An Accessible Verification Environment for UML Models of Services

Federico Banti, Rosario Pugliese and Francesco Tiezzi

Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy

Abstract

Service-Oriented Architectures (SOAs) provide methods and technologies for modelling, programming and deploying software applications that can run over globally available network infrastructures. Current software engineering technologies for SOAs, however, remain at the descriptive level and lack rigorous foundations enabling formal analysis of service-oriented models and software. To support automatised verification of service properties by relying on mathematically founded techniques, we have developed a software tool that we called Venus (Verification ENvironment for UML models of Services). Our tool takes as an input service models specified by UML 2.0 activity diagrams according to the profile UML4SOA, while its theoretical bases are the process calculus COWS and the temporal logic SocL. A key feature of Venus is that it provides access to verification functionalities also to those users not familiar with formal methods. Indeed, the tool works by first automatically translating UML4SOA models and natural language statements of service properties into, respectively, COWS terms and SocL formulae, and then by automatically model checking the formulae over the COWS terms. In this paper we present the tool, its architecture and its enabling technologies by also illustrating the verification of a classical 'travel agency' scenario.

Key words: Service-Oriented Architectures, CASE tools, UML, Formal methods, Model checking, process calculi

⋆ This work is a revised and extended version of (Banti et al., 2009c) and is partially based on a preliminary submitted paper (Banti et al., 2009b). The work has been partially supported by the EU project SENSORIA, IST-2005-016004.

Email address: fbanti@gmail.com, {pugliese,tiezzi}@dsi.unifi.it (Federico Banti, Rosario Pugliese and Francesco Tiezzi).

1. Introduction

Service-Oriented Architectures (SOAs) provide methods and technologies for programming and deploying software applications that can run over globally available network infrastructures. The most successful implementations of the SOA paradigm are probably the so-called *web services*, sort of autonomous computational entities accessible by humans and other services through the Web. They are, in general, loosely coupled and heterogeneous, widely differing in their internal architecture and, possibly, in their implementation languages. Different services are often combined together to form service-based systems, also called *service orchestrations*, where service components are usually implemented by different software developing groups. Service orchestrations may themselves become services, making composition a recursive operation. Both stand alone and composed services usually have requirements like, e.g., service availability, functional correctness, and protection of private data. Implementing services satisfying these requirements demands the use of rigorous software engineering methodologies encompassing the various phases of the software development process, and exploiting formal techniques for qualitative and quantitative verification. The goal is to initially specify the services using a high-level, possibly graphical, modelling language driving the software development process towards implementations complying with the specification. Service models also provide a human understandable common view of service systems. It is then fundamental to guarantee the correctness of the initial models. Therefore, in this paper we put forward an approach for verifying behavioral properties of service models by relying on formal methods.

Currently, the most widely used language for modelling software systems is UML (Object Management Group, 2007a). It is intuitive, powerful, and extensible. Recently, a UML 2.0 profile, christened UML4SOA (Mayer et al., 2008b), has been designed for modeling SOAs. In particular, the behavioral aspects of services are mainly represented by UML4SOA *activity diagrams*. Inspired to WS-BPEL (OASIS, 2007), the OASIS standard for orchestrating web services, UML4SOA activity diagrams integrate UML with specialized actions for exchanging messages among services, specialized structured activity nodes and activity edges for representing scopes equipped with event, fault and compensation handlers. However, UML in general, and UML4SOA in particular, falls short of providing formal semantics and formal verification techniques for its models and their properties.

Formal verification methods usually exploit theoretical frameworks like, e.g. process calculi, state machines, Petri nets, temporal and modal logics. Recently, many research efforts have been devoted to investigate feasibility and effectiveness of using process calculi (see, e.g., (Bocchi et al., 2003; Butler et al., 2005; Geguang et al., 2005; Guidi et al., 2006; Ferrari et al., 2006; Lapadula et al., 2007a; Lanese et al., 2007; Prandi and Quaglia, 2007; Carbone et al., 2007; Boreale et al., 2008; Vieira et al., 2008; Bartoletti et al., 2008)) and temporal logics (see, e.g., (Fantechi et al., 2008; De Nicola et al., 2009)) for formally specifying, simulating and verifying service behaviours. On the one hand, process calculi are a cornerstone of specification and verification of concurrent, distributed and mobile systems. In fact, due to their algebraic nature, process calculi convey in a distilled form the SOA compositional programming style. On the other hand, temporal logics have long been used to represent properties of concurrent and distributed systems owing to their ability of expressing notions of necessity, possibility, eventuality, etc. (see e.g. (Huth and Ryan, 2004)). These logics have proved suitable to reason about complex software systems because they only provide abstract specifications of these systems and can thus be used for describing system properties rather than system behaviours. One successful application of temporal logics to the analysis of systems, often supported by efficient software tools (see e.g. (Clarke et al., 1999; Grumberg and Veith, 2008)), is *model checking*.

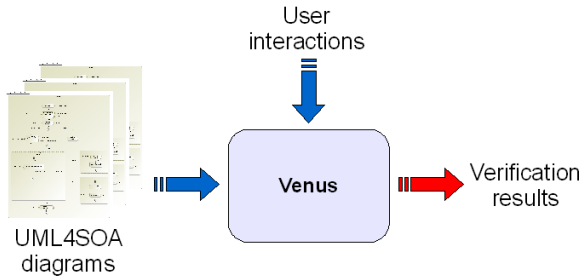


Fig. 1. Venus interactions

While UML is quite well known and widely used within industrial contexts, the same cannot be said of the formal methods we mentioned before. Thus, our aim is not only to devise a viable approach for verifying behavioral properties of UML models of services by exploiting process calculi and temporal logics, but also to make such an approach accessible to people not necessarily familiar with the formal methods which it relies on. The solution we illustrate hereafter is to almost completely automatise the verification process by means of a suitable software environment shepherding the (non-expert) users to set the behavioral service properties they want to verify. We present a prototypical implementation of this approach called Venus (Verification ENvironment for UML models of Services).

Venus interactions with the external world are schematized in Figure 1. Firstly, the tool asks the user to upload the UML4SOA activity diagrams modelling the service scenario to be analysed. Then, the user is required to select the service properties he wants to verify out of a predefined list of intuitive behavioral properties (described in Section 3 and taken from (Fantechi et al., 2008)) that are written in natural language and are relevant in most of service scenarios. For instance, the user might choose to check service *availability*, i.e. if the service is always able to accept client requests, or service *responsiveness*, i.e. if the service guarantees at least one response to each accepted request. An expert user may himself also write down additional (domain specific) properties he wants to check. All properties will be internally expressed as logical formulae in terms of the actions the modelled service is expected to perform and of the propositions that are true in its states. Next, the user is asked to select which are the operations of the UML4SOA diagrams corresponding to the actions occurring in the selected properties. For instance, to check service availability, the user is requested to specify which operation in the diagrams corresponds to the action of accepting a client request. Finally, Venus checks the validity of the properties by exploiting an internal model checker. In case a property does not hold, the tool can provide a detailed explanation of the result, that is a so-called *counterexample*.

The above external behaviour hides from the users the technical details of what really happens inside Venus during the verification process. The cornerstone of our tool is an automatised encoding from the modelling language UML4SOA to the service-oriented calculus COWS (Lapadula et al., 2007a). This encoding permits to pass from a semi-formal graphical notation to an internal representation with a formal operational semantics. The rational for our choice of COWS as the target calculus of the encoding is its proximity to UML4SOA. In fact, in (Banti et al., 2009a) we used COWS for translating *by hand* UML4SOA activity diagrams to enable a subsequent analysis phase. There, we have experimented that the specific mechanisms and primitives of COWS are particularly suitable for encoding services specified by UML4SOA activity diagrams. This is not surprising if one consider that both UML4SOA and COWS are inspired to WS-BPEL. The next step has been to define an encoding of UML4SOA diagrams into COWS terms and implement it as an automatic tool (this is the topic of (Banti et al., 2009b)). The encoding is compositional, in

the sense that the encoding of a service scenario is the parallel composition of the encoding of its individual services and, moreover, the encoding of a UML4SOA activity diagram is the COWS term resulting from the parallel composition of the encodings of its components. Additionally, our encoding defines a transformational operational semantics for UML4SOA which is, at the best of our knowledge, the only formal semantics of this modelling language.

The properties selected by the user are, in turn, translated into formulae of the branching-time temporal logic SocL (Fantechi et al., 2008). This logic permits to express properties of services (like the general properties predefined in Venus) in terms of states and state changes, and of the actions that are performed when moving from one state to the other. SocL can express in an easy way peculiar aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses. This, together with the existence of a model checker for SocL formulae over COWS terms, has motivated our choice of SocL out of the several temporal logics proposed in the literature.

The rest of the paper is structured as follows. Section 2 first provides an overview of UML4SOA by means of a classical 'travel agency' example and then presents our proposal of a BNF-like syntax for UML4SOA. Section 3 presents the Venus tool and illustrates its usage by resorting to the travel agency example. Section 4 briefly reviews COWS, while Section 5 presents the COWS-based transformational semantics of UML4SOA. Section 6 introduces SocL and its model checker. Finally, Section 7 touches upon comparisons with related work and directions for future developments.

2. An overview of UML4SOA

We start by informally presenting UML4SOA through a realistic but simplified example, illustrated in Figure 2, based on the classical 'travel agency' scenario.

A travel agency exposes a service that automatically books a flight and a hotel according to the requests of the user. The activity starts with a *receive* action for a message from a client containing a request for a flight and a hotel (stored in reqData). Whenever prompted by a client request, the service creates an instance to serve that specific request and is immediately ready to concurrently serve other requests. Typically, in a service-oriented scenario, to ensure that each message is delivered to the proper service instance, along with the message some *correlation data* are exchanged that are required to match the corresponding data known by the message receiver's instance. In our scenario, each service instance is uniquely identified by the value stored in the write once variable id, which is, from now onwards, included in every exchanged message.

Afterwards, by a *send&receive* action, the request is sent to a flight searching service (flightService) and the service awaits for a response message that will be stored in the variables flightAnswer and fData. As soon as this action is executed, a *compensation handler* is installed. The compensation consists of a *send* action to the flight searching service of a message asking to delete the request. The received answer is then checked by a decision node. If the answer is positive, similar actions are undertaken for booking a hotel by means of a hotel searching service (hotelService). If also this service replies positively, the reservation data (stored in fData and hData) are forwarded to the client and the activity successfully terminates. Instead, if at least one answer is negative, an exception is raised by a *raise* action. An exception may also be raised in response to an *event* consisting of an incoming message from the client, and requiring to cancel his own request. All exceptions are caught by the *exception handler* that through the action *compensate all* triggers all the compensations installed so far in reverse order w.r.t. their completion, and notifies the client that his requests have not been fulfilled.

