The SENSORIA Approach Applied to the Finance Case Study*

Stefania Gnesi[1], Rosario Pugliese[2], and Francesco Tiezzi[2]

[1] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", ISTI - CNR, Pisa
stefania.gnesi@isti.cnr.it
[2] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
rosario.pugliese@unifi.it,tiezzi@dsi.unifi.it

Abstract. This chapter provides an effective implementation of (part of) the SENSORIA approach, specifically *modelling* and *formal analysis* of service-oriented software based on mathematically founded techniques. The 'Finance case study' is used as a test bed for demonstrating the feasibility and effectiveness of the use of the process calculus COWS and some of its related analysis techniques and tools. In particular, we report the results of an application of a temporal logic and its model checker for expressing and checking functional properties of services and a type system for guaranteeing confidentiality properties of services.

1 Introduction

The SENSORIA approach encompasses the whole development process of service-oriented software, from systems specified in high-level languages to deployment and re-engineering. In fact, as part of the project the partners have developed a large set of languages, methods, techniques and tools that can be applied during the development of service-oriented applications. Each of these project's outcomes has been designed to solve a certain type of problems and is applicable to some specific situations. It is thus difficult to identify the 'best' technique or tool that solves a particular problem arising in the development process.

To shepherd the prospective user through the selection procedure, as a result of a collaboration among several people involved in the project, a *catalogue of patterns* has recently started to be developed (see Chapter 7-5). Several patterns have been already catalogued that address a broad spectrum of SOA engineering aspects such as modelling, specification, analysis, verification, orchestration, deployment. Besides as an index to SENSORIA outcomes, this catalogue serves as a guidance for using them and for better understanding relative advantages and disadvantages.

Since we want to demonstrate the feasibility and effectiveness of the use of the process calculus COWS [**?**] (see also Chapter 2-1), we consider relevant to this chapter those patterns involving process calculi as specification formalisms and their related techniques for qualitative and quantitative analysis, and present solutions to these patterns in terms of COWS and its related analysis techniques and tools. As a test bed, we use the 'Finance case study' (its UML4SOA modelling can be found in Chapter 7-1).

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

This way the chapter provides an effective implementation of (part of) the SENSORIA approach, specifically *modelling* and *formal analysis* of service-oriented software based on mathematically founded techniques.

Hence, this chapter contains the following contributions. Section 2 presents a COWS specification of the Finance case study. Section 3 illustrates a solution to the *Functional Service Verification* pattern, where service behavioural properties are expressed using the temporal logic SocL and verified using the model checker CMC [**?**] (see also Chapters 4-3 and 4-2). Section 4 illustrates a solution to the *Service Specification and Analysis* pattern, where confidentiality properties are checked using the type system of [**?**]. Section 5 reports on an ongoing effort for devising an integrated approach that can lead to verifiable implementations of service components from abstract architectural models of business activities. To this aim we are developing software tools that can provide access to verification functionalities also to users not familiar with formal methods. Section 6 briefly reviews some feedbacks from an application of COWS and its related analysis techniques to the Finance case study.

2 A COWS Specification of the Finance Case Study

In this section, we present a relevant part of a COWS specification modelling the Finance case study (the whole specification is reported in [**?**]). We will gently introduce COWS's syntax and semantics in a step-by-step fashion while commenting upon the specification and refer the interested reader to Chapter 2-1 for a presentation of COWS's syntax and an informal explanation of its semantics.

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

The COWS term representing the overall scenario is

$$\mathit{CreditInstitute} \mid \mathit{RatingProvider} \mid \mathit{BalanceAnalysisProvider} \mid \mathit{SecurityAnalysisProvider} \mid \mathit{Portal}$$

The services above are composed by using the parallel composition operator $\_|\_$ that allows the different components to be concurrently executed and to interact with each other.

CreditInstitute is defined as follows.

$$[customerManagement, creditManagement]\,(\,CreditRequest \mid CustomerManagement \mid CreditManagement\,)$$

The term is the parallel composition of the (considered) subservices of the credit institute. The delimitation operator $[\_]\_$ is used here to declare that *customerManagement* and *creditManagement* are shared partner names known to *CreditRequest*, *CustomerManagement* and *CreditManagement*, and only to them. Basically, this ensures that external services cannot directly interact with *CustomerManagement* and *CreditManagement*, which are indeed 'internal' subservices of *CreditInstitute*. Service *CreditRequest* is publicly invocable and can interact with *Portal* and other external services, other than with the two above internal services.

Hereafter we only focus on service *CreditRequest*, which is defined as follows.

$$\begin{array}{l}
*\,[k, raise, x_{Id}, x_{Name}, x_{Password}] \\
\quad creditReq \cdot initialize?\langle x_{Id}, x_{Name}, x_{Password}\rangle. \\
\quad (\,customerManagement \cdot checkUser!\langle x_{Id}, x_{Name}, x_{Password}\rangle \\
\qquad \mid [x_{UserOK}]\, creditReq \cdot checkUser?\langle x_{Id}, x_{UserOK}\rangle. \\
\qquad\quad (\,\{\!|\, portal \cdot initialize!\langle x_{Id}, x_{UserOK}\rangle \,|\!\}\, \\
\qquad\quad \mid [if, then] \\
\qquad\quad\quad (\,if \cdot then!\langle x_{UserOK}\rangle \\
\qquad\quad\quad \mid if \cdot then?\langle \mathbf{false}\rangle.\,(\,\mathbf{kill}(k) \mid \{\!|\, raise \cdot abort!\langle\rangle \,|\!\}\,) \\
\qquad\quad\quad + if \cdot then?\langle \mathbf{true}\rangle. \\
\qquad\quad\quad\quad (\,customerManagement \cdot getCustomerData!\langle x_{Id}, x_{Name}, x_{Password}\rangle \\
\qquad\quad\quad\quad \mid [x_{LoginName}, x_{FirstName}, x_{LastName}] \\
\qquad\quad\quad\quad\quad creditReq \cdot getCustomerData?\langle x_{Id}, x_{LoginName}, x_{FirstName}, x_{LastName}\rangle. \\
\qquad\quad\quad\quad\quad\quad Main\,)\,)\,)\,)
\end{array}$$

The replication operator $*\,\_$, that spawns in parallel as many copies of its argument term as necessary, is exploited to model the fact that, whenever prompted by a customer request, *CreditRequest* creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Each such instance has a private name k, a reserved partner name *raise* to raise fault signals, and its own copies of variables x_{Id}, x_{Name} and $x_{Password}$. Name k introduces a named scope that groups together all the activities of the instance, making it possible to associate with such scope suitable termination activities, as well as ad hoc fault and compensation handlers. Each interaction with the service starts with a receive activity of the form $creditReq \cdot initialize?\langle x_{Id}, x_{Name}, x_{Password}\rangle$ corresponding to reception of a request emitted by *Portal* on behalf of a customer. The receive activity creates a new service instance and initializes the variables x_{Id}, x_{Name} and $x_{Password}$, declared local to the instance by the delimitation operator, with data provided by a customer. In particular, variable x_{Id} is used to store a fresh datum, generated by *Portal*, univocally identifying a session of

the process (which, in COWS, coincides with an instance of the service). The identifier allows *CreditRequest* to safely communicate with the involved services. In fact, in each interaction among them, 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 the messages to the instances identified by the same datum.

