Towards a Framework for the Verification of UML Models of Services

Federico Banti, Rosario Pugliese, and Francesco Tiezzi
Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
Viale Morgagni 65, 50134 Firenze, Italia
fbanti@gmail.com, {pugliese,tiezzi}@dsi.unifi.it

Abstract. We make a connection between different layers of abstraction of the engineering process of Service-Oriented Architectures (SOAs) by presenting an encoding of UML4SOA, a UML profile for modeling SOAs, in COWS, a process calculus for specifying service-oriented systems. The encoding provides a rigorous semantics for UML4SOA and paves the way for the verification of UML4SOA models by exploiting the reasoning mechanisms and analysis techniques that are available for COWS.

1 Introduction

Service-Oriented Architectures (SOAs) provide methods and technologies for programming and deploying software applications that can run over globally available computational network infrastructures. The most successful implementations of the SOA paradigm are probably the so called web services, sort of independent 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. Both stand alone web services and web service-based systems 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 that encompass all the phases of the software development process, from modelling to deployment, and exploit formal techniques for qualitative and quantitative verification of systems. The goal is to initially specify the services by exploiting a high-level modelling language and then to transform the specification towards the final deployment. This methodology should guarantee the properties of the implementation code by means of the application of formal methods to test the behavioral and quantitative properties of the specification.

As a matter of fact, UML [20] is by now a widely used modelling language for specifying software systems. It is intuitive, powerful, and extensible. Recently, a UML 2.0 profile, christened UML4SOA [17, 16], has been designed for modeling SOAs. In particular, we focus our attention on UML4SOA activity diagrams since they express the behavioral aspects of services, which we are mainly interested to. Inspired to WS-BPEL [19], the OASIS standard language for orchestrating web services, UML4SOA activity

* This work has been supported by the EU project SENSORIA, IST-2 005-016004.
diagrams integrate UML with specialized actions for exchanging messages among services, specialized structured activity nodes and activity edges for representing scopes, fault and compensation handlers. Currently, UML4SOA lacks formal semantics and methods of analysis, and must hence be regarded as an informal modelling language.

On the contrary, several process calculi for the specification of SOA systems have been recently designed (see, e.g., [15, 6, 11, 5]) that provide linguistic primitives supported by mathematical semantics, and analysis and verification techniques for qualitative and quantitative properties. To exploit previous work on process calculi, in this paper we define an encoding of UML4SOA in COWS (Calculus for the Orchestration of Web Services) [15], a recently proposed process calculus for specifying and combining services while modelling their dynamic behaviour. Indeed, in [1] we have first used UML4SOA activity diagrams to specify the behaviour of a financial service and then translated by hand these diagrams to COWS terms to enable a subsequent analysis phase. In that context, we experimented that the specific mechanisms and primitives of COWS are particularly suitable for encoding services specified by UML4SOA activity diagrams. In fact, this is not surprising if one consider that, like UML4SOA, COWS is inspired to WS-BPEL. The encoding we introduce in this paper formalizes those intuitions and supports a more systematic and mathematically well-founded approach to engineering of SOA systems where developers can concentrate on modelling the high-level behaviour of the system and use transformations for analysis purposes.

Besides defining a transformational semantics for UML4SOA, our encoding enables the use of the tools and methodologies developed for COWS for the analysis of UML4SOA models of services. Thus, given a service specification, one can check confidentiality properties by using the type system of [13], information flow properties using the static analysis of [3], functional properties using the logic and the model checker of [9], and quantitative properties using the stochastic extension introduced in [22].

Moreover, the encoding we propose is compositional, in the sense that the encoding of a UML4SOA activity diagram is (approximately) the COWS term resulting from the parallel composition of the encodings of its components. The encoding is thus easily expandable, applicable to large and complex real applications, and suitable for automatic implementation. In fact, we are currently developing a software tool for automatically translating UML4SOA orchestrations into COWS terms. A first prototype of the software can be downloaded from the COWS’s web page (http://rap.dsi.unifi.it/cows/).