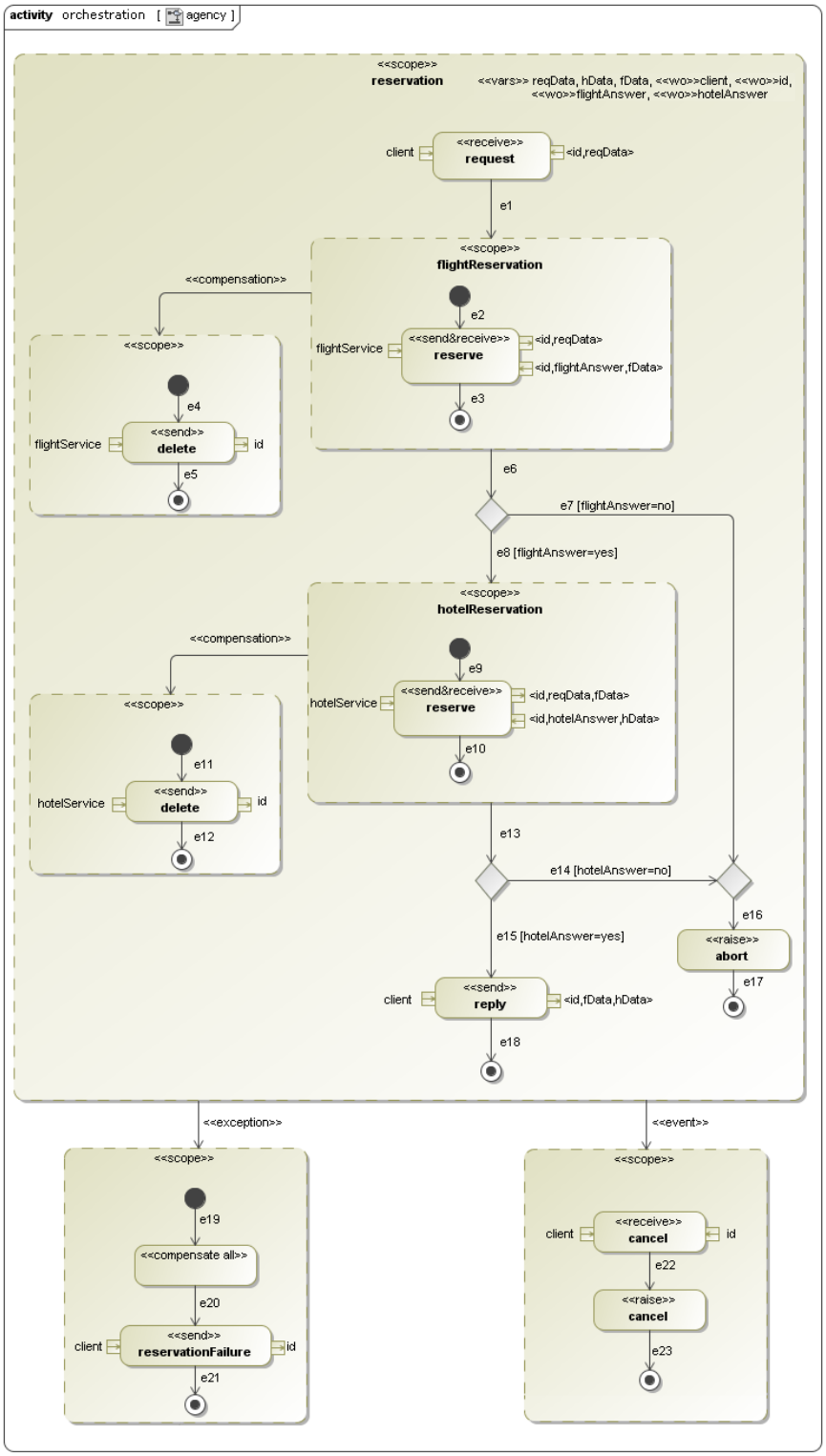


Fig. 2. Travel agency scenario: agency service

The other participants involved in the travel agency scenario are a client, a flight booking service and a hotel booking service. For the sake of simplicity, we have considered a client that simply invokes the agency service and waits for a response (it never requires a cancellation), and flight/hotel booking services that, when invoked, non-deterministically reply either yes or no to every request and then allow the agency service to delete the reservation. The UML4SOA diagrams modelling such services are shown in Figure 3.

The syntax of UML4SOA is given in (Mayer et al., 2008b) by a metamodel in classical UML-style. In Figure 4 we provide an alternative BNF-like syntax that is more suitable for defining an encoding by induction on the syntax of constructs. Each row of the table represents a production of the form SYMBOL ::= ALTER_1 | ... | ALTER_n, where the non-terminal SYMBOL is in the top left corner of the row (highlighted by a gray background), while the alternatives ALTER_1, ..., ALTER_n are the other elements of the row.

To simplify the encoding and its exposition we adopt some mild restrictions. We assume that every action and scope has one incoming and one outgoing control flow edge (except for receiving actions that may have no incoming edge), that a fork or decision node has one incoming edge, and that a join or merge node has one outgoing edge. These restrictions do not compromise expressiveness of the language and are usually implicitly adopted by most of UML users for sake of clarity. We also omit many classical UML constructs, in particular object flows, exception handlers, expansion regions and several UML actions, since UML4SOA offers specialized versions of such constructs. Regarding object flows, used for passing values among nodes, they become unnecessary since, for inter-service communications, UML4SOA relies on input and output pins, while data are shared among the elements of a scope by storing them in variables.

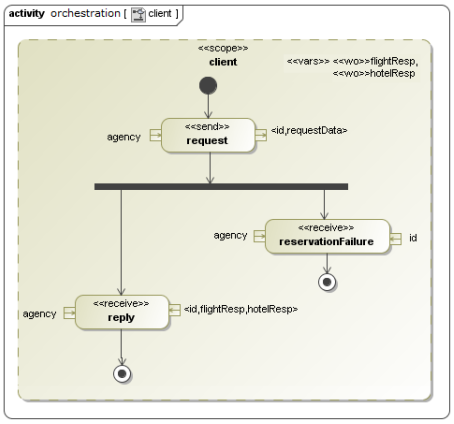
A UML4SOA *application* is a finite set of orchestrations ORC. We use orc to range over orchestration names. An *orchestration* is an activity enclosing one top level scope with, possibly, several nested scopes. A *scope* SCOPE is a structured activity that permits explicitly grouping activities together with their own associated variables, references to partner services, event handlers, and a fault and a compensation handler. A list of *variables* is generated by the following grammar:

$$\mathsf{VARS} ::= \mathsf{nil} \mid \mathsf{X}, \mathsf{VARS} \mid \ll\mathsf{wo}\gg \mathsf{X}, \mathsf{VARS}$$

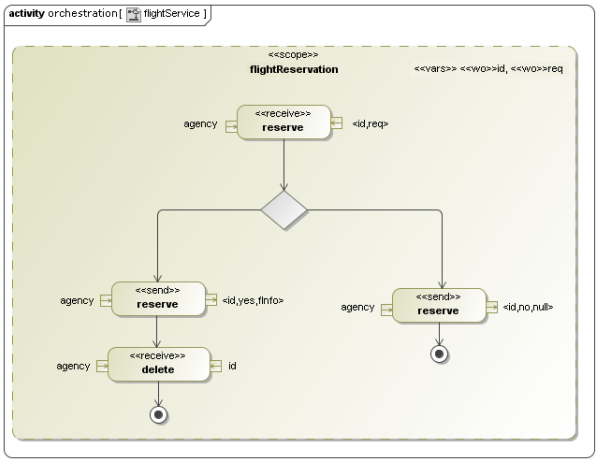
We use X to range over variables and the symbol $\ll\mathsf{wo}\gg$ to indicate that a variable is 'write once', i.e. a sort of late bound constant that can be used, e.g., to store a correlation datum (see (OASIS, 2007, Sections 7 and 9) for further details) or a reference to a partner service. Lists of variables can be inductively built from nil (the empty list) by application of the comma operator ",". Graphical editors for specifying UML4SOA diagrams usually permit declaring local variables as properties of a scope activity, but they are not depicted in the corresponding graphical representations. Instead, here we explicit the variables local to a scope because such information is needed for the translation in COWS. For a similar reason, we show the edge names in the graphical representation of a graph. Notably, to obtain a compositional translation, each edge is divided in two parts: the part outgoing from the source activity and the part incoming into the target activity. In the outgoing part a guard is specified; this is a boolean expression and can be omitted when it is **true**.

A *graph* GRAPH can be built by using edges to connect *initial nodes* (depicted by large black spots), *final nodes* (depicted as circles with a dot inside), control flow nodes, actions and scopes. It is worth noticing that for all incoming edges there should exist an outgoing edge with the same name, and vice-versa. Moreover, we assume that (pairs of incoming and outgoing) edges in orchestrations are pairwise distinct. These properties are guaranteed for all graphs generated by using any UML graphical editor. If a receiving action, namely a RECEIVE or a RECEIVE&SEND,

(a)



(b)



(c)

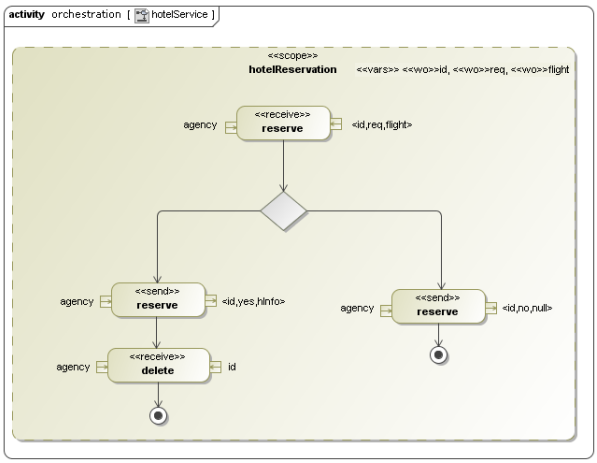


Fig. 3. Travel agency scenario: (a) client, (b) flight booking service and (c) hotel booking service

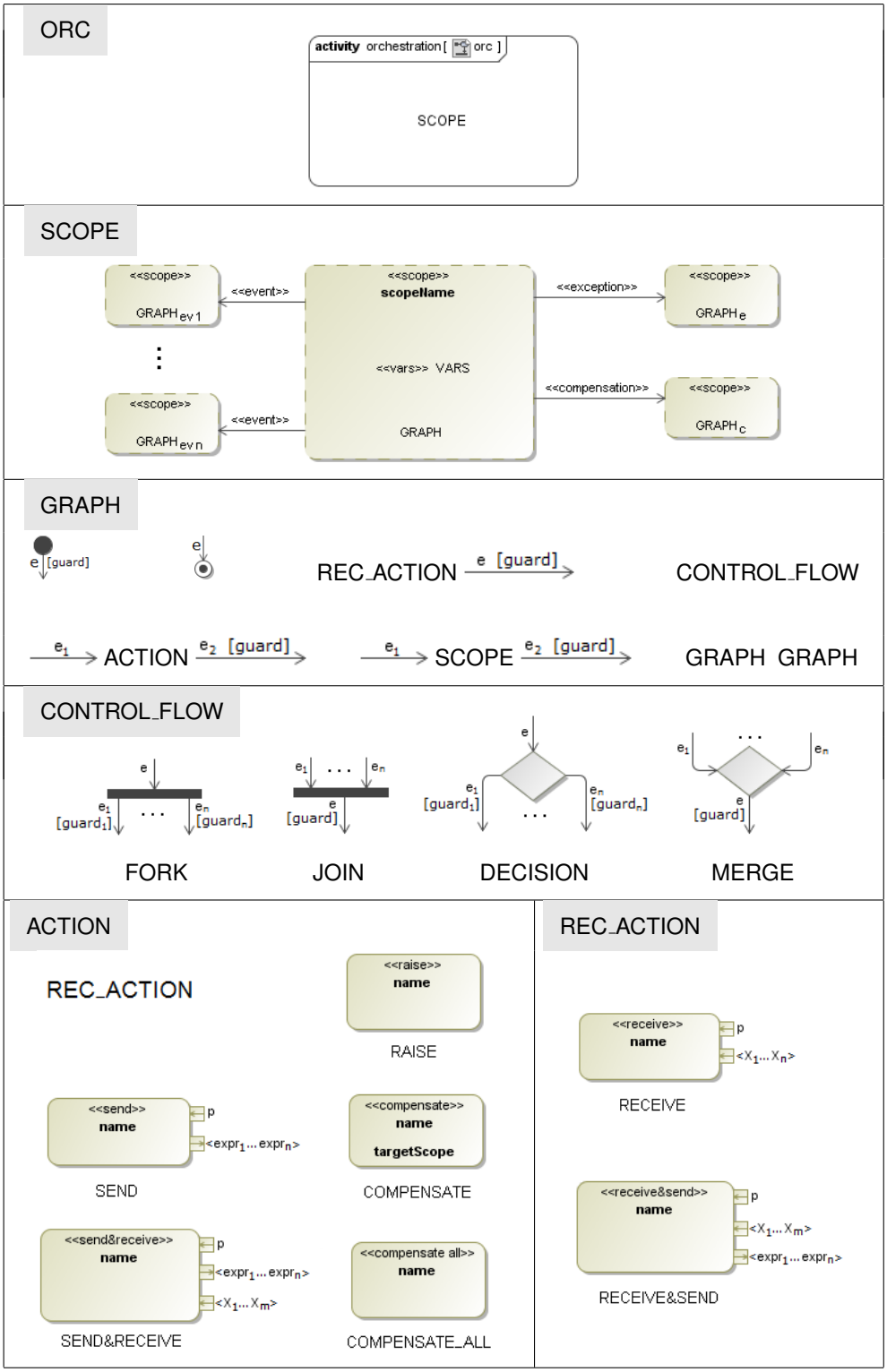
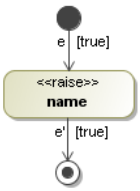
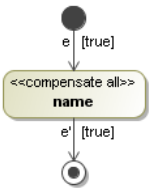


Fig. 4. UML4SOA syntax

has no incoming edges, then it starts when a message is received and remains enabled to wait for other messages (like a UML *AcceptEventAction* (Object Management Group, 2007a, Section 12.3.1)). This kind of action permits specifying *persistent* services, i.e. services capable of creating multiple instances to serve several requests simultaneously, such as the travel agency service depicted in Figure 2.