Once created, a *CreditRequest*'s instance requires *CustomerManagement* to check the customer login data, by invoking the operation *checkUser* provided by the 'internal' partner name *customerManagement* through the invoke activity $customerManagement \cdot checkUser!\langle x_{Id}, x_{Name}, x_{Password}\rangle$, and waits for a reply. The answer is forwarded to the customer by means of the invoke activity $portal \cdot initialize!\langle x_{Id}, x_{UserOK}\rangle$. To guarantee eventual execution of this invoke, it is protected by the protection operator $\{\!|\_|\!\}$ that prevents it to be cancelled due to an abrupt termination of its enclosing scope k. Concurrently, by exploiting the receive-guarded choice operator $\_+\_$ and the private names *if* and *then*, the instance can make a conditional choice based on the answer. A negative answer forces the immediate termination of the instance, through the execution of the activity $\mathbf{kill}(k)$, and the emission of an (internal) fault signal $raise \cdot abort!\langle\rangle$. Notice that, in this specific case, the fault signal is not caught and dealt with by any fault handler. In case of a positive answer, the service instance gets the customer data from *CustomerManagement*, by means of a pair of invoke-receive activities over the operation *getCustomerData*, and activates the term *Main*, which is defined as follows.

$$
\begin{array}{l}
[raise, comp] \\
(\ [k_{Main}] \\
\quad (\ [repeat, until, update, desired] \\
\quad\quad (\ repeat \cdot until!\langle\rangle \\
\quad\quad\quad \mid * repeat \cdot until?\langle\rangle. \\
\quad\quad\quad\quad [k_{loop}] \\
\quad\quad\quad\quad (\ Creation \\
\quad\quad\quad\quad\quad \mid update \cdot desired?\langle \mathbf{true}\rangle.\ (\ \mathbf{kill}(k_{loop}) \mid \{\!|\ repeat \cdot until!\langle\rangle\ |\!\}\ ) \\
\quad\quad\quad\quad\quad\quad + update \cdot desired?\langle \mathbf{false}\rangle.\ Finalize\ )\ ) \\
\quad\quad \mid\ creditReq \cdot cancel?\langle x_{Id}\rangle.\ (\ \mathbf{kill}(k_{Main}) \mid \{\!|\ raise \cdot abort!\langle\rangle\ |\!\}\ )\ ) \\
\quad \mid\ raise \cdot abort?\langle\rangle. \\
\quad\quad [end] \\
\quad\quad (\ comp \cdot creation!\langle creation, end\rangle \\
\quad\quad\quad \mid comp \cdot end?\langle\rangle. \\
\quad\quad\quad\quad (\ comp \cdot handleBalanceAndSecurityData!\langle handleBalanceAndSecurityData, end\rangle \\
\quad\quad\quad\quad\quad \mid comp \cdot end?\langle\rangle.\ portal \cdot abortProcess!\langle x_{Id}\rangle\ )\ )\ ) \\
\quad \mid * [x, y]\ comp \cdot creation?\langle x, y\rangle.\ comp \cdot y!\langle\rangle \\
\quad\quad \mid * [x, y]\ comp \cdot handleBalanceAndSecurityData?\langle x, y\rangle.\ comp \cdot y!\langle\rangle\ )
\end{array}
$$

This term models a 'scope' activity named k_{Main} which is equipped with an event and a fault handler. When the scope starts, the handlers are enabled. The event handler (highlighted by a dark gray background) is activated by an invocation of the operation *cancel*; this forces the immediate termination of all (unprotected) activities representing the normal behaviour of the scope, by means of activity $\mathbf{kill}(k_{Main})$, and the execution

of activity $raise \cdot abort!\langle\rangle$, which activates the fault handler. Then, the fault handler (highlighted by a light gray background) sends two compensation signals along endpoints of the form $comp \cdot scopeName$, where *scopeName* is replaced by *creation* and *handleBalanceAndSecurityData*, and terminates by sending a message notifying the customer that the process has correctly aborted. The private name *end* permits sequentializing the above activities. It is worth noticing that, if no compensation handler has yet been installed, the compensation activities have to immediately terminate without doing nothing. To this aim, the two (replicated) receive activities $comp \cdot scopeName?\langle x, y\rangle$ catch the compensation signals, only if no compensation handlers are ready to do so[1], and reply with the corresponding termination signals.

The normal behaviour of the scope consists of a repeat-until loop, implemented by using the replication operator together with the private names *repeat* and *until*. At each iterative step, the term *Creation* is executed which upon termination allows a conditional choice to be taken: if the customer has requested an update (i.e. activity $update \cdot desired?\langle \mathbf{true}\rangle$ is executed), the remaining activities of the current iterative step are stopped (by the activity $\mathbf{kill}(k_{loop})$) and the loop is restarted (by the signal $repeat \cdot until!\langle\rangle$); if no update has been requested (i.e. activity $update \cdot desired?\langle \mathbf{false}\rangle$ is executed), the term *Finalize* is activated.

Finalize is simply the invoke activity

$$portal \cdot goodbye!\langle x_{Id}\rangle$$

that informs the portal that the process is concluded.

Creation is defined as follows.

$$
\begin{array}{l}
[x_{CustomerId}, x_{CreditAmount}, x_{CreditType}, x_{MonthlyInstalment}] \\
creditReq \cdot createNewCreditRequest?\langle x_{Id}, x_{CustomerId}, x_{CreditAmount}, x_{CreditType}, x_{MonthlyInstalment}\rangle. \\
(\ creditManagement \cdot initCreditData!\langle x_{Id}, x_{CustomerId}, x_{CreditAmount}, x_{CreditType}, x_{MonthlyInstalment}\rangle \\
\quad \mid creditReq \cdot initCreditData?\langle x_{Id}\rangle. \\
\quad\quad (\ portal \cdot createNewCreditRequest!\langle x_{Id}, working\rangle \\
\quad\quad\quad \mid HandleBalanceAndSecurityData \\
\quad\quad\quad \mid [x_{End}]\, \{\!|\, comp \cdot creation?\langle creation, x_{End}\rangle. \\
\quad\quad\quad\quad\quad (\ creditManagement \cdot removeData!\langle x_{Id}\rangle \\
\quad\quad\quad\quad\quad\quad \mid creditReq \cdot removeData?\langle x_{Id}\rangle.\, comp \cdot x_{End}!\langle\rangle\ )\ |\!\}\ )\ )\ )
\end{array}
$$

After the data for a new credit request have been received, the service forwards them to the credit management service and waits for an acknowledgement. Then, it replies to the portal to notify that the system is working on the request, activates the term *HandleBalanceAndSecurityData*, and installs a compensation handler for undoing the activities previously performed along the operation *initCreditData*. The compensation handler (highlighted by a gray background) is a protected term waiting for a compensation request, i.e. a signal along $comp \cdot creation$. When this signal is received, the compensation handler becomes active and invokes the operation *removeData* provided by *creditManagement*.

[1] Indeed, because of the semantics of parallel composition, the receives $comp \cdot scopeName?\langle x, y\rangle$ are assigned a lower priority than that assigned to the receives $comp \cdot scopeName?\langle scopeName, x_{End}\rangle$ performed by the compensation handlers.

HandleBalanceAndSecurityData is defined as follows.

$$
\begin{array}{l}
[\mathit{flow}, \mathit{end}] \\
(\ (\ \mathit{portal} \cdot \mathit{enterBalanceData}!\langle x_{Id} \rangle \\
\quad \mid [x_{BalancePackage}] \\
\quad\ \mathit{creditReq} \cdot \mathit{enterBalanceData}?\langle x_{Id}, x_{BalancePackage} \rangle . \\
\quad\quad (\ \mathit{balance} \cdot \mathit{updateBalanceRating}!\langle x_{Id}, x_{LoginName}, x_{FirstName}, x_{LastName}, x_{BalancePackage} \rangle \\
\quad\quad\quad \mid \mathit{creditReq} \cdot \mathit{updateBalanceRating}?\langle x_{Id} \rangle . \ \mathit{flow} \cdot \mathit{end}!\langle \rangle\ )\ ) \\
\quad \mid (\ \mathit{portal} \cdot \mathit{enterSecurityData}!\langle x_{Id} \rangle \\
\quad\quad \mid [x_{SecurityPackage}] \\
\quad\quad\ \mathit{creditReq} \cdot \mathit{enterSecurityData}?\langle x_{Id}, x_{SecurityPackage} \rangle . \\
\quad\quad\quad (\ \mathit{security} \cdot \mathit{updateSecurityRating}!\langle x_{Id}, x_{LoginName}, x_{FirstName}, x_{LastName}, x_{SecurityPackage} \rangle \\
\quad\quad\quad\quad \mid \mathit{creditReq} \cdot \mathit{updateSecurityRating}?\langle x_{Id} \rangle . \ \mathit{flow} \cdot \mathit{end}!\langle \rangle\ )\ ) \\
\quad \mid \mathit{flow} \cdot \mathit{end}?\langle \rangle . \\
\quad\ \mathit{flow} \cdot \mathit{end}?\langle \rangle . \\
\quad\ (\ \mathit{RatingCalculation} \\
\quad\quad \mid [x_{End}] \\
\quad\quad\ \{\!|\ \mathit{comp} \cdot \mathit{handleBalanceAndSecurityData}?\langle \mathit{handleBalanceAndSecurityData}, x_{End} \rangle . \\
\quad\quad\quad [\mathit{completed}] \\
\quad\quad\quad (\ \mathit{balance} \cdot \mathit{clearData}!\langle x_{Id} \rangle \mid \mathit{creditReq} \cdot \mathit{clearData}?\langle x_{Id}, b \rangle . \ \mathit{comp} \cdot \mathit{completed}!\langle \rangle \\
\quad\quad\quad\quad \mid \mathit{security} \cdot \mathit{clearData}!\langle x_{Id} \rangle \mid \mathit{creditReq} \cdot \mathit{clearData}?\langle x_{Id}, s \rangle . \ \mathit{comp} \cdot \mathit{completed}!\langle \rangle \\
\quad\quad\quad\quad \mid \mathit{comp} \cdot \mathit{completed}?\langle \rangle . \ \mathit{comp} \cdot \mathit{completed}?\langle \rangle . \ \mathit{comp} \cdot x_{End}!\langle \rangle\ )\ |\!\}\ )\ )
\end{array}
$$

