An Eclipse-based SCA design framework to support coordinated execution of services

Fabio Albani[1] Elvinia Riccobene[2] Patrizia Scandurra[1]

[1] Università degli Studi di Bergamo, DIIMM, Dalmine (BG), Italy
patrizia.scandurra@unibg.it
[2] Università degli Studi di Milano, DTI, Crema (CR), Italy
elvinia.riccobene@unimi.it

Abstract. This paper presents a design framework for coordinated execution of service-oriented applications. The framework is based on the OSOA open standard model *Service Component Architecture* (SCA) for heterogeneous service assembly and on the formal method *Abstract State Machine* (ASM) for modeling notions of service behavior, interactions, orchestration, and compensation in an abstract but executable way. The framework was developed by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany, and the simulator AsmetaS of the ASM toolset ASMETA.

1 Introduction

Service-oriented applications are playing an important role in several application domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few) since they offer complex and flexible functionalities in widely distributed environments by composing, possibly dynamically "on demand", different types of services. Web Services is the most notable example of technology for implementing such components. On top of these service-oriented components, business processes and workflows can be (re-)implemented as composition of services – *service orchestration* or *service coordination*.

However, early designing, prototyping, and testing of the functionality of such assembled service-oriented applications is hardly feasible since services are discoverable, loosely-coupled, and heterogeneous (i.e. they differ in their implementation/middleware technology) components that can only interact with others on compatible interfaces. Concurrency and coordination aspects that are already difficult to address in component-based system design (though extensively studied), are even more exacerbated in service-oriented system design. In order to support the engineering of service-oriented applications, to withstand inevitable faults, and to improve the service quality (such as efficiency and reliability), established foundational theories and high-level formal notations and analysis techniques traditionally used for component-based systems should be revisited and integrated with service development technologies.

This paper proposes a formal framework for coordinated execution of heterogeneous service-oriented applications. It relies on the *SCA-ASM* language [21]

that combines the OSOA open standard model *Service Component Architecture* (SCA) [18] for heterogeneous service assembly in a technology agnostic way, with the formal method *Abstract State Machine* (ASM) [8] able to model notions of service behavior, interactions, orchestration, and compensation [6, 5, 7] in an abstract but executable way. The framework is based on the Eclipse environment. It was developed by integrating the Eclipse-based SCA Composite Designer [22], the SCA runtime platform Tuscany [25], and the simulator AsmetaS of the ASM analysis toolset ASMETA [2].

A designer may use the proposed framework to provide abstract implementations in SCA-ASM of (i) *mock components* (possibly not yet implemented in code or available as off-the-shelf) or of (ii) *core components* containing the main service composition or process that coordinates the execution of other components (possibly implemented using different technologies) providing the real computation. He/she can then validate the behavior of the overall assembled application, by configuring these SCA-ASM models *in place* within an SCA-compliant runtime platform as implementation of (mock or core) components and then execute them together with the other (local or remote) components implementations according to the chosen SCA assembly.

This paper is organized as follows. Section 2 provides background on SCA and ASMs. Section 3 describe the proposed framework. Section 4 surveys some related work. Finally, Section 5 concludes the paper.

2 Background on SCA and ASMs

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

Fig. 1 shows an *SCA composite* (or *SCA assembly*) as a collection of SCA components. 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 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; *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.). Services and references are typed by *interfaces*. An interface describes a set of related oper-

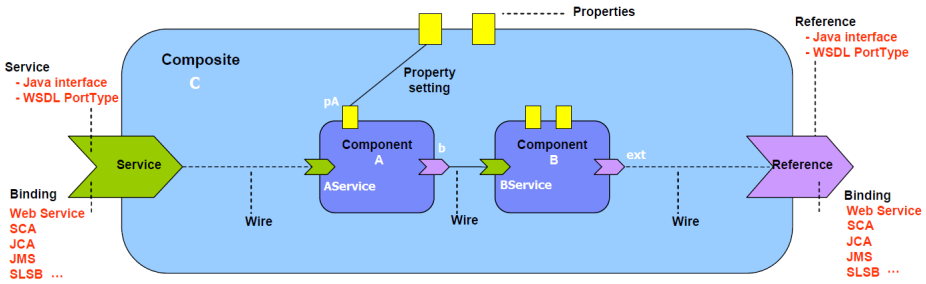

Fig. 1. An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)

ations (or business functions) which as a whole make up the service offered or required by a component. The provider may respond to the requester client of an operation invocation with zero or more messages. These messages may be returned synchronously or asynchronously.

Assemblies of components deployed together are called *composite* components and consist of: properties, services, services organized as sub-components, required services as references, and wires connecting sub-components.

Abstract State Machines ASMs are an extension of Finite State Machines (FSMs) [8] where unstructured control states are replaced by states comprising arbitrary complex data. 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. The *transition relation* is specified by rules describing how functions change from one state to the next. There is a limited but powerful set of ASM *rule constructors*, but the basic transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed[3] when *Condition* is true. Distributed computation can be modeled by means of *multi-agent ASMs*: multiple agents interact in parallel in a synchronous/asynchronous way. Each agent's behavior is specified by a basic ASM.

Besides ASMs comes with a rigorous mathematical foundation [8], ASMs can be read as pseudocode on arbitrary data structures, and can be defined as the tuple (*header*, *body*, *main rule*, *initialization*): *header* contains the *signature*[4] (i.e. domain, function and predicate declarations); *body* consists of domain and function definitions, state invariants declarations, and transition rules; *main rule* represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body); *initialization* defines initial values for domains and functions declared in the signature.