Event, exception and compensation handlers are activities linked to a scope by respectively an event, a compensation and an exception activity edge. An *event handler* is a scope triggered by an event in the form of incoming message. For each event handler, indeed, we assume that its graph $\mathsf{GRAPH}_{\mathsf{ev\,i}}$ takes the form $\mathsf{REC\_ACTION} \xrightarrow{\mathsf{e\ [guard]}} \mathsf{GRAPH}$. A *compensation handler* is a scope that is installed when execution of the related main scope completes and is executed in case of failure to semantically roll back the execution of this latter scope. An *exception handler* is an activity triggered by a raised exception whose main purpose is to trigger execution of the installed compensations. *Default* event handlers are empty graphs, while default exception and compensation handlers, are respectively, as follows:





For readability sake, these handlers sometimes will not be represented.

It is worth noticing that, handling of exceptions in UML4SOA differs from UML 2.0. Indeed, the former can execute compensations of completed nested scopes in case of failure, while the latter can only provide an alternative way to successfully complete an activity in case an exception is raised. See Section 5 for a formal explanation of the behavior of the UML4SOA construct.

Control flow nodes CONTROL_FLOW are the standard UML ones: *fork* nodes (depicted by bars with 1 incoming edge and *n* outgoing edges), *join* nodes (depicted by bars with *n* incoming edges and 1 outgoing edge), *decision* nodes (depicted by diamonds with 1 incoming edge and *n* outgoing edges), and *merge* nodes (depicted by diamonds with *n* incoming edges and 1 outgoing edge).

Finally, UML4SOA provides seven specialized actions ACTION for exchanging data, for raising exceptions and for triggering scope compensations. SEND sends the message resulting from the evaluation of expressions $\mathsf{expr}_1,\ldots,\mathsf{expr}_n$ to the partner service identified by p. UML4SOA is parametric with respect to the language of the expressions, whose exact syntax is deliberately omitted; we just assume that expressions contain, at least, variables. RECEIVE permits receiving a message, stored in $\mathsf{X}_1,\ldots,\mathsf{X}_n$, from the partner service identified by p. Send actions do not block the execution flow, while receive actions block it until a message is received. The other two actions for message exchanging, i.e. SEND&RECEIVE and RECEIVE&SEND, are shortcuts for, respectively, a sequence of a send and a receive action from the same partner and vice-versa. RAISE causes normal execution flow to stop and triggers the associated exception handler. COMPENSATE triggers compensation of its argument scope, while COMPENSATE_ALL, only allowed inside a compensation or an exception handler, triggers compensation of all scopes (in the reverse order of their completion) nested directly within the same scope to which the handler containing the action is related.

In the next Section we will show how to use Venus for analyzing UML4SOA models of services.

3. Venus: a verification environment for UML models of services

As anticipated in the Introduction, we tackle the problem of making verification of service behavioral properties accessible to people that are familiar with UML but not necessarily with formal verification methods, like process calculi and temporal logics.

To this purpose, we have developed Venus [1], a software tool that aims at automatising, as much as possible, the verification process of service models specified by using the UML4SOA profile, actually hiding to the (non-expert) user the underlying formal methods and theories. This way, developers can concentrate on modelling the high-level behaviour of the system and use our tool at an intuitive level for analysing it. A prototype of Venus can be downloaded at `http://rap.dsi.unifi.it/cows`.

We present the functionalities of Venus by illustrating how the tool can be used to analyse the travel agency scenario introduced in Section 2.

First of all, Venus requires the user to provide the UML4SOA specification, consisting of a set of activity diagrams. Each diagram can be edited by using the graphical UML editor MagicDraw (No Magic Inc., 2009) where, to allow users to specify UML4SOA activity diagrams, the UML4SOA profile (Mayer et al., 2008a) must have been previously installed. Figures 2 and 3 show examples of UML4SOA diagrams edited using MagicDraw. More specifically, Venus accepts as an input a set of files XMI (Object Management Group, 2007b), storing UML4SOA diagrams, that can be automatically generated by MagicDraw (Figure 5).

The user can then select the properties that he wants to verify out of a predefined list of twelve general properties written in natural language (Figure 6). The properties focus on the dynamics of service-client interaction and can be grouped in three categories. Properties of the first group describe the behavior of the service w.r.t. incoming requests from potential service clients. Regarding this aspect a service is said to be

Available if it is always capable to accept an incoming request;

Parallel if, after accepting a request, it can accept further requests before giving a response to the initial request;

Sequential if it must deliver a response to a request before accepting the next request.

Most of the services are expected to be Available, in order to be able to concurrently provide answers to a number of clients as largest as possible. Of course, to be Available, a service must be at least Parallel. Sometimes however a service can only be sequential, i.e. it cannot serve more than one client at a time (a cash machine is an example of sequential service).

Properties of the second group regard the service responses to client requests. A service is said to be

Responsive if it guarantees at least one response to each accepted request;

One-shot if, after a positive response to a request, it cannot accept any further requests;

Single-response if it provides at most one response to each accepted request;

Multiple-response if it provides more than one response to each accepted request;

Broken if it always provide an unsuccessful response to each accepted request;

No-response if it never provides a response to each accepted request;

Reliable if it provides a successful response to each accepted request.

---

[1] Venus is a free software; it can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.

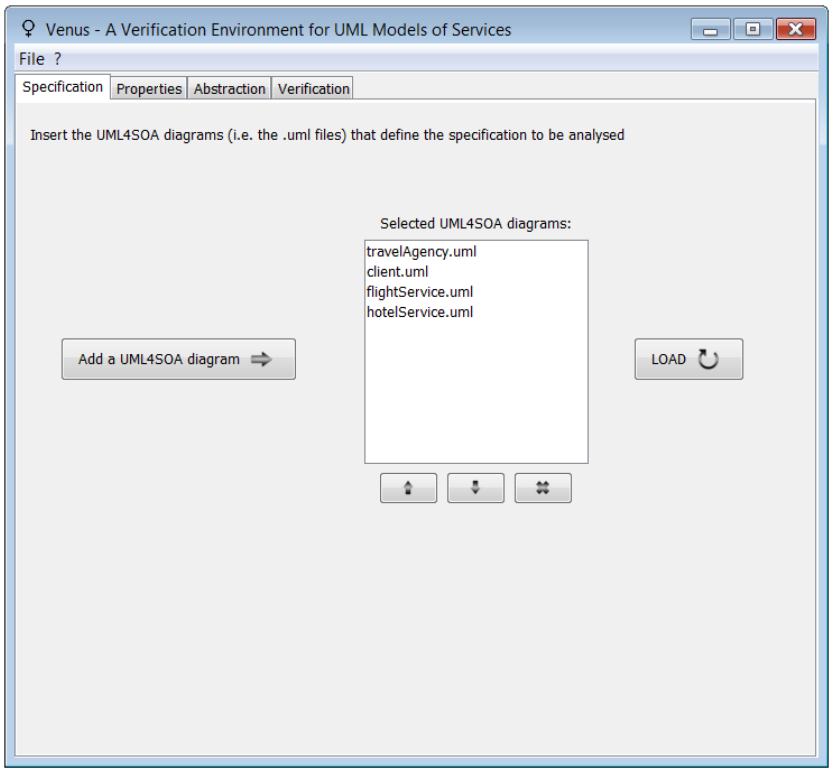


Fig. 5. Venus interface: insertion of UML4SOA diagrams

Usually a service is expected to be Responsive, in order to guarantee that the clients know the response to their requests. One-shot services are not persistent, since they lose their full functionalities after providing a positive response to a request. Services of this kind usually provide access to limited resources. To prevent ambiguity, most of the times a service is also expected to be Single-response. In some cases, however, a service is expected to provide multiple answers to a request. For instance, a service responsible of accepting paper submissions to a conference is expected to answer to a submission with several messages for, respectively, submission acceptance/rejection, paper acceptance/rejection notification, camera-ready solicitations and so on. A service is, in general, not supposed to be Broken, i.e. not able to deliver a positive response: this property is usually checked for verifying that, indeed, it is not satisfied. Similarly, a service is usually supposed to fail the check for the No-response property. Notably, to demand a service to be Reliable is a very strong requirement, since it requires every possible request to have a positive response and in most scenarios this property is supposed not to be satisfied.

Properties of the third group regard the possibility to withdraw a request. A service is said to be

Cancelable if it allows to cancel a pending request before a response has been provided;

Revocable if it allows to cancel a request after a successful response has been provided.

In many application domains it is usually desirable for a service to provide the possibility to withdraw a request. Cancelable services are said to be fair to the user. Whether a service is supposed to be Revocable depends from the intended policy. For instance, a company offering an on-line booking service may decide to provide or not the service with the possibility to cancel a booking.

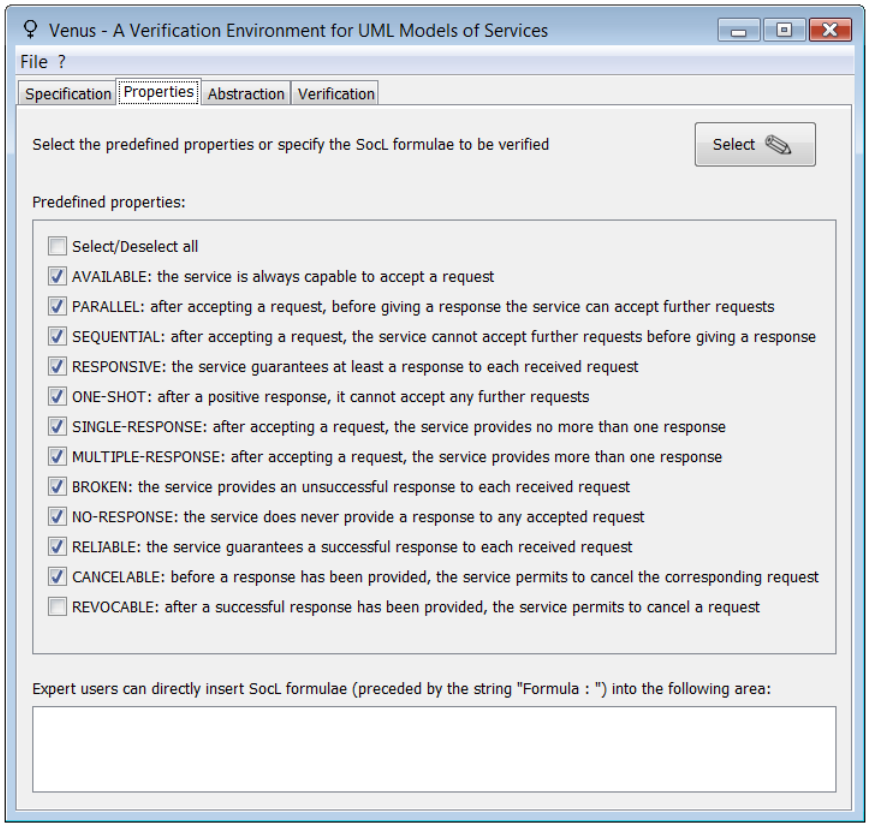


Fig. 6. Venus interface: selection of service properties

Besides these general properties, expert users can write down additional (domain specific) properties in the textarea at the bottom of the window (see again Figure 6). These properties must be expressed as formulae of the temporal logic SocL described in Section 6.

Once the properties to be verified have been selected, the user has to specify, within the loaded UML4SOA models, which are the relevant operations and what is their role w.r.t. the selected properties. More specifically, he has to specify the operations representing initial requests, positive responses, negative responses, cancellations and revocations. Moreover, he can add, for each operation, the corresponding correlation identifier [2] , which can be either a value or a write once variable. To shepherd the user for selecting the proper correlation identifier, the tool appends to the operation name its type (e.g. r for receive, s for send, s&r_r for the receiving activity of a send&receive, ...) and the number of its arguments. Notice that, we need to identify the receiving and sending activities of send&receive and receive&send actions, since they are equipped with two distinct tuples of arguments.

In our example (Figure 7) we specify that an invocation of operation request corresponds to sending an initial request to the agency service and the value that will be assigned to variable id will be used to correlate subsequent positive responses, negative responses and cancellations to such request (sent by operations reply, reservationFailure and cancel, respectively). Notice

[2] For the time being, Venus only supports the usage of a single correlation datum, which is however adequate in most cases.

