# Automated Verification of UML Models of Services*

Federico Banti, Rosario Pugliese, and Francesco Tiezzi

Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze
{fbanti,tiezzi.f}@gmail.com, rosario.pugliese@unifi.it

**Abstract.** We build a bridge between different layers of abstraction of the engineering process of Service-Oriented Architectures. We present an encoding of the UML profile UML4SOA in the process calculus COWS and the software tool UStoC that implements the encoding. The encoding provides a rigorous semantics for UML4SOA and its implementation enables the verification of UML4SOA models of services by exploiting the tools and methodologies available for COWS. We demonstrate the effectiveness of our approach by means of the translation and analysis of an automotive scenario.

## 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 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. 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.

The most widely used language for modelling software systems is probably UML [20]. It is intuitive, powerful, and extensible. Recently, a UML 2.0 profile, christened UML4SOA [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 [18], 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. Currently, UML4SOA lacks formal semantics and

* This work has been supported by the EU project SENSORIA, IST-2 005-016004.

must hence be regarded as an *informal* modelling language. Furthermore, since it is a static model, UML4SOA specifications are not suitable for direct automated analysis.

On the contrary, several *process calculi* have been recently designed (e.g., [12, 10, 5, 11]) providing linguistic primitives for the specification of SOAs, and formal techniques and software tools for verification of their qualitative and quantitative properties. These calculi are based on formal and rigorous semantics and provide a higher level of abstraction w.r.t. actual web service languages and platforms. However, they might still be too low level and impractical for developers accustomed to work with abstract architectural models of services, but not with the technicalities of process calculi.

To pave the way for an integrated approach that can lead to a verifiable development of service components by exploiting the work on process calculi, in this paper we define an encoding of UML4SOA in COWS [12]. In fact, 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. We have thus experimented that COWS's distinctive features are particularly suitable for encoding services specified by UML4SOA diagrams. This is not surprising if one consider that both UML4SOA and COWS are inspired to WS-BPEL. Our encoding 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 exploit the encoding for analysis purposes.

The presented encoding is compositional, i.e. the translation of a UML4SOA activity diagram is the COWS term resulting from the parallel composition of the translations of its components, which supports expansibility and applicability to large systems. However, since the semantics of UML4SOA is only informally specified, correctness cannot be treated, i.e. it cannot be formally proved that the semantics of the COWS term resulting from application of the encoding conforms to the intended semantics of the original UML4SOA diagram. On the contrary, the encoding defines a precise 'transformational' semantics for the UML4SOA profile. In fact, it is quite intuitive and direct, which makes us confident to have correctly rendered the original informal semantics.

We have then empirically assessed the quality of our UML4SOA's semantics while developing, and then using, UStoC, a software tool that implements the encoding. The automated translation provided by UStoC enables the use of the techniques and tools developed for COWS to verify UML4SOA models of services. Thus, e.g., given a service specification, one can check confidentiality properties by using the type system of [13], information flow properties using the static analysis of [2], behavioural properties using the logic and the model checker of [9], and quantitative properties using the stochastic extension introduced in [22]. Here, to illustrate the approach, a UML4SOA model of a service system is first translated into a COWS term by exploiting UStoC, then its formal properties are verified by using the COWS model checker CMC [9].

The rest of the paper is structured as follows. Section 2 presents an overview of UML4SOA, also by means of an automotive scenario, and our proposal for a BNF-like (graphical) syntax of UML4SOA. Section 3 briefly reviews COWS. Section 4 presents the COWS-based transformational semantics of UML4SOA. Section 5 uses the automotive scenario for illustrating the usage of UStoC and the automated verification techniques supported by CMC. Finally, Section 6 touches upon related and future work.

## 2 An overview of UML4SOA

We start by informally presenting UML4SOA through a realistic but simplified scenario in the automotive area. The example is inspired to a case study from the EU project SENSORIA [27] on developing sound methodologies for SOAs. In our scenario, a car is equipped with a service-oriented application that, when a severe failure occurs and the car is no longer drivable, tries to book a garage for repairing the car and a tow truck for hauling the car to the garage, and to rent a substitutive car. The application operates by orchestrating the booking services of the garage, the car rental and the tow truck. To simplify the exposition, we assume that diagnostic data, car location and communication endpoints of the services mentioned above have been already collected and used to initialize the service orchestrator.

The orchestrator, illustrated in Figure 1, after its initialization, starts with a *send&receive* action sending the diagnosticData of the failure and identification data id univocally identifying the car to the garage service (garage). The orchestrator then awaits for a response message from the garage, containing an answer (yes or no) saying if the garage can repair the failure or not, the coordinates of the garage and other data. These information are stored, respectively, in the variables garageAnswer, garageGps and gData. As soon as this action is executed, a *compensation handler* is installed. The compensation consists of a *send* action delivering a message asking to delete the reservation. The garage answer is then checked by a decision node. If the answer is positive, another *send&receive* action is performed for reserving a tow truck by the towTruck service. If also this service replies positively, a third *send&receive* action allows the orchestrator to contact the car rental service (rentalCar) and to rent a car that will be handed over to the driver at the garage location. If either the garage or the tow truck service answer is negative, an exception is raised by performing an action *raise*. A merge node is used for merging the two branches of the workflow dealing with negative answers. The exception is caught by an exception handler that triggers the necessary compensations by an action *compensateAll* and then rents a car that will be handed over at the current location of the broken car (rather than at the garage location).

The garage service is illustrated in Figure 2. Since we are mainly interested in the interaction with the client, the schema of the service internal behaviour is simplistic. A new instance is created whenever a reservation request is received containing the client identity (stored in the variable id) and other data (stored in the variable data). The instance performs a non-deterministic choice by means of a decision node and, afterwards, either replies negatively to the client and terminates, or provides a positive answer and awaits for a possible request for deletion from the client. The tow truck service behaves similarly to the garage service, while the car rental service always replies positively. Their modelling UML4SOA diagrams are reported in Appendix C.

The syntax of UML4SOA is given in [16] by a metamodel in classical UML-style. We provide in Table 1 an alternative BNF-like syntax that is more suitable for defining an encoding by induction on the syntax of constructs; it is the exact syntax accepted by the tool, which returns an error when the input does not comply with it. Each row of the table represents a production of the form SYMBOL ::= $\mathsf{ALTER}_1$ | ... | $\mathsf{ALTER}_n$, where the non-terminal SYMBOL is in the top left corner of the row (with a gray background), while the alternatives $\mathsf{ALTER}_1, \ldots, \mathsf{ALTER}_n$ are the other elements of the row.