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 briefly reviews COWS. Section 4 presents the COWS-based transformational semantics of UML4SOA. Finally, Section 5 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 1, 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,
Fig. 1. Travel agency scenario
a message from a client containing a request for a flight (flightReq) and a hotel (hotelReq) is received. Then, the workflow forks in two parallel branches. In the left branch, by a send&receive action, the flight request is sent to a flight searching service (flightService) and the service awaits for a response message that will be stored in the variable flightAnswer. As soon as this action is executed, a compensation handler is installed. The compensation consists of a send action to the flight searching service with a message asking to delete the request. The received answer is then sorted by a decision node. In the right branch, similar actions are undertaken in order to book a hotel by contacting a hotel searching service (hotelService). If both the answers are positive, the two branches join, the answers are forwarded to the client and the activity successfully terminates. 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.

The syntax of UML4SOA is given in [16] by a metamodel in classical UML-style. In Table 1 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 ::= ALTER1 | . . . | ALTERN, where the non-terminal SYMBOL is in the top left corner of the row (highlighted by a gray background), while the alternatives ALTER1, . . . , ALTERN are the other elements of the row.

To simplify the encoding and its exposition we adopt some restrictions on the language. We assume that every action and scope has one incoming and one outgoing control flow 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 expressivity 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. The rational for this choice is that UML4SOA offers specialized versions of most of these 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 a UML activity enclosing one top level scope with, possibly, several nested scopes. A scope is a UML 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:

\[
\text{VARS ::= nil | X, VARS | \text{\langle wo\rangle X, VARS}}
\]

We use X to range over variables and the symbol \text{\langle wo\rangle} 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 [19, 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
Table 1. UML4SOA syntax
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 name of edges 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 holds true.

A 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.

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. A compensation handler is a scope whose execution semantically rolls back the execution of the related main scope. It is installed when execution of the related main scope completes and is executed in case of failure. An exception handler is an activity triggered by a raised exception whose main purpose is to trigger execution of the installed compensations.

Default event, exception and compensation handlers are respectively as follows: a graph composed of an initial node directly connected to a final node, a graph composed of a RAISE action preceded and followed by initial and final nodes, and a graph composed of a COMPENSATE_ALL action preceded and followed by initial and final nodes. For readability sake, these handlers will be sometimes omitted from the representation.

It is worth noticing that, UML4SOA exception handler differs from the corresponding UML 2.0 construct. 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 4 for a formal explanation of the behavior of these UML4SOA constructs.

Control flow nodes 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 for exchanging data, for raising exceptions and for triggering scope compensations. SEND sends the message resulting from the evaluation of expressions 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 X₁,...,Xₙ, 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.

3 An overview of COWS

COWS is a formalism for specifying and combining services that has been influenced by the principles underlying WS-BPEL. 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 states, processes playing more than one partner role, and stateful sessions made by several correlated service interactions, inter alia.

The syntax of COWS is presented in Table 2. 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 \(e\), 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 `\(\bar{\ldots}\)` denotes tuples (ordered sequences) of homogeneous elements, e.g. \(\bar{x}\) is a compact notation for denoting a tuple of variables as \(<x_1, \ldots, x_n>\). 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 \(0\) and write \([e_1, \ldots, e_n]\) in place of \([e_1]\) \(\ldots\)[\(e_n]\). 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 \(p \cdot o\) can be interpreted as a specific implementation of operation \(o\) provided by the service identified by the logic name \(p\). An invoke \(p \cdot o!e\) can proceed as soon as the evaluation of the expressions \(\bar{e}\) in its argument returns the corresponding values. A receive \(p \cdot o?\bar{w}.s\) offers an invocable operation \(o\) along a given

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

Table 2. COWS syntax
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 $s$. Differently from the scope of names and variables, that of killer labels cannot be extended (indeed, killer labels are not communicable values). Delimitation can be used to generate ‘fresh’ private names (like the restriction operator of the $\pi$-calculus [18]) and to delimit the field of action of kill activities. Execution of a kill activity $\text{kill}(k)$ causes termination of all parallel terms inside the enclosing $[k]$, which stops the killing effect. Critical activities can be protected from a forced termination by using the protection operator $\{s\}$.