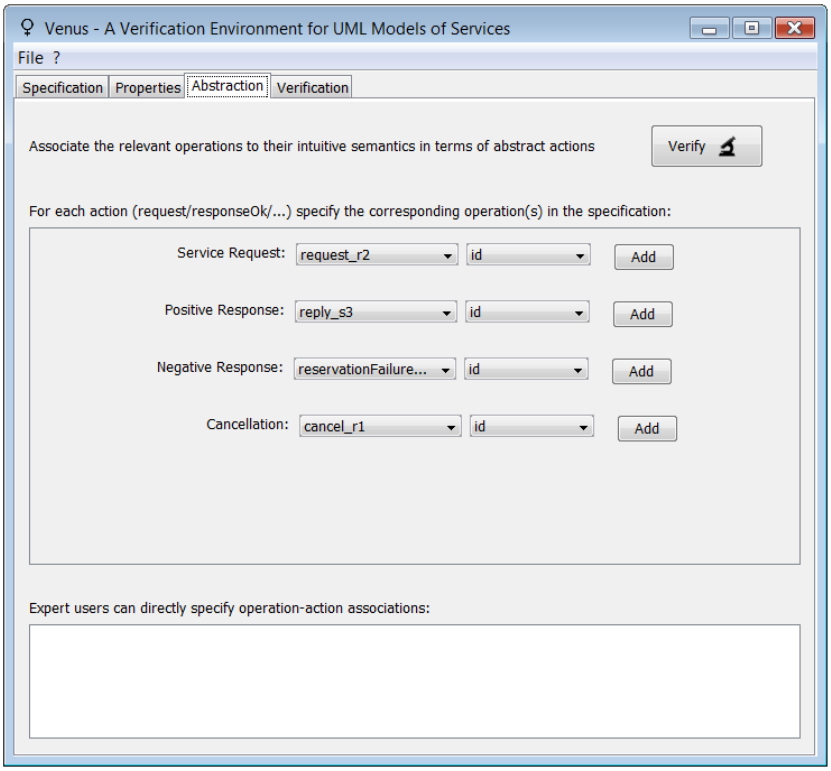


Fig. 7. Venus interface: definition of the intuitive semantics of the relevant operations

that Venus requires to specify such operations only for the roles that are needed for checking the properties previously selected (in our example, initial request, positive response, negative response and cancellation). Moreover, for each role, more than one operation can be specified by using the associated 'Add' button, thus covering the possibility that more than one operation might play the same role. The formal mechanism supporting the association of UML4SOA operations to roles is known as *abstraction rules* and is described in Section 6. Expert users can also create their own roles and associate operations to them by specifying suitable abstraction rules in the textarea at the bottom of the window.

Finally, the tool allows the user to check the validity of each property (Figure 8) and, in case of a negative result, to require an explanation. Notably, Venus also displays the SocL formula corresponding to each predefined property that has been selected for model checking.

For instance, the agency service of Section 2 is Available, Parallel, Responsive and Single-Response. It is not Reliable, since it rejects a request whenever the flight or the hotel cannot be booked. More interestingly, the service is not Cancelable; the explanation provided by the tool shows that this happens because, after the exception abort is raised, the event handler containing the operation cancel is disabled before starting the fault handler.

Venus architecture. Venus is implemented in Java to guarantee its portability across different platforms and to exploit the well-established libraries for parsing XML documents and developing graphical interfaces. As shown in Figure 9, the tool is composed of three main components: the *graphical user interface* (GUI), implemented by resorting to Java Swing library (Sun Mi-

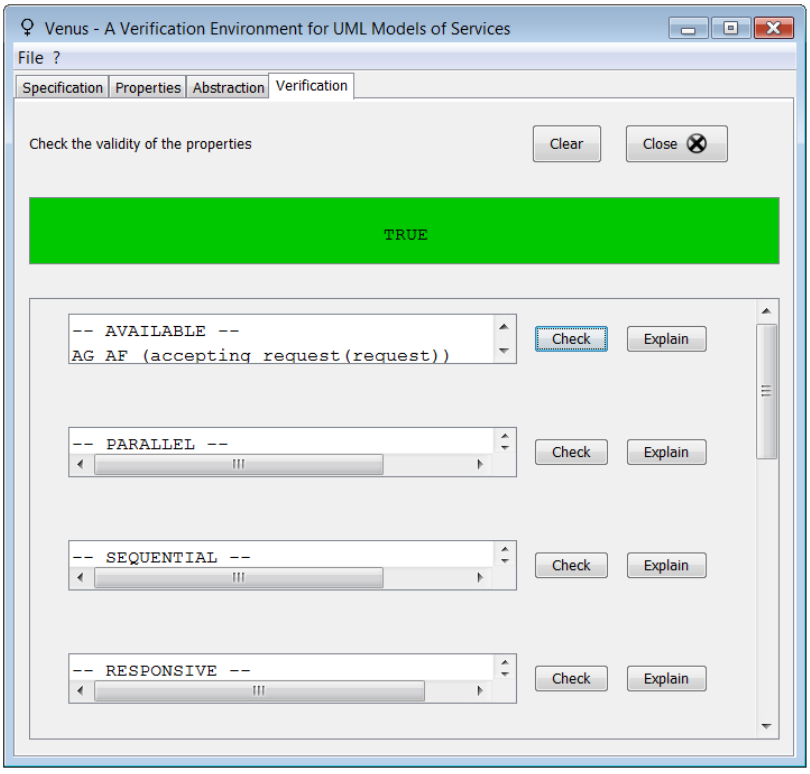


Fig. 8. Venus interface: verification of the service properties

crosystems, 2009); the automatic translator from UML4SOA into COWS, called UStoC; the model checker CMC [3] supporting verification of SocL formulae over COWS terms.

UStoC takes the XMI files storing the UML4SOA diagrams and encodes them into a COWS term. The term is then passed as an input to the model checker CMC. Similarly, the properties and the associations that the user has selected by means of the GUI are translated into, respectively, SocL formulae and abstraction rules and passed to CMC as an additional input. Finally, CMC checks the properties and displays the results and, when requested, their explanations to the user.

In our exposition, besides the automatic verification of general properties, we mentioned advanced functionalities of Venus targeted to expert users. These functionalities require the user to be familiar with the underlying logical and computational framework of Venus. The purpose of the following sections is to unveil this framework.

4. COWS: a Calculus for Orchestration of Web Services

COWS (Calculus for the Orchestration of Web Services, (Lapadula et al., 2007a)) is a recently proposed process calculus for specifying and combining services while modelling their dynamic behaviour. It provides a novel combination of constructs and features borrowed from well-known calculi such as non-binding receiving activities, asynchronous communication, polyadic synchronization, pattern matching, protection, and delimited receiving and killing activities. These features make it easier to model service instances with shared state, processes playing more than one

[3] Venus currently incorporates the version v0.7g of CMC.

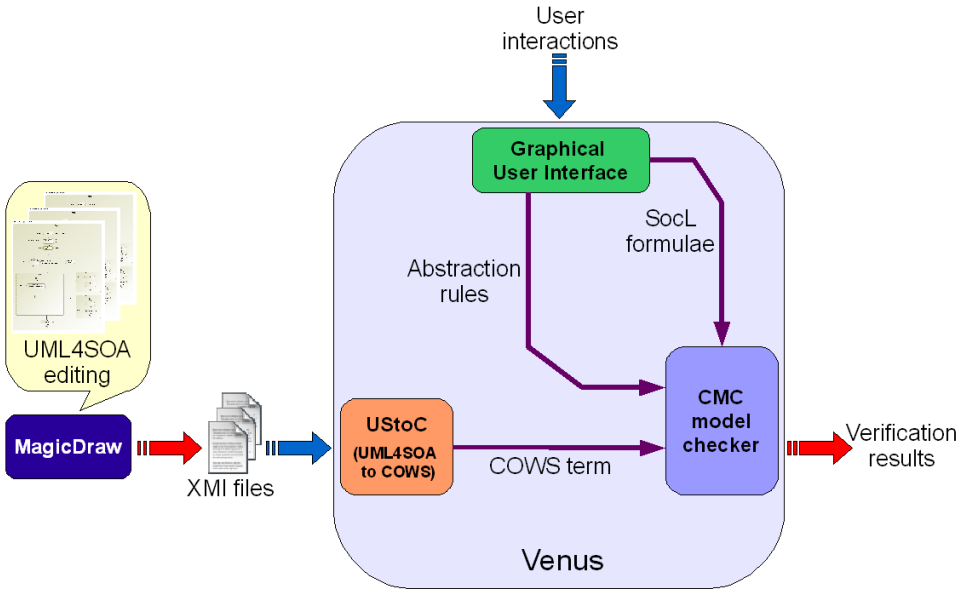


Fig. 9. Venus architecture

$$
\begin{array}{rcll}
s & ::= & u \cdot u'!\bar{\epsilon} \mid g & \text{(invoke, receive-guarded choice)} \\
& \mid & [e]\, s \mid s \mid s \mid * s & \text{(delimitation, parallel composition, replication)} \\
& \mid & \mathbf{kill}(k) \mid \{\!|s|\!\} & \text{(kill, protection)} \\
g & ::= & \mathbf{0} \mid p \cdot o?\bar{w}.s \mid g + g & \text{(empty, receive prefixing, choice)}
\end{array}
$$

Fig. 10. COWS syntax

partner role, stateful sessions made by several correlated service interactions, and long-running transactions, inter alia.

The syntax of COWS is presented in Figure 10. 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 $v, v', \ldots$) and the set of 'write once' *variables* (ranged over by $x, y, \ldots$). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by $n, m, o, p, \ldots$, mainly used to represent partners and operations. The syntax of *expressions*, ranged over by ϵ, is deliberately omitted; we just assume that they contain, at least, values and variables, but do not include killer labels (that, hence, can *not* be exchanged in communication).

We use w to range over values and variables, u to range over names and variables, and e to range over *elements*, i.e. killer labels, names and variables. The *bar* $\bar{}$ denotes tuples (ordered sequences) of homogeneous elements, e.g. $\bar{x}$ is a compact notation for denoting a tuple of variables as $\langle x_1, \ldots, x_n \rangle$. We assume that variables in the same tuple are pairwise distinct. We adopt the following conventions for operators' precedence: monadic operators bind more tightly than parallel, and prefixing more tightly than choice. We omit trailing occurrences of $\mathbf{0}$ and write $[e_1, \ldots, e_n]\, s$ in place of $[e_1] \ldots [e_n]\, s$. Finally, we write $I \triangleq s$ to assign a name I to the term s.

Invoke and *receive* are the basic communication activities provided by COWS. 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. An invoke $p \bullet o!\bar{\epsilon}$ can proceed as soon as the evaluation of the expressions $\bar{\epsilon}$ in its argument returns the corresponding values. A receive $p \bullet o?\bar{w}.s$ offers an invocable operation o along a given partner name p. Execution of a receive within a *choice* permits to take a decision between alternative behaviours. 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 makes it easier to model many service interaction and reconfiguration patterns.

The *delimitation* operator is the *only* binder of the calculus: $[e]\, s$ binds e in the scope s. The scope of names and variables can be extended while that of killer labels cannot (in fact, they are not communicable values). Besides for generating 'fresh' private names (as 'restriction' in π-calculus Milner et al. (1992)), delimitation can be used for introducing a named scope for grouping certain activities. It is then possible to associate suitable termination activities to such a scope, as well as ad hoc fault and compensation handlers, thus laying the foundation for guaranteeing *transactional properties* in spite of services' loose coupling. This can be conveniently done by relying on the *kill* activity $\mathbf{kill}(k)$, that causes immediate termination of all concurrent activities inside the enclosing $[k]$ (which stops the killing effect), and the *protection* operator $\{\!|s|\!\}$, that preserves intact a critical activity s also when one of its enclosing scopes is abruptly terminated.

Delimitation can also be used to regulate the range of application of the substitution generated by an inter-service communication. This takes place when the arguments of a receive and of a concurrent invoke along the same endpoint match and causes each variable argument of the receive to be replaced by the corresponding value argument of the invoke within the whole scope of variable's declaration. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables.

Execution of concurrent terms is interleaved, but when a kill activity or a communication can be performed. Indeed, the *parallel* operator is equipped with a priority mechanism which allows some actions to take precedence over others. Kill activities are assigned greatest priority 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 compensation handling. Additionally, receive activities are 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. In the end, this permits to correlate different service communications, thus implicitly creating interaction sessions.