[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.

Executing an ASM M means executing its main rule starting from a specified initial state. A computation 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 (*header*, *body*) without a main rule, without a characterization of the set of initial states, and the body may have no rule declarations.

An open framework, the *ASMETA tool set* [2], based on the Eclipse/EMF platform and developed around the *ASM Metamodel*, is also available for editing, exchanging, simulating, testing, and model checking models. *AsmetaL* is the textual notation to write ASM models within the ASMETA tool-set.

The SCA-ASM modeling language By adopting a suitable subset of the SCA standard and exploiting the notion of *distributed multi-agent ASMs*, the SCA-ASM modeling language [21] complements the SCA assembly model with the ASM model of computation to provide ASM-based formal and executable description of the services internal behavior, services orchestration, interactions, and compensations. According to this implementation type, a service-oriented component is an ASM endowed with (at least) one agent (a business partner or role instance) able to interact with other agents by providing and requiring services to/from other service-oriented components' agents. The service behaviors encapsulated in an SCA-ASM component are captured by ASM transition rules.

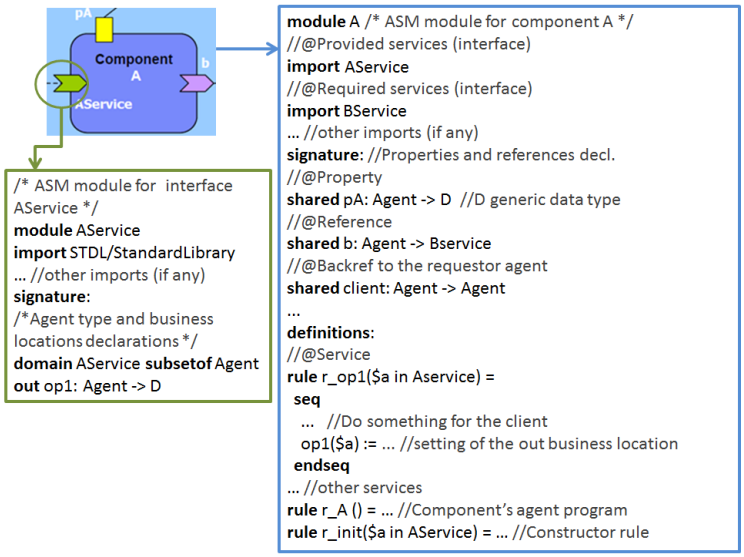

Fig. 2. SCA-ASM component shape

Fig. 2 shows the shape of an SCA-ASM component `A` using the graphical SCA notation, and the corresponding ASM modules for the provided interface

COMPUTATION AND COORDINATION		
Skip rule	**skip**	do nothing
Update rule	$f(t_1, \ldots, t_n) := t$	update the value of f at $t_1, \ldots, t_n$ to t
Call rule	$R[x_1, \ldots, x_n]$	call rule R with parameters $x_1, \ldots, x_n$
Let rule	**let** $x = t$ **in** R	assign the value of t to x and then execute R
Cond rule	**it** ϕ **then** R_1 **else** R_2	if ϕ is true, then execute R_1, otherwise R_2
Iterate rule	**while** ϕ **do** R	execute rule R until ϕ is true
Seq rule	**seq** $R_1 \ldots R_n$ **endseq**	rules $R_1 \ldots R_n$ are executed in sequence without exposing intermediate updates
Parallel rule	**par** $R_1 \ldots R_n$ **endpar**	rules $R_1 \ldots R_n$ are executed in parallel
Forall rule	**forall** x **with** ϕ **do** $R(x)$	forall x satisfying ϕ execute R
Choose rule	**choose** x **with** ϕ **do** $R(x)$	choose an x satisfying ϕ and then execute R
Split rule	**forall** $n \in N$ **do** $R(n)$	split N times the execution of R
Spown rule	**spawn** child **with** R	create a child agent with program R
COMMUNICATION		
Send rule	**wsend**[*lnk,R,snd*]	send data *snd* to *lnk* in reference to rule R (no blocking, no acknowledgment)
Receive rule	**wreceive**[*lnk,R,rcv*]	receive data *rcv* from *lnk* in reference to rule R (blocks until data are received, no ack)
SendReceive rule	**wsendreceive** [*lnk,R,snd,rcv*]	send data *snd* to *lnk* in reference to rule R waits for data *rcv* to be sent back (no ack)
Reply rule	**wreplay**[*lnk,R,snd*]	returns data *snd* to *lnk*, as response of R request received from *lnk* (no ack)

Table 1. SCA-ASM rule constructors for computation, coordination, communication

`AService` (on the left) and the skeleton of the component itself (on the right) using the textual AsmetaL notation of the ASMETA toolset. Details on the meaning of these concepts can be found in [21].

The ASM rule constructors and predefined ASM rules (i.e. named ASM rules made available as model library) used as basic SCA-ASM behavioral primitives are recalled in Table 1. In particular, communication primitives provide both synchronous and asynchronous interaction styles (corresponding, respectively, to the *request-response* and *one-way* interaction patterns of the SCA standard). Communication relies on a dynamic domain *Message* that represents messages managed by an *abstract message-passing* mechanism: components communicate over wires according to the semantics of the communication primitives and a message encapsulates information about the partner link, the referenced service name, and data. We abstract, therefore, from the SCA notion of *binding*. Indeed, we adopt the default SCA binding (`binding.sca`) for message delivering, i.e. the SOAP/HTTP or the Java method invocations (via a Java proxy) depending