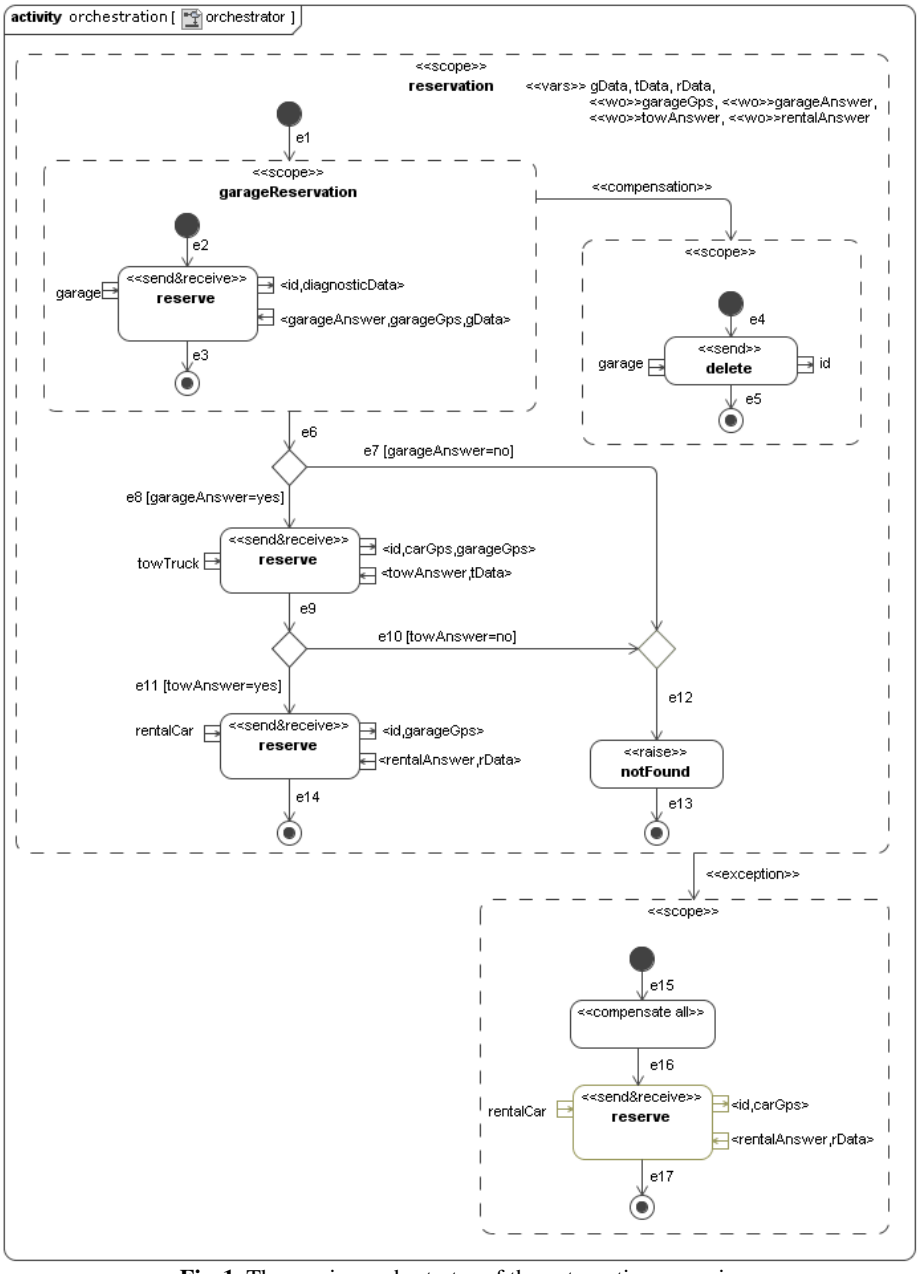



**Fig. 1.** The service orchestrator of the automotive scenario

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 expressivity of the language and are often implic-

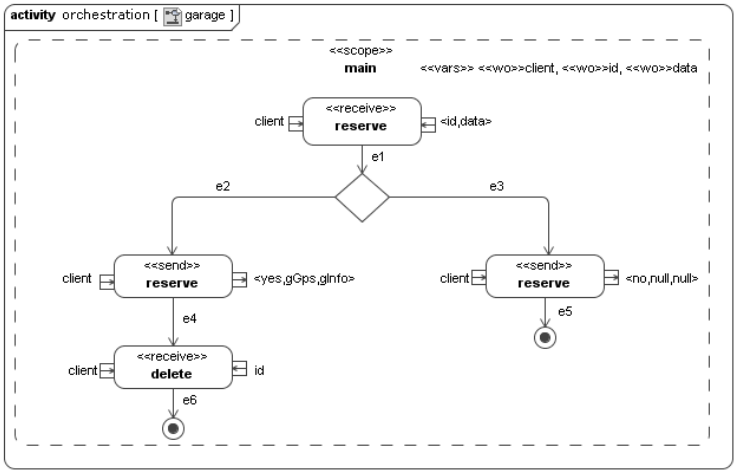



**Fig. 2.** The garage service

itly adopted in practice for sake of clarity for highlighting convergence and divergence of flows. For instance, an action node with one incoming and one outgoing edge preceded and followed by, respectively, a join and a fork node is often used in place of an action node with multiple incoming and outgoing edges. 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.

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:

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

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 [18, 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 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. 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 each incoming edge there should exist an outgoing edge with the same name, and vice-versa. Moreover, we assume that (pairs of

