An ASM-based executable formal model of service-oriented component interactions and orchestration*

Elvinia Riccobene
Università degli Studi di Milano
DTI, via Bramante, 65 - Crema (CR), Italy
elvinia.riccobene@unimi.it

Patrizia Scandurra
Università degli Studi di Bergamo
DIIMM, via Marconi, 5 - Dalmine (BG), Italy
patrizia.scandurra@unibg.it

ABSTRACT

Formal design methods, that might serve as a basis for specifying and analyzing abstract models of service orchestrations, are needed to complement the wide range of domain-specific languages (mainly based on graphical notations) that are currently being defined for engineering service-oriented systems. This paper presents a formal and executable semantic framework for UML4SOA models of service-oriented systems. The UML4SOA language is a UML profile developed in the EU SENSORIA project for modeling services behavior focusing on service orchestration aspects. We complement the graphical model of a service orchestration scenario with a formal description that is suitable for rigorous execution-platform-independent analysis. We map the behavioral primitives of UML4SOA activity diagrams into a particular class of Abstract State Machines (ASMs) able to model notions of service interactions and orchestrations.

Keywords

Service-oriented Computing, Service behaviour modeling, UML4SOA, Abstract State Machines

1. INTRODUCTION

Service-oriented computing (SOC) is an emerging paradigm for developing loosely coupled, interoperable, evolvable systems and applications relying on the basic unification principle that "Everything is a service". *Services* are loosely coupled computational entities available in a distributed environment. On top of these services, business processes and technical workflows can be (re-)implemented as compositions of services – *service orchestration*. The architectural foundation for SOC is provided by the Service-Oriented Architecture (SOA), which states that applications expose their functionality as services in a uniform and technology-independent way such that they can be discovered and invoked over the network. This new programming style relies on interface-based design, composition and reusability.

In order to support software engineers with intuitive and easy to adopt design and implementation techniques for service-oriented software, modeling notations specific to the SOA domain are required.

Among existing ones [19, 20], within the european project SENSORIA [18], the UML4SOA profile [22] has been developed as a UML 2 extension for the SOA domain. It provides a conservative extension of UML 2 activity diagrams for modeling service orchestrations on a high-level of abstraction, and allows a fully automated, model-driven approach for transforming orchestration down to code [21].

Besides domain-specific notations, appropriate design methodologies are also required in the context of SOA. Indeed, service-based systems usually have requirements like, e.g., service availability, functional correctness, protection of private data. Implementing services satisfying these requirements demands the use of rigorous software engineering methodologies that encompass all phases of the software development process, from modeling to deployment, but also exploit formal techniques for qualitative and quantitative verification of systems [2]. An appropriate design methodology should therefore combine a *development process*, consisting into initially specifying the services by high-level modeling languages and then transforming the specification towards the final deployment, with an *analysis process* able to guarantee the properties of the implementation code by means of the application of formal methods to verify the behavioral and quantitative properties of the specification.

The UML4SOA models have great value for communicating the orchestration workflow. However, even a fully automated transformation of the orchestration down to code is guaranteed by the authors of the profile [22], UML4SOA models are not yet executable and the use of this profile should be integrated within a precise engineering methodology for SOA, which, for analysis purposes, requires a formal counterpart of the graphical UML4SOA description. This is a first result of our ongoing work towards the development of a *back-end* framework, based on the Abstract State Machines (ASMs) formal method, for the specification and analysis of service-oriented component systems in a high level of abstraction and technology agnostic way (i.e. independently of the hosting middleware and runtime platforms and of the programming languages in which services are programmed).

In this paper, a transformational semantic mapping is provided to transform a UML4SOA description of a services

*This work was partially supported by the Italian Government under the project PRIN 2007 D-ASAP (2007XKE-HFA)

The Second Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010) 15 June 2010, Paris, France

orchestration into a formal executable description based on the Abstract State Machines (ASMs) formal method [9]. Here, we focus on the workflow describing the orchestration for services, and we leave abstract the specification of the services internal behavior, which is addressed as future work. In particular, for modeling services interaction, we exploit the precise high-level models for eight fundamental service interaction patterns, given by Barros and Boerger [5] in terms of the ASMs. They model arbitrarily complex interaction patterns of distributed service-based (business) processes, that go beyond simple request-response sequences and may involve a dynamically evolving number of participants. The UML4SOA interaction primitives can be viewed – in accordance with the authors' claim – as "combinations of refinements of" some of "the eight bilateral and multilateral service interaction pattern ASMs defined" in [5].

ASM expressiveness and executability allow for the definition and analysis of complex structured services interaction protocols in a formal way but without overkill. The ASM design method is supported by several tools [13] for validation and verification which can be used to analyze ASM-based models of services.