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 (which is different from most process calculi).

Execution of parallel terms is interleaved, except when a kill activity or a communication can be performed. Indeed, the former must be executed eagerly while the latter must ensure that, if more than one matching receive is ready to process a given invoke, only one of the receives with greater priority (i.e. the receives that generate the substitution with ‘smaller’ domain, see [15] for further details) is allowed to progress. 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.

### 4 A transformational semantics for UML4SOA through COWS

Hereafter we present an encoding of UML4SOA diagrams in COWS. The encoding disambiguates the meaning of the individual diagrams. It is compositional, in the sense that the translation of an activity diagram is given by the (parallel) composition of the encodings of all its individual elements. We first underline the general layout, then provide specific explanations along with the presentation of the individual encodings. We refer the reader to Table 1 for the names of the encoded UML4SOA elements.

At top level, an orchestration ORC is encoded through an encoding function $\llbracket \cdot \rrbracket$ that returns a COWS term. Function $\llbracket \cdot \rrbracket$ is in turn defined by another encoding function $\llbracket\cdot\rrbracket_{\text{spec}}^\text{orc}_{\text{VARS}}$ that, given an element of a diagram, returns a COWS term and has the two additional arguments, the name $\text{orc}$ of the enclosing orchestration and the names of the variables defined at the level of the encoded element. The argument $\text{orc}$ is used for translating the communication activities, by specifying who is sending/receiving messages. The variable names $\text{VARS}$ 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.

We start by providing the encoding of the graph elements, i.e. nodes with incoming and outgoing edges, treating actions and scopes as black boxes and focusing on the encoding of passage of control among nodes. We provide then the encoding of actions, of the variables delimited within scopes and of scopes (and related handlers) themselves. We end with the translation of whole orchestrations.

Graphs. The encoding of a GRAPH is given simply by the parallel execution of all the CWS processes resulting from the encoding of its elements.

\[
\text{or}_{\text{VARS}}^{\text{GRAPH}_1 \text{ GRAPH}_2} = \text{or}_{\text{VARS}}^{\text{GRAPH}_1} \mid \text{or}_{\text{VARS}}^{\text{GRAPH}_2}
\]

Control flow nodes. 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 invoke and receive activities, where each edge name e is encoded by a COWS endpoint e. A guard is encoded by a COWS (boolean) expression \(\epsilon_{\text{guard}}\). 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

\[
\text{or}_{\text{VARS}}^{e ! (\epsilon_{\text{guard}})}
\]

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.

\[
\text{or}_{\text{VARS}}^{\ast e ? (\text{true}). (e_1 ! (\epsilon_{\text{guard}_1}) \mid \ldots \mid e_n ! (\epsilon_{\text{guard}_n})}
\]

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.

\[
\text{or}_{\text{VARS}}^{\ast e_1 ? (\text{true}). \ldots . e_n ? (\text{true}). e ! (\epsilon_{\text{guard}})}
\]

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 \(n_1, \ldots, n_n\) (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.

\[
\text{or}_{\text{VARS}}^{e ? (\text{true}). [n_1, \ldots, n_n] (n_1 ! (\epsilon_{\text{guard}_1}) \mid \ldots \mid n_n ! (\epsilon_{\text{guard}_n})}
\]

\[
\mid n_1 ? (\text{true}). e_1 ! (\text{true}) + \ldots + n_n ? (\text{true}). e_n ! (\text{true})}
\]

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 \texttt{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 \(t!\langle \rangle\) sends a termination signal to start the execution of (possible) subsequent activities.