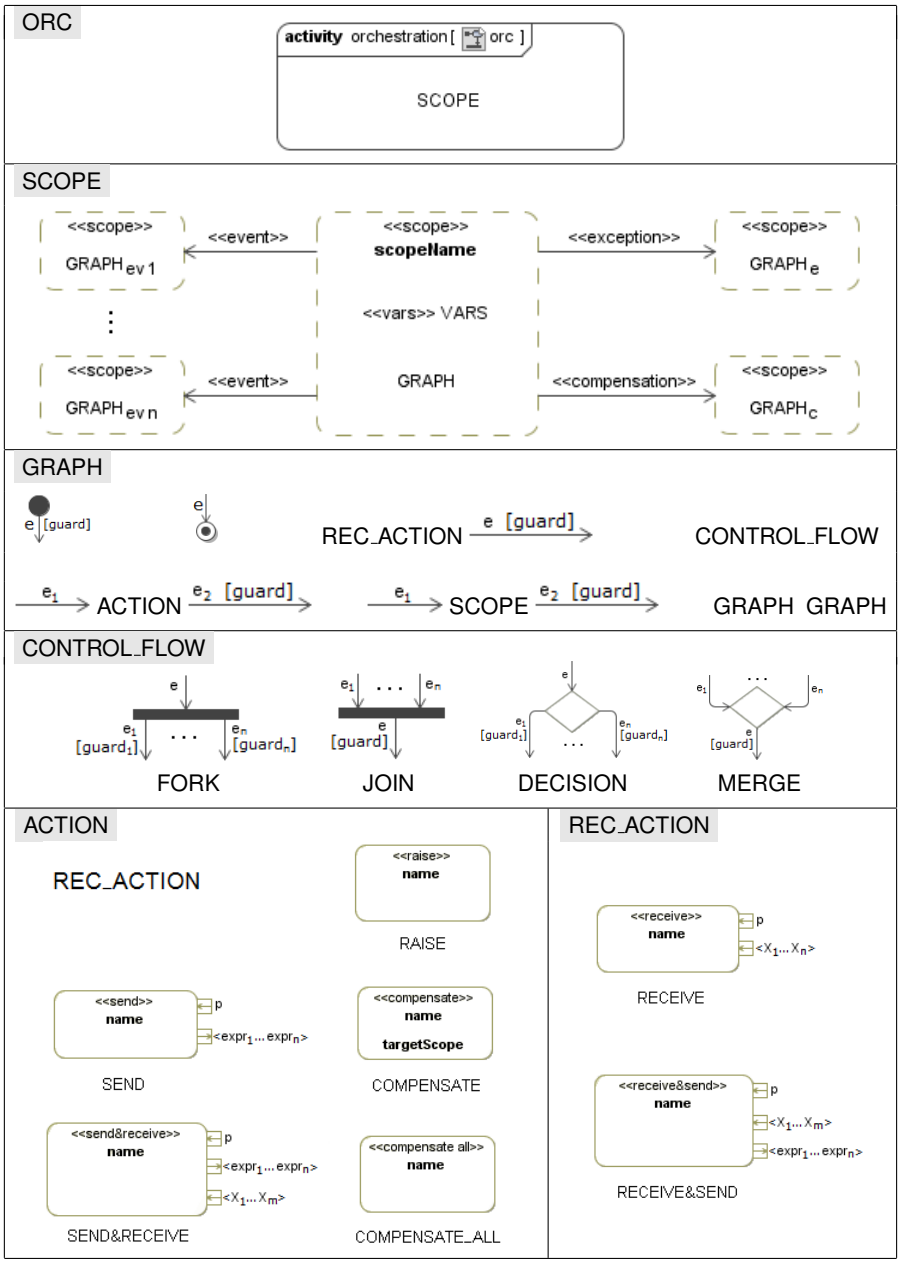



**Table 1.** UML4SOA syntax

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, has no incoming edges, then it starts when a message is received and remains enabled to wait for other messages (like a UML *AcceptEventAction* [20, 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 garage 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 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 handlers are empty graphs, while default exception and compensation handlers are, respectively, as follows: 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.

*Control flow* nodes CONTROL_FLOW are the standard UML ones: *fork* and *join* nodes (depicted by bars), *decision* and *merge* nodes (depicted by diamonds).

Finally, UML4SOA provides seven specialized actions. SEND sends the message resulting from the evaluation of expressions $\mathsf{expr}_1,\ldots,\mathsf{expr}_n$ (whose exact syntax is deliberately omitted, since UML4SOA is parametric w.r.t. their language) to the partner service identified by p. 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. Actions 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 (Calculus for the Orchestration of Web Services, [12]) is a recently proposed process calculus for specifying and combining services. Indeed, the calculus permits readily modelling several specific features of SOAs as, e.g., service instances with shared states, processes playing more than one partner role, and stateful sessions made by several correlated service interactions. This is achieved by means of a novel combination of constructs, some of which borrowed from well-known process calculi, as e.g. non-binding receiving activities, asynchronous communication, polyadic synchronization, pattern-matching, protection, and delimited receiving and killing activities.

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 $v$, $v'$, ...) and the set of 'write-once' *variables* (ranged over by $x$, $y$, ...). The set of values is left unspecified; however, we assume that it includes

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

**Table 2.** COWS syntax

the set of *names*, ranged over by $n$, $m$, $o$, $p$, ..., mainly used to represent partners and operations. The syntax of *expressions*, ranged over by $\epsilon$, is deliberately omitted; we just assume that they contain, at least, values and variables, but do not include killer labels (that, hence, cannot 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. $\bar{x}$ stands for 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. Finally, we omit trailing occurrences of $\mathbf{0}$ and write $[e_1, \ldots, e_n]\, s$ in place of $[e_1] \ldots [e_n]\, s$.

*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 $p$ and continues as $s$. 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 $\pi$-calculus), 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 *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 [12] 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, can be exploited to model persistent services.

## 4 A transformational semantics for UML4SOA through COWS

We start underlining the general layout of our encoding of UML4SOA diagrams in COWS, then we provide specific explanations along with the presentation of each case. 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 $[\![\cdot]\!]$ that returns a COWS term. Function $[\![\cdot]\!]$ is in turn defined in terms of another encoding function $[\![\cdot]\!]^{\mathsf{orc}}_{\mathsf{VARS}}$ that additionally takes as 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 essential since, as we will show, they enable sharing received messages among the various elements of a scope and storing names of partner links.

**Graphs.** We start by providing in Table 3 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 $\mathtt{e}$. A guard is encoded by a COWS (boolean) expression $\epsilon_{guard}$. 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