This paper is organized as follows. Some background concerning the UML4SOA profile and the ASM formal method are given in Sect. 2 and 3, respectively. The semantic mapping from the UML profile to the formal executable description in terms of the ASMs is presented in Sect. 4, together with an illustrative case study taken from the literature. Sect. 5 provides a description of related work along the same direction. Finally, Sect. 6 concludes the paper and outlines some future directions of our work.

2. MODELING SERVICE ORCHESTRATION IN UML4SOA

The UML4SOA [22] is a conservative extension of the UML for modeling service orchestrations on a high level of abstraction, and allowing an automated, model-driven approach for transforming orchestrations down to code [21]. UML4SOA builds on the basic structural notions of another OMG UML profile, named SoaML, and adds behavioral descriptions. The idea is to specify orchestrations (workflows) in detail trough UML2 activity diagrams and their required and provided services only as protocols to be fulfilled by (externally) implemented services.

2.1 Service orchestrations as UML Activities

The UML4SOA profile extends the UML2 activity diagrams with service-specific model elements, providing special elements for service interactions, long running transactions and their compensation[1]. A brief description of the UML4SOA *stereotypes* (extended UML modeling constructs) follows.

An *orchestration* is a specialized UML `Activity` for modeling service orchestrations. Each orchestration contains a root «scope»(or «serviceActivity»). A *scope* (or *serviceActivity*) is a UML `StructuredActivityNode` that contains arbitrary `ActivityNodes`, and may have an associated compensation handler.

Specialized actions for service interactions have been defined for sending and receiving data. In particular, a «send»action is an UML `CallBehaviourAction` that sends a message without blocking (typically used to invoke an operation of a partner asynchronously). A «receive»action is a UML `AcceptCallAction`, receiving a message (typically used to receive an operation call request from an external partner); it blocks until a message is received. Service interaction actions may have interaction pins for sending or receiving data. In particular, «lnk»is an UML `Pin` that holds a reference to the service involved in the interaction, «snd»is a `Pin` that holds a container with data to be sent, and «rcv»is a `Pin` that holds a container for data to be received. A «replay»action is a UML `ReplyAction` that sends out data in reply (as result value of a certain functional activity) to the request point of a previous receive action. Finally, a «send&receive»action is a complete synchronous operation call execution with a partner. Some data (stored in the send pins) is sent, then the action waits for data to be sent back, which is stored in the receive pins.

Specialized edges connecting scopes with handlers are also supported in UML4SOA. For example, compensation is modeled by using compensation handlers, which are activities or structured activity nodes themselves and are attached with a «compensation»edge (a UML `ActivityEdge`) to an element (an activity, a service interaction action, a structured activity node) to be compensated. A «compensateAll»action stereotype is also supported to invoke all compensation handlers nested in the current activity node. In this case, the inner elements with compensation handlers are compensated in reverse order of their completion, i.e. the last completed element first. The profile also contains elements for event and exception handling. For a complete overview, see [22].

Remark. UML4SOA adopts Protocol State Machines to specify the (declarative) externally visible behavior of services, as provided to or required from a partner. Stereotyped transitions with «receive», «send», «send&Receive», and «replay», are used to denote operations of a participant a UML4SOA protocol state machines belongs to. We here to not tackle the specification of services internal behavior which can be captured by a particular class of ASMs, the *control-state ASMs*. They would allow modeling the (possibly external) agent life cycle when engaged in service interactions. In particular, service operations could be modeled by *turbo transition rules* executed by the provider agent. This is address as future work.

2.1.1 Orchestration example

As orchestration example in UML4SOA, Fig. 1 (adapted from [17]) shows an UML 2 activity diagram for the orchestration of services in the *On road assistance scenario* case study of the SENSORIA project. UML 2 stereotyped actions indicate the type of interactions («send», «receive», etc.). In addition, stereotypes are used to model compensation of long running transactions: «compensate»and «compensationEdge». Actions match operations of required and provided interfaces of the services, which are defined as ports of UML 2 components. Action names include the name of the provider separated by a dot from the action name. The process starts with a request from the *Orchestrator* to the *Bank* to charge the driver's credit card with the security deposit payment, which is modeled by an asynchronous «send»action `RequestCardCharge` that takes the card number as an input parameter other than the specific amount. In parallel to the interaction with the bank, the orchestrator

[1]Compensation handling is the process of rolling back successfully completed actions.

initiates a synchronous interaction to get the current position of the car from the *GPS* service. The current location is modelled as input of the «sendAndReceive»action and subsequently used by the `FindLocalServices` action which retrieves a list of services. In case no local services are available, an action `FindServices` on the *RemoteDiscovery* is started. If services cannot be found an action to compensate the credit card charge will be launched. For the selection of services, the *Orchestrator* synchronizes with the *Reasoner* to obtain the most appropriate (best) services. Service ordering is modeled by the actions `OrderGarage`, `OrderTowTruck` and `RentalCar` following a parallel and sequential process, respectively.