\[
\llbracket e \rrbracket_{\text{VARS}}^{\text{orc}} = \llbracket e \rrbracket_{\text{VARS}}^{\text{true}}. \langle \text{kill}(k_t) \mid t!\langle \rangle \rangle
\]

\textbf{Action and scope nodes.} An \texttt{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 \texttt{ACTION} and, in parallel, a process waiting for a termination signal sent from the encoding of \texttt{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.

\[
\llbracket e \rrbracket_{\text{VARS}}^{\text{orc}} = \ast \llbracket e_1 ?(\text{true}) \rrbracket_{\text{VARS}}^{\text{true}}. \langle \text{kill}(k_t) \mid t!\langle \rangle \rangle
\]

The encoding of a \texttt{SCOPE} node is similar to the previous one, with two main additions. When a \texttt{SCOPE} terminates, the encoding of its node sends a signal \(i!\langle \rangle\) enabling the compensation related to the scope. Moreover, it sends its name to the local \texttt{Stack} process in case compensation activities are started (see the encoding of compensation handlers below for further explanations).

\[
\llbracket e \rrbracket_{\text{VARS}}^{\text{orc}} = \ast \llbracket e_1 ?(\text{true}) \rrbracket_{\text{VARS}}^{\text{true}}. \langle \text{t!\langle \rangle} \rangle
\]

Function \texttt{scopeName}(\cdot), given a scope, returns its name.

\textit{Sending and receiving actions.} Sending and receiving actions are translated by relying on, respectively, COWS \texttt{invoke} and \texttt{receive} activities. Special care must be taken to ensure that a sent message is received only by the intended \texttt{RECEIVE} action and partner link. For this purpose, the action names are used as operation names in encoded terms. Thus, a \texttt{SEND} and a \texttt{RECEIVE} action can exchange messages only if they share the same name. Moreover, the partner name along which the communication takes place is the name \texttt{orc} of the enclosing orchestration.

Action \texttt{SEND} is an asynchronous call: message \(\langle \text{expr}_1, \ldots, \text{expr}_n \rangle\) 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 \(\langle \text{orc}, \epsilon_{\text{expr}_1}, \ldots, \epsilon_{\text{expr}_n} \rangle\), where \texttt{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 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 \(t\) is sent for allowing the computation to proceed.

\[
\llbracket \text{SEND} \rrbracket_{\text{VARS}}^{\text{orc}} = \llbracket p \rrbracket_{\text{VARS}}^{\text{true}} \cdot \langle \text{name}!\langle \text{orc}, \epsilon_{\text{expr}_1}, \ldots, \epsilon_{\text{expr}_n} \rangle \rangle \langle t!\langle \rangle \rangle
\]
where $[p]_\text{VARS}^{\text{orc}}$ is $p$ if $\ll x_1 \gg p \notin \text{VARS}$, and $x_p$ otherwise; similarly, each $\epsilon_{\text{expr}}$ is obtained from $\text{expr}$ by replacing each $X$ in the expression such that $\ll x_1 \gg X \in \text{VARS}$ with $x_X$.

Unlike SEND, 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 $\text{orc} \cdot \text{name}$, 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 $\ll x_1 \gg X \in \text{VARS}$ 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.

$$[[\text{RECEIVE}]]_{\text{VARS}}^{\text{orc}} = \text{orc} \cdot \text{name}?([p]_{\text{VARS}}^{\text{orc}}, [X_1]_{\text{VARS}}^{\text{orc}}, \ldots, [X_n]_{\text{VARS}}^{\text{orc}}). t!()$$

The encodings of actions SEND&RECEIVE and RECEIVE&SEND are basically the composition of the encodings of actions SEND and RECEIVE, and vice versa.