It requires the customer to enter (in parallel) balance and security data and, then, sends them to the *balance* and *security* services that store such data and, when requested, will compute the corresponding ratings. When the parallel computation ends, i.e. after that two signals along $\mathit{flow} \cdot \mathit{end}$ have been consumed, the term *RatingCalculation* is activated and a compensation handler (highlighted by a gray background) for undoing the already executed activities is installed. Notably, since the compensation activities share the same operation name *clearData*, names b and s are used in the receiving activities to distinguish the responses.

RatingCalculation is defined as follows.

$$
\begin{array}{l}
\mathit{rating} \cdot \mathit{calculateRating}!\langle x_{Id}, x_{LoginName}, x_{FirstName}, x_{LastName} \rangle \\
\mid [x_{Result}, x_{RatingData}]\ \mathit{creditReq} \cdot \mathit{calculateRating}?\langle x_{Id}, x_{Result}, x_{RatingData} \rangle . \ \mathit{Decision}
\end{array}
$$

It invokes the service *rating* for getting the rating of the customer request. When an answer is returned, it activates the term *Decision*, which is defined as follows.

$$
\begin{array}{l}
[\mathit{if}, \mathit{then}, \mathit{end}, x_{ManualAcceptance}, x] \\
(\ \mathit{if} \cdot \mathit{then}!\langle x_{Result} \rangle \\
\quad \mid (\ \mathit{if} \cdot \mathit{then}?\langle \mathit{aaa} \rangle . \\
\quad\quad\ [\mathit{var}, \mathit{set}]\ (\ \mathit{var} \cdot \mathit{set}!\langle \mathit{undef} \rangle \mid \mathit{var} \cdot \mathit{set}?\langle x_{ManualAcceptance} \rangle . \ \mathit{approval} \cdot \mathit{end}!\langle \rangle\ ) \\
\quad\quad + \mathit{if} \cdot \mathit{then}?\langle x \rangle . \ \mathit{Approval}\ ) \\
\quad \mid \mathit{approval} \cdot \mathit{end}?\langle \rangle . \\
\quad\quad (\ \mathit{if} \cdot \mathit{then}!\langle x_{Result}, x_{ManualAcceptance} \rangle
\end{array}
$$

$$
\begin{array}{l}
\quad | \, [x_1, x_2, x_3, x_4] \\
\qquad\qquad (\, \mathit{if} \cdot \mathit{then}?\langle \mathit{aaa}, x_1 \rangle . \mathit{Accept} \\
\qquad\qquad\quad + \mathit{if} \cdot \mathit{then}?\langle x_2, \mathbf{true} \rangle . \mathit{Accept} \\
\qquad\qquad\quad + \mathit{if} \cdot \mathit{then}?\langle x_3, x_4 \rangle . \mathit{Decline} \,) \,)
\end{array}
$$

Firstly, it checks the rating result. If it is *aaa*, the service assigns the value *undef* to $x_{ManualAcceptance}$ and skips the approval phase; otherwise, the term *Approval* starts. Then, if the rating result is *aaa* or $x_{ManualAcceptance}$ has been set to **true** (i.e. either $\mathit{if} \cdot \mathit{then}?\langle \mathit{aaa}, x_1 \rangle$ or $\mathit{if} \cdot \mathit{then}?\langle x_2, \mathbf{true} \rangle$ is executed), the term *Accept* is activated; otherwise, *Decline* is executed.

Approval is defined as follows.

$$
\begin{array}{l}
[\mathit{if}, \mathit{then}, x] \\
(\, \mathit{if} \cdot \mathit{then}!\langle x_{Result} \rangle \\
\quad | \, (\, \mathit{if} \cdot \mathit{then}?\langle \mathit{bbb} \rangle . \\
\qquad\quad \mathit{portal} \cdot \mathit{requestClerkApproval}!\langle x_{Id}, x_{RatingData} \rangle \\
\qquad + \mathit{if} \cdot \mathit{then}?\langle x \rangle . \\
\qquad\quad \mathit{portal} \cdot \mathit{requestSupervisorApproval}!\langle x_{Id}, x_{RatingData} \rangle \,) \\
\quad | \, [x_{ApprovalData}] \\
\qquad \mathit{creditReq} \cdot \mathit{approvalResult}?\langle x_{Id}, x_{ManualAcceptance}, x_{ApprovalData} \rangle . \, \mathit{approval} \cdot \mathit{end}!\langle\rangle \,)
\end{array}
$$

It checks if the rating result is equal to *bbb*. In the positive case, it requests a clerk approval, otherwise a supervisor approval. After a response from either a clerk or a supervisor, it terminates the approval phase by sending the signal $\mathit{approval} \cdot \mathit{end}!\langle\rangle$.

Decline is defined as follows.

$$
\begin{array}{l}
\mathit{creditManagement} \cdot \mathit{generateDecline}!\langle x_{Id}, x_{RatingData} \rangle \\
| \, [x_{DeclineData}] \, \mathit{creditReq} \cdot \mathit{generateDecline}?\langle x_{Id}, x_{DeclineData} \rangle . \\
\qquad\qquad (\, \mathit{portal} \cdot \mathit{declineToClient}!\langle x_{Id}, x_{DeclineData} \rangle \\
\qquad\qquad\quad | \, [x_{UpdateDesired}] \, \mathit{creditReq} \cdot \mathit{declineToClient}?\langle x_{Id}, x_{UpdateDesired} \rangle . \\
\qquad\qquad\qquad\qquad \mathit{update} \cdot \mathit{desired}!\langle x_{UpdateDesired} \rangle \,)
\end{array}
$$

It requires the credit management service to generate the decline data and forwards them to the customer. The customer will reply by indicating if he desires or not a data update, and such response will be sent to the main scope (see the definition of the term *Main*) by means of the invoke activity $\mathit{update} \cdot \mathit{desired}!\langle x_{UpdateDesired} \rangle$.

Finally, *Accept* is defined as follows.

$$
\begin{array}{l}
\mathit{creditManagement} \cdot \mathit{generateOffer}!\langle x_{Id}, x_{RatingData} \rangle \\
| \, [x_{AgreementData}] \\
\quad \mathit{creditReq} \cdot \mathit{generateOffer}?\langle x_{Id}, x_{AgreementData} \rangle . \\
\quad (\, \mathit{portal} \cdot \mathit{offerToClient}!\langle x_{Id}, x_{AgreementData} \rangle \\
\qquad | \, [x_{Accepted}] \\
\qquad\quad \mathit{creditReq} \cdot \mathit{offerToClient}?\langle x_{Id}, x_{Accepted} \rangle . \\
\qquad\quad [\mathit{if}, \mathit{then}, \mathit{end}] \\
\qquad\quad (\, \mathit{if} \cdot \mathit{then}!\langle x_{Accepted} \rangle \\
\qquad\qquad | \, \mathit{if} \cdot \mathit{then}?\langle \mathbf{false} \rangle . \, \mathit{update} \cdot \mathit{desired}!\langle \mathbf{false} \rangle \\
\qquad\qquad + \mathit{if} \cdot \mathit{then}?\langle \mathbf{true} \rangle . \\
\qquad\qquad\quad (\, \mathit{creditManagement} \cdot \mathit{acceptOffer}!\langle x_{Id}, x_{Accepted} \rangle \\
\qquad\qquad\qquad | \, \mathit{portal} \cdot \mathit{acceptOffer}?\langle x_{Id} \rangle . \, \mathit{update} \cdot \mathit{desired}!\langle \mathbf{false} \rangle \,) \,) \,)
\end{array}
$$

It behaves similarly to the previous term, except for the fact that an offer is generated instead of a decline and, moreover, the acceptance of the offer by the customer is sent to the credit management service. Notice that, whether the customer accepted or not, the activity $update \cdot desired!\langle \mathbf{false} \rangle$ is executed to indicate to the main scope that the data update is not desired.

