationUpload(createInst,reqProcessing)  =
   * [ID] portal.createInst?<ID>.
        [k] [fault#] [abort#]
        (
         [abortFault]
         (
          [CUST_DATA] [SEC_DATA] [FINAL_BALANCE] [AMOUNT] [CUST]
          portal.getCreditRequest?<ID,CUST_DATA,AMOUNT,CUST>.
            [par#] [end#]
            ( -- Activities 1)
              portal.securities?<ID,SEC_DATA>. par.end!<>
              |
              -- Activities 2)
              [repeat#] [loop#]
              ( repeat.loop!<>
                | * repeat.loop?<>.
                    [BALANCE] portal.balance?<ID,BALANCE>.
                       -- invoke validation service
                       (validation.validateBalance!<ID,portal,BALANCE>
                         | portal.validateBalance?<ID,no>.
                         -- notify the customer that balances
                         -- are not valid and cycles
                            ( CUST.balanceNotValid!<ID> | repeat.loop!<> )
                           +
                           portal.validateBalance?<ID,yes>. par.end!<BALANCE>
                    )
              )
              |
              -- Activities 1) and 2) terminates successfully
              par.end?<>. par.end?<FINAL_BALANCE>.
                -- invokes RequestProcessing
                (kill(k) | {portal.reqProcessing!<ID,CUST_DATA,
                                  SEC_DATA,FINAL_BALANCE,AMOUNT,CUST>})
             )
          | portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
         )
         |
         -- fault handler
         fault.abort?<>. nil
       )


InformationUpdate(reqProcessing,reqUpdate) =
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT] [CUST] [MOTIVATIONS]
     portal.reqUpdate?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST,MOTIVATIONS>.
        [k] [fault#] [abort#]
        (
         [abortFault] [NEW_AMOUNT] [NEW_SEC_DATA] [assign#] [ment#] [par#] [end#]
         ( -- notifies the customer of needing to update the data
          customer.update!<ID,MOTIVATIONS>
          | (portal.updAnswer?<ID,no>. kill(k)
             + portal.updAnswer?<ID,yes>.
               ( ( -- updates the amount
                  portal.updAmount?<ID,no>. (assign.ment!<AMOUNT>
                    | assign.ment?<NEW_AMOUNT>. par.end!<>)
                  + portal.updAmount?<ID,yes>.
                      portal.newAmount?<ID,NEW_AMOUNT>. par.end!<>
                 )
                 |
                 ( -- updates the securities
                  portal.updSecurities?<ID,no>. (assign.ment!<SEC_DATA>
                    | assign.ment?<NEW_SEC_DATA>. par.end!<>)
                  + portal.updSecurities?<ID,yes>.
                      portal.newSecurities?<ID,NEW_SEC_DATA>. par.end!<>
                 )
                 |
                 -- Updating terminated
                 par.end?<>. par.end?<>.
                   -- invokes RequestProcessing
                   (kill(k) | {portal.reqProcessing!<ID,CUST_DATA,
                                 NEW_SEC_DATA,BALANCE,NEW_AMOUNT,CUST>})
               )
            )
          | portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
         )
         |
         -- fault handler
         fault.abort?<>. nil
        )