3. ABSTRACT STATE MACHINES

Abstract State Machines (ASMs) are an extension of FSMs, where unstructured control states are replaced by states comprising arbitrary complex data [7]. Although the ASM method comes with a rigorous mathematical foundation [9], ASMs provides accurate yet practical industrially viable behavioral semantics for pseudocode on arbitrary data structures. This specification method is tunable to any desired level of abstraction, and provides rigor without formal overkill.

The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them, while the *transition relation* is specified by rules describing how functions change from one state to the next.

Functions are classified as *derived* functions, i.e. those coming with a specification or computation mechanism given in terms of other functions, and *basic* functions which can be *static* (never change during any run of the machine) or *dynamic* (may change as a consequence of agent actions or *updates*). Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* and *output* (only write) functions.

Basically, a transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* are a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Condition* is true. f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms.

To fire this rule in a state s_i, $i \geq 0$, all terms $t_1, \ldots, t_n, t$ are evaluated at s_i to their values, say $v_1, \ldots, v_n, v$, then the value of $f(v_1, \ldots, v_n)$ is updated to v, which represents the value of $f(v_1, \ldots, v_n)$ in the next state s_{i+1}. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \ldots, v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. They represent the abstract ASM concept of basic object containers (memory units), which abstracts from particular memory addressing and object referencing mechanisms. Location-value pairs (loc, v) are called *updates* and represent the basic units of state change. Locations are updated by rule firing only when no inconsistent updates occur, namely for any location loc and all elements v, w, it is true that if (loc, v) and (loc, w) are simultaneously updates, then $v = w$.

The (unique) *main rule* is a transition rule and represents the starting point of the machine program (i.e. it calls all the other ASM transition rules). Initial values for domains and functions are defined in a set of *initial states*. *Executing* an ASM means executing its main rule starting from a specified initial state.

A *computation* of an ASM M is a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of M, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing simultaneously all of the transition rules which are enabled in S_n.

These is a limited but powerful set of *rule constructors* that allow to express simultaneous parallel actions (`par`) of a single agent *self*, either in an atomic way, *Basic ASMs*, or in a structured and recursive way, *Structured or Turbo ASMs*, by sequential actions (`seq`), iterations (`iterate`, `while`, `recwhile`), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification `choose`) and unrestricted synchronous parallelism (universal quantification `forall`). Furthermore, it supports a generalization where multiple agents interact in parallel in a synchronous/asynchronous way, *Synch/Asynch Multi-agent ASMs*.

The increasing application of the ASM formal method for academic and industrial projects has caused a rapid development of tools [13] around ASMs of various complexity and goals: tools for mechanically verifying properties using theorem provers or model checkers, and execution engines for simulation and testing purposes.

4. ASM-BASED SERVICE-ORIENTED COMPONENTS BEHAVIOR

A service-based system is a distributed system: a system made of collection of distributed computational components (computers, software applications, devices, etc.) perceived by a user as a single system. However, compared with classical distributed systems, service-based systems are open systems, and therefore rather unpredictable as many parts may be unknown at a given time. Indeed services are volatile distributed entities; they may be searched, discovered, and dynamically linked with the remained part of the system environment, and unlinked at a later moment. A business process may be provided that acts as an orchestrator, i.e. an active entity that invokes available services according to a given set of rules to meet some business requirements. A service orchestration is a composition specification showing how services are composed in a workflow.

We represent in ASM a service-based system exploiting the notion of *distributed multi-agent ASMs*. Essentially, each business participant (or partner role) has an associated ASM agent with a *program* (a set of transition rules) to execute. A service-oriented component is an ASM endowed with (at least one) agent able to be engaged in conversational interactions with other external agents by providing/requiring services to/from other (partner) service-oriented components. The notion of service (operation) is captured in ASM by the notion of named *turbo rule* defined in an ASM for a service-oriented component. In this way we provide atomic (zero-time) parallel execution of entire (sub)machines as services whose computations, analyzed in isolation, may have duration and may access the needed state portion (interface), thus combining the atomic black box and the durative white box view of service-oriented components. Moreover, we assume that in our ASM component model there is an agent that acts as "orchestrator", the *orchestrator agent*, by executing (as its own program) an ASM rule capturing the behavior of the overall orchestration workflow.