Finally, the *replication* operator $* s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to model persistent services, i.e. services which can create multiple instances to serve several requests simultaneously.

The rigorous syntax and semantics of COWS provide a formalism for encoding and simulating UML4SOA models of services. In the next section we will show the actual encoding implemented in UStoC, one of the Venus components shown in Figure 9, to transform UML4SOA diagrams into COWS terms.

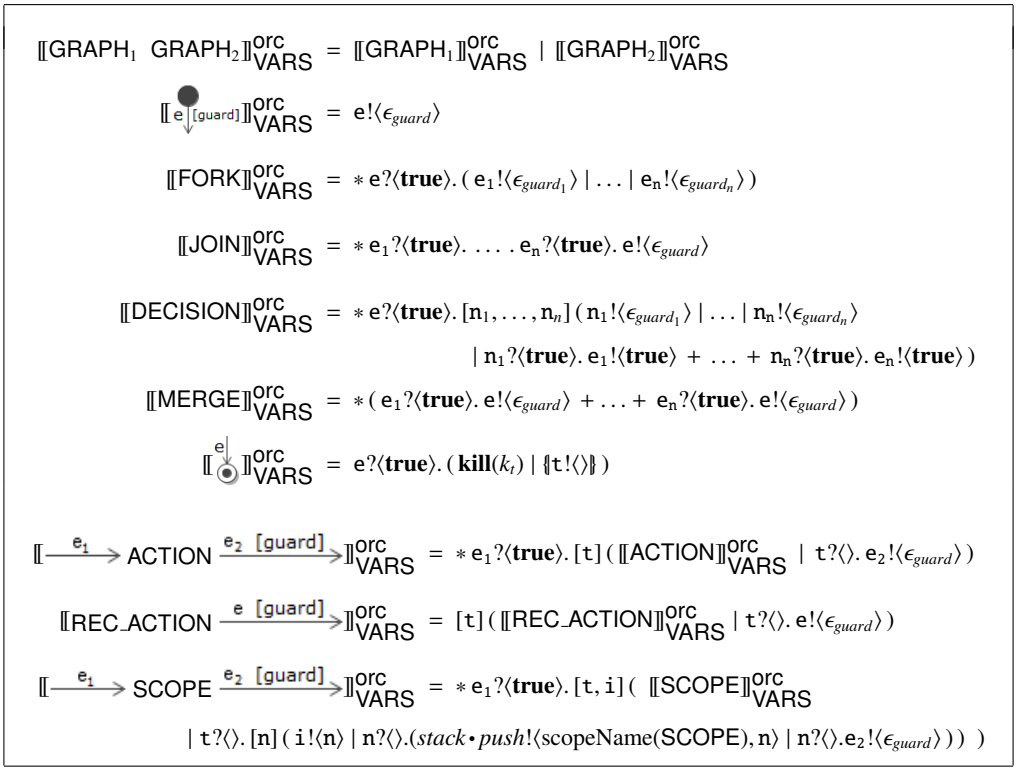


Fig. 11. Encoding of graph elements

5. A translation of UML4SOA diagrams into COWS terms

The encoding of UML4SOA diagrams in COWS illustrated herein was firstly presented in (Banti et al., 2009c). The encoding is *compositional*, in the sense that the translation of an activity diagram is given by the (parallel) composition of the encodings of all its elements. We first underline the general layout, then provide specific explanations along with the presentation of each case. We refer the reader to Figure 4 for the names of the encoded UML4SOA elements.

At top level, an orchestration ORC is encoded through an encoding function $[\![\cdot]\!]$ that returns a COWS term. Function $[\![\cdot]\!]$ is in turn defined by another encoding function $[\![\cdot]\!]^{\mathsf{orc}}_{\mathsf{VARS}}$ that, given an element of a diagram, returns a COWS term and has the two additional arguments, the name orc of the enclosing orchestration and the names of the variables defined at the level of the encoded element. The argument orc is used for translating the communication activities, by specifying who is sending/receiving messages. The variable names VARS are necessary for delimiting the scope of the variables used by the translated element. Variables are fundamental since, as we shall show, they are used to share received messages among the various elements of a scope and, moreover, they can also be instantiated as names of partner links.

Graphs. We start by providing in Figure 11 the encoding of the graph elements, i.e. nodes with incoming and outgoing edges, treating for now actions and scopes as black boxes and focusing on the encoding of passage of control among nodes. The encoding of a GRAPH is given simply by the parallel execution of all the COWS processes resulting from the encoding of its elements. An element of a graph is encoded as a process receiving and sending signals by its

incoming and outgoing edges, respectively. These edges are respectively translated as receive and invoke activities, where each edge name e is encoded by a COWS endpoint e. A guard is encoded by a COWS (boolean) expression ϵ_{guard}. Guards are exchanged as boolean values between invoke and receive activities and the communication is allowed only if the evaluation of a guard is **true**. With the exception of initial and final nodes, the encoding of every node is a COWS process made persistent by using replication, since a node can be visited several times in the same workflow (this may occur if the activity diagram contains cycles). Practically, an initial node is translated as a signal along its outgoing edge. The encoding of a FORK node is a COWS service that can be instantiated by performing a receive activity corresponding to the incoming edge. After the synchronization, an invoke activity is simultaneously activated for each outgoing edge. The encoding of a JOIN node is a service performing a sequence of receive activities, one for each incoming edge, and of an activity invoking its outgoing edge. The order of the receive activities does not matter, since, anyway, to complete its execution, i.e. to invoke the outgoing edge, synchronization over all incoming edges is required. In the encoding of a DECISION node, the endpoints $\mathtt{n}_1, \ldots, \mathtt{n}_n$ (one for each outgoing edge) are locally delimited and used for implementing a non-deterministic guarded-choice that selects *one* endpoint among those whose guard evaluates to **true**, thus enabling the invocation of the corresponding outgoing edge. A MERGE node is encoded as a choice guarded by all its incoming edges; all guards are followed by an invoke of its outgoing edge. Final nodes, when reached, enable a kill activity $\mathbf{kill}(k_t)$, where the killer label k_t is delimited at scope level, that instantly terminates all the unprotected processes in the encoding of the enclosing scope (but without affecting other scopes). Simultaneously, the protected term $\mathtt{t}!\langle\rangle$ sends a termination signal to start the execution of (possible) subsequent activities.

An ACTION node with an incoming and an outgoing edge is encoded as a service performing a receive on the incoming edge followed by the encoding of ACTION and, in parallel, a process waiting for a termination signal sent from the encoding of ACTION along the internal endpoint t and then performing an invoke on the outgoing edge. Of course, t is delimited to avoid undesired synchronization with other processes. A REC_ACTION node with an outgoing edge and without an incoming one is encoded as a service performing the encoding of REC_ACTION in parallel with the process handling the termination signal. The encoding of a SCOPE node is similar to that of an ACTION node, with two main additions. When a SCOPE terminates, the encoding of its node sends a fresh endpoint n along i enabling the compensation related to the scope, awaits for an acknowledgement along n and sends its name and n to the local *Stack* process in case compensation activities are started (see the encoding of compensation handlers below for further explanations). After another acknowledgement, it performs an invoke on the outgoing edge. Function scopeName(·), given a scope, returns its name.

Actions. The encoding of actions is shown in Figure 12. Sending and receiving actions are translated by relying on, respectively, COWS invoke and receive activities. Special care must be taken to ensure that a sent message is received *only* by the intended RECEIVE action and partner. For this purpose, in encoded terms, the action names are used as operation names, and the name orc of the orchestration enclosing the receive action is used as partner name. A SEND and a RECEIVE action can exchange messages only if they share the same name.

Action SEND is an asynchronous call: message $\langle \mathsf{expr}_1, \ldots, \mathsf{expr}_n \rangle$ is sent to the partner p and the process proceeds without waiting for a reply. This is encoded in COWS by an invoke activity sending the tuple $\langle \mathsf{orc}, \epsilon_{expr_1}, \ldots, \epsilon_{expr_n} \rangle$, where orc indicates the sender of the message and will be used by the receiver to (possibly) provide a reply. The invoked partner p is rendered either as

$$
\begin{aligned}
[\![\mathsf{SEND}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= \{\!|\, [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \bullet \mathsf{name}!\langle \mathsf{orc}, \epsilon_{expr_1}, \ldots, \epsilon_{expr_n} \rangle \,|\!\} \mid \mathtt{t}!\langle\rangle \\
[\![\mathsf{RECEIVE}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= \mathsf{orc} \bullet \mathsf{name}?\langle [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, [\![\mathsf{X}_1]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, \ldots, [\![\mathsf{X}_n]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \rangle.\, \mathtt{t}!\langle\rangle \\
[\![\mathsf{SEND\&RECEIVE}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= \{\!|\, [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \bullet \mathsf{name}!\langle \mathsf{orc}, \epsilon_{expr_1}, \ldots, \epsilon_{expr_n} \rangle \,|\!\} \\
&\quad \mid \mathsf{orc} \bullet \mathsf{name}?\langle [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, [\![\mathsf{X}_1]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, \ldots, [\![\mathsf{X}_m]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \rangle.\, \mathtt{t}!\langle\rangle \\
[\![\mathsf{RECEIVE\&SEND}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= \mathsf{orc} \bullet \mathsf{name}?\langle [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, [\![\mathsf{X}_1]\!]^{\mathsf{orc}}_{\mathsf{VARS}}, \ldots, [\![\mathsf{X}_m]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \rangle. \\
&\quad (\, \{\!|\, [\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} \bullet \mathsf{name}!\langle \mathsf{orc}, \epsilon_{expr_1}, \ldots, \epsilon_{expr_n} \rangle \,|\!\} \mid \mathtt{t}!\langle\rangle \,) \\
[\![\mathsf{RAISE}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= \mathbf{kill}(k_r) \mid \{\!|\, \mathtt{r}!\langle\rangle \,|\!\} \\
[\![\mathsf{COMPENSATE}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= c \bullet \mathsf{scopeName}!\langle \mathsf{scopeName} \rangle \mid \mathtt{t}!\langle\rangle \\
[\![\mathsf{COMPENSATE\_ALL}]\!]^{\mathsf{orc}}_{\mathsf{VARS}} &= [\mathtt{n}]\, (\, \mathit{stack} \bullet \mathit{compAll}!\langle \mathtt{n} \rangle \mid \mathtt{n}?\langle\rangle.\mathtt{t}!\langle\rangle \,)
\end{aligned}
$$

Fig. 12. Encoding of actions

the link p, in case p is a constant, or as the COWS variable x_{p} in case p is a write-once variable. In parallel, a termination signal along the endpoint $\mathtt{t}$ is sent for allowing the computation to proceed. $[\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}}$ is p if $\ll\mathsf{wo}\gg \mathsf{p} \notin \mathsf{VARS}$, and x_{p} otherwise; similarly, each ϵ_{expr_i} is obtained from $\mathsf{expr_i}$ by replacing each X in the expression such that $\ll\mathsf{wo}\gg \mathsf{X} \in \mathsf{VARS}$ with x_{X}. Unlike SEND, action Action RECEIVE is a blocking activity, preventing the workflow to go on until a message is received. It is encoded as a COWS receive along the endpoint orc • name, with input pattern a tuple where the first element is the encoding of the link pin p and the others are either COWS variables x_{X} if $\ll\mathsf{wo}\gg \mathsf{X} \in \mathsf{VARS}$ or variables X otherwise. This way, a message can be received if its correlation data match with those of the input pattern and, in this case, the other data are stored as current values of the corresponding variables. The encodings of actions SEND&RECEIVE and RECEIVE&SEND simply results from the composition of the encodings of actions SEND and RECEIVE.

The behavior, and thus the encoding, of a RAISE action is somehow similar to that of a final node. In both cases a kill activity is enabled, in parallel with a protected termination signal invoking an exception handler. They differ for the killer label and the endpoint along which the termination signal is sent. In this way, a RAISE action terminates all the activities in its enclosing scope (where k_r is delimited) and triggers related the exception handler (by means of signal $\mathtt{r}!\langle\rangle$). An exception can be propagated by an exception handler that executes another RAISE action. Notably, since default exception handlers simply execute a RAISE action and terminate, not specifying exception handlers results in the propagation of the exception to the further enclosing scope until eventually reaching the top level and thus terminating the whole orchestration. Action COMPENSATE is encoded as an invocation of the compensation handler installed for the target scope. Action COMPENSATE_ALL is encoded as an invocation of the local *Stack* process requiring it to execute (in reverse order w.r.t. scopes completion) all the compensation handlers installed within the enclosing scope.