3 A Logical Methodology for Checking Functional Properties

In this section, we present a solution to the *Functional Service Verification* pattern (see Chapter 7-5), where service behavioural properties are expressed using the action- and state-based, branching-time, temporal logic SocL and verified using the on-the-fly model checker CMC. Both SocL and CMC are part of a methodology for verifying functional properties of services introduced in [**?**] and also described in Chapters 4-3 and 4-2; there, among many other SENSORIA tools, the tool UMC is also described which is based on the same CMC's underlying computational model but uses UML statecharts, rather than COWS, as an input specification language. Here we briefly report the main ingredients of the logic and refer the interested reader to [**?**] (and Chapter 4-2) for a formal account of the semantics of SocL formulae.

This approach takes an abstract point of view: services are thought of as software entities which may have an internal state and can perform actions, by which they can also interact with each other. A service is thus characterized in terms of states and atomic propositions that are true over them, and of state changes and actions performed when moving from one state to another. Atomic propositions express the potential capability of the service to perform a specific action, i.e. that in a given state the action is enabled.

An action has a *type*, e.g. accept a request, provide a response, etc., and is part of a possibly long-running *interaction* started when a client firstly invokes one of the operations exposed by the service. Thus, according to this view, an interaction identifies a collection of actions, each of them corresponding to a single invocation of a service operation. Since service operations can be independently invoked by several clients, multiple instances of a same interaction can be simultaneously active. To univocally identify an action, *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. SocL atomic propositions have the form $p(i, c)$, where p is the name, while i and c are as above. We will use π as a generic atomic proposition.

For example, action $request(cr, 1234, 1)$ could stand for a *request* action for starting an (instance of the) interaction *cr* which will be identified through the correlation

tuple $\langle 1234, 1\rangle$. A *response* action corresponding to the request above could be written as, e.g. $response(cr, 1234, 1)$. If some correlation value is unknown at design time, e.g. the identifier 1, a (binder for a) correlation variable id can be used instead, as in the action $request(cr, 1234, \underline{id})$. A corresponding response action could be written as $response(cr, 1234, id)$, where the (free) occurrence of the correlation variable id indicates the connection with the action where the variable is bound. Similarly, actions like $cancel(cr, 1234, id)$, $fail(cr, 1234, id)$ and $undo(cr, 1234, id)$ could indicate *cancellation*, *failure* and *compensation* notification for the same request. As regards atomic propositions, $accepting\_request(login)$ indicates that a state can accept requests for interaction $login$, while proposition $accepting\_cancel(cr, 1234, id)$ indicates that a state permits to cancel those requests for interaction cr identified by the correlation tuple $\langle 1234, id\rangle$.

The syntax of SocL formulae is defined as follows:

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

where state formulae are the main syntactic category.

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 operators for negation and conjunction, respectively. As usual, we will use ff to abbreviate $\neg tt$, $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$ and $\phi_1 \Rightarrow \phi_2$ to abbreviate $\neg\phi_1 \vee \phi_2$. 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). E and A are existential and universal (respectively) *path quantifiers*. X and U are the *next* and *(strong) until* operators [**?**], while W is the *weak until* operator [**?**]. Intuitively, the formula $X_\gamma\phi$ says that in the next state of the path, reached by an action satisfying γ, the formula ϕ holds. The formula $\phi \,{}_\chi U_\gamma\, \phi'$ 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. The formula $\phi \,{}_\chi W_\gamma\, \phi'$ holds on a path either if the corresponding formula with 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.

Other useful operators can be derived as usual; those that we use in the sequel are:

- $[\gamma]\,\phi$ stands for $\neg\, EX_\gamma \neg\, \phi$ and means that no matter how a process performs an action satisfying γ, the state it reaches in doing so will *necessarily* satisfy ϕ.
- $EF\phi$ stands for $\phi \vee E(true \,{}_{tt} U_{tt}\, \phi)$ and means that there is some path that leads to a state at which ϕ holds; i.e., ϕ *potentially* holds.
- $EF_\gamma\, \phi$ stands for $E(true \,{}_{tt} U_\gamma\, \phi)$ and means that there is some path that leads to a state at which ϕ holds reached by a last action satisfying γ; if ϕ is *true*, we say that an action satisfying γ will *eventually* be performed.
- $AF_\gamma\, \phi$ stands for $A(true \,{}_{tt} U_\gamma\, \phi)$ and means that an action satisfying γ will be performed in the future along every path and at the reached states ϕ holds; if ϕ is *true*, we say that an action satisfying γ is *inevitable*.

– $AG\phi$ stands for $\neg\, EF \neg\, \phi$ and means that ϕ holds at every state on every path; i.e., ϕ holds *globally*.

Properties of the case study specified with SocL. In [**?**] we have singled out many significant classes of desirable properties of the externally observable behaviour of services. Over the COWS specification presented in Section 2, by using the model checker CMC, we have checked the following properties:

Availability:

$AG(accepting\_request(login))$

This formula means that the service *CreditRequest* is *available*, i.e. it is always capable to accept a login request.

Responsiveness:

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

This formula means that *CreditRequest* is *responsive*, i.e. it always guarantees an answer (i.e. an offer or a decline, sent by means of action $response(cr, id)$) to each received credit request, unless the customer cancels his own request (by means of action $cancel(cr, id)$). The answer from *CreditRequest* and the request of cancellation from *Portal* belong to the same interaction cr of the credit request and are properly *correlated* by variable id.

Interruptibility:

$AG\,[request(cr, \underline{id})]\,A(accepting\_cancel(cr, id)\,_{tt}\,U_{cancel(cr,id)\vee response(cr,id)}\,true)$

The system can accept a cancellation of a credit request, after that the customer has sent his credit request and until he cancels the request or receives an answer.

Compensability:

$AG\,[request(rating, \underline{id})]\,EF_{cancel(cr,id)}\,AF_{undo(cr,id)}\,AF_{undo(cr,id)}\,AF_{undo(cr,id)}\,true$

We want to ensure that if a cancellation is requested after the rating calculation has started, then all compensation activities of services *balance*, *security* and *creditManagement* will be executed. Each such compensation corresponds to performing action $undo(cr, id)$. Thus, by exploiting the fact that any compensation activity can be executed at most once (this can be easily checked separately for each compensation activity), we require all computations after a cancellation to contain three occurrences of $undo(cr, id)$.

Fault handling:

$AG\,(raising\_abort(cr)\;\Rightarrow\;AF_{fail(cr)}\,true)$

Whenever an abort exception is raised (atomic proposition $raising\_abort(cr)$), the failure is notified to the customer (by means of action $fail(cr)$).

Model checking SocL *formulae.* The formulae presented in Section 3 are stated in terms of *abstract* actions and atomic propositions, meaning that, e.g., a credit is requested or the system is ready to accept a login. In other words, the properties we want to verify are formalized as SocL formulae in a completely independent way of the service specification. This is a key feature of the verification methodology introduced in [**?**]. To perform the verification, these formulae must be tailored to the COWS specification of Section 2 that is expressed in terms of *concrete* actions, i.e. communication of data tuples along endpoints. This is done by defining an appropriate set of *abstraction rules*

that relate the actions in the specification to the actions and atomic propositions in the SocL formulae. These rules are provided as an input to CMC, together with the COWS specification and the SocL formula to be checked, and are used by the tool to transform the labels of the Doubly Labelled Transition System (L^2TS) corresponding to the COWS specification during its on-the-fly generation. It is worth noticing that in the L^2TS corresponding to a COWS term, each transition is labelled with the actions performed when moving from the source state to the target one, while each state is labelled with the actions enabled in that state. CMC supports the overall verification process.

