Specifying Formal executable Behavioral Models for Structural Models of Service-oriented Components*

Elvinia Riccobene[1] and Patrizia Scandurra[2]

[1] DTI - Università degli Studi di Milano, Milan, Italy

[2] DIIMM - Università degli Studi di Bergamo, Bergamo, Italy

Abstract. This paper presents a behavioral formalism based on the *Abstract State Machine* (ASM) formal method and intended for high-level, platform-independent, executable specification of *Service-oriented Components*. We complement the recent *Service Component Architecture* – a graphical notation able to provide the overall and the components structure – with an ASM-based formalism able to describe the workflow of the service orchestration and the services internal behavior. The resulting *service-oriented component model* provides an ASM-based representation of both the structural and behavioral aspects of service-oriented systems, like service interactions, service orchestration, service tasks and compensation. The ASM formal description of a service-oriented system is suitable for rigorous execution-platform-independent analysis.

1 Introduction

The Service-Oriented paradigm is emerging as a new way to engineer applications that are exposed as *services* for possible use through standardized protocols. Services are loosely coupled, interoperable, evolvable, computational components available in a distributed environment. On top of these services, business processes and workflows are used to compose services as *service orchestration*. The Service-Oriented Architecture (SOA) is the architectural foundation for the Service-Oriented paradigm. SOA 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. It also requires specific modeling notations able to support the service-oriented system engineering with intuitive and easy to adopt design and implementation techniques.

Recently, the Service Component Architecture (SCA) [21] project is proposed to implement service construction based on the SOA principles. SCA provides a metamodel-based visual notation to construct and assemble service components in a platform independent manner. The SCA initiative is divided into several specification documents, such as the SCA assembly model specification, the SCA policy framework, etc. The assembly model specifies the concept of service components and focuses on the relationship between service components in a particular assembly. However, the SCA

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

Riccobene E. and Scandurra P. (2010).
Specifying Formal executable Behavioral Models for Structural Models of Service-oriented Components.
In *Proceedings of the 4th International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing*, pages 3-15
DOI: 10.5220/0003042200030015

assembly model lacks of a precise definition. As a service programming model, it is not enough for SCA to provide informal definition. A rigorous semantic model for SCA is necessary to specify the dynamic behavior of a service-oriented system, which can provide a formal foundation for the service component assembly and support to verify the compatibility of the assembled components. Moreover, the use of the SCA notation should be integrated within a precise engineering methodology for SOA, which, for high-level analysis purposes, requires a formal counterpart of the SCA description. Indeed, service-based systems usually have requirements such as service availability, functional correctness, protection of private data, etc. Implementing services satisfying these requirements demands the use of 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.

This paper presents a behavioral formalism based on the Abstract State Machine (ASM) [5] formal method and intended for the specification and analysis of service-oriented systems at a high level of abstraction and in a technology agnostic way (i.e. independently of the hosting middleware and runtime platforms and of the programming languages in which services are programmed). This is a first result of our ongoing work towards the development of an ASM-based *back-end* framework, for high-level specification and analysis of SCA descriptions of service-oriented component systems. ASMs expressiveness and executability allow for the definition and analysis of behavioral aspects of services (and complex structured interaction protocols) in a formal way but without overkill. Moreover, the ASM design method is supported by a set of tools (developed through model-driven engineering technology), the ASMETA toolset [13, 2], useful for validation and verification (essentially simulation, scenario-based validation, model-based testing, and model-checking) of ASM-based models of services.

A *service-oriented component model* is introduced to provide an ASM-based representation of both the structural and behavioral aspects of service-oriented systems like service interactions, service orchestration, service tasks and compensation. In particular, the component model integrates the orchestration modeling with the specification of service behaviors, so integrating *intra-* and *inter-* component behavior in one formalism; this is especially useful for analysis purposes to verify "global properties" that depend on "local properties". We start from the SCA standard [21] for the structural aspects of service-oriented components, and we complement the graphical view of a service-oriented system with a formal description which is then enriched with the executable specification of the services internal behavior and services orchestration. In particular, for modeling services behavior, the ASMs provide atomic (zero-time) parallel execution of entire (sub)machines – used to model service tasks – whose computations, analyzed in isolation, may have duration and may access the needed state portion, thus combining the atomic black box and the white box view of service-oriented components. For modeling services interaction, we exploit high-level communication patterns defined in [26] and adapted from [3]. They model in terms of the ASMs complex interactions of distributed service-based (business) processes that go beyond simple request-response sequences and may involve a dynamically evolving number of participants.