Variables, scopes and orchestrations. The encoding of the variables delimited within scopes, scopes (and related handlers) themselves, and whole orchestrations is shown in Figure 13.

$$[\![\mathsf{nil}]\!] = \mathbf{0} \qquad [\![\mathsf{X}, \mathsf{VARS}]\!] = \mathit{Var}_{\mathsf{X}} \mid [\![\mathsf{VARS}]\!] \qquad [\![\ll \mathsf{wo} \gg \mathsf{X}]\!], \mathsf{VARS} = [\![\mathsf{VARS}]\!]$$

$$[\![\mathsf{SCOPE}]\!]^{\mathsf{orc}}_{\mathsf{VARS}'} = [\mathtt{e}, \mathrm{vars}(\mathsf{VARS})]$$
$$([\mathit{stack}]\,([\mathtt{r}]\,([k_r, k_t]\,([\![\mathsf{GRAPH}]\!]^{\mathsf{orc}}_{\mathsf{VARS}',\mathsf{VARS}} \mid \{\!|\mathit{Stack}|\!\}$$
$$\mid *[\mathtt{t}, k_t]\,[\![\mathsf{GRAPH}_{\mathsf{ev}\,1}]\!]^{\mathsf{orc}}_{\mathsf{VARS}',\mathsf{VARS}}$$
$$\mid \ldots \mid *[\mathtt{t}, k_t]\,[\![\mathsf{GRAPH}_{\mathsf{ev}\,n}]\!]^{\mathsf{orc}}_{\mathsf{VARS}',\mathsf{VARS}})\mid \mathtt{r}?\langle\rangle.\,\mathtt{e}!\langle\rangle)$$
$$\mid [\![\mathsf{VARS}]\!] \mid \mathtt{e}?\langle\rangle.\,\{\!|[\mathtt{t}, k_t]\,[\![\mathsf{GRAPH}_{\mathsf{e}}]\!]^{\mathsf{orc}}_{\mathsf{VARS}',\mathsf{VARS}}|\!\}))$$
$$\mid [y]\,\mathtt{i}?\langle y\rangle.\,\{\!|\,y!\langle\rangle \mid c \bullet \mathsf{scopeName}?\langle \mathsf{scopeName}\rangle.$$
$$[\mathtt{t}]\,([k_t]\,[\![\mathsf{GRAPH}_{\mathsf{C}}]\!]^{\mathsf{orc}}_{\mathsf{VARS}',\mathsf{VARS}} \mid \mathtt{t}?\langle\rangle.\,\mathit{stack} \bullet \mathit{end}!\langle \mathsf{scopeName}\rangle)$$
$$\mid *[x]\,c \bullet \mathsf{scopeName}?\langle x\rangle.\,\mathit{stack} \bullet \mathit{end}!\langle \mathsf{scopeName}\rangle\,|\!\}))$$

$$[\![\mathsf{ORC}]\!] = \mathrm{isPersistent}(\mathsf{SCOPE})\,[k_r, c, \mathtt{t}, \mathtt{r}, \mathtt{i}, \mathit{stack}, \mathrm{edges}(\mathsf{SCOPE})]\,[\![\mathsf{SCOPE}]\!]^{\mathsf{orc}}_{\mathsf{nil}}$$

Fig. 13. Encoding of variables, scopes and orchestrations

Variables declared write-once (by means of $\ll \mathsf{wo} \gg$) directly corresponds to COWS variables (as we have seen, e.g., in the encoding of SEND). The remaining variables, i.e. variables that store values and can be rewritten several times (as usual in imperative programming languages), are encoded as internal services accessible only by the elements of the scope. Specifically, a variable X is rendered as a service $\mathit{Var}_{\mathsf{X}}$ providing 'read' and 'write' functionalities along the public partner name X. When the service variable is initialized (i.e. the first time the 'write' operation is used), an instance is created that is able to provide the value currently stored. When this value must be updated, the current instance is terminated and a new instance is created which stores the new value.

$$\begin{aligned}\mathit{Var}_{\mathsf{X}} \triangleq\ & [x_v, x_a]\,\mathsf{X} \bullet \mathit{write}?\langle x_v, x_a\rangle.\\ & [\mathtt{n}]\,(\mathtt{n}!\langle x_v, x_a\rangle\\ & \quad \mid *[x, y]\,\mathtt{n}?\langle x, y\rangle.\,(y!\langle\rangle \mid [k]\,(*[y']\,\mathsf{X} \bullet \mathit{read}?\langle y'\rangle.\,\{\!|\,y'!\langle x\rangle|\!\}\\ & \qquad\qquad \mid [x', y']\,\mathsf{X} \bullet \mathit{write}?\langle x', y'\rangle.\\ & \qquad\qquad\quad (\mathbf{kill}(k) \mid \{\!|\,\mathtt{n}!\langle x', y'\rangle|\!\}))))\end{aligned}$$

Service $\mathit{Var}_{\mathsf{X}}$ provides two operations: *read*, for getting the current value; *write*, for replacing the current value with a new one. To access the service, a user must invoke these operations by providing a communication endpoint for the reply and, in case of *write*, the value to be stored. The *write* operation can be invoked along the public partner X; the first time, it corresponds to initialization of the variable. $\mathit{Var}_{\mathsf{X}}$ uses the delimited endpoint n to store the current value of the variable. This permits to implement further *read* operations in terms of forced termination and re-instantiation. Delimitation $[k]$ is used to confine the effect of the kill activity to the current instance, while protection $\{\!|\_|\!\}$ avoids forcing termination of pending replies and of the invocation that will trigger the new instance.

Variables like X may (temporarily) occur in expressions used by invoke and receive activities within COWS terms obtained as result of the encoding. To get rid of these variables and finally

obtain 'pure' COWS terms, we exploit the following encodings:

$$\begin{array}{rcll}
\langle\!\langle u \bullet u'!\bar{\epsilon} \rangle\!\rangle & = & [\mathtt{m}, \mathtt{n}_1, \ldots, \mathtt{n}_m] & \text{if } \bar{\epsilon} \text{ contains } \mathsf{X}_1, \ldots, \mathsf{X}_m \\
& & (\, \mathsf{X}_1 \bullet \mathit{read}!\langle \mathtt{n}_1 \rangle \mid \ldots \mid \mathsf{X}_m \bullet \mathit{read}!\langle \mathtt{n}_m \rangle & \\
& & \mid [x_1, \ldots\ x_m]\, \mathtt{n}_1?\langle x_1 \rangle.\ \ldots\ .\, \mathtt{n}_m?\langle x_m \rangle.\ \mathtt{m}!\, \bar{\epsilon} \cdot \{\mathsf{X}_i \mapsto x_i\}_{i \in \{1,\ldots,m\}} & \\
& & \quad \mid [\bar{x}]\, \mathtt{m}?\bar{x}.\, u \bullet u'!\bar{x}\,) &
\end{array}$$

$$\begin{array}{rcll}
\langle\!\langle p \bullet o?\bar{w}.s \rangle\!\rangle & = & [x_1, \ldots\ x_m] & \text{if } \bar{w} \text{ contains } \mathsf{X}_1, \ldots, \mathsf{X}_m \\
& & p \bullet o?\bar{w} \cdot \{\mathsf{X}_i \mapsto x_i\}_{i \in \{1,\ldots,m\}}\, . & \\
& & \quad [\mathtt{n}_1, \ldots, \mathtt{n}_m]\, (\mathsf{X}_1 \bullet \mathit{write}!\langle x_1, \mathtt{n}_1 \rangle \mid \ldots \mid \mathsf{X}_m \bullet \mathit{write}!\langle x_m, \mathtt{n}_m \rangle & \\
& & \qquad \mid \mathtt{n}_1?\langle\rangle.\ \ldots\ .\, \mathtt{n}_m?\langle\rangle.\ \langle\!\langle s \rangle\!\rangle) &
\end{array}$$

where $\{\mathsf{X}_i \mapsto x_i\}$ denotes substitution of X_i with x_i, and endpoint $\mathtt{m}$ returns the result of evaluating $\bar{\epsilon}$ (of course, we are assuming that $\mathtt{m}$, $\mathtt{n}_i$ and x_i are fresh).

A SCOPE is encoded as the parallel execution, with proper delimitations, of the processes resulting from the encoding of all its components. Function vars(·), given a list of variables VARS, returns a list of COWS variables/names, where a COWS name X corresponds to a variable X in VARS, while a COWS variable x_{X} corresponds to a variable ≪wo≫ X in VARS. The (private) endpoint r catches signals generated by RAISE actions and activate the corresponding handler, by means of the (private) endpoint e. Killer labels k_r and k_t are used to delimit the field of action of kill activities generated by the translation of action RAISE or of final nodes, respectively, within GRAPH. When a scope successfully completes, its compensation handler is installed by means of a signal along the endpoint i. Installed compensation handlers are protected to guarantee that they can be executed despite of any exception. Afterwards, the compensation can be activated by means of the partner name *c*. Notably, a compensation handler can be executed only once. After that, the term $* [x]\, c \bullet \mathsf{scopeName}?\langle x \rangle.\, \mathit{stack} \bullet \mathit{end}!\langle \mathsf{scopeName} \rangle$ permits to ignore further compensation requests (by also taking care not to block the compensation chain).

The (protected) *Stack* service associated to a scope offers, along the partner name *stack*, three operations: *end* to catch the termination of the scope specified as argument of the operation, *push* to stack the scope name specified as argument of the operation into the associated *Stack*, and *compAll* that triggers the compensation of all scopes whose names are in *Stack*. The specification of *Stack* is as follows:

$$\begin{array}{l}
[q]\, (\, \mathit{Lifo} \mid\ * [x, y]\, \mathit{stack} \bullet \mathit{push}?\langle x, y \rangle.\, q \bullet \mathit{push}!\langle x, y \rangle \\
\qquad \mid\ * [x]\, \mathit{stack} \bullet \mathit{compAll}?\langle x \rangle.\, [\mathtt{loop}]\, (\, \mathtt{loop}!\langle\rangle \mid * \mathtt{loop}?\langle\rangle.\, \mathit{Comp}\,)\,)
\end{array}$$

where loop is used to model a while cycle executing *Comp*. The term *Comp* pops a scope name scopeName out of *Lifo* and invokes the corresponding compensation handler (by means of $c \bullet \mathsf{scopeName}!\langle \mathsf{scopeName} \rangle$); in case *Lifo* is empty, the cycle terminates and a termination signal is sent along the argument *x* of the operation *compAll*.

$$\begin{array}{rcl}
\mathit{Comp} & \triangleq & [\mathtt{r}, \mathtt{e}]\, (\, q \bullet \mathit{pop}!\langle \mathtt{r}, \mathtt{e} \rangle \mid [y]\, (\mathtt{r}?\langle y \rangle.\, (c \bullet y!\langle y \rangle \mid \mathit{stack} \bullet \mathit{end}?\langle y \rangle.\, \mathtt{loop}!\langle\rangle) \\
& & \qquad + \ \mathtt{e}?\langle\rangle.\, x!\langle\rangle)\,)
\end{array}$$

Lifo is an internal queue providing 'push' and 'pop' operations. *Stack* can push and pop a scope name into/out of *Lifo* via $q \bullet \mathit{push}$ and $q \bullet \mathit{pop}$, respectively. To push, *Stack* sends the value to be inserted, while to pop sends two endpoints: if the queue is not empty, the last inserted value is removed from the queue and returned along the first endpoint, otherwise a signal along the

second endpoint is received. Each value in the queue is stored as a triple made available along the endpoint $\mathtt{h}$ and composed of the actual value, and two correlation values working as pointers to the previous and to the next element in the queue. The correlation value retrieved along $\mathtt{m}$ is associated with the element on top of the queue, if this is not empty, otherwise it is *empty*.

$$
\begin{aligned}
\mathit{Lifo} \triangleq\ & [\mathtt{m},\mathtt{h}]\,(*\,[y_v, y_r, y_e, y] \\
& \quad (\,q \cdot \mathit{push}?\langle y_v, y\rangle.[z]\,(\mathtt{m}?\langle z\rangle.\,[c]\,(\,\mathtt{h}!\langle y_v, z, c\rangle \mid \mathtt{m}!\langle c\rangle \mid y!\langle\rangle\,)\,) \\
& \qquad + q \cdot \mathit{pop}?\langle y_r, y_e\rangle.[z]\,(\mathtt{m}?\langle z\rangle.[y_v, y_t]\,\mathtt{h}?\langle y_v, y_t, z\rangle.(\,\mathtt{m}!\langle y_t\rangle \mid y_r!\langle y_v\rangle\,) \\
& \qquad\qquad + \mathtt{m}?\langle \mathit{empty}\rangle.(\,\mathtt{m}!\langle \mathit{empty}\rangle \mid y_e!\langle\rangle\,)\,)\,) \\
& \quad \mid \mathtt{m}!\langle \mathit{empty}\rangle\,)
\end{aligned}
$$

Notice that, because of the COWS's (prioritized) semantics, whenever the queue is empty, the presence of receive $\mathtt{m}?\langle \mathit{empty}\rangle$ prevents taking place of the synchronization between $\mathtt{m}!\langle \mathit{empty}\rangle$ and $\mathtt{m}?\langle z\rangle$.