The abstraction rules we have used for our analysis are

$$\begin{aligned}
\mathit{Action}\ \mathit{createNewCreditRequest}\langle \$id, *, *, *, * \rangle &\rightarrow \mathit{request}(cr, \$id)\\
\mathit{Action}\ \mathit{offerToClient}\langle \$id, * \rangle &\rightarrow \mathit{response}(cr, \$id)\\
\mathit{Action}\ \mathit{declineToClient}\langle \$id, * \rangle &\rightarrow \mathit{response}(cr, \$id)\\
\mathit{Action}\ \mathit{cancel}\langle \$id \rangle &\rightarrow \mathit{cancel}(cr, \$id)\\
\mathit{Action}\ \mathit{calculateRating}\langle \$id, *, *, * \rangle &\rightarrow \mathit{request}(\mathit{rating}, \$id)\\
\mathit{Action}\ \mathit{clearData}\langle \$id \rangle &\rightarrow \mathit{undo}(cr, \$id)\\
\mathit{Action}\ \mathit{removeData}\langle \$id \rangle &\rightarrow \mathit{undo}(cr, \$id)\\
\mathit{Action}\ \mathit{abortProcess} &\rightarrow \mathit{fail}(cr)\\
\mathit{State}\ \mathit{abort}! &\rightarrow \mathit{raising_abort}(cr)\\
\mathit{State}\ \mathit{initialize}? &\rightarrow \mathit{accepting_request}(\mathit{login})\\
\mathit{State}\ \mathit{cancel}?\langle \$id \rangle &\rightarrow \mathit{accepting_cancel}(cr, \$id)
\end{aligned}$$

The metavariable "$id" is used to capture the corresponding argument of the operation so that it can be used in the abstract action, while the wildcard "$*$" is used as a placeholder for any argument.

We comment on some of the rules, the remaining ones can be interpreted similarly. The first rule prescribes that whenever a concrete action involving the operation *createNewCreditRequest* (with any five arguments) occurs in the label of a transition, then it is replaced by the abstract action *request*(*cr*, 1234) (where we suppose that 1234 is the value passed as the first argument to *createNewCreditRequest*). This way, while the first datum exchanged when executing operation *createNewCreditRequest* is preserved (that is the session identifier), the other four data are discharged in the 'abstraction process'. Similarly, the second rule prescribes that whenever an action involving the operation *offerToClient* (with any pair of arguments) occurs in the label of a transition, then it is replaced by the abstract action *response*(*cr*, 1234). Again, the preserved datum is the session identifier which is used to correlate responses from the contacted *CreditRequest* service. To correlate cancellations to the corresponding credit requests, the fourth rule permits replacing an action involving the operation *cancel* (with one argument) by the abstract action *cancel*(*cr*, 1234). The last three rules work similarly, but they relate concrete actions labelling states (rather than transitions) to atomic propositions. The symbols "!" and "?" permit specifying if a rule applies to invoke actions or to receive ones, respectively.

The verification process shows that all the abstract properties we presented in Section 3 do hold for the COWS specification of the Finance case study presented in Section 2, except for the last property. Indeed, if during the login phase

CustomerManagement replies to *CreditRequest* that the customer username and password are not correct, *CreditRequest* raises a fault that is not caught by any fault handler. Thus, no message is sent to the customer to notify him that the process has been aborted. This can be remedied by associating a fault handler behaving as the *Main*'s fault handler to the activities for initialization performed within the term *CreditReq*.

4 A Type System for Checking Confidentiality Properties

In this section, we present a solution to the *Service Specification and Analysis* (see Chapter 7-5) through the type system for COWS introduced in [**?**]. This type system permits expressing and forcing policies regulating the exchange of data among interacting services and ensuring that, in that respect, services do not manifest unexpected behaviours. This enables us to check confidentiality properties, e.g., that critical data such as personal information are shared only with authorized partners.

The types express the policies for data exchange in terms of *regions*, i.e. sets of service partner names attachable to each single datum. Service programmers can thus settle the partners usable to exchange any given datum (and, then, the services that can share it), thus avoiding the datum being accessed (by unwanted services) through unauthorised partners. Then, a type inference system (statically) performs some coherence checks (e.g. the service used in an invocation must belong to the regions of all data occurring in the argument of the invocation) and annotates variable declarations with the minimal regions that ensure consistency of services initial configuration. COWS operational semantics uses these annotations in very efficient checks (i.e. subset inclusions) to authorise or block transitions, in order to guarantee that computations proceed according to them. This property, called *soundness*, can be stated as follows: a service s is *sound* if, for any datum v in s associated to region r and for all evolutions of s, it holds that v can be exchanged only by using services in r. As a consequence of the type soundness of the language, it follows that well-typed services always comply with the policies regulating the exchange of data among interacting services.

We illustrate now some relevant properties for the Finance case study. We first consider the point of view of the customer, then that of the service.

From the customer point of view, the service programmer can specify policies stating that the customer's personal information and the credit request data cannot become available to unauthorised users. Thus, for example, the balance data *balancePackage*, communicated by *Portal* to *CreditRequest* and, then, forwarded to service *BalanceAnalysisProvider*, gets annotated with the policy {*creditReq*, *balance*}, that allows *CreditRequest* and *BalanceAnalysisProvider* to receive the datum but prevents them from transmitting the datum to other services. Other non-critical data, e.g. *customerId*, can be transmitted without an attached policy. The service invocations performed by *Portal* get annotated as follows:

$$creditReq \cdot createNewCreditRequest!\langle id, customerId, \{amount\}_{\{creditReq,\, x_{creditMng}\}}, \{mortgage\}_{\{creditReq,\, x_{creditMng}\}}, \{instalment\}_{\{creditReq,\, x_{creditMng}\}}\rangle$$

$$creditReq \cdot enterBalanceData!\langle id, \{balancePackage\}_{\{creditReq,\, balance\}}\rangle$$

$$creditReq \cdot enterSecurityData!\langle id, \{securityPackage\}_{\{creditReq,\, security\}}\rangle$$

Notice that, while it is perfectly reasonable to assume that the partner names *balance* and *security* are known a priori by *Portal*, the partner name of the credit management service, since it is private, must be communicated by *CreditRequest* to *Portal* at runtime. Indeed, besides policies fixed at design time, the type system permits to express also policies that depend on values discovered at runtime. Thus, in our example, to support communication of the partner name initially unknown, the invoke activity $portal \cdot initialize!\langle x_{Id}, x_{UserOK}\rangle$ performed by *CreditRequest*, which notifies the result of the login check to the customer, has to be modified as follows

$$portal \cdot initialize!\langle x_{Id}, x_{UserOK}, customerManagement\rangle$$

The annotations set by programmers are written as a subscript of the datum to which they refer to. Instead, the annotations put by the type inference, to better distinguish them from those put by the programmers, are written as a superscript of the variable declaration to which they refer to. Thus, the syntax of variable delimitation becomes $[\{x\}^r]\ s$, which means that the datum that dynamically will replace x will be used in s at most by the partners belonging to the region r. Hence, for example, once the type inference phase ends, the term *HandleBalanceAndSecurityData* (subterm of *CreditRequest*) gets annotated as follows

$$\begin{array}{l}
[flow, end] \\
(\ (\ portal \cdot enterBalanceData!\langle x_{Id}\rangle \\
\quad |\ [\{x_{BalancePackage}\}^{\{creditReq,balance\}}] \\
\quad\ creditReq \cdot enterBalanceData?\langle x_{Id}, x_{BalancePackage}\rangle. \\
\quad\quad (\ balance \cdot updateBalanceRating!\langle x_{Id}, \{x_{LoginName}\}_{\{balance\}}, \{x_{FirstName}\}_{\{balance\}}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{x_{LastName}\}_{\{balance\}}, \{x_{BalancePackage}\}_{\{balance\}}\rangle \\
\quad\quad\quad |\ creditReq \cdot updateBalanceRating?\langle x_{Id}\rangle.\ flow \cdot end!\langle\rangle\ )\ ) \\
\quad |\ (\ portal \cdot enterSecurityData!\langle x_{Id}\rangle \\
\quad\quad |\ [\{x_{SecurityPackage}\}^{\{creditReq,security\}}] \\
\quad\quad\ creditReq \cdot enterSecurityData?\langle x_{Id}, x_{SecurityPackage}\rangle. \\
\quad\quad\quad (\ security \cdot updateSecurityRating!\langle x_{Id}, \{x_{LoginName}\}_{\{security\}}, \{x_{FirstName}\}_{\{security\}}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{x_{LastName}\}_{\{security\}}, \{x_{SecurityPackage}\}_{\{security\}}\rangle \\
\quad\quad\quad\quad |\ creditReq \cdot updateSecurityRating?\langle x_{Id}\rangle.\ flow \cdot end!\langle\rangle\ )\ ) \\
\quad |\ flow \cdot end?\langle\rangle.\ \ldots\ )
\end{array}$$