$$[[\text{SEND}&\text{RECEIVE}]]_{\text{VARS}}^{\text{orc}} = \{[[p]_{\text{VARS}}^{\text{orc}} \cdot \text{name}!([\text{orc}, \epsilon_{\text{expr}}, \ldots, \epsilon_{\text{expr}}]) \\
| \text{orc} \cdot \text{name}?([p]_{\text{VARS}}^{\text{orc}}, [X_1]_{\text{VARS}}^{\text{orc}}, \ldots, [X_n]_{\text{VARS}}^{\text{orc}}). t!()$$

$$[[\text{RECEIVE}&\text{SEND}]]_{\text{VARS}}^{\text{orc}} = \text{orc} \cdot \text{name}?([p]_{\text{VARS}}^{\text{orc}}, [X_1]_{\text{VARS}}^{\text{orc}}, \ldots, [X_n]_{\text{VARS}}^{\text{orc}}).
(\{[[p]_{\text{VARS}}^{\text{orc}} \cdot \text{name}!([\text{orc}, \epsilon_{\text{expr}}, \ldots, \epsilon_{\text{expr}}]) | t!()\})$$

Actions for fault and compensation handling. The behavior, and thus the encoding, of a RAISE is 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.

$$[[\text{RAISE}]]_{\text{VARS}}^{\text{orc}} = \text{kill}(k_r) | \{t!()\}$$

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 $t!()$). 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.

$$[[\text{COMPENSATE}]]_{\text{VARS}}^{\text{orc}} = \epsilon \cdot \text{scopeName}!([\text{scopeName}]) t!()$$

$$[[\text{COMPENSATE_ALL}]]_{\text{VARS}}^{\text{orc}} = [n] (\text{stack} \cdot \text{compAll}!([n]) | n?(). t!())$$

Variables. The encoding of scope variables is as follows.

$$[[\text{nil}]] = 0 \\
[[X, \text{VARS}]] = \text{Var}_X | [[\text{VARS}]] \\
[[\ll \text{wo} \gg X], \text{VARS}] = [[\text{VARS}]]$$
Thus, variables declared write-once (by means of \( \text{w} \times \text{w} \)) directly correspond 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 \( \text{Var}_X \) providing two operations along the public partner name \( X: \text{read} \), for getting the current value; \( \text{write} \), for replacing the current value with a new one. When the service variable is initialized (i.e., the first time the \( \text{write} \) 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. To access the service, a user must invoke operations \( \text{read} \) and \( \text{write} \) by providing a communication endpoint for the reply and, in case of \( \text{write} \), the value to be stored. Due to lack of space, the service \( \text{Var}_X \) has been omitted (we refer the interested reader to [2]).

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:

\[
\langle \langle u \cdot u' \cdot \xi \rangle \rangle = \left[ m, n_1, \ldots, n_m \right] \quad \text{if } \xi \text{ contains } X_1, \ldots, X_m
\]

\[
\left( X_1 \cdot \text{read}^! \langle n_1 \rangle \right) \ldots \left( X_m \cdot \text{read}^! \langle n_m \rangle \right)\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{read}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{read}^! \langle x_m \rangle \right)
\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{write}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{write}^! \langle x_m \rangle \right)
\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{read}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{read}^! \langle x_m \rangle \right)
\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{write}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{write}^! \langle x_m \rangle \right)
\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{read}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{read}^! \langle x_m \rangle \right)
\]

\[
\left[ x_1, \ldots, x_m \right] \cdot \text{write}^! \langle x_1 \rangle \ldots \left( X_m \cdot \text{write}^! \langle x_m \rangle \right)
\]

where \( \{ X_i \mapsto x_i \} \) denotes substitution of \( X_i \) with \( x_i \), and endpoint \( m \) returns the result of evaluating \( \xi \) (of course, we are assuming that \( m, n_i \) and \( x_i \) are fresh).