The encoding of an orchestration is that of its top-level scope. Function isPersistent$(\cdot)$ returns either the replication symbol $*$ if the top-level scope directly contains at least a REC_ACTION node or the empty string otherwise; function edges$(\cdot)$ returns the names of all the edges of the graphs contained within its argument scope.

The encoding presented in this section permits to transform UML4SOA diagrams, with an informal semantics, into COWS terms equipped with a formal semantics and a precise behavior. The logic SocL described in the next section permits to express behavioral properties to be checked over the COWS terms generated from the encoding by means of the model checker CMC.

6. The logic SocL

The service properties informally described in Section 3 require a rigorous definition in order to be verified over a COWS specification. Venus internally represents these properties as SocL formulae. SocL is a an action- and state-based, branching time logic that makes use of high level temporal operators drawn from mainstream logics like CTL (Clarke and Emerson, 1981), ACTL (De Nicola and Vaandrager, 1990) and ACTLW (Meolic et al., 2008). SocL has been specifically designed to express in an effective way distinctive aspects of services. Indeed, by taking inspiration from the SOC emerging standard WS-BPEL, SocL permits to link together actions executed as part of the same interaction by means of a correlation mechanism. SocL formulae can be checked over a COWS term by the on-the-fly model checker CMC. Both SocL and CMC are part of a methodology for verifying functional properties of services introduced in (Fantechi et al., 2008). Here we briefly report the main ingredients of the logic and refer the interested reader to (Fantechi et al., 2008) for a formal account of the semantics of SocL formulae.

The SocL 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 propositions that are true over them, and of state changes and actions performed when moving from one state to another. In our approach, each proposition expresses the service capability to perform an action. In fact, the interpretation domain of SocL formulae are *Doubly Labelled Transition Systems* (L^2TSs, (De Nicola and Vaandrager, 1995)), namely extensions of *Labelled Transition Systems* (LTSs), where labels represent executed actions, enriched with a labelling function from states to sets

$$\gamma ::= \underline{\alpha} \mid \chi \qquad \chi ::= tt \mid \alpha \mid \tau \mid \neg\chi \mid \chi \wedge \chi \qquad \text{(action formulae)}$$

$$\phi ::= true \mid \pi \mid \neg\phi \mid \phi \wedge \phi' \mid E\Psi \mid A\Psi \qquad \text{(state formulae)}$$

$$\Psi ::= X_\gamma \phi \mid \phi \,{}_\chi U_\gamma\, \phi' \mid \phi \,{}_\chi W_\gamma\, \phi' \qquad \text{(path formulae)}$$

Fig. 14. SocL syntax

of propositions. An action has a *type*, e.g. accept a request, provide a response, etc., and is part of 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. 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 characterized 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 $\underline{\cdot}$ emphasizes the fact that the action may contain variable binders), and α as a generic action without variable binders.

For example, action $request(tr, 1234, 1)$ could stand for a *request* action for starting an (instance of the) interaction *tr* which will be identified through the correlation tuple $\langle 1234, 1\rangle$. A *response* action corresponding to the request above, for example, could be written as $response(tr, 1234, 1)$. Moreover, 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 $request(tr, 1234, \underline{id})$. A corresponding response action could be written as $response(tr, 1234, id)$, where the (free) occurrence of the correlation variable *id* indicates the connection with the action where the variable is bound. Similarly, actions like $cancel(tr, 1234, id)$, $fail(tr, 1234, id)$ and $undo(tr, 1234, id)$ could indicate *cancellation*, *failure* and *compensation* notification for the same request.

The syntax of SocL formulae is presented in Figure 14. 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 $\neg tt$, $\chi \vee \chi'$ to abbreviate $\neg(\neg\chi \wedge \neg\chi')$ and $\phi_1 \Rightarrow \phi_2$ to abbreviate $\neg\phi_1 \vee \phi_2$. π denotes a *proposition*, that is a property that can be true over the states of services. 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 (resp.) *path quantifiers*. X, U and W are the *next*, *(strong) until* and *weak until* operators drawn from those firstly introduced in (De Nicola and Vaandrager, 1990) and subsequently elaborated in (Meolic et al., 2008). 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 logic 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*.

6.1. A few templates of service properties specified with SocL

We now show how the service properties that can be selected in Venus can be expressed as formulae in SocL. These properties use the following service actions *request*(i, var) (accepting a request), *responseOk*(i, var) (positively answering to a request), *responseFail*(i, var) (negatively answering to a request), *cancel*(i, var) (accepting a cancellation of a request), *undo*(i, var) (accepting a revocation of a request), and the following propositions, expressing the potential capability of the service to perform an action, *accepting_request*(i) (capability to accept a request), *accepting_cancel*(i, var) (capability to accept a cancellation) and *accepting_undo*(i, var) (capability to accept a revocation). The properties are formalised as follows:

(1) - - *Available* service - -

$$AG\,AF\,(accepting\_request(i)).$$

This formula means that in every state the service eventually accepts a request.

(2) - - *Parallel* service - -

$$AG\ [request(i, \underline{var})]$$
$$E(true_{\neg\,(responseOk(i,var)\vee responseFail(i,var))}\,U\ accepting\_request(i)).$$

This formula means that the service can serve several requests simultaneously. Indeed, in every state, if a request is accepted then, in some future state, a further request for the same interaction can be accepted before giving a response to the first accepted request. Notably, the responses belongs to the same interaction i of the accepted request and they are correlated by the variable *var*.

(3) - - *Sequential* service - -

$$AG\ [request(i, \underline{var})]$$
$$A(\neg\, accepting\_request(i)\,{}_{tt}U_{responseOk(i,var)\vee responseFail(i,var)}true).$$

In this case, the service can serve at most one request at a time. Indeed, after accepting a request, it cannot accept further requests for the same interaction before replying to the accepted request.

(4) - - *One-shot* service - -

$$AG\,[responseOk(i, var)]\,AG\,\neg\, accepting\_request(i).$$

This formula states that, after positive response to a request has been provided, in all future states, the service cannot accept any further request.

(5) - - *Broken* service - -

$$AG\,[request(i, \underline{var})]\,AF_{responseFail(i,var)}\,true.$$

This formula states that whenever the service accept a request, it always eventually provides an unsuccessful response.

(6) - - *Cancelable* service - -

$$AG\,[request(i, \underline{var})]$$
$$A(accepting\_cancel(i, var)\,_{tt}W_{responseOk(i,var)\vee responseFail(i,var)}true).$$

This formula means that the service is ready to accept a cancellation required by the client (fairness towards the client) before possibly providing a response to the accepted request.

(7) - - *Revocable* service - -

$$EF_{responseOk(i,\underline{var})}\,EF(accepting\_undo(i, var))$$

The meaning of this formula is that, after a successful response have been provided, the service can eventually accept an undo of the corresponding request.

(8) - - *Responsive* service - -

$$AG\,[request(i, \underline{var})]\,AF_{responseOk(i,var)\vee responseFail(i,var)}\,true.$$

The formula states that whenever the service accepts a request, it always eventually provides at least a (successful or unsuccessful) response.

(9) - - *Single-response* service - -

$$AG\,[request(i, \underline{var})]$$
$$\neg EF_{responseOk(i,var)\vee responseFail(i,var)}\,EF_{responseOk(i,var)\vee responseFail(i,var)}\,true.$$

The formula means that whenever the service accepts a request, it cannot provide two or more correlated (successful or unsuccessful) responses, i.e. it can only provide at most a single response.

(10) - - *Multiple-response* service - -

$$AG\,[request(i, \underline{var})]$$
$$AF_{responseOk(i,var)\vee responseFail(i,var)}\,AF_{responseOk(i,var)\vee responseFail(i,var)}\,true.$$

Differently from the previous formula, here the service always eventually provides two or more responses.

(11) - - *No-response* service - -

$$AG\,[request(i, \underline{var})]\,\neg EF_{responseOk(i,var)\vee responseFail(i,var)}\,true.$$

This formula means that the service never provides a (successful or unsuccessful) response to any accepted request.

(12) - - *Reliable* service - -

$$AG\,[request(i, \underline{var})]\,AF_{responseOk(i,var)}\,true.$$

This formula guarantees that in every state the service eventually provides a successful response to each accepted request.

In (Fantechi et al., 2008), semantically slightly different interpretations of the properties are provided. Users familiar with SocL may use Venus to verify these alternative versions of the properties or write down their own formulae from scratch.

6.2. CMC and the abstraction mechanism

As mentioned above, the interpretation domain of SocL are L^2TS. Hence, a model checker engine that assists the verification process of SocL formulae has to rely on such internal representation. While, in principle, UML4SOA specification could have been directly translated using this class of transition systems, these translations would have lacked of compositionality and, moreover, the obtained transition systems could have been non-finite. By relying on the process calculus COWS, instead, a service scenario is translated as the parallel composition of the translations of the individual services. The terms of a process calculus are syntactically finite, even when the corresponding semantic model, usually defined in terms of labelled transition systems, is not. Therefore, the tool CMC has been specifically developed for checking SocL formulae over L^2TSs generated from COWS terms.

A fundamental mechanism for filling the gap between COWS terms and L^2TSs is based on the so-called *abstraction rules*. In fact, the SocL formulae previously presented are stated in terms of 'abstract' actions and propositions, meaning that, e.g., a reservation is requested or the system is ready to accept a reservation request. In other words, the properties we want to verify are formalized as SocL formulae in a completely independent way of the service specification. The abstraction rules permit to link these abstract actions and propositions to the 'concrete' operations of COWS specifications, which in their turn encode UML4SOA operations. We refer the interested reader to (Fantechi et al., 2008) for a comprehensive account of this step.

To automatically generate the abstraction rules required as input by CMC, Venus shepherds the user into providing the necessary data by selecting the UML4SOA operations (and, consequently, their COWS encoding) corresponding to SocL action types (i.e. accept a request, provide a positive response, etc.). Such information are also used to instantiate the formulae templates presented in Section 6.1 to the actual formulae checked by CMC.

7. Concluding remarks and related work

We have presented Venus, a tool for automatic verification of service models specified by UML4SOA. The tool implements an automatic translation of UML4SOA activity diagrams into the process calculus COWS, and shepherds the user in the specifications of properties, internally represented as SocL formulae. The properties are then checked over the COWS term resulting from the translation by exploiting the model checker CMC. Venus allows users without any knowledge of COWS and SocL to select the properties they want to check out of a predefined list of general properties whose meaning is intuitively explained in natural language. An expert user familiar with SocL (but not necessarily with COWS) can also define its own properties directly as SocL formulae. Both expert and non-expert users are shepherded by the tool into selecting which concrete operations in the considered service scenario correspond to the abstract actions and propositions in the SocL formulae.

There are several works in the literature whose goal is the automatic verification of UML specifications. However, to the best of our knowledge, ours is the first effective (although still demonstrative) tool allowing a user solely familiar with UML activity diagrams to verify properties of services. In particular none of the existing works considers service-oriented UML profiles and, most importantly, implements facilities for formalising properties of services, like the abstraction mechanism and the menu of predefined properties presented in this work.