Indeed, the annotations inferred for variables $x_{BalancePackage}$ and $x_{SecurityPackage}$ are derived from the use of these variables made by *HandleBalanceAndSecurityData*. Thus, they are assigned regions $\{creditReq, balance\}$ and $\{creditReq, security\}$, respectively, because they are only used in the receives along $creditReq \cdot enterBalanceData$ and $creditReq \cdot enterSecurityData$, and in the invokes along $balance \cdot updateBalanceRating$ and $security \cdot updateSecurityRating$. Hence, the partner name of such endpoints must belong to the region of the corresponding variables.

Now, *Portal* can safely communicate balance data (respectively, security data) to *CreditRequest*, since the region $\{creditReq, balance\}$ (resprespectively $\{creditReq, security\}$) of the data contains the region of the receiving variable (in fact, they coincide). More in general, the typed version of the credit request service, respects all above defined policies.

Suppose instead that service *CreditRequest* (accidentally or maliciously) attempts to reveal the balance data through some internal operation such as $int \cdot o!\langle\{x_{BalancePackage}\}_r\rangle$,

for some region r. For *CreditRequest* to successfully complete the type inference phase, we should have $int \in r$. Then, as result of the inference, we would get the annotated variable declaration $[\{x_{BalancePackage}\}^{r'}]$, for some region r' with $r \subseteq r'$. Now, the interaction between the typed terms *Portal* and *CreditRequest* would be blocked by the runtime checks because the datum sent by *Portal* would be annotated as $\{balancePackage\}_{\{creditReq,\, balance\}}$ while the region r' of the receiving variable $x_{BalancePackage}$ is such that $int \in r \subseteq r' \not\subseteq \{creditReq, balance\}$.

From the *CreditRequest*'s point of view, the service programmer can require the customer not to pass to other services the offer that has been specifically computed for the customer demands. Therefore, the corresponding invocation performed by *CreditRequest* gets annotated as follows:

$$portal \bullet offerToClient!\langle x_{Id}, \{x_{AgreementData}\}_{\{portal\}}\rangle$$

For what concerns the type inference of the involved terms we can reason as before.

5 Automated Verification of UML4SOA Models of Services

Although the logical verification methodology described in Section 3 is effective and automated, people willing to use it are required to be able to understand and deal with algebraic and logical tools, i.e. the process calculus COWS and the temporal logic SocL. Sometimes this may not be the case, especially within industrial contexts. To make the verification of service properties more accessible, we then put forward the idea of exploiting translations of languages at different abstraction levels, i.e. modelling languages and process calculi, as those defined in [**?**,**?**]. Here, we report on an ongoing effort for devising an approach that integrates our verification methodology with language translations aiming at obtaining verifiable implementations of service components from abstract architectural models of business activities. To this aim, we are developing two software tools[2]: UStoC, that supports translation from UML4SOA to COWS, and Venus, that, by closely integrating UStoC and CMC, provides access to verification functionalities also to those users not familiar with formal methods.

In Section 2, the UML4SOA activity diagrams specifying the behaviour of the services involved in the Finance case study (presented in Chapters 0-2 and 7-1) are translated 'by hand' into COWS terms to enable a subsequent analysis phase. By accomplishing this task, we have experimented how the specific mechanisms and primitives of COWS are particularly suitable for encoding services specified by UML4SOA activity diagrams. This is not surprising if one considers that both UML4SOA and COWS are inspired by WS-BPEL. To formalize those intuitions and support a more systematic and mathematically well-founded approach to engineering of SOA systems, we have defined a compositional encoding of UML4SOA activity diagrams into COWS terms. This way, developers can concentrate on modelling the high-level behaviour of the system and use the encoding for analysis purposes. Such encoding is implemented by UStoC, a software tool that given a UML4SOA specification, consisting of a set of XMI files [**?**] automatically generated by the UML editor MagicDraw [**?**], returns a

[2] Both tools are freely downloadable from `http://rap.dsi.unifi.it/cows/` and can be redistributed and/or modified under the terms of the GNU General Public License.

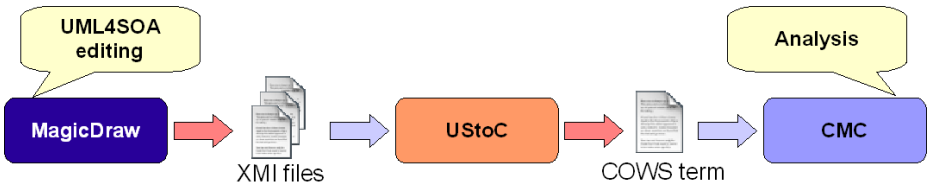

Fig. 1. Verification process of UML4SOA models of services

COWS term written in the syntax accepted by CMC. UStoC's workflow is graphically depicted in Fig. 1.

UStoC works properly with activity diagrams specified by using version 1.2 of the UML4SOA profile. Therefore, to use the tool for translating the Finance case study, we need to specify the case study using the profile mentioned above. For the sake of simplicity, we consider here just an excerpt of the scenario, which is composed of three services: creditRequest that performs the initialization activities and terminates upon receiving the data for a new credit request, customerManagementService that, when invoked, non-deterministically replies either yes or no to every request, and portalService that, if the login succeeds, non-deterministically sends either a credit request or a cancellation request. The UML4SOA diagram modelling service creditRequest is shown in Fig. 2. To analyse this scenario, firstly we generate a file XMI (saved with extension `.uml`) for each UML4SOA diagram by using MagicDraw. Then, we load the three created files into UStoC (by pushing the 'Add' button on the right-hand side of the graphical interface, a screenshot of which is shown in Fig. 3) and encode them into a COWS term (by pushing the 'Start encoding' button). Finally, we can export the COWS term from UStoC to CMC and start to analyse it exactly as described in Section 3.

The above example shows how UStoC simplifies the modelling phase of the verification process by enabling the use of the abstract modelling language UML4SOA. However, the problem of making more accessible our verification methodology to people without significative expertise on process calculi and logics is not resolved. To remedy this, we are developing Venus, a software tool that integrates UStoC and CMC in order to hide the use of COWS and SocL and, hence, make the verification process as much transparent as possible for developers. The issue of tailoring and reflecting the (low-level) results obtained by the verification of COWS terms to the corresponding (high-level) UML4SOA specifications is tackled by exploiting abstraction rules that permit specifying a 'bridge' between the two specification levels.