RequestProcessing(reqProcessing,reqUpdate,contractProcessing) =
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT] [CUST]
     portal.reqProcessing?<ID,CUST_DATA,SEC_DATA,BALANCE,AMOUNT,CUST>.
        [k] [fault#] [abort#] [undo#]
        (
         [abortFault]
         (
          -- adds request to employee task list
          portal.addToETL!<ID,SEC_DATA,BALANCE,AMOUNT>
          | portal.taskAddedToETL?<ID>.
             (
              -- installs the compensation handler
              {portal.undo?<empTaskList>. portal.removeTaskETL!<ID>}
              |
              -- receives evaluation from an employee
              [RATING] [ADDITIONAL_INFO] [DECISION]
                portal.empEvaluation?<ID,RATING,ADDITIONAL_INFO,DECISION>.
                  [cond#] [choice#] (
                    cond.choice!<DECISION>
                    |
                    -- 1) negative evaluation
                    cond.choice?<no>.
                      (kill(k) | {CUST.negativeResp!<ID,ADDITIONAL_INFO>})
                    +
                    -- 2) ask to update
                    cond.choice?<update>. (kill(k)
                    | {portal.reqUpdate!<ID,CUST_DATA, SEC_DATA,
                                         BALANCE,AMOUNT,CUST,ADDITIONAL_INFO>})
                    +
                    -- 3) positive evaluation
                    cond.choice?<yes>.
                      ( -- adds request to supervisor task list
                       portal.addToSTL!<ID,SEC_DATA,BALANCE,AMOUNT,ADDITIONAL_INFO>
                       | portal.taskAddedToSTL?<ID>.
                         ( -- installs the compensation handler
                          {portal.undo?<supTaskList>. portal.removeTaskSTL!<ID>}
                          |
                          -- receives evaluation from a supervisor
                          [OFFER] [MOTIVATIONS] [SUP_DECISION]
                            portal.supEvaluation?<ID,OFFER,
                                                  MOTIVATIONS,
                                                  SUP_DECISION>.
                             [cond#] [choice#] (
                               cond.choice!<SUP_DECISION>
                               |
                               -- 1) negative evaluation
                               cond.choice?<no>.
                                 (kill(k) | {CUST.negativeResp!<ID,MOTIVATIONS>})
                               +
                               -- 2) ask to update
                               cond.choice?<update>.
                                 (kill(k) | {portal.reqUpdate!<ID,CUST_DATA,
                                               SEC_DATA,BALANCE,AMOUNT,
                                               CUST,MOTIVATIONS>})
                               +
                               -- 3) positive evaluation
                               cond.choice?<yes>.
                                  ( -- sends the unrated offer to the customer
                                   CUST.offer!<ID,OFFER,MOTIVATIONS>
                                   | -- receives customer's answer
                                     (portal.answer?<ID,yes>.
                                         (kill(k)
                                          | {portal.contractProcessing!<ID,
                                              CUST_DATA, SEC_DATA, BALANCE,
                                              AMOUNT, CUST, RATING,
                                              ADDITIONAL_INFO, OFFER,
                                              MOTIVATIONS>}
                                         )
                                      + portal.answer?<ID,no>. kill(k)
                                     )
                                  )
                             )
                         )
                    )
               )
            )
           |
           -- receive cancellation from customer
           portal.cancel?<ID>. (kill(abortFault) | {fault.abort!<>})
          )
          |
          -- fault handler
          fault.abort?<>.
            -- compensateAll
            (portal.undo!<empTaskList> | portal.undo!<supTaskList> )
         )


ContractProcessing(contractProcessing)=
   * [ID] [CUST_DATA] [SEC_DATA] [BALANCE] [AMOUNT]
     [CUST] [RATING] [ADDITIONAL_INFO] [OFFER] [MOTIVATIONS]
     portal.contractProcessing?<ID,CUST_DATA,SEC_DATA,BALANCE,
       AMOUNT,CUST,RATING,ADDITIONAL_INFO,OFFER,MOTIVATIONS>.
       -- ... contract processing ...
       nil


ValidationService  =
   * [ID] [BANK][BALANCE]
     validation.validateBalance?<ID,BANK,BALANCE>.
     [nonDet#] [choice#](nonDet.choice!<>
                        | nonDet.choice?<>. BANK.validateBalance!<ID,yes>
                          + nonDet.choice?<>. BANK.validateBalance!<ID,no> )
```

```
 EmployeeTaskList =
    * [ID] [SEC_DATA] [BALANCE] [AMOUNT]
      portal.addToETL?<ID,SEC_DATA,BALANCE,AMOUNT>.
        ( portal.taskAddedToETL!<ID>
          | [EMP] (portal.askTaskETL?<EMP>.
                     EMP.getTaskETL!<ID,SEC_DATA,BALANCE,AMOUNT>
                   +
                   portal.removeTaskETL?<ID>. nil
                  )
        )


 Employee(employee) =
    [repeat#] [loop#]
      ( repeat.loop!<>
        | * repeat.loop?<>.
            ( portal.askTaskETL!<employee>
              | [ID] [SEC_DATA] [BALANCE] [AMOUNT]
                employee.getTaskETL?<ID,SEC_DATA,BALANCE,AMOUNT>.
                   -- ... evaluates the request ...
                   [nonDet#] [choice#](nonDet.choice!<>
                     | -- sends the evaluation
                     nonDet.choice?<>.
                       (portal.empEvaluation!<ID,rating,additionalInfo,yes>
                         | repeat.loop!<>)
                     + nonDet.choice?<>.
                       (portal.empEvaluation!<ID,rating,additionalInfo,no>
                         | repeat.loop!<>)
                     + nonDet.choice?<>.
                       (portal.empEvaluation!<ID,rating,additionalInfo,update>
                         | repeat.loop!<>)
                   )
            )
      )


 SupervisorTaskList =
    * [ID] [SEC_DATA] [BALANCE] [AMOUNT] [ADDITIONAL_INFO]
      portal.addToSTL?<ID,SEC_DATA,BALANCE,AMOUNT,ADDITIONAL_INFO>.
        (portal.taskAddedToSTL!<ID>
         | [SUP] (portal.askTaskSTL?<SUP>. SUP.getTaskSTL!<ID,SEC_DATA,
                                             BALANCE,AMOUNT,
                                             ADDITIONAL_INFO>
                  +
                  portal.removeTaskSTL?<ID>. nil
                 )
        )