This paper is organized as follows. Some background concerning the SCA standard and the ASM formal method are given in Sect. 2 and 3, respectively. The ASM-based

service-oriented component model is presented in Sect. 4, while an illustrative case study is reported in Sect. 5. Sect. 6 provides a description of related work along the same direction and outlines some future directions of our work.

2 Service Component Architecture

The Service Component Architecture (SCA) [21] is an XML-based metadata model that describes the relationships and the deployment of services independently from SOA platforms and middleware programming APIs (such as Java, C++, Spring, PHP, BPEL, Web services, etc.). SCA is also supported by a graphical notation (a metamodel-based language developed with the Eclipse-EMF environment) and runtime environments (like Apache Tuscany and FRAscaTI) that enable developers to create service components, assemble them into composite applications, and run/debug them.

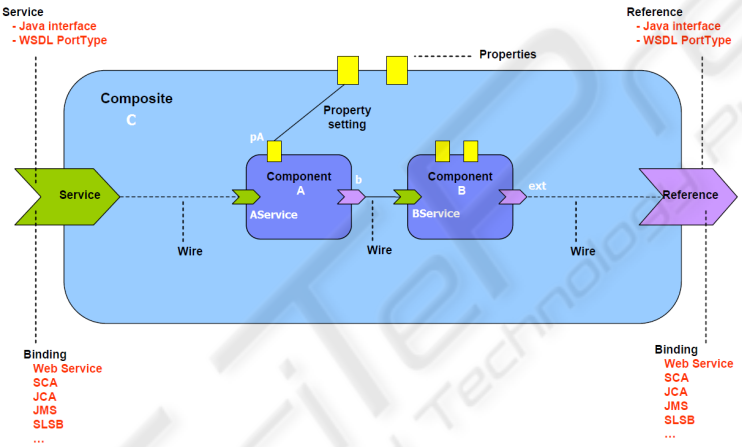

Fig. 1. An SCA composite example (adapted from the SCA Assembly Model V1.00 spec.)

To get an overview of the architecture of SCA, we will now look at its basic building blocks and their (inter-)relations. Fig. 1 shows an *SCA composite* (or *SCA assembly*) as a collection of SCA components using the SCA graphical notation. Following the principles of SOA, loosely coupled service components are used as atomic units or building blocks to build an application.

An *SCA component* is a configured piece of software that has been configured to provide its business functions (operations) for interaction with the outside world. This interaction is accomplished through: *services* that are externally visible functions provided by the component; *references* (functions required by the component) wired to services provided by other components or to references of the composite component containing the component; *properties* allowing for the configuration of a component implementation with externally set data values; and *bindings* that specify access mechanisms used by services and references according to some technology/protocol (e.g. WSDL binding to consume/expose web services, JMS binding to receive/send Java Message Service, etc.). Service and references are typed by *interfaces* that describe the

business function (operation). A particular business function is typically grouped with a set of other related operations, as defined by an interface, which as a whole make up the service offered by a provider component. Each invocation by a client component on a reference operation causes one invocation of the operation on one service provider. The provider may respond to the operation invocation with zero or more messages. These messages may be returned synchronously or asynchronously to the requester client.

As unit of composition and hierarchical design, assemblies of service components deployed together are supported in terms of *composite* components. A composite consisting of: properties, services, service implementations organized as sub-components, required services as references, wires connecting sub-components.

3 Abstract State Machines

Abstract State Machines (ASMs) are an extension of FSMs [6], where unstructured control states are replaced by states comprising arbitrary complex data. Although the ASM method comes with a rigorous mathematical foundation [5], 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. 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[3] when *Condition* is true.

There 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*.

Based on [5], an ASM can be defined as the tuple:

(*header*, *body*, *main rule*, *initialization*)

The *header* contains the *name* of the ASM and its *signature*[4], namely all domain, function and predicate declarations. Function 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*).

[3] f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule to a state S_i, $i \geq 0$, evaluate all terms $t_1, \ldots, t_n, t$ at S_i and update the function f to t on parameters $t_1, \ldots, t_n$. This produces another state S_{i+1} which differs from S_i only in the new interpretation of the function f.

[4] *Import* and *export* clauses can be also specified for modularization.

Dynamic functions are further classified into: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* (read and write by an agent and by the environment or by another agent) and *output* (only write) functions.