We distinguish three basic kinds of service activities:

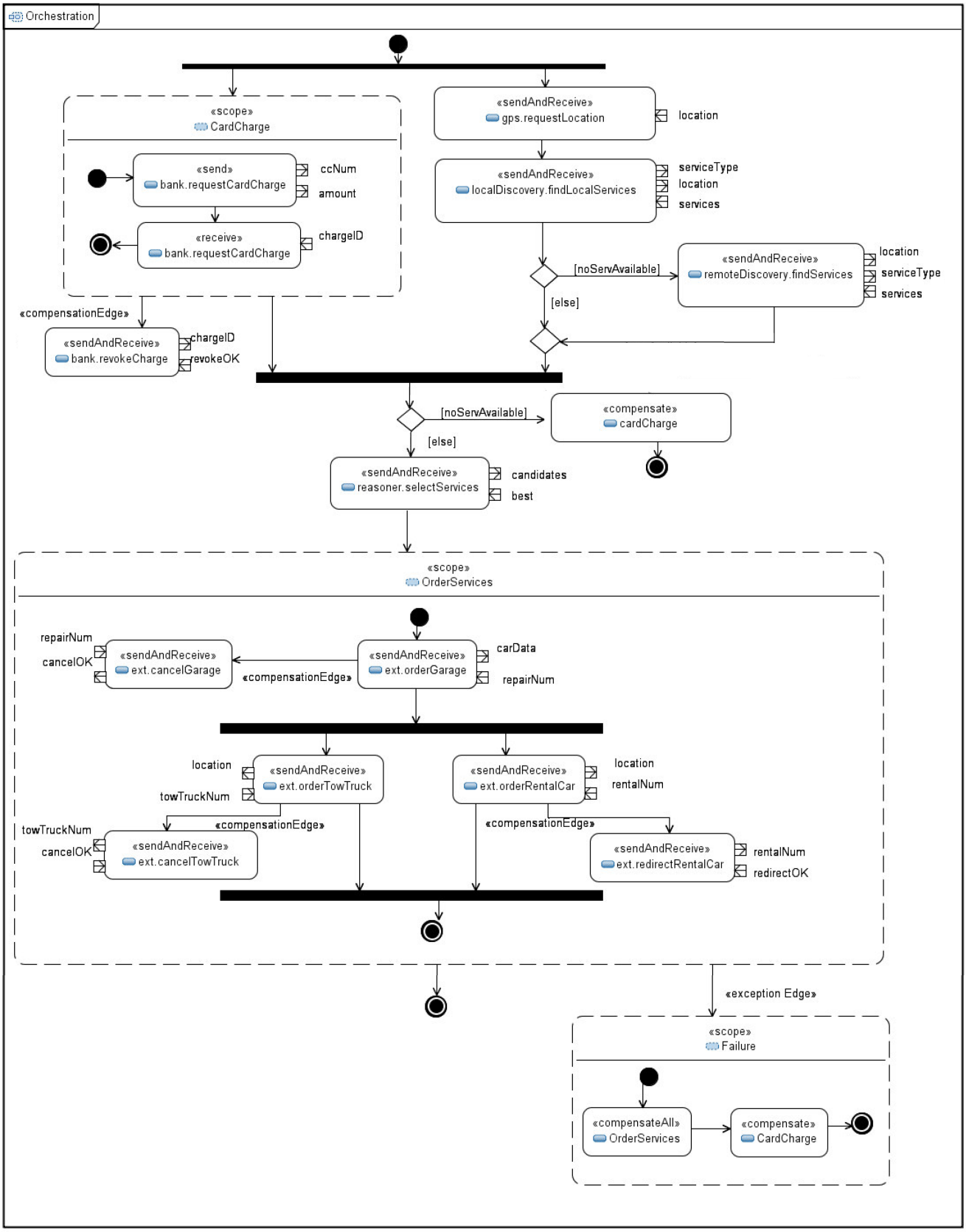


Figure 1: Service orchestration example adapted from [17]

(i) *functional activities*: they deal with data manipulation;
(ii) *communication activities*: they deal with patterns of service interactions pertaining to orchestration [4];
(iii) *fault activities*: they deal with faults or exceptions, and error recovery (by compensation or exception handlers).

The resulting system is therefore an asynchronous multi-agent ASM that will behave accordingly to the behavior of each service (ASM agent) involved in. This main ASM also provides the necessary initialization (such as appropriated component *bindings*) and initial startup of all agents' programs (in the main ASM rule) to make the system model executable.

In the following subsection, a transformational *semantic mapping* [12] is provided to transform UML4SOA descriptions of services orchestrations into formal executable descriptions based on the ASMs.

4.1 Service orchestration

The UML4SOA orchestration activity is semantically mapped into a *structured ASM rule* to be executed by the orchestrator ASM agent. The transformation patterns used to map each modeling element of the UML4SOA activity diagram into ASM concepts are reported in Fig. 2.