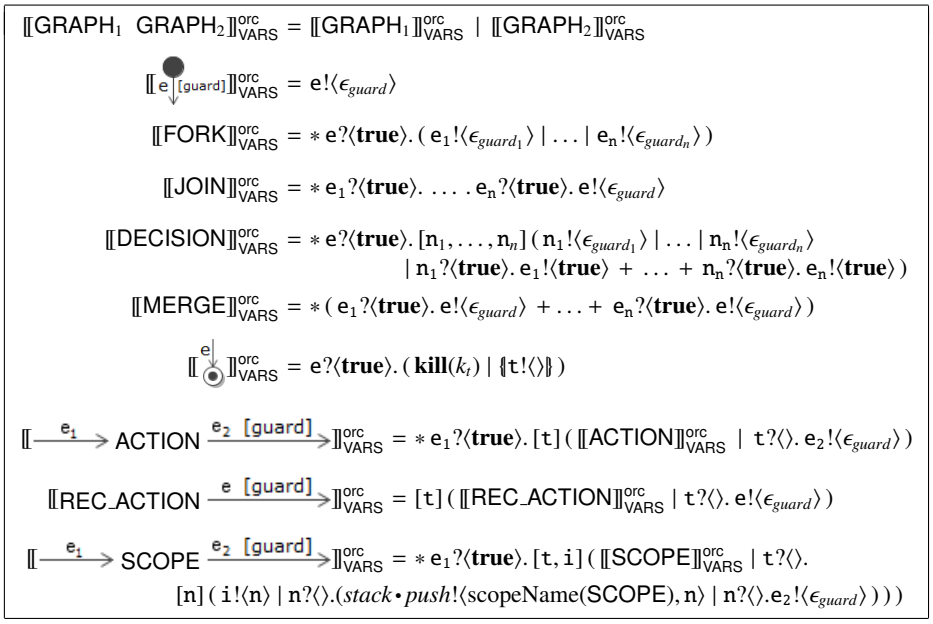


**Table 3.** Encoding of graph elements

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 $\mathtt{t}$ and then performing an invoke on the outgoing edge. Of course, $\mathtt{t}$ 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 $\mathtt{n}$ along $\mathtt{i}$ enabling the compensation related to the scope, awaits for an acknowledgement along $\mathtt{n}$ and sends its name and $\mathtt{n}$ 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 Table 4. 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

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

**Table 4.** Encoding of actions

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 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 the link p, in case p is a constant, or as the COWS variable $x_{\mathsf{p}}$ 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. $[\![\mathsf{p}]\!]^{\mathsf{orc}}_{\mathsf{VARS}}$ is p if $\ll\mathsf{wo}\gg \mathsf{p} \notin \mathsf{VARS}$, and $x_{\mathsf{p}}$ otherwise; similarly, each $\epsilon_{expr_i}$ is obtained from $\mathsf{expr}_\mathsf{i}$ by replacing each X in the expression such that $\ll\mathsf{wo}\gg \mathsf{X} \in \mathsf{VARS}$ with $x_{\mathsf{X}}$. Action RECEIVE is encoded as a COWS receive along the endpoint $\mathsf{orc} \bullet \mathsf{name}$, with input pattern a tuple where the first element is the encoding of the link pin p and the others are either COWS variables $x_{\mathsf{X}}$ 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 have been omitted, because they 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 Table 5. 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

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

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

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

**Table 5.** Encoding of variables, scopes and orchestrations

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 two operations along the public partner name X: *read*, for getting the current value; *write*, for replacing the current value with a new one. Variables like X may (temporarily) occur in expressions used by invoke and receive activities within COWS terms obtained as result of the translation. To get rid of these variables and finally obtain 'pure' COWS terms, we exploit the encoding introduced in [12] that, for the sake of completeness, we report in the Appendix A.

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 $x_{\mathsf{X}}$ 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*. Due to lack of space, we relegate the specification of *Stack* to the Appendix A.

The encoding of an orchestration is that of its top-level scope. Function isPersistent(·) 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(·) returns the names of all the edges of the graphs contained within its argument scope.

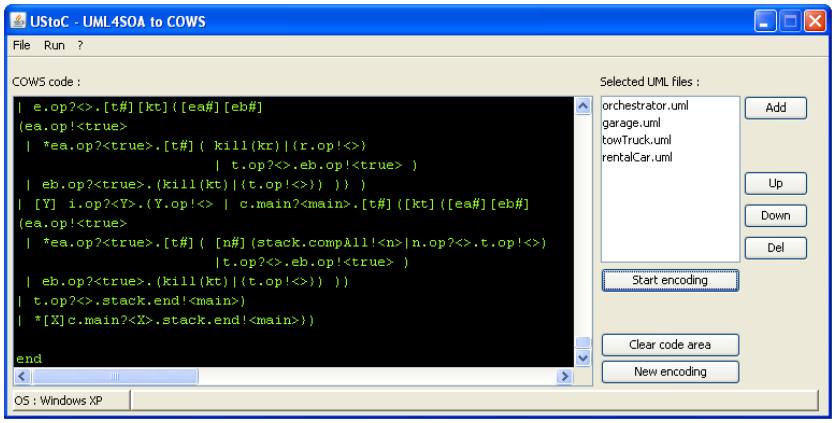



**Fig. 3.** A screenshot of UStoC interface

## 5 Verification of the automotive scenario

We have developed a software tool, called UStoC[1], to automatise the encoding illustrated in Section 4. The COWS terms generated by UStoC are written in the syntax accepted by CMC[2] [9], a model checker supporting analysis of COWS specifications.

UStoC accepts as an input a set of files XMI (XML Metadata Interchange [19]) storing a UML4SOA specification. Such files can be automatically generated by the UML modeler MagicDraw[3] [17] where, to allow users to graphically specify UML4SOA activity diagrams, the UML4SOA profile [15] must have been previously installed. In fact, the diagrams of Figures 1 and 2 have been edited by using MagicDraw. For the time being, MagicDraw is the only CASE tool supporting the UML4SOA profile. However, the use of XMI as the input format makes UStoC independent of the tool used for the high-level system specification. Thus, it should also be able to support XMI files produced by other tools for which UML4SOA plug-ins would be properly developed.

UStoC is written in Java to guarantee portability across different platforms and to exploit well-established libraries for parsing XML documents. The tool comprises three main components: a *graphical user interface*, a screenshot of which is shown in Figure 3; an XMI *parser*, which creates a Java object for each element of a diagram; and a *translator*, which consists of a set of classes, one for each UML4SOA element, each of which providing a method `toCOWS()` that returns the COWS translation of the corresponding element. When `toCOWS()` is invoked on an object representing a UML4SOA orchestration, it translates the orchestration and, recursively, the enclosed elements.

Figure 4 illustrates an example of automated verification process of UML4SOA models of services where the tools MagicDraw, UStoC and CMC are used in combina-

[1] UStoC is a free software; it can be downloaded from http://rap.dsi.unifi.it/cows/tools/ustoc.zip and redistributed and/or modified under the terms of the GNU General Public License.

[2] For the experiments carried out in this paper, we have used the Java standalone version of CMC (v0.7g), which is downloadable at http://fmt.isti.cnr.it/cmc.

[3] We have used MagicDraw Academic Personal Edition 16.5, which is freely available for qualifying institutions. One can also use MagicDraw Community Edition, which provides a minimal functionality set and is free for developers working on non-commercial projects.