The *body* of an ASM consists of (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM. It also contains declarations (definitions) of transition rules. The body may also contains definitions of *axioms* for invariants to assume over domains and functions of the ASM.

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 defined in the body). The main rule is *closed* (i.e. it does not have parameters) and since there are no free global variables in the rule declarations of an ASM, the notion of a move does not depend on a variable assignment, but only on the state of the machine.

The *initialization* of an ASM is a characterization of the initial states. An initial state defines initial values for domains and functions declared in the signature of the ASM. *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.

A lightweight notion of module is also supported. An *ASM module* is an ASM without a main rule and without a characterization of the set of initial states.

In addition to its mathematical-based foundation, a general framework, the *ASMETA tool set* [14, 2], based on the Eclipse/EMF modeling platform is also available for developing, exchanging, simulating, testing and model checking ASM models.

4 Modeling Service-oriented Systems in ASMs

A *service-oriented 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 rather non predictable 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. Moreover, in a service *assembly* component (a composite component made of other internal or external service-oriented components), an agent may act as "orchestrator" by executing (as part of its own program)

```
module A
import STDL/StandardLibrary //domains and functions for standard data types
import STDL/CommonBehavior //predefined rules for services interactions
import AService //provided services (interface)
import BService //required services (interface)
export * //all functions and rules are exported
signature:
//Property
shared pA: Agent -> D //D is a domain for a data type
//Reference
shared b: Agent -> BService
//Client agent to which the component's agent will be linked to
shared client: Agent -> Agent
//Other user-defined domains and functions (if any)
controlled rcv: Agent -> String
...
definitions:
//Axioms (if any), i.e. assumptions and constraints on functions
...
//Rule for the provided business function getPA in the AService interface
rule r_getPA($a in AService, client in String) =
  seq
    ... //Do something for the client
    getPA($a,client) := ... //setting of the out business function location
  endseq
//Other utility rules
...
//Agent's program (life cycle): receive a request and handle it
rule r_A () =
  seq
    r_wreceive(client(self),"getPA",rcv)
    r_getPA(self,rcv) //direct service invocation
    r_wreplay(client(self),"getPA",getPA(self,client))
  endseq
//Constructor rule (invokable by the container composite)
macro rule r_init($a in AService) = ... //do initial properties settings and other
```

```
module AService
import STDL/StandardLibrary
export *
signature:
//decl. for business roles and functions
signature:
domain AService subsetof Agent
out getPA: Prod(Agent,String) -> D
...
```

Fig. 2. ASM modules for an SCA component `A` and its provided service interface `AService`.

an ASM rule capturing the behavior of the orchestration workflow. 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.

4.1 Service-oriented Components and their Assemblies

A transformational semantic mapping is provided to transform SCA descriptions of service structures into ASM-based formal descriptions. Listings in Fig. 2 and in Fig. 3 report the templates of an ASM module corresponding to an SCA component (like the component `A` in Fig. 1) with its provided service interface (like the `AService` interface provided by the `A` component) and to an SCA composite (like the composite `C` in Fig. 1), respectively, using the AsmetaL notation of the ASMETA toolset.

Appropriate transformation rules map the SCA key modeling elements (components, properties, services, references, wires, and composites) into ASM concepts. Essentially, an SCA service-oriented *component* is mapped into an *ASM module* endowed with (read: provides a type declaration for) at least one agent able to interact with other

```
module C
import A,B //import of ASM modules for subcomponents
export *
signature:
//Agents of the sub−components
static compA: AService
static compB: BService
//Properties
shared pA: D //D is a domain for a data type
...
shared ext: Agent //external reference
shared client: Agent −> Agent //Client agent to which the component's agent A will be linked to
//Other user−defined domains and functions (if any)
...
definitions:
... //Axioms (if any), i.e. assumptions and constraints on functions
//Constructor rule
rule r_init =
  par
    //wires setting
    client(compA) := client
    b(compA):= compB
    ...
    //Properties setting
    pA(compA):= pA
    ...
    //Agents program assignment
    program(compA) := r_A()
    program(compA) := r_B()
    //execution of agents initialization routines
    r_init(compA)
    r_init(compB)
  endpar
```

Fig. 3. ASM template for an SCA composite `C`.

external service-oriented components. Each service-oriented component with its business role has, therefore, an associated ASM and an ASM agent with a *program* to execute. An SCA component's *property* is straightforwardly mapped into an ASM function. An *interface* is a description of business functions. Services and references of a component are typed by interfaces. An interface is mapped into an ASM module containing only a collection of declarations of signature elements (domains and functions) for the business roles, declared in terms of subdomains of the predefined ASM `Agent` domain, and business functions, declared as parameterized ASM *out* functions. This ASM module is imported (through *import clauses*) by both the "provider" ASM module and the "requester" ASM module in order to "provide" (by giving definitions for those elements), respectively to "require" (by exposing an explicit reference typed by the declared agent subdomain), the declared business functions.