Services are invoked through communication activities of the orchestration workflow. We here focus only on bilateral interactions (or two-party interactions); however, additional communication patterns can be supported in ASM (e.g. multilateral interactions or multi-party interactions) as formalized in [5] allowing therefore more expressiveness also in the service interactions specification. Moreover, other control flow nodes (not reported here) can be easily supported in ASM as formalized in [8]. For example, the fork and merge nodes can be used separately. The fork node is to be intended as an *asynchronous parallel split* that spawns finitely many sub-agents using as underlying parallelism the concept of asynchronous ASMs. In addition, other ASM rule constructors may be used to allow for more expressiveness in describing orchestration workflows; for example, the *choice rule* can be used to define non deterministic selection patterns [8]. Moreover, more complicated workflow patterns like those introduced in the recent OMG initiative *Business Process Management Notation* (BPMN)[23] notation on business process modeling can be captured by ASM rule-patterns as well [10].

A detailed description follows on how communication activities and fault activities are represented in terms of ASMs. Functional activities do not require a special treatment as they can be intuitively captured by means of ASM rules with no special rule constructor or rule patterns.

4.1.1 Communication activities

For mapping communication actions, we take advantage of the precise high-level models for fundamental bilateral service interaction patterns, given by Barros and Boerger in [5] in terms of the ASMs. They define turbo ASM rules $\textsc{Send}_s$, $\textsc{Receive}_t$, $\textsc{SendReceive}_{s,t}$ and $\textsc{ReceiveSend}_{s,t}$ to capture the semantics of both asynchronous and synchronous message passing (the non-blocking and blocking mode) and the semantics of service interactions beyond simple request-response sequences by involving acknowledgment, resending, etc. All these variants are denoted by parameters $s \in \mathit{SendType} = \{\mathit{noAck},\ \mathit{ackNonBlocking},\ \mathit{ackblocking}\} \cup \{\mathit{noAckResend},\ \mathit{ackNonBlockingResend},\ \mathit{ackBlockingResend}\}$ and $t \in \mathit{ReceiveType} = \{\mathit{blocking},\ \mathit{buffer},\ \mathit{discard}\} \cup \{\mathit{noAckBlocking},\ \mathit{noAckBuffer},\ \mathit{ackBlocking},\ \mathit{ackBuffer}\}$.

These mentioned communication patterns are more general than those of the UML4SOA, but the semantics of the UML4SOA interaction actions *send*, *receive*, *send&receive*, and *replay*, can be captured by ASM submachines defined as *wrappers* of the turbo rules already formalized in [5]. We report below the definition of these wrapper rules. Each of these rules describes one side of the interaction and relies on a dynamic domain *Message* that represents message instances managed by an abstract message passing mechanism. Each message is characterized by dynamic context-dependent information provided by the following dynamic functions:

recipient:Message → *Agent* denoting the recipient agent;
sender:Message → *Agent* denoting the sender agent;
serviceOp:Message → *Rule* denoting the service operation;
serviceData:Message → *D* denoting data of some generic type *D* to be send or received.

The requirement that two messages have to be unequivocally related to one another when one is a request message and the other one is its response message, is captured (as in [5]) by two dynamic predicates[2] *RequestMsg* and *ResponseMsg* with a function *requestMsg*, which identifies for every $m \in \mathit{ResponseMsg}$ the $\mathit{requestMsg}(m) \in \mathit{RequestMsg}$ to which m is the *responceMsg*.

The wrapper rule WSend.

The wrapper rule WSend uses a simplified version of the pattern $\textsc{Send}_s$ in [5] resulting from the instantiation of the parameter $s = \mathit{noAck}$ so denoting a non-blocking action with no ack. An abstract machine $\textsc{BasicSend}(m)$ has the intended interpretation that the message m is sent to *recipient*(m). Possible faults at the sender's side during an attempt to send message m are captured by a machine $\textsc{HandleSendFault}(m)$ typically triggered by a condition **not** *OkSend*(m). Both these two machines use as guards the abstract monitored predicates *SendMode*(m) and *SendFaultMode*(m), respectively. As typical assumption *SendMode*(m) **and not** *OkSend*(m) implies *SendFaultMode*(m). Note that as in [5], for notational succinctness we assume the firing of the rules $\textsc{FirstSend}(m)$ and *HandleSendFault*(m) is preemptive. This means that the guard *SendMode*(m) (resp. *SendFaultMode*(m)) automatically becomes false after firing the $\textsc{FirstSend}(m)$ (resp. $\textsc{HandleSendFault}(m)$) rule.

```
rule WSend(lnk, R_A, snd) =
extend Message with m do
    seq
        par
            recipient(m):= lnk
            sender(m):= self
            serviceOp(m):= R_A
            serviceData(m):= snd
            RequestMsg(m):= true
            SendMode(m):= true
        endpar
        Send(m)_noAck

rule Send_noAck(m) =
par
    FirstSend(m)
    HandleSendFault(m)
endpar
```

[2]We identfy sets with unary predicates.