**Scopes.** A SCOPE is encoded as the parallel execution, with proper delimitations, of the processes resulting from the encoding of all its components.

\[
\llbracket \text{SCOPE} \rrbracket = \left[ e, \text{stack}, \text{vars}(\text{VARS}) \right]
\]

(\( \llbracket \text{VARS} \rrbracket \)

| \( \text{r}^?(). \text{e}(). \) \n
| \( \llbracket \text{VARS} \rrbracket \)

| \( \text{e}().[t,k]. \) \( \llbracket \text{GRAPH}_{t,k} \rrbracket_{\text{VARS},\text{VARS}} \)

| \( \text{r}^?(). \) \( \text{e}().[t,k]. \) \( \llbracket \text{GRAPH}_{t,k} \rrbracket_{\text{VARS},\text{VARS}} \)

| \( \text{r}^?(). \) \( \text{e}().[t,k]. \) \( \llbracket \text{GRAPH}_{t,k} \rrbracket_{\text{VARS},\text{VARS}} \)

| \( \text{r}^?(). \) \( \text{e}().[t,k]. \) \( \llbracket \text{GRAPH}_{t,k} \rrbracket_{\text{VARS},\text{VARS}} \)

Function \( \text{vars}() \), given a list of variables \( \text{VARS} \), returns a list of COWS variables/names, where a COWS name \( X \) corresponds to a variable \( X \) in \( \text{VARS} \), while a COWS variable \( X_k \) corresponds to a variable \( \llbracket \text{w} \times \text{w} \rangle 

The (private) endpoint \( r \) catches signals generated by RAISE actions and activates the corresponding handler, by means of the (private) endpoint \( e \). Killers \( k_e \) and \( k_r \)
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 \( \ast [x] c \cdot \text{scopeName}?(x). \text{stack} \cdot \text{end}!(\text{scopeName}) \) 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:

\[
[q]( \text{Lifo} \mid * [x] \text{stack} \cdot \text{push}?(x). q \cdot \text{push}!(x) \\
* [x] \text{stack} \cdot \text{compAll}?(x). [\text{loop}] (\text{loop}!()) \mid * \text{loop}!(). \text{Comp} )
\]

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 \( c \cdot \text{scopeName}!(\text{scopeName}) \)); in case Lifo is empty, the cycle terminates and a termination signal is sent along the argument \( x \) of the operation compAll.

\[
\text{Comp} \triangleq [r, e] ( q \cdot \text{pop}!(r, e) \mid [y] (r?y). (c \cdot y)!y \mid \text{stack} \cdot \text{end}?(y). \text{loop}!()) + e?(). x!())
\]

Lifo is an internal queue providing ‘push’ and ‘pop’ operations. Stack can push and pop a scope name into/out of Lifo via q·push and q·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 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 m is associated with the element on top of the queue, if this is not empty, otherwise it is empty.

\[
\text{Lifo} \triangleq [m, h] (* [y_r, y_r, y_e] (q \cdot \text{push}?(y_r). [z] m?z) \cdot [c] (h!y_r, c) \mid m!z() ) \\
+ q \cdot \text{pop}?(y_r, y_e). [z] (m?z) \cdot [y_r, y_r, y_e] h?y_r, y_e, y_r, z, (m!y_r) \mid y_r!y_e() ) + m?(\text{empty}). (m!\text{empty} \mid y_e!)() ))
\]

Notice that, because of the COWS’s (prioritized) semantics, whenever the queue is empty, the presence of receive m!(empty) prevents taking place of the synchronization between m!(empty) and m?(z).

Orchestrations. The encoding of an orchestration is that of its top-level scope.

\[
\llbracket \text{ORC} \rrbracket = [k, c, t, i, \text{edges}((\text{SCOPE}))][\llbracket \text{SCOPE} \rrbracket^\text{or}}
\]

where function edges(\( \cdot \)), given a scope, returns the names of all the edges of the graphs contained within the scope.
5 Concluding remarks

We have presented an encoding of UML4SOA activity diagrams into COWS. Both languages have been defined within the European project SENSORIA [28] on developing methodologies and tools for dealing with SOAs. As far as we know, our encoding is the first (transformational) semantics of UML4SOA. It can be the cornerstone of a future framework for verification of service models specified through UML4SOA. With a similar objective, in [4] we have defined a translation from the modelling language SRML [10] into COWS.