The ASM module `A` shown in the left of Fig. 2 (corresponding to the component `A` in Fig. 1), for example, provides definitions for the business functions declared in the imported `AService` ASM module (corresponding to the provided `AService` interface) shown in the right of Fig. 2. The `A` module also provides declarations for the property `pA`, the reference `b` to a `BService` agent, a reference `client` to a generic client agent, and other functions. The agent domain `AService` declared in the `AService` module and the rule `r_A` characterize the agent associated to the component `A`.

The notion of *service* operation provided by a component is captured by a named ASM *turbo rule*. It models the notion of submachine computation in a black-box view,

hiding the internals of the subcomputation by compressing them into one step. The name of such a rule – it is a convention – is the same name of the out business function declared in the typing service interface. In case of a return value, the body of such a rule must contain, among other things, an update of such out function (location); the value of such location denotes the value to be returned to the client. See, e.g., the `r_getPA` rule in the ASM module `A` in Fig. 2 and the occurrence within it of the business function `getPA` (declared in the `AService` module) on the left-side of an update-rule.

Services can be accessed through *references* in SCA. These are abstract access endpoints to services that will be possibly discovered at runtime. In the ASMs, references are represented in terms of functions that have as codomain a subset of the *Agent* domain named with the name of the reference's typing interface (see, e.g., the reference `b` to a `BService` agent in the ASM module `A` in Fig. 2). This domain is declared in the ASM module corresponding to the reference's typing interface, and the ASM module corresponding to the component exposing the interface has also to import the ASM module for the interface. In this way we identify (even if it is not known at design time) the partner's business role (i.e. the agent type).

An SCA *composite* component (made of an assembly of components) is represented by a composite ASM module that embeds (through import clauses) the ASM modules corresponding to the sub-components of the SCA composite. Communication links between components are denoted in SCA by appropriated *wires* as configured by the assembly. These links are created in the initial state or in an initialization (constructor) rule of the ASM corresponding to the assembly component in terms of function (reference) assignments. The ASM module `C` shown in Fig. 3 (corresponding to the composite `C` in Fig. 1), for example, imports the ASM modules for the sub-components `A` and `B`, and declares two references `compA` and `compB` to the agents of the subcomponents. It also carries out in the constructor rule `r_init` the wires setting, properties setting, agents' program assignment, and initialization of the sub-components. A "top-level" composite containing the overall assembly is mapped into composite ASM (read: the *main ASM*) with a possible ASM initial state to initialize the ASM modules and their agents as dictated by the configured sub-components.

We abstract from the SCA notion of *binding*, i.e. from several access mechanisms used by services and references (e.g. WSDL binding, JMS binding, etc.). We assume that components communicate over the communication links through an abstract asynchronous and message-oriented mechanism (see next subsection), where a message encapsulates information about the partner link and the referenced service name and data.

4.2 Service Behavior: Orchestration and Interactions

The behavior of a service-oriented system is the description of the involved service activities composed in a workflow (orchestration). Here we adopt a simple service composition technique. We compose services by embedding more than one service component into a top-level composite component (the main ASM). A component embeds an ASM agent executing (as its own program) an appropriate interactive behavior or a "piece" of orchestration workflow. The overall orchestration is, therefore, spread throughout the

internal components[5] and consists of the patterns of interactions (or communication).

For modeling service orchestration, basic control-flow constructs are easily supported in the ASMs by rule constructors such as the *seq-rule* for executing activities sequentially, the *par-rule* for synchronous parallel split of activities, the *conditional rule* for alternative flows, etc.. Other control flow patterns (not reported here) can be easily supported in ASM as formalized in [7]. For example, the "fork" and "merge" nodes (using the same terminology of the UML activity diagrams) can be used separately; a fork node is to be intended as an *asynchronous parallel split* [7] that spawns finitely many sub-agents using as underlying parallelism the concept of asynchronous ASMs. As another example, the *choice rule* can be used to define non deterministic selection patterns [7]. Moreover, more complicated workflow patterns like those introduced in the recent OMG initiative *Business Process Management Notation* (BPMN)[24] on business process modeling can be captured by ASM rule-patterns as well (some formalization work for BPMN has been already done; see for example [8]).