Let us see how Venus can be used to analyse the excerpt of the Finance case study previously introduced. First of all, similarly to UStoC, Venus requires the user to provide the XMI files storing the UML4SOA diagrams (Fig. 4). Then, it requires the user to select the properties that he wants to verify out of a list of predefined general properties written in natural language (Fig. 5). Notably, expert users can add in the text area at the bottom of the window further properties directly expressed as SocL formulae. Now, the user has to define the intuitive semantics of the relevant operations of the loaded UML4SOA specification. This is done by specifying the operations representing initial requests, positive responses, negative responses, cancellations, ..., and by possibly indicating the corresponding correlation data (Fig. 6). In our example, we specify that

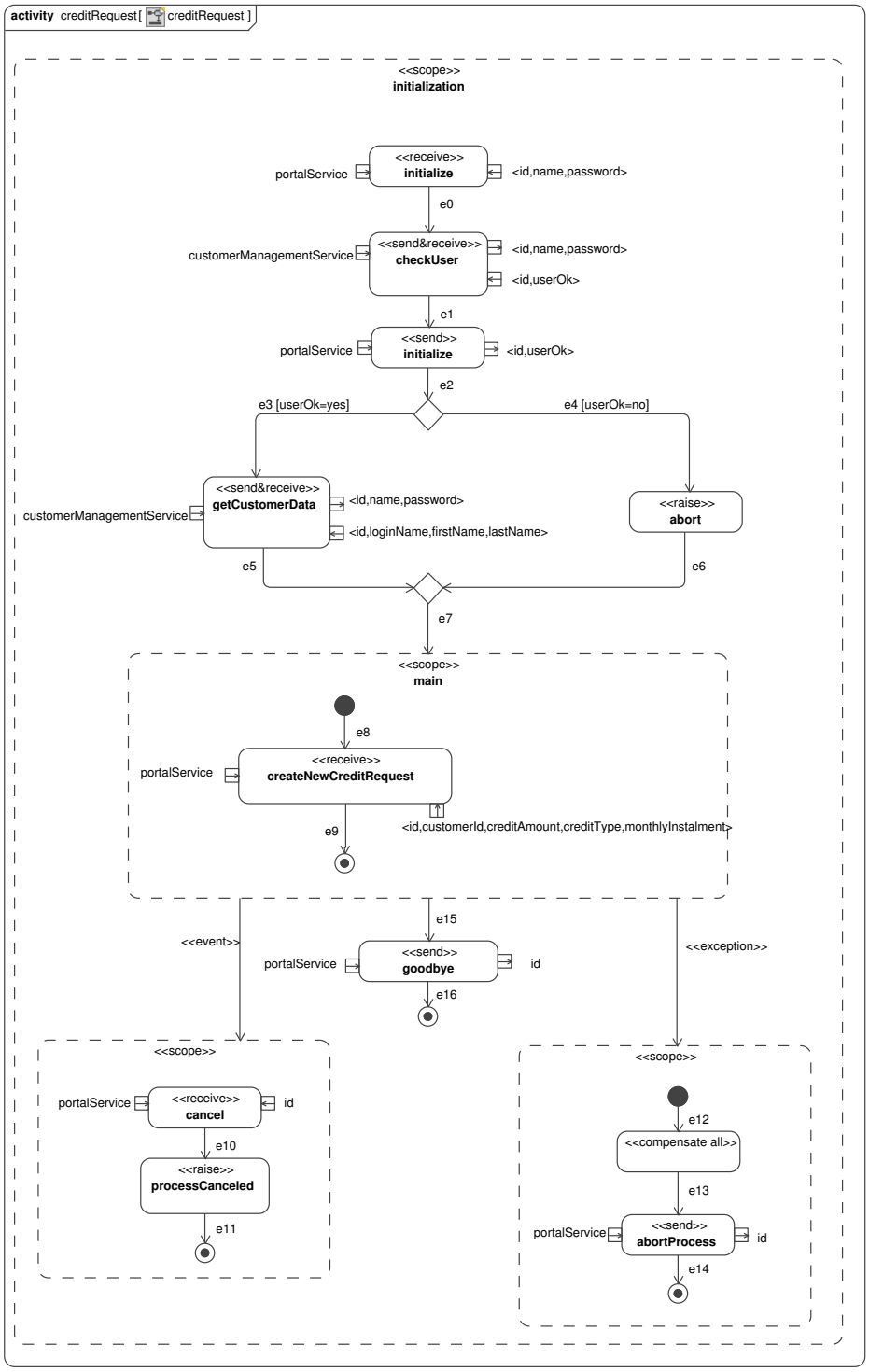

Fig. 2. Excerpt of the Finance case study: creditRequest

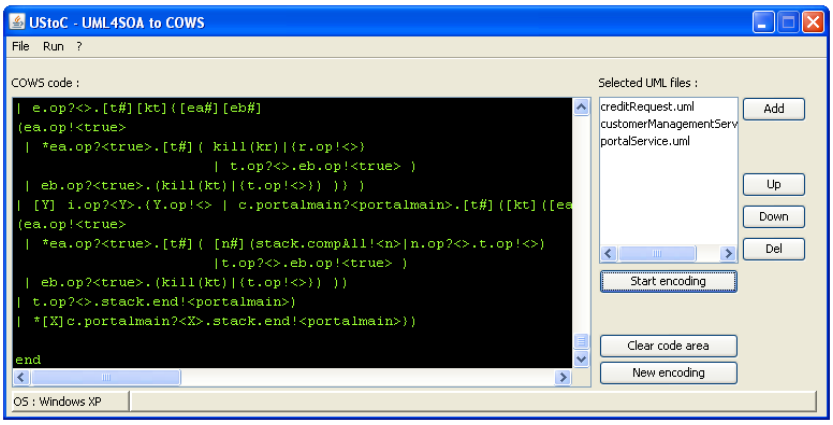

Fig. 3. A screenshot of UStoC interface

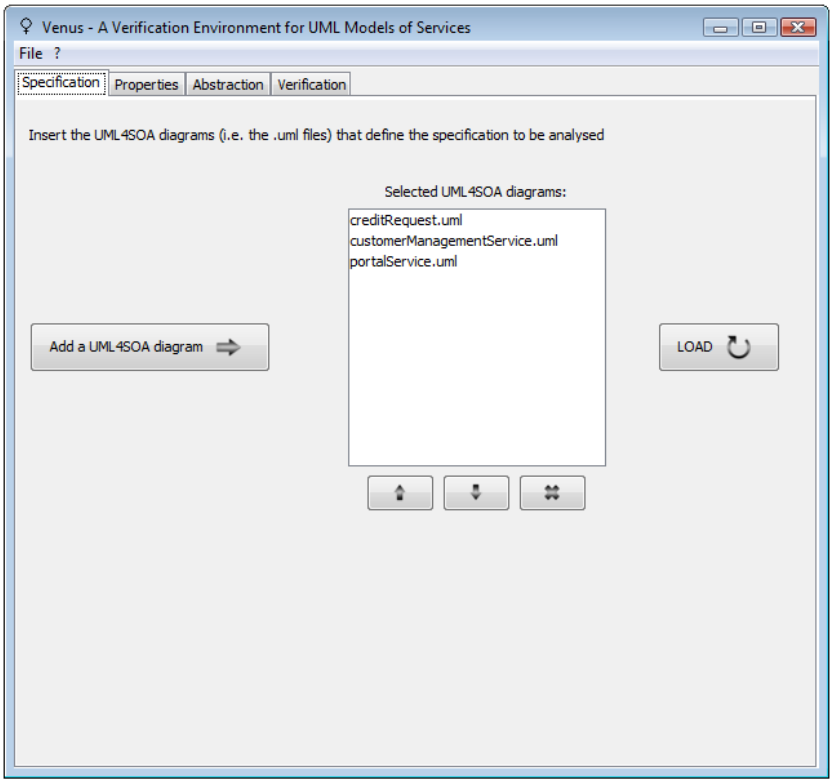

Fig. 4. Venus interface: loading of UML4SOA diagrams

an invocation of operation initialize corresponds to sending an initial request to the service and the value that will be assigned to variable id will be used to correlate positive and negative responses to such request. Notice that Venus requires to specify such operations only for the categories that are needed for checking the properties previously