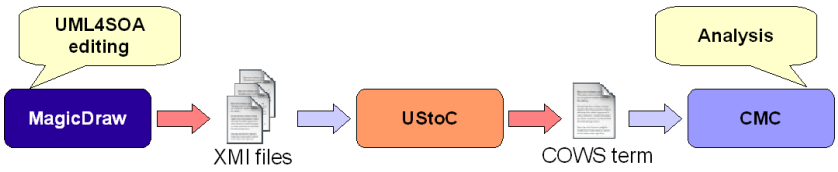



**Fig. 4.** Verification process of UML4SOA models of services

tion. We have applied such process to the scenario introduced in Section 2. To analyse the scenario, firstly we have generated a file XMI (saved with extension `.uml`) for each UML4SOA diagram by using MagicDraw. Then, we have loaded the four created files into UStoC (by using the 'Add' button on the right-hand side of the graphical interface) and encoded them into a COWS term (by pressing the 'Start encoding' button). An excerpt of the obtained COWS term is reported in Appendix B. Afterwards, we have exported the COWS code from UStoC to CMC. This can be done either by simply coping the code from the UStoC text area and pasting it into the CMC text area or by saving the code in a `.cow` file (by selecting 'Save as .COW file ...' from the 'File' menu) and opening it in CMC. Some logical formulae that we have verified by using CMC over the COWS specification of the automotive scenario follow:

- `AF {reserve(rentalCar,yes,*)} true`
  This formula means that the orchestrator eventually succeeds in renting a car, i.e. it *eventually* (operator `AF`) receives a positive answer (action `reserve(rentalCar,yes,*)`) from the rental car service.
- `AG [reserve(towTruck,no,*)] AF {delete} true`
  This formula means that, if the tow truck booking fails, the garage booking is compensated. That is, it holds *globally* (operator `AG`) that *if* (operator `[·]`) the tow truck service's answer is negative (action `reserve(towTruck,no,*)`), then it holds that *eventually* (operator `AF`) the compensation is executed (action `delete`).
- `EF {reserve(garage,yes,*,*)}`
  `EF {reserve(towTruck,yes,*)} EF {reserve(rentalCar,yes,*)} true`
  This formula means that the orchestration *can* succeed in making all reservations. In fact, `EF formula` means that `formula` *can eventually* hold. A stronger form of this property, meaning that the orchestration *always* succeeds, is given by the formula:
  `AF {reserve(garage,yes,*,*)}`
  `AF {reserve(towTruck,yes,*)} AF {reserve(rentalCar,yes,*)} true`

All the above properties hold for the considered scenario, but for the stronger form of the last property, because a service might reply negatively to a reservation request.

## 6 Concluding remarks

Through the presented encoding of UML4SOA in COWS, two languages recently defined within the EU project SENSORIA [27], and the related implementation, we have laid down the basis for an integrated approach that, by exploiting the work on process calculi, can lead to a verifiable development of service components starting from abstract architectural models (a similar aim has led to [4]). To pursue our long-term goal, we are currently developing a tool that closely integrates UStoC and CMC. A challenging issue

we are tackling is to tailor and reflect the (low-level) results obtained by the verification of COWS terms to the corresponding (high-level) UML4SOA specifications.

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., [8, 23]). Although Petri Nets can be a natural choice for encoding workflows, they seem however 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. [25, 6]), stochastic semantics [24], and transformation into SMV specifications [3], but none of them considers the UML4SOA profile and, above all, seems to be adequate for encoding its specific constructs.

An environment for verifying UML diagrams based on the VIATRA framework is presented in [7]. However, to the best of our knowledge, VIATRA does not support the UML4SOA profile. The same applies for the various graph transformation tools considered in [26] that translate high-level UML activity diagrams into CSP processes.

A software tool translating UML4SOA models into WS-BPEL is presented in [16]. The translation, however, does not apply to all possible UML4SOA diagrams and is not compositional. Furthermore, WS-BPEL code has not a univocal semantics (see [14]), thus the translation does not provide a formal semantics to UML4SOA models. Indeed, as far as we know, our encoding is the first (transformational) semantics of UML4SOA.

As target of our encoding, we have singled COWS out of several similar process calculi for its distinctive primitives and mechanisms, 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 interaction, preventing to mix messages from different service instances. Defining a transformational semantics for UML4SOA using a session-based calculus (e.g. [5, 11]) appears to be problematic and less intuitive, mainly because 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., [10]), COWS seems to be more adequate since it relies on more basic constructs; furthermore, it already provides a number of verification tools.

Recently, another UML profile for designing SOAs, 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. 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.

## References

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

2. J. Bauer, F. Nielson, H.R. Nielson, and H. Pilegaard. Relational Analysis of Correlation. In *SAS*, LNCS 5079, pp. 32–46. Springer, 2008.
3. M. E. Beato, M. Barrio-Solrzano, C. E. Cuesta, and P. de la Fuente. UML automatic verification tool with formal methods. In *VLFM*, ENTCS 127 (4), pp. 3–16, Elsevier, 2005.
4. L. Bocchi, J.L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From Architectural to Behavioural Specification of Services. In *FESCA*, ENTCS. Elsevier, 2009. To appear.
5. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In *FMOODS*, LNCS 5051, pp. 19–38. Springer, 2008.
6. M.L. Crane and J. Dingel. Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In *CASCON*, pp. 96–110. ACM, 2008.
7. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE*, pp. 267–270. IEEE, 2002.
8. C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In *SDL*, LNCS 3530, pp. 133–148, 2005.
9. A. Fantechi et al. A model checking approach for verifying COWS specifications. In *FASE*, LNCS 4961, pp. 230–245. Springer, 2008.
10. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *ICSOC*, LNCS 4294, pp. 327–338. Springer, 2006.
11. I. Lanese, F. Martins, A. Ravara, and V.T. Vasconcelos. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *SEFM*, pp. 305–314. IEEE, 2007.
12. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, Università di Firenze, 2007. http://rap.dsi.unifi.it/cows/papers/cows-esop07-full.pdf. An extended abstract appeared in *ESOP*, LNCS 4421, pp. 33-47, Springer.
13. A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, LNCS 4767, pp. 223–239. Springer, 2007.
14. A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *COORDINATION*, LNCS 5052, pp. 199–215. Springer, 2008.
15. P. Mayer, N. Koch, and A. Schroeder. The UML4SOA profile (version 1.2), December 2008. Available at http://www.uml4soa.eu/profile.
16. P. Mayer, A. Schroeder, and N. Koch. Mdd4soa: Model-driven service orchestration. In *EDOC*, pp. 203–212. IEEE, 2008.
17. No Magic. MagicDraw UML academic personal edition 16.5. http://www.magicdraw.com/.
18. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007.
19. Object Management Group. XMI Mapping Specification, v2.1.1.
20. Object Management Group. Unified Modeling Language (UML), v2.1.2, 2007.
21. Object Management Group. Sevice oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS), 2008.
22. D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, LNCS 4749, pp. 245–256, 2007.
23. H. Störrle and J.H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In *Software Engineering*, LNI 64, pp. 117–128. GI, 2005.
24. N. Tabuchi, N. Sato, and H. Nakamura. Model-driven performance analysis of UML design models based on stochastic process algebra. In *ECMDA-FA*, LNCS 3748, pp. 41–58. Springer, 2005.
25. M.H. ter Beek, S. Gnesi, and F. Mazzanti. Formal verification of an automotive scenario in service-oriented computing. In *ICSE*, pp. 613–622. ACM, 2008.
26. D. Varró et al. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In *AGTIVE*, LNCS 5088, pp. 540–565. Springer, 2008.
27. M. Wirsing et al. *At your service: Service Engineering in the Information Society Technologies Program*, chapter SENSORIA: Engineering for Service-Oriented Overlay Computers, pp. 159–182. MIT Press, 2009.