In addition to control-flow patterns, we define three basic kinds of service activities:
(i) *functional activities*: they deal with data manipulation (assignments);
(ii) *fault activities*: they deal with faults or exceptions, and error recovery (by compensation or exception handlers);
(iii) *communication or interaction activities*: they deal with message exchange between services to interact. Activities (i) and (ii) 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. *Compensation handlers* can be, for example, specified in terms of named ASM rules associated to certain services to be executed in case of faults. Communication activities (iii) deserve more explanation, as better explained below.

Services are invoked (i.e. interact) through communication activities. To this purpose, we take advantage of the precise high-level models for eight fundamental service interaction patterns, given by Barros and Boerger in [3] in terms of the ASMs. They define turbo ASM rules SEND_s, RECEIVE_t, $\text{SENDRECEIVE}_{s,t}$ and $\text{RECEIVESEND}_{s,t}$ 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 $s \in \{$*noAck, ackNonBlocking, ackblocking, noAckResend, ackNonBlockingResend, ackBlockingResend*$\}$ and $t \in \{$*blocking, buffer, discard, noAckBlocking, noAckBuffer, ackBlocking, ackBuffer*$\}$.

Therefore, we capture the semantics of common interaction actions *send*, *receive*, *send&receive*, and *replay* by the following ASM submachines (turbo rules):

– WSEND_{noAck}(lnk, op, snd): sends data *snd* without blocking to the partner link *lnk* in reference to the service operation *op*.

– $\text{WRECEIVE}_{noAckBlocking}$(lnk, op, rcv): receives data in the location *rcv* from the

[5] For the specification of the externally visible behavior of service components as provided to or required from a partner, some proposals (such as [23]) adopt a (declarative) *Protocol State Machine* formalism to specify which interaction a component can be engaged in which state and under which condition. Similarly, in ASM the (unknown) behavior of an external required component may be captured by a class of ASMs, named *control-state ASMs*, that specifies in an abstract way the external partner agent's life cycle when engaged in service interactions.

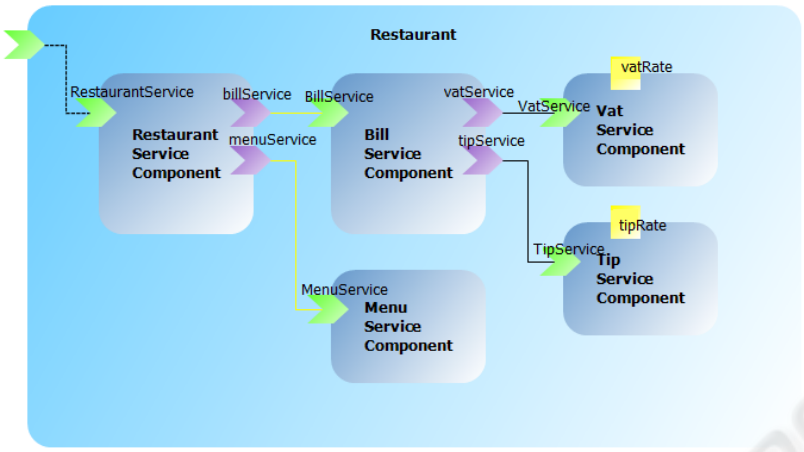

Fig. 4. SCA structure of the Restaurant case study.

partner link *lnk* in reference to the service operation *op*; it blocks until data are received.

– WREPLAY$_{noAck}$(lnk, op, snd): returns some data *snd* to the partner link *lnk*, as response of a previous *op* request received from the same partner link.

– WSENDRECEIVE$_{noAck,noAckBlocking}$(lnk, op, snd,rcv): in reference to the service operation *op*, some data *snd* are sent to the partner link *lnk*, then the action waits for data to be sent back, which are stored in the receive location *rcv*.

These submachines have been already defined in [26] as "wrappers" of the general patterns originally presented in [3]. Each of these communication 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.

Note that additional communication patterns can be supported in ASM (e.g. for *multi-party interactions*) as specializations of the more abstract patterns formalized in [3], allowing, therefore, more expressiveness in the service interactions specification.