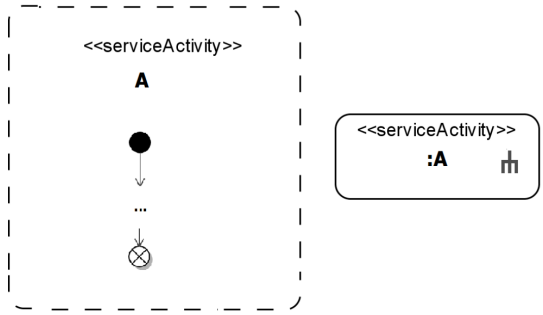
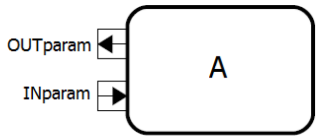
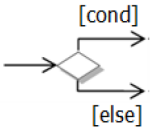
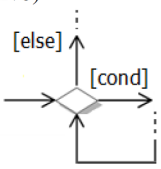
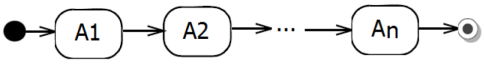
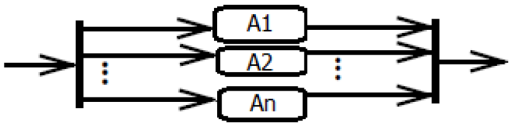
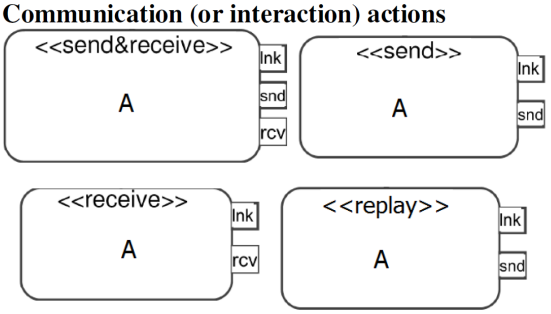
UML4SOA	ASM
Service activity (or scope) 	**Named rule** R_A **rule** RA = <rule definition>
Functional action (data manipulation) with one or more input/output parameters 	**Named rule RA** • INparam is a *logical variable* (parameter of R_A) or a *function term* (read within R_A) • OUTparam is a *location variable* or a *location term* appearing in the left side of an assignment rule within R_A, or the *return value* of R_A (if R_A is a *turbo rule* with return value)
Decision node 	**Conditional rule** **if** cond **then** R1 **else** R2 where cond is a *boolean term*
Decision node (iterative) 	**IterativeWhileRule** **while** cond **do** ... where cond is a *boolean term*
Sequence control flow 	**Seq rule** **seq** RA1 RA2... RAn **endseq**
(Coupled) **For-join nodes** 	**Synchronous parallel split** where all activities A1, A2, .., An are executed simultaneously as actions of one agent **par** RA1 RA2 ... RAn **endpar**
Communication (or interaction) actions  *lnk*: service partner link *snd,rcv*: data to be send or received	**Predefined submachines for interaction** WSEND (lnk, R_A, snd) WSENDRECEIVE (lnk, R_A, snd,rcv) WRECEIVEt (lnk, R_A, rcv) WREPLAY(lnk, R_A, snd)

Figure 2: From UML4SOA to ASMs

```
rule FirstSend(m) =
if SendMode(m) and OkSend(m)
then BasicSend(m)

rule HandleSendFault(m) =
if SendFaultMode(m) then SendFaultHandler(m)
```

The wrapper rule WReceive.

This wrapper uses a simplified version of the pattern Receive$_t$ in [5] resulting from the instantiation of the parameter $t = noAckBlocking$ so denoting a blocking action with no ack.

```
rule WReceive(lnk, R_A, rcv) = Receive_noAckBlocking(m)
where recipient(m) = self and sender(m) = lnk and
      serviceOp(m) = R_A and ResponseMsg(m)

rule Receive_noAckBlocking(m) =
if Arriving(m) and ReadyToReceive(m)
then Consume(m)
```

Note that the submachine Consume(m) left abstract in [5], is here refined as

$$\text{Consume}(m) \equiv rcv := serviceData(m)$$

in order to store in the location rcv the received data embedded in the received message.

The wrapper rule WReplay.