 Supervisor(supervisor) =
    [repeat#] [loop#]
      ( repeat.loop!<>
        | * repeat.loop?<>.
            ( portal.askTaskSTL!<supervisor>
              | [ID] [SEC_DATA] [BALANCE] [AMOUNT] [INFO]
                supervisor.getTaskSTL?<ID,SEC_DATA,BALANCE,AMOUNT,INFO>.
                   -- ... evaluates the request ...
                   [nonDet#] [choice#](nonDet.choice!<>
                      | -- sends the evaluation
                      nonDet.choice?<>.
                        (portal.supEvaluation!<ID,offer,motivations,yes>
                          | repeat.loop!<>)
                      + nonDet.choice?<>.
                        (portal.supEvaluation!<ID,offer,motivations,no>
                          | repeat.loop!<>)
                      + nonDet.choice?<>.
                        (portal.supEvaluation!<ID,offer,motivations,update>
                          | repeat.loop!<>)
                   )
            )
      )


in
 [key#]
 ( Customer(key,francesco,sensoria,15000,10000)
   | [createInst#] [reqProcessing#] [reqUpdate#] [contractProcessing#]
     ( [authentication#] [notAuthorized#] [authorized#] (
         Portal(key,authentication,notAuthorized,authorized,createInst)
         | Authentication(authentication,notAuthorized,authorized) )
       | InformationUpload(createInst,reqProcessing)
       | InformationUpdate(reqProcessing,reqUpdate)
       | RequestProcessing(reqProcessing,reqUpdate,contractProcessing)
       | ContractProcessing(contractProcessing)
       | EmployeeTaskList()
       | SupervisorTaskList()
     )
 )
 | ValidationService()
 | Employee(employee)
 | Supervisor(supervisor)
end
```

Abstraction rules

The abstraction rules used for our analysis are the following.

```
Abstractions {
    Action creditRequest<$1> -> request(cr,$1)
    Action balanceNotValid<$1> -> fail(cr,$1)
    Action negativeResp<$1,*> -> fail(cr,$1)
    Action offer<$1,*,*> -> response(cr,$1)
    Action update<$1,*> -> fail(cr,$1)
    Action cancel<$1> -> cancel(cr,$1)
    Action supEvaluation<$1,*,*,yes> -> response(seval,$1)
    Action supEvaluation<$1,*,*,no> -> fail(seval,$1)
    Action empEvaluation<$1,*,*,yes> -> response(eeval,$1)
    Action empEvaluation<$1,*,*,no> -> fail(eeval,$1)
    Action validateBalance<$1,yes> -> response(beval,$1)
    Action validateBalance<$1,no> -> fail(beval,$1)
    Action taskAddedToETL<$1> -> request(eval,$1)
    Action taskAddedToSTL<$1> -> request(eval,$1)
    Action removeTaskSTL<$1> -> cancel(eval,$1)
    Action removeTaskETL<$1> -> cancel(eval,$1)
    Action taskAddedToSTL<$1> -> request(tostl,$1)
    Action reqUpdate<$1,*,*,*,*,*,*> -> request(upd,$1)
    Action update<$1,*> -> response(upd,$1)
    Action securities<$1,*> -> request(sec,$1)
    Action balance!<$1,*> -> request(bal,$1)
    Action reqProcessing<$1,*,*,*,*> -> request(rproc,$1)
    State login -> accepting_request(login)
}
```

SocL properties

We report the SocL formulae expressing the properties that the case study is expected to fulfill, written in the syntax of CMC. The difference between the syntax used here and that introduced in Section 5 is that, given a correlation variable `var`, its binding occurrence (i.e. $\underline{\texttt{var}}$) is written `$var`, while its free occurrences `%var`. Moreover, logical operators $\vee$ and $\neg$ are written `or` and `not`, respectively.

```
(Availability) AG accepting_request(login)

(Responsiveness and correlation soundness)
  AG [request(cr,$id)]
     AF {response(cr,%id) or (fail(cr,%id) or cancel(cr,%id))} true

(Interruptibility) AG [request(cr,$id)] EF {cancel(cr,%id)} true

(i) AG [request(cr,$id)]
     not E[true {not response(seval,%id)} U {response(cr,%id)} true]

(ii) AG [request(cr,$id)]
    not E[true {not (fail(seval,%id) or fail(eeval,%id)
                 or fail(beval,%id))} U {fail(cr,%id)} true]

(iii) AG [request(eval,$id)] EF [cancel(cr,%id)]
        AF {cancel(eval,%id)} true

(iv) AG [request(upd,$id)]
       AF {cancel(cr,%id) or response(upd,%id)} true

(v) AG [request(cr,$id)]
    not E[true {not request(sec,%id)
             or request(bal,%id)} U {request(rproc,%id)}true]

(vi) AG [ request(cr,$id) ]
```

```
        AF {not cancel(cr,%id) or response(cr,%id)} true

(vii) AG [request(cr,$id)]
         AF {not cancel(cr,%id) or request(tostl,%id)}true
```