# A Auxiliary encodings

We report here the definitions of some cases of the encoding that, due to lack of space, have been left out from the paper. We will write $I \triangleq s$ to assign a name $I$ to the term $s$.

### Encoding of rewritable variables

A rewritable variable $\mathsf{X}$ is rendered as a service $Var_{\mathsf{X}}$ providing 'read' and 'write' functionalities along the public partner name $\mathsf{X}$. 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.

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

Service $Var_{\mathsf{X}}$ 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 $\mathsf{X}$; the first time, it corresponds to initialization of the variable. $Var_{\mathsf{X}}$ uses the delimited endpoint $\mathtt{n}$ 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 $\mathsf{X}$ 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:

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

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

where $\{\mathsf{X}_i \mapsto x_i\}$ denotes substitution of $\mathsf{X}_i$ 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).

**Stack**

The *Stack* service associated to a scope is rendered in COWS as follows:

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

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 $c \bullet \mathsf{scopeName}!\langle\mathsf{scopeName}\rangle$); in case *Lifo* is empty, the cycle terminates and a termination signal is sent along the argument $x$ of the operation *compAll*.

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

*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 `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*.

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

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 $\mathtt{m}?\langle z\rangle$.

## B COWS translation of the automotive scenario

The UML4SOA specification of the automotive scenario introduced in Section 2 is translated in COWS, by means of the encoding illustrated in Section 4, as follows[4]:

$$\mathit{Orchestrator} \mid \mathit{Garage} \mid \mathit{TowTruck} \mid \mathit{RentalCar}$$

The term is the parallel composition of the service orchestrator and the three booking services. Hereafter, we only focus on the term *Orchestrator*, which is defined as follows:

[4] For the sake of presentation, we have not applied here the encoding presented at page 17 that permits removing the rewritable variables used by a COWS term.

$$
\begin{array}{l}
[k_r, c, \mathtt{t}, \mathtt{r}, \mathtt{i}, \mathit{stack}, \mathtt{e}_1, \ldots, \mathtt{e}_{17}, \mathtt{e}, \mathsf{gData}, \mathsf{tData}, \mathsf{rData},\\
x_{\mathsf{garageAnswer}}, x_{\mathsf{garageGps}}, x_{\mathsf{towAnswer}}, x_{\mathsf{rentalAnswer}}]\\
\quad (\,[\mathit{stack}]\,(\,[\mathtt{r}]\,(\,[k_r, k_t]\,(\,\mathit{ReservationGraph} \mid \{\!|\,\mathit{Stack}\,|\!\}\,)\\
\qquad\qquad \mid \mathtt{r}?\langle\rangle.\,\mathtt{e}!\langle\rangle\,)\\
\qquad\quad \mid \mathit{Var}_{\mathsf{gData}} \mid \mathit{Var}_{\mathsf{tData}} \mid \mathit{Var}_{\mathsf{rData}} \mid \mathtt{e}?\langle\rangle.\,\{\!|\,[\mathtt{t}, k_t]\,\mathit{ExceptionGraph}\,|\!\}\,)\\
\quad \mid [y]\,\mathtt{i}?\langle y\rangle.\,\{\!|\,y!\langle\rangle \mid c \cdot \mathsf{reservation}?\langle\mathsf{reservation}\rangle.\\
\qquad\qquad\qquad [\mathtt{t}]\,(\,[k_t]\,\mathit{DefaultComp} \mid \mathtt{t}?\langle\rangle.\,\mathit{stack} \cdot \mathit{end}!\langle\mathsf{reservation}\rangle\,)\\
\qquad\quad \mid *\,[x]\,c \cdot \mathsf{reservation}?\langle x\rangle.\,\mathit{stack} \cdot \mathit{end}!\langle\mathsf{reservation}\rangle\,|\!\}\,)
\end{array}
$$

where the handlers are

$$
\begin{array}{rl}
\mathit{ExceptionGraph} \triangleq & \mathtt{e}_{15}!\langle\mathbf{true}\rangle\\
& \mid *\,\mathtt{e}_{15}?\langle\mathbf{true}\rangle.\,[\mathtt{t}]\,(\,[\mathtt{n}]\,(\mathit{stack} \cdot \mathit{compAll}!\langle\mathtt{n}\rangle \mid \mathtt{n}?\langle\rangle.\,\mathtt{t}!\langle\rangle)\\
& \qquad\qquad\qquad \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_{16}!\langle\mathbf{true}\rangle\,)\\
& \mid *\,\mathtt{e}_{16}?\langle\mathbf{true}\rangle.\\
& \quad [\mathtt{t}]\,(\,\{\!|\,\mathsf{rentalCar} \cdot \mathsf{reserve}!\langle\mathsf{orchestrator}, \mathsf{id}, \mathsf{carGps}\rangle\,|\!\}\\
& \qquad \mid \mathsf{orchestrator} \cdot \mathsf{reserve}?\langle\mathsf{rentalCar}, x_{\mathsf{rentalAnswer}}, \mathsf{rData}\rangle.\,\mathtt{t}!\langle\rangle\\
& \qquad \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_{17}!\langle\mathbf{true}\rangle\,)\\
& \mid \mathtt{e}_{17}?\langle\mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)
\end{array}
$$

$$
\begin{array}{rl}
\mathit{DefaultComp} \triangleq & [\mathtt{e}_a, \mathtt{e}_b]\\
& (\,\mathtt{e}_a!\langle\mathbf{true}\rangle\\
& \quad \mid *\,\mathtt{e}_a?\langle\mathbf{true}\rangle.\,[\mathtt{t}]\,(\,[\mathtt{n}]\,(\mathit{stack} \cdot \mathit{compAll}!\langle\mathtt{n}\rangle \mid \mathtt{n}?\langle\rangle.\,\mathtt{t}!\langle\rangle)\\
& \qquad\qquad\qquad \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_b!\langle\mathbf{true}\rangle\,)\\
& \quad \mid \mathtt{e}_b?\langle\mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)\,)
\end{array}
$$