This wrapper uses the pattern Send$_{noAck}$ to return values to a request point within a message which is the response of a previous message of service request.

```
rule WREPLAY(lnk, R_A, snd) =
    let msg = m' ∈ Message | RequestMsg(m') and
        serviceOP(m') = R_A in
        extend Message with m do
        seq
            par
                recipient(m):= lnk
                sender(m):= self
                serviceOp(m):= R_A
                serviceData(m):= snd
                ResponseMsg(m):= true
                requestMsg(m):= msg
                SendMode(m):= true
            endpar
            responseMsg(msg):= m
            SEND_noAck(m)
```

The wrapper rule WSendReceive.

This wrapper uses a simplified version of the pattern Send$_s$Receive$_t$ in [5] resulting from the instantiation of the parameter $s = noAck$ and $t = noAckBlocking$ so denoting a non-blocking send action with no ack of a service request, followed by a blocking receive action with no ack of a service response.

```
rule WSendReceive(lnk, R_A, snd, rcv) =
extend Message with m do
    seq
        par
            recipient(m):= lnk
            sender(m):= self
            serviceOp(m):= R_A
            serviceData(m):= snd
            RequestMsg(m):= true
            SendMode(m):= true
        endpar
        SendReceive_noAck,noAckBlocking(m)

rule SendReceive_noAck,noAckBlocking(m) =
par
    Send_noAck(m)
    Receive_noAckBlocking(m')
endpar
where m' ∈ Message and ResponseMsg(m') and
      recipient(m') = self and serviceOp(m') = R_A and
      requestMsg(m') = m
```

4.1.2 Fault activities

UML4SOA *compensation handlers* can be also specified in terms of rules to be executed in case of fault. More precisely, executing a «compensate»action (not shown in the mapping table) for a certain (named) service activity A, that is given as a parameter, corresponds to invoke an ASM compensation rule associated to the corresponding service rule R_A through a predefined function $compensate(R_A)$ (denoting the «compensation»edge of UML4SOA). In order to support also «compensateAll»actions of the UML4SOA with the semantics of invoking all installed compensation handlers that are nested in the current service activity node, we require that an appropriated rule $compensateAll(R_A)$ is explicitly defined. The body of such a rule must consist of the sequential invocations of all compensation handlers rules for all service actions inner in the scope, in reverse order of their completion, i.e. the last completed element first. UML4SOA event (exception) handling mechanism is treated similarly.

4.1.3 Orchestration example

As example of the application of our mapping from UML4SOA to the ASMs, we here report the resulting ASM rule (the orchestrator agent's program) from the UML4SOA orchestration model in Fig. 1:

```
rule Orchestration() =
seq
    par
     CardCharge()
     seq
         WSENDRECEIVE(gps,«requestLocation»,undef,location)
         WSENDRECEIVE(localDiscovery,«findLocalServices»,
                      (serviceType,location),services)
         if noServAvailable
         then WSENDRECEIVE(remoteDiscovery,«findServices,
                           (serviceType,location),services)
    endpar
    if noServAvailable then compensate(«cardCharge»)
    else seq
            WSENDRECEIVE(reasoner,«SelectServices»,
                         candidates,best)
            OrderServices()
```

where $compensate(\text{«cardCharge»}) \equiv$ *WSENDRECEIVE(bank, «revokeCharge»,chargeID,revokeOK).*

The encoding for the sub-activities *CardCharge* and *OrderServices* is straightforward.

5. RELATED WORK

In this paper we provided an ASM-based executable formal description of the UML4SOA models. The UML4SOA language is a UML 2 extension for SOA, namely, a high-level domain-specific modeling language service orchestrations as an extension of UML2 activity diagrams. In order to make UML4SOA models executable, some code generators for low level target languages (such as BPEL/WSDL, the Jolie language, and Java) already exist [21]; however the

target languages do not provide the same rigor and preciseness of a formal method necessary for early design execution and analysis.

Other lightweight notations for service modeling have been proposed such as the OMG SoaML UML profile [19]. SoaML profile is more focused on architectural aspects of services and relies on the standard UML 2 activity diagrams for behavioral aspects without further specialization. Another example is the Service Component Architecture (SCA) standard [20]. This is an XML-based metadata model that describes the relationships and the deployment of services; it is a joint collaboration between major software vendors with the goal of delivering a language independent programming model for SOA. The SCA standard is supported by a visual notation and runtime environments (like Apache Tuscany and FRAscaTI) that enables developers to create service components that can be assembled into composite applications. However, it does not provide behavioral modeling abstractions.