(Latella et al., 1999; Latella and Massink, 2001) presents a translation of UML State Chart diagrams (which differ from activity diagrams) into Promela, the input language of the Spin verification environment (Holzmann, 2003). A translation of UML state chart diagrams is also at

the base of (Jürjens and Shabalin, 2004), which focusses on security aspects of system specifications. (ter Beek et al., 2008; Dong et al., 2001; Knapp et al., 2002) presents alternative verification tools implementing a semantic for UML state chart diagrams. (Compton et al., 2000) describes a verification tool for both state chart and activity diagrams. However, the tool supports version 1.0 of UML, rather than the current version 2.0. (Arons et al., 2004) proposes a framework for automatic verification of UML state machine specifications based on (semi-)automated theorem proving over a temporal linear logic. However, the proposed approach requires a user to have high skills in automated theorem proving techniques and the ability to devise creative solutions related to the specific model and properties, while our approach tries to provide a verification environment easily accessible also to non-expert users.

In (Csertán et al., 2002) an environment for the verification of UML class diagrams based on the VIATRA framework (VIATRA2 Developer Team, 2009) is presented. Being based on class diagrams, the system is focused on structural properties rather than behavioral properties which are instead the target of our work.

The proposed encoding of UML4SOA activity diagrams into COWS also provides, as far as we know the first (transformational) formal semantics of UML4SOA. The problem of defining a formal semantics for (subsets of) UML activity diagrams has been tackled by many authors. A largely followed approach is based on (extensions of) Petri Nets (see, e.g., (Eichner et al., 2005; Störrle and Hausmann, 2005)). However, although Petri Nets can be a natural choice for encoding workflows, they seem not to fit well for such constructs as compensation, message correlation and shared variables, that are more relevant for UML4SOA. Other approaches have introduced operational semantics through transition systems (e.g. (ter Beek et al., 2008; Crane and Dingel, 2008)) and stochastic semantics (Tabuchi et al., 2005), but none of them considers the UML4SOA profile and, above all, seems to be adequate for encoding its specific constructs. Regarding UML4SOA, a software tool translating UML4SOA models into WS-BPEL is presented in (Mayer et al., 2008b). The translation, however, does not apply to all possible UML4SOA diagrams and is not compositional. Furthermore, WS-BPEL code has not an univocal semantics, thus the translation does not provide a formal semantics to UML4SOA models.

As target of our encoding, we have singled COWS out of several similar process calculi for its peculiar features, specifically the termination constructs and the correlation mechanism. In fact, kill activities are suitable for representing ordinary and exceptional process terminations, while protection permits to naturally 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. Modelling such a feature by using session-oriented calculi designed for SOA (e.g. (Lanese et al., 2007; Boreale et al., 2008)) seems to be quite cumbersome. The main reason is that UML4SOA is not session-oriented, thus the specific features of these calculi are of little help. Compared to other correlation-oriented calculi (like, e.g., (Guidi et al., 2006)), COWS seems more adequate since it relies on more basic constructs and provides verification tools.

There are a number of directions along which the presented work could evolve and extend. The current prototypical implementation of Venus still lacks the efficiency for managing the analysis of large system specifications. The way to improve this aspect of the tool it is to optimize both the automatic translation of UML4SOA diagrams into COWS terms, in order to obtain a more tractable COWS specification, and the underlying CMC model checker. Recently, another UML profile for SOA design, named SoaML (Object Management Group, 2008), has been introduced. With respect to UML4SOA, SoaML is more focused on architectural aspects

of services and relies on the standard UML 2.0 activity diagrams without further specializing them. A new version of the UML4SOA profile has been then released, which basically integrates Protocol State Machine Diagrams for modelling services external to a given orchestration. We plan to study the feasibility of extending our encoding, and the related implementation, to the new UML4SOA profile.

The current version of Venus only provides a fixed set of predefined properties. We plan to extend this functionality by allowing a user to load sets of customized predefined properties written in proper text files (by means of an interface similar to that shown in Figure 5). This way, expert users may define new predefined properties (each of which expressed both in natural language and in SocL) that can be subsequently used also by non-expert users. Another facility for non-expert users could be to allow the specification of user defined properties also in pseudo-natural language, beside the SocL temporal logic.

Venus provides an explanation for violation of a property when the 'Explain' button is pressed. Due to the abstraction mechanism, actions used within such explanation are already expressed in terms of UML4SOA operations. However, to provide a feedback easily understandable also by non-expert users and help them to improve the original UML4SOA specification, explanations generated by CMC need to be further refined.

Our tool supports verification of behavioral properties of services. The translation of UML4SOA diagrams into COWS and the research results already obtained for this calculus could permit to extend our framework to other classes of properties by integrating the appropriate tools. Thus, for example, we could verify confidentiality properties by using the type system of (Lapadula et al., 2007b), information flow properties using the static analysis of (Bauer et al., 2008) and quantitative properties using the stochastic extension of COWS introduced in (Prandi and Quaglia, 2007).

References

Arons, T., Hooman, J., Kugler, H., Pnueli, A., van der Zwaag, M., 2004. Deductive Verification of UML Models in TLPVS. In: UML. Vol. 3273 of LNCS. Springer, pp. 335–349.

Banti, F., Lapadula, A., Pugliese, R., Tiezzi, F., 2009a. Specification and Analysis of SOC Systems using COWS: A Finance Case Study. In: WWV. Vol. 235 of ENTCS. Elsevier, pp. 71–105.

Banti, F., Pugliese, R., Tiezzi, F., 2009b. Automated Verification of UML Models of Services. Tech. rep., DSI, Università di Firenze, submitted for publication. Available at `http://rap.dsi.unifi.it/cows`.

Banti, F., Pugliese, R., Tiezzi, F., 2009c. Towards a Framework for the Verification of UML Models of Services. In: WWV.

Bartoletti, M., Degano, P., Ferrari, G. L., Zunino, R., 2008. Semantics-Based Design for Secure Web Services. IEEE Trans. Software Eng. 34 (1), 33–49.

Bauer, J., Nielson, F., Nielson, H., Pilegaard, H., 2008. Relational Analysis of Correlation. In: SAS. Vol. 5079 of LNCS. Springer, pp. 32–46.

Bocchi, L., Laneve, C., Zavattaro, G., 2003. A Calculus for Long-Running Transactions. In: FMOODS. Vol. 2884 of LNCS. Springer, pp. 124–138.

Boreale, M., Bruni, R., De Nicola, R., Loreti, M., 2008. Sessions and Pipelines for Structured Service Programming. In: FMOODS. Vol. 5051 of LNCS. Springer, pp. 19–38.

Butler, M., Hoare, C., Ferreira, C., 2005. A Trace Semantics for Long-Running Transactions. In: 25 Years Communicating Sequential Processes. Vol. 3525 of LNCS. Springer, pp. 133–150.

Carbone, M., Honda, K., Yoshida, N., 2007. Structured Communication-Centred Programming for Web Services. In: ESOP. Vol. 4421 of LNCS. Springer, pp. 2–17.

Clarke, E., Emerson, E., 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs. Vol. 131 of LNCS. Springer, pp. 52–71.

Clarke, E. M., Grumberg, O., Peled, D., 1999. Model Checking. MIT Press.

Compton, K., Gurevich, Y., Huggins, J., Shen, W., 2000. An Automatic Verification Tool for UML. Tech. rep., University of Michigan.

Crane, M., Dingel, J., 2008. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In: CASCON. ACM, pp. 96–110.

Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D., 2002. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: ASE. IEEE, pp. 267–270.

De Nicola, R., Latella, D., Loreti, M., Massink, M., September 2009. Service-oriented stochastic logic. Tech. rep., Università di Firenze.

De Nicola, R., Vaandrager, F., 1990. Action versus State based Logics for Transition Systems. In: Proc. of the Ecole de Printemps on Semantics of Concurrency. Vol. 469 of LNCS. Springer, pp. 407–419.

De Nicola, R., Vaandrager, F., 1995. Three logics for branching bisimulation. Journal of the ACM 42 (2), 458–487.

Dong, W., Wang, J., Qi, X., Qi, Z., 2001. Model Checking UML Statecharts. In: APSEC. IEEE, pp. 363–370.

Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., Stehno, C., 2005. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In: SDL. Vol. 3530 of LNCS. Springer, pp. 133–148.

Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F., 2008. A model checking approach for verifying COWS specifications. In: FASE. Vol. 4961 of LNCS. Springer, pp. 230–245.

Ferrari, G. L., Guanciale, R., Strollo, D., 2006. Event Based Service Coordination over Dynamic and Heterogeneous Networks. In: ICSOC. Vol. 4294 of LNCS. Springer, pp. 453–458.

Geguang, P., Xiangpeng, Z., Shuling, W., Zongyan, Q., 2005. Towards the semantics and verification of bpel4ws. In: WLFM 2005. Elsevier.

Grumberg, O., Veith, H. (Eds.), 2008. 25 Years of Model Checking - History, Achievements, Perspectives. Vol. 5000 of LNCS. Springer.

Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G., 2006. SOCK: A Calculus for Service Oriented Computing. In: ICSOC. Vol. 4294 of LNCS. Springer, pp. 327–338.

Holzmann, G., 2003. The Spin Model Checker - Primer and Reference Manual. Addison-Wesley.

Huth, M., Ryan, M., 2004. Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press.

Jürjens, J., Shabalin, P., 2004. Automated Verification of UMLsec Models for Security Requirements. In: UML. Vol. 3273 of LNCS. Springer, pp. 365–379.

Knapp, A., Merz, S., Rauh, C., 2002. Model Checking - Timed UML State Machines and Collaborations. In: FTRTFT. Vol. 2469 of LNCS. Springer, pp. 395–416.

Lanese, I., Martins, F., Ravara, A., Vasconcelos, V., 2007. Disciplining Orchestration and Conversation in Service-Oriented Computing. In: SEFM. IEEE, pp. 305–314.

Lapadula, A., Pugliese, R., Tiezzi, F., 2007a. A calculus for orchestration of web services. Tech. rep., Università di Firenze, `http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf`. An extended abstract appeared in *ESOP*, LNCS 4421, pages 33-47, Springer.

Lapadula, A., Pugliese, R., Tiezzi, F., 2007b. Regulating data exchange in service oriented applications. In: FSEN. Vol. 4767 of LNCS. Springer, pp. 223–239.

Latella, D., Majzik, I., Massink, M., 1999. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Asp. Comput. 11 (6), 637–664.

Latella, D., Massink, M., 2001. A Formal Testing Framework for UML Statechart Diagrams Behaviours: From Theory to Automatic Verification. In: HASE. IEEE, pp. 11–22.

Mayer, P., Koch, N., Schroeder, A., December 2008a. The UML4SOA profile (version 1.2). Available at http://www.uml4soa.eu/profile.

Mayer, P., Schroeder, A., Koch, N., 2008b. Mdd4soa: Model-driven service orchestration. In: EDOC. IEEE, pp. 203–212.

Meolic, R., Kapus, T., Brezocnik, Z., 2008. ACTLW - an Action-based Computation Tree Logic With Unless Operator. Elsevier Information Sciences 178 (6), 1542–1557.

Milner, R., Parrow, J., Walker, D., 1992. A Calculus of Mobile Processes, I and II. Information and Computation 100 (1), 1–40, 41–77.

No Magic Inc., 2009. MagicDraw UML personal edition 16.5. Available at http://www.magicdraw.com/.

OASIS, April 2007. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS.

Object Management Group, 2007a. Unified Modeling Language (UML), version 2.1.2.

Object Management Group, 2007b. XMI Mapping Specification, v2.1.1.

Object Management Group, 2008. Sevice oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS). Tech. rep., OMG, available at http://www.omg.org/docs/ad/08-08-04.pdf.

Prandi, D., Quaglia, P., 2007. Stochastic COWS. In: ICSOC. Vol. 4749 of LNCS. Springer, pp. 245–256.

Störrle, H., Hausmann, J., 2005. Towards a Formal Semantics of UML 2.0 Activities. In: Software Engineering. Vol. 64 of LNI. GI, pp. 117–128.

Sun Microsystems, 2009. The Swing Tutorial. Available at http://java.sun.com/docs/books/tutorial/uiswing.

Tabuchi, N., Sato, N., Nakamura, H., 2005. Model-driven performance analysis of UML design models based on stochastic process algebra. In: ECMDA-FA. Vol. 3748 of LNCS. Springer, pp. 41–58.

ter Beek, M., Gnesi, S., Mazzanti, F., 2008. Formal verification of an automotive scenario in service-oriented computing. In: ICSE. ACM, pp. 613–622.

VIATRA2 Developer Team, 2009. VIATRA2 Project Overview. Available at http://eclipse.org/gmt/VIATRA2/.

Vieira, H., Caires, L., Seco, J. C., 2008. The Conversation Calculus: A Model of Service-Oriented Computation. In: ESOP. Vol. 4960 of LNCS. Springer, pp. 269–283.