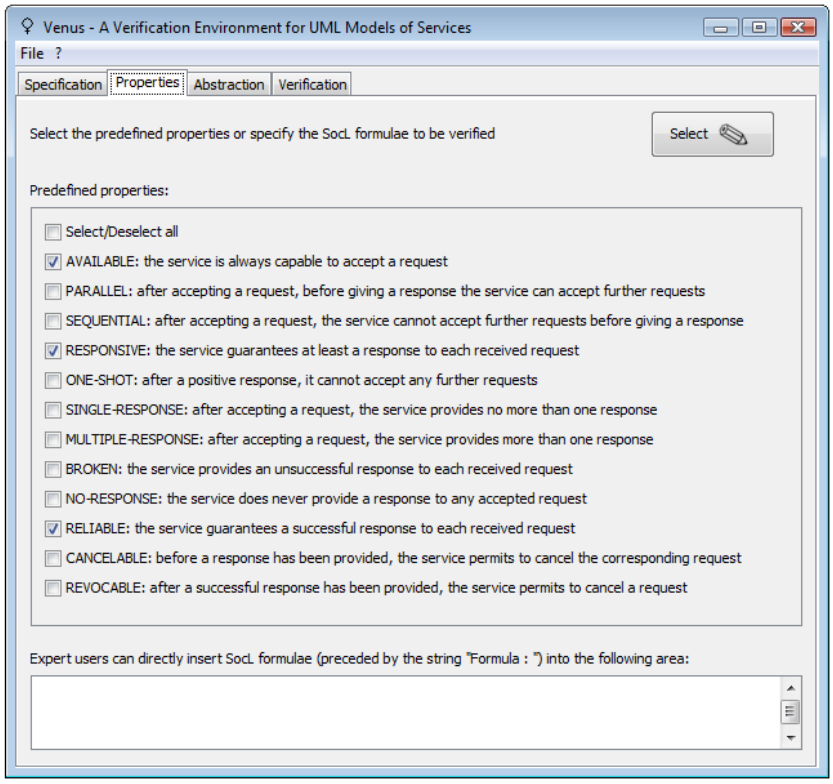

Fig. 5. Venus interface: selection of service properties

selected (in this case, e.g., initial request, positive response and negative response). Moreover, for each category, more than one operation can be specified by using the associated 'Add' button. The information provided at this step are used, on the one hand, to express the selected general properties as SocL formulae and, on the other hand, to generate the abstraction rules that will be applied to the COWS term resulting from the translation of the UML4SOA diagrams provided at the initial step. Expert users can also provide here custom abstraction rules. Finally, Venus properly arranges all data, loads them into CMC, and allows the user to check the validity of each property and, possibly, to require an explanation in case of a negative result (Fig. 7).

6 Concluding Remarks

We have presented a COWS specification of the Finance case study and two analysis techniques, namely a temporal logic and its model checker for expressing and checking functional properties and a type system for guaranteeing confidentiality properties.

The specification of the case study demonstrates that COWS's distinctive features, as e.g. the termination constructs and the correlation mechanism, are effective tools for specifying service-oriented systems. In fact, kill activities are suitable for representing ordinary and exceptional process terminations, while protection permits to naturally

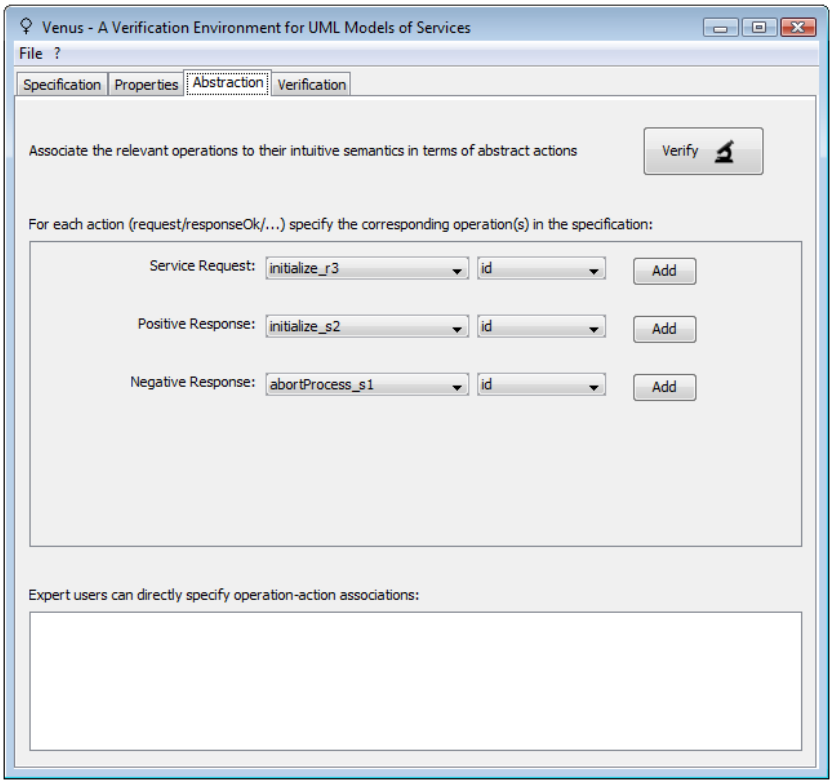

Fig. 6. Venus interface: definition of the intuitive semantics of the relevant operations

represent exception and compensation handlers that are supposed to run after normal computations terminate. Even more crucially, the correlation mechanism permits to automatically correlate messages belonging to the same long-running interaction, preventing to mix messages from different service instances. Also the encoding of UML4SOA in COWS, which is at the basis of the tools UStoC and Venus, has greatly benefitted from COWS's distinctive features. The definition of such an encoding appears to be problematic and less intuitive if one use a different, e.g. session-based, calculus.

There are several requirements and properties concerning to liveness, correctness, and security that an implementation of the Finance case study is expected to fulfill. The methodology reported in Section 3, and then exploited by the tools presented in Section 5, has proven to be very effective to check a large spectrum of behavioural properties. With respect to the many other temporal logics proposed in the literature, one important advantage of SocL is that the service properties can be formulated in a way which is independent from individual service domains and specifications. Security properties regarding the exchange of data among service components can be instead insured by means of the type system reported in Section 4. Since it is not realistic to assume complete knowledge of the whole system and access to the internal implementation of all the involved services, a practical implementation of this approach would require services to declare how they use the data they exchange and should rely on a

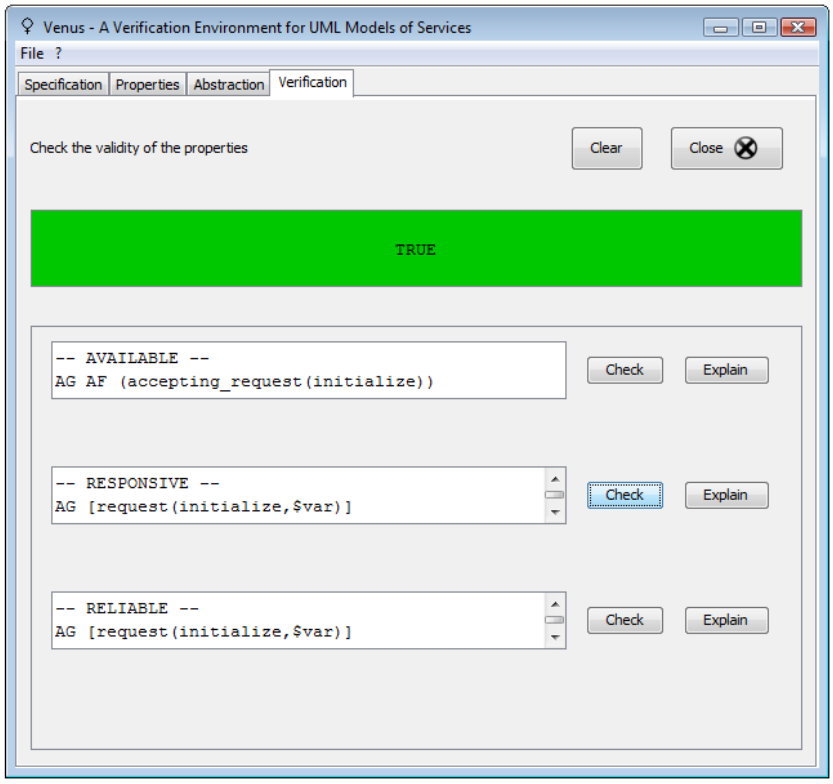

Fig. 7. Venus interface: verification of the service properties

mechanism ensuring that service behaviours do comply with their declaration. The run-time support should also take charge of performing the checks described in Section 4 to authorise or block transitions.

Some other analysis techniques for COWS terms have been developed as a result of the SENSORIA project. In particular, a Flow Logic for checking information flow properties is presented in [**?**], a stochastic extension of COWS that enables verification of quantitative properties is presented in [**?**] (see also Chapter 5-5), and a few observational semantics for checking interchangeability of COWS terms and conformance against service specifications are presented in [**?**] (see also Chapter 2-2). However, we have not yet results on the application of these techniques to the COWS specification of the Finance case study.