The term *ReservationGraph*, modelling the main behaviour of the top level scope, is as follows:

$$
\begin{array}{l}
\mathtt{e}_1!\langle\mathbf{true}\rangle\\
\mid *\,\mathtt{e}_1?\langle\mathbf{true}\rangle.\,[\mathtt{t}, \mathtt{i}]\,(\,\mathit{GarageScope}\\
\qquad\qquad \mid \mathtt{t}?\langle\rangle.\,[\mathtt{n}]\,(\,\mathtt{i}!\langle\mathtt{n}\rangle \mid \mathtt{n}?\langle\rangle.(\mathit{stack} \cdot \mathit{push}!\langle\mathsf{garageReservation}, \mathtt{n}\rangle\\
\qquad\qquad\qquad\qquad \mid \mathtt{n}?\langle\rangle.\mathtt{e}_6!\langle\mathbf{true}\rangle\,)\,)\,)\\
\mid *\,\mathtt{e}_6?\langle\mathbf{true}\rangle.\,[\mathtt{n}_1, \mathtt{n}_2]\,(\,\mathtt{n}_1!\langle x_{\mathsf{garageAnswer}} = \mathsf{yes}\rangle \mid \mathtt{n}_2!\langle x_{\mathsf{garageAnswer}} = \mathsf{no}\rangle\\
\qquad\qquad \mid \mathtt{n}_1?\langle\mathbf{true}\rangle.\,\mathtt{e}_8!\langle\mathbf{true}\rangle + \mathtt{n}_2?\langle\mathbf{true}\rangle.\,\mathtt{e}_7!\langle\mathbf{true}\rangle\,)\\
\mid *\,\mathtt{e}_8?\langle\mathbf{true}\rangle.\,[\mathtt{t}]\,(\,\mathit{TowTruck\_s\&r} \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_9!\langle\mathbf{true}\rangle\,)\\
\mid *\,\mathtt{e}_9?\langle\mathbf{true}\rangle.\,[\mathtt{n}_3, \mathtt{n}_4]\,(\,\mathtt{n}_3!\langle x_{\mathsf{towAnswer}} = \mathsf{yes}\rangle \mid \mathtt{n}_4!\langle x_{\mathsf{towAnswer}} = \mathsf{no}\rangle\\
\qquad\qquad \mid \mathtt{n}_3?\langle\mathbf{true}\rangle.\,\mathtt{e}_{11}!\langle\mathbf{true}\rangle + \mathtt{n}_4?\langle\mathbf{true}\rangle.\,\mathtt{e}_{10}!\langle\mathbf{true}\rangle\,)\\
\mid *\,(\,\mathtt{e}_7?\langle\mathbf{true}\rangle.\,\mathtt{e}_{12}!\langle\mathbf{true}\rangle + \mathtt{e}_{10}?\langle\mathbf{true}\rangle.\,\mathtt{e}_{12}!\langle\mathbf{true}\rangle\,)\\
\mid *\,\mathtt{e}_{11}?\langle\mathbf{true}\rangle.\,[\mathtt{t}]\,(\,\mathit{RentalCar\_s\&r} \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_{14}!\langle\mathbf{true}\rangle\,)\\
\mid *\,\mathtt{e}_{12}?\langle\mathbf{true}\rangle.\,[\mathtt{t}]\,(\,\mathbf{kill}(k_r) \mid \{\!|\,\mathtt{r}!\langle\rangle\,|\!\} \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_{13}!\langle\mathbf{true}\rangle\,)\\
\mid \mathtt{e}_{13}?\langle\mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)\\
\mid \mathtt{e}_{14}?\langle\mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)
\end{array}
$$

Intuitively, the above term is the parallel composition of the activities that form (from the top to the bottom) the main graph of scope reservation: the initial node, the scope garageReservation (modelled by the COWS term *GarageScope*), the first decision node, the receive&send action reserve having towTruck as partner (modelled by

the COWS terms *TowTruck_s&r*), the second decision node, the merge node, the receive&send action reserve having rentalCar as partner (modelled by the COWS terms *RentalCar_s&r*), the raise action notFound and the two final nodes.

The term *GarageScope*, modelling the scope garageReservation, is

$$
\begin{array}{l}
[\mathtt{e}] \\
(\,[\mathit{stack}]\ (\,[\mathtt{r}]\,(\,[k_r, k_t]\,(\,\mathit{GarageGraph}\ \mid\ \{\!|\,\mathit{Stack}\,|\!\}\,)\ \mid\ \mathtt{r}?\langle\rangle.\,\mathtt{e}!\langle\rangle\,) \\
\qquad\qquad \mid\ \mathtt{e}?\langle\rangle.\,\{\!|\,[\mathtt{t}, k_t]\,\mathit{DefaultExc}\,|\!\}\,) \\
\quad \mid\ [y]\,\mathtt{i}?\langle y\rangle.\,\{\!|\,y!\langle\rangle \mid c \cdot \mathsf{garageReservation}?\langle \mathsf{garageReservation}\rangle. \\
\qquad\qquad\qquad\qquad [\mathtt{t}]\,(\,[k_t]\,\mathit{GarageComp}\ \mid\ \mathtt{t}?\langle\rangle.\,\mathit{stack} \cdot \mathit{end}!\langle \mathsf{garageReservation}\rangle\,) \\
\qquad\qquad\qquad\quad \mid\ *\,[x]\,c \cdot \mathsf{garageReservation}?\langle x\rangle.\,\mathit{stack} \cdot \mathit{end}!\langle \mathsf{garageReservation}\rangle\,|\!\}\,)\,)
\end{array}
$$