Recently, another UML profile for SOA design, named SoaML [21], 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. We believe it is worth to study the feasibility of defining an encoding from SoaML into COWS, but leave it as a challenge for future work.

In this work, we focused on those constructs that are more relevant for UML4SOA, namely workflow-related constructs and the specialized UML4SOA constructs. Several works propose formal semantics for (subsets of) UML activity diagrams. Among these works the most relevant ones are those based on (extensions of) Petri Nets (see, e.g., [8, 23]). Although we regard Petri Nets as a natural choice for encoding such aspects of UML activity diagrams as workflows, other aspects turned out to be hardly representable in this formalism. For instance, in [23], the authors themselves deem the encoding of classical UML exception handling into Petri Nets as not completely satisfactory. Also, variables are not treated by the Petri Nets-based semantics of UML activity diagrams nor, at the best of our knowledge, by any other semantics.

The UMC framework [26] and the virtual machine-based approach of [7] provide operational semantics for (subsets of) UML activity diagrams by transition systems. Although these approaches are clearly less expressive, it could be interesting to compare them with the correspondent fragments of our encoding. In [25] the authors use model checking to verify a UML activity diagram of a SOA case study. In fact, the analysis is done on a handmade translation in UMC of the activity diagram. The authors themselves point out that an automatic translation like the one presented in this paper would be highly desirable. In [24], a stochastic semantics for a subset of UML activity diagrams is proposed. It could be interesting to compare this approach with a stochastic extension, along the line of [22], of the encoding proposed in this paper. Anyway, none of these proposals attempts to encode the UML4SOA profile and, most notably, none of them seems to be adequate for encoding its specific constructs like message exchanging actions, scopes, exceptions and compensation handlers.

We have singled COWS out of several similar process calculi for its peculiar features, specifically the termination constructs and the correlation mechanism. 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 (inspired by that of WS-BPEL) permits to automatically correlate messages belonging to the same interaction, preventing to mix messages from different service instances. Modelling such a feature by using session-oriented calculi designed for SOA (e.g. [5, 27, 12]) 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., [11]), COWS seems more adequate since it relies on more basic constructs and provides analysis tools and a stochastic extension.

In [16], a software tool for generating WS-BPEL code from UML4SOA models is presented. This work and the related tool had been very useful for verifying the intended meaning of UML4SOA constructs by their equivalent WS-BPEL code. However, the encoding algorithm is not capable of translating all the possible diagrams and is not compositional. Also, WS-BPEL code has not, in general, an univocal semantics and indeed the very same code generates different computations when running on different WS-BPEL implementations [14]. Thus, the proposed encoding in WS-BPEL does not provide a formal semantics to UML4SOA models.

Our long-term goal is to build a complete framework for verifying UML4SOA models. To this aim quite some work remains to be done. Currently, the encoding is implemented by a software tool (described in the full version of this paper [2]) that accepts as input a UML2 EMF XMI 2.1 file storing a UML4SOA specification. Such file can be automatically generated by the UML editor MagicDraw (http://www.magicdraw.com) where, to allow users to graphically specify UML4SOA activity diagrams, the UML4SOA profile (available at http://www.mdd4soa.eu) must be previously installed. Our tool automatically translates the XMI description into a COWS term, written in the syntax accepted by CMC [26], a model checker supporting analysis of COWS terms. As a further step in the development of a verification framework for UML4SOA models, we plan to more closely integrate the tool implementing the encoding with CMC. We also intend to investigate the challenging issue of how to tailor the (low-level) results obtained by the analysis of COWS terms to the corresponding (high-level) UML4SOA specifications, thus making the verification process as much transparent and, hence, usable as possible for developers. Other planned extensions include the encoding of a larger subset of UML 2.0 activity diagrams and its stochastic extension by relying on the stochastic variant [22] of COWS.

References

20. OMG. Unified Modeling Language (UML), version 2.1.2.