5 Running Case Study

Fig. 4 shows the SCA assembly of the *Restaurant case study* taken from the SCA distribution [21]. The `Restaurant` composite is a composition of five components: `RestaurantServiceComponent` that allows a client to see the menus proposed by the restaurant and also to compute the bill for a particular menu; `MenuService-Component` that provides different menus; a `Menu` (as data type) is defined by a description and the price without taxes; `BillServiceComponent` that computes the price of a menu with the different taxes; `VATServiceComponent` that computes the VAT (Value Added Tax); and `TipServiceComponent` that computes the tip.

As example, Fig. 5 reports the ASM module for the `BillServiceComponent`. The bill agents' program and the service rule are a small orchestration example for the coordination of the two helper services VAT and Tip.

```
module BillServiceComponent
import STDL/StandardLibrary
import STDL/CommonBehavior
import BillService //provided interface
import TipService //required interface
import VatService //required interface
export *
signature:
shared vatService: Agent -> VatService //reference
shared tipService: Agent -> TipService //reference
shared clientBillService: Agent -> Agent //Client agent to which the component is linked to
//Other functions used for internal computations
controlled priceWithTaxRate: Agent -> Real
controlled priceWithTipRate: Agent -> Real
controlled menuprice : Agent -> Real
definitions:
//Rule for the provided service operation getBill
rule r_getBill($a in Agent, $menuPrice in Real) =
seq
r_wsendreceive(vatService($a),"getPriceWithVat",$menuPrice,priceWithTaxRate($a))
r_wsendreceive(tipService($a),"getPriceWithTip",priceWithTaxRate($a),priceWithTipRate($a))
getBill($a,$menuPrice) := priceWithTipRate($a) //setting of the out business function location
endseq
rule r_BillServiceComponent = //Agent program (life cycle)
 seq
  r_wreceive(clientBillService(self),"getBill",menuprice(self))
  r_getBill(self,menuprice(self)) //direct service invocation
  r_wreplay(clientBillService(self),"getBill",getBill(self,menuprice(self)))
 endseq
//Constructor rule
macro rule r_init($a in BillService) = skip //do nothing
```

Fig. 5. ASM module of the `BillServiceComponent`.

6 Related Work and Future Directions

On the formalization of the SCA component model, some previous works, like [9, 10] to name a few, exist. However, they do not rely on a practical and executable formal method like ASMs. In [18], an analysis tool, *Wombat*, for SCA applications is presented; their approach is similar to our as their tool is used to perform simulation and verification tasks by transforming each SCA module into one composed Petri net. We are, however, not sure that their methodology scales effectively to large systems.

Lightweight visual notations for service modeling have been proposed such as the OMG SoaML UML profile [20]. The SoaML profile, like the SCA initiative, is more focused on architectural aspects of services.

Another UML extension for service modeling, named UML4SOA [23], has been developed within the EU project SENSORIA [19]. The UML4SOA language is fo-

cused on modeling 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, Jolie, and Java) already exist [22]; however the target languages do not provide the same rigor and preciseness of a formal method necessary for early design exploration and analysis.

Within the EU project SENSORIA, another modeling notation specific to the SOA domain, named SRML [25], 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 UML4SOA[6] (or 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 UML4SOA (SRML) models in terms of the ASMs and then explore ASM-based analysis tools.

Several *process calculi* for the specification of SOA systems have been designed (see, e.g., [17, 15, 16, 4]). They provide linguistic primitives supported by mathematical semantics, and verification techniques for qualitative and quantitative properties. In particular, in [11] an encoding of UML4SOA in COWS (Calculus for the Orchestration of Web Services), a recently proposed process calculus for specifying services while modeling their dynamic behavior, is presented. Compared to these notations, the ASMs have the advantage to be executable and formal without mathematical overkill.

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, like [8, 7, 12, 1] to name a few. Some of these previous formalization efforts are at the basis of our work.

As future work, we propose to complete the proposed ASM-based service-oriented component model towards different directions. We have been developing several case studies, some taken from the SCATuscany distribution and some other from the EU SENSORIA project [19], in order to assure the approach scales effectively to large and different systems. We have been also extending the Eclipse-based SCA Tools and exploiting the Tuscany runtime that allows extension modules to be plugged in, to provide a direct support of the ASM-based component model and automate the transformation from SCA to ASMs. We aim also at defining and developing synthesis patterns to generate code automatically (at least for some critical parts) from ASM models of services.