where

$$
\begin{array}{rl}
\mathit{GarageGraph} \triangleq & \mathtt{e}_2!\langle \mathbf{true}\rangle \\
& \mid\ *\,\mathtt{e}_2?\langle \mathbf{true}\rangle. \\
& \quad [\mathtt{t}]\,(\,\{\!|\mathsf{garage} \cdot \mathsf{reserve}!\langle \mathsf{orchestrator}, \mathsf{id}, \mathsf{diagnosticData}\rangle|\!\} \\
& \qquad \mid \mathsf{orchestrator} \cdot \mathsf{reserve}?\langle \mathsf{garage}, x_{\mathsf{garageAnswer}}, x_{\mathsf{garageGps}}, \mathsf{gData}\rangle. \\
& \qquad\ \mathtt{t}!\langle\rangle \\
& \qquad \mid\ \mathtt{t}?\langle\rangle.\,\mathtt{e}_3!\langle \mathbf{true}\rangle\,) \\
& \mid\ \mathtt{e}_3?\langle \mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)
\end{array}
$$

$$
\begin{array}{rl}
\mathit{DefaultExc} \triangleq & [\mathtt{e}_\mathsf{a}, \mathtt{e}_\mathsf{b}] \\
& (\,\mathtt{e}_\mathsf{a}!\langle \mathbf{true}\rangle \\
& \ \mid\ *\,\mathtt{e}_\mathsf{a}?\langle \mathbf{true}\rangle.\,[\mathtt{t}]\,(\,\mathbf{kill}(k_r) \mid \{\!|\,\mathtt{r}!\langle\rangle\,|\!\} \mid \mathtt{t}?\langle\rangle.\,\mathtt{e}_\mathsf{b}!\langle \mathbf{true}\rangle\,) \\
& \ \mid\ \mathtt{e}_\mathsf{b}?\langle \mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)\,)
\end{array}
$$

$$
\begin{array}{rl}
\mathit{GarageComp} \triangleq & \mathtt{e}_4!\langle \mathbf{true}\rangle \\
& \mid\ *\,\mathtt{e}_4?\langle \mathbf{true}\rangle.\,[\mathtt{t}]\,(\,\{\!|\mathsf{garage} \cdot \mathsf{delete}!\langle \mathsf{orchestrator}, \mathsf{id}\rangle|\!\} \mid \mathtt{t}!\langle\rangle \\
& \qquad\qquad\qquad \mid\ \mathtt{t}?\langle\rangle.\,\mathtt{e}_5!\langle \mathbf{true}\rangle\,) \\
& \mid\ \mathtt{e}_5?\langle \mathbf{true}\rangle.\,(\,\mathbf{kill}(k_t) \mid \{\!|\,\mathtt{t}!\langle\rangle\,|\!\}\,)
\end{array}
$$

The remaining terms of *ReservationGraph* are as follows:

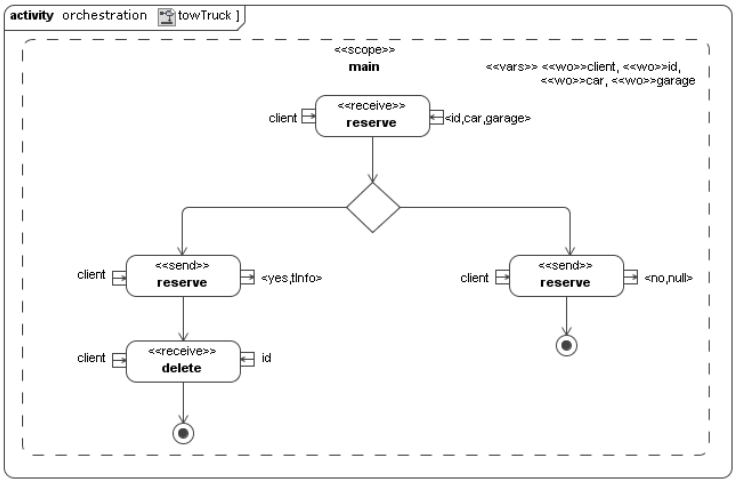
$$
\begin{array}{rl}
\mathit{TowTruck\_s\&r} \triangleq & \{\!|\mathsf{towTruck} \cdot \mathsf{reserve}!\langle \mathsf{orchestrator}, \mathsf{id}, \mathsf{carGps}, x_{\mathsf{garageGps}}\rangle|\!\} \\
& \mid \mathsf{orchestrator} \cdot \mathsf{reserve}?\langle \mathsf{towTruck}, x_{\mathsf{towAnswer}}, \mathsf{tData}\rangle.\,\mathtt{t}!\langle\rangle
\end{array}
$$

$$
\begin{array}{rl}
\mathit{RentalCar\_s\&r} \triangleq & \{\!|\mathsf{rentalCar} \cdot \mathsf{reserve}!\langle \mathsf{orchestrator}, \mathsf{id}, x_{\mathsf{garageGps}}\rangle|\!\} \\
& \mid \mathsf{orchestrator} \cdot \mathsf{reserve}?\langle \mathsf{rentalCar}, x_{\mathsf{rentalAnswer}}, \mathsf{rData}\rangle.\,\mathtt{t}!\langle\rangle
\end{array}
$$

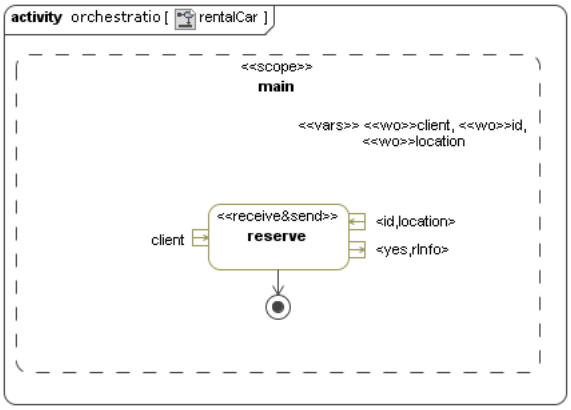
## C Other services of the automotive scenario

The remaining UML4SOA diagrams of the external services involved in the automotive scenario, i.e. (a) the tow truck service and (b) the car rental service are as follows.

(a)


activity orchestration towTruck ]
<<scope>>
main
<<vars>> <<wo>>client, <<wo>>id,
<<wo>>car, <<wo>>garage
client
<<receive>>
reserve
<id,car,garage>
client
<<send>>
reserve
<yes,tInfo>
client
<<receive>>
delete
id
client
<<send>>
reserve
<no,null>

(b)


activity orchestratio [ rentalCar ]
<<scope>>
main
<<vars>> <<wo>>client, <<wo>>id,
<<wo>>location
client
<<receive&send>>
reserve
<id,location>
<yes,rInfo>