Within the EU project SENSORIA [18], another high-level modeling notation specific to the SOA domain, named SRML [24], has been developed. SRML is a declarative modeling language for service-oriented systems with a computation and coordination model. We believe it is worth to study the feasibility of defining an encoding from SRML into ASMs, but we leave it as a challenge for future work. The goal of this activity would be the definition of an executable operational semantics of SRML models in terms of the ASMs.

Several *process calculi* for the specification of SOA systems have been recently designed (see, e.g., [16, 14, 15, 6]). They provide linguistic primitives supported by mathematical semantics, and analysis and verification techniques for qualitative and quantitative properties. In particaulr, in [3] an encoding of UML4SOA in COWS (Calculus for the Orchestration of Web Services), a recently proposed process calculus for specifying and combining services while modeling their dynamic behavior, is presented.

Within the ASM community, the ASMs have been used in the SOA domain for the purpose of formalizing business process modeling languages and middleware technologies related to web services [10, 8, 11, 1].

6. CONCLUSION AND FUTURE WORK

We proposed an ASM-based executable specification for service orchestrations modeled with the UML4SOA profile. The application of our semantic mapping from UML4SOA to ASMs provides a formal counterpart of the graphical UML4SOA description. The resulting rigorous model is useful for analysis purposes, at first instance for platform-independent executability of models. The specification here provided has to be intended as a foundational model not specifically tight to current web services technologies.

As future work, we plan to integrate the orchestration modeling with the specification of service behaviors, so integrating *intra-* and *inter-component* behavior. Moreover, we want to address the behavioral aspects of service discovery (for the lookup of service provider interfaces and service locations) and self-adaptability.We plan to specify and reason about "classes of properties" of services models through the analysis (validation and verification) tools supporting the ASMs; for example, for checking that the services resulting from a composition meet desirable correctness properties and do not manifest unexpected behaviors. We aim also at defining and developing synthesis patterns to generate code automatically (at least for some critical parts) from ASM models of services.

7. REFERENCES

[1] M. Altenhofen, A. Friesen, and J. Lemcke. Asms in service oriented architectures. *Journal of Universal Computer Science*, 14(12):2034–2058, 2008.

[2] C. Attiogbé. Can component/service-based systems be proved correct? *CoRR*, abs/0910.1901, 2009.

[3] F. Banti, R. Pugliese, and F. Tiezzi. Automated Verification of UML Models of Services. Tech. Rep., di Sistemi e Informatica, Univ. Firenze. Submitted for pubblication, 2009.

[4] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-Based Business Process Interconnection. Technical Report FIT-TR-2005-02, Faculty of IT, Queensland University of Technology, April, 2005.

[5] A. P. Barros and E. Börger. A compositional framework for service interaction patterns and interaction flows. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.

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

[7] E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *LNCS*, pages 264–283. Springer, 2005.

[8] E. Börger. Modeling Workflow Patterns from First Principles. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *ER*, volume 4801 of *LNCS*, pages 1–20. Springer, 2007.

[9] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[10] E. Börger, O. Sörensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *J. of Universal Computer Science*, 15(1):3–32, 2009.

[11] R. Farahbod, U. Glässer, and M. Vajihollahi. A formal semantics for the business process execution language for web services. In S. Bevinakoppa, L. F. Pires, and S. Hammoudi, editors, *WSMDEIS*, pages 122–133. INSTICC Press, 2005.

[12] A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *J. of Automated Software Engineering*, 16(3-4), 2009.

[13] ASMs web site. `http://www.eecs.umich.edu/gasm/`, 2008.

[14] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. : A calculus for service oriented computing. In A. Dan and W. Lamersdorf, editors, *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[15] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*, pages 305–314. IEEE, 2007.

[16] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *LNCS*, pages 33–47. Springer, 2007.

[17] EU project SENSORIA, ist-2 005-016004. tech. rep. d8.2.a, automotive case study: Requirements modelling and analysis of selected scenarios. `www.sensoria-ist.eu/`.

[18] EU project SENSORIA, ist-2 005-016004 `www.sensoria-ist.eu/`.

[19] OMG. Service oriented architecture Modeling Language (SoaML), ptc/2009-04-01, april 2009 `http://www.omg.org/spec/soaml/1.0/beta1/`.

[20] OSOA. Service Component Architecture (SCA) `www.osoa.org`.

[21] P. Mayer, A. Schroeder, and N. Koch. A model-driven approach to service orchestration. In *IEEE SCC (2)*, pages 533–536. IEEE, 2008.

[22] P. Mayer, A. Schroeder, N. Koch, and A. Knapp. The UML4SOA Profile. In *Technical Report, LMU Muenchen*, 2009.

[23] OMG, Business Process Management Notation (BPMN). `www.bpmn.org/`, 2008.

[24] SRML: A Service Modeling Language. `http://www.cs.le.ac.uk/srml/`, 2009.