We plan to revise our component model (if necessary) to take in consideration also the changes recently made to the SCA Assembly specification [21] to introduce some extensions for Event Processing. Moreover, since service-oriented components can be discovered and bound to other components at run-time to produce configurations, we want to address the behavioral aspects of *service discovery* (for the lookup of service provider interfaces and service locations) and *self-adaptability* by extending the service-oriented component model in ASMs with specific "roles" of service agents.

In the future, we aim also at specifying and reasoning about "classes of properties" of services through the ASMETA analysis tools, for example, to verify the compatibility of the assembled components and check that the services resulting from a composition meet desirable properties without manifesting unexpected behaviors.

[6] Such an encoding would be natural to carry out for the UML4SOA since we also inspired from the UML4SOA communication activities for our interaction patterns.

References

1. M. Altenhofen, A. Friesen, and J. Lemcke. Asms in service oriented architectures. *J. of Universal Computer Science*, 14(12):2034–2058, 2008.
2. The ASMETA tooset website. http://asmeta.sf.net/, 2006.
3. Alistair P. Barros and Egon Börger. A compositional framework for service interaction patterns and interaction flows. In *ICFEM'05 Proc.*, *LNCS* 3785, pages 5–35. Springer, 2005.
4. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *FMOODS Proc.*, *LNCS* vol. 5051, pages 19–38. Springer, 2008.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
6. Egon Börger. The ASM method for system design and analysis. A tutorial introduction. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005 Proc.*, *LNCS* vol. 3717, pages 264–283. Springer, 2005.
7. Egon Börger. Modeling Workflow Patterns from First Principles. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *ER*, *LNCS* vol. 4801, pages 1–20. Springer, 2007.
8. E. Brger, O. Srensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *J. of Universal Computer Science*, 15(1):3–32, 2009.
9. Zuohua Ding, Zhenbang Chen, and Jing Liu. A rigorous model of service component architecture. *Electr. Notes Theor. Comput. Sci.*, 207:33–48, 2008.
10. Dehui Du, Jing Liu, and Honghua Cao. A rigorous model of contract-based service component architecture. In *CSSE (2)*, pages 409–412. IEEE Computer Society, 2008.
11. F. Tiezzi F. Banti, R. Pugliese. Automated verification of UML models of services. Submitted for publication, 2009.
12. R. Farahbod, U. Glässer, and M. Vajihollahi. A formal semantics for the business process execution language for web services. In Savitri Bevinakoppa, Luís Ferreira Pires, and Slimane Hammoudi, editors, *WSMDEIS*, pages 122–133. INSTICC Press, 2005.
13. A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *Int. Conf. on Software Engineering Advances, ICSEA* 2008.
14. Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A metamodel-based simulator for ASMs. In Andreas Prinz, editor, *14th Int. ASM Workshop Proc.*, 2007.
15. C. Guidi et al. : A calculus for service oriented computing. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, *LNCS* 4294, pages 327–338. Springer, 2006.
16. I. Lanese, F. Martins, V. Thudichum Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM*, pages 305–314. IEEE, 2007.
17. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *LNCS*, pages 33–47. Springer, 2007.
18. Axel Martens and Simon Moser. Diagnosing sca components using wombat. In *Business Process Management Proc.*, *LNCS* 4102, pages 378–388. Springer, 2006.
19. EU project SENSORIA, ist-2 005-016004 www.sensoria-ist.eu/.
20. OMG. The SoaML Profile, ptc/2009-04-01
21. OSOA. Service Component Architecture (SCA) www.osoa.org.
22. P. Mayer, A. Schroeder, and N. Koch. A model-driven approach to service orchestration. In *IEEE SCC (2)*, pages 533–536. IEEE, 2008.
23. P. Mayer et al. The UML4SOA Profile. *Tech. Rep., LMU Muenchen*, 2009.
24. OMG, Business Process Management Notation (BPMN). www.bpmn.org/, 2008.
25. SRML: A Service Modeling Language. http://www.cs.le.ac.uk/srml/, 2009.
26. E. Riccobene and P. Scandurra. An ASM-based executable formal model of service-oriented component interactions and orchestration. Workshop on Behavioural Modelling - Foundations and Application (BM-FA 2010), ACM DL Proc. ISBN 978-1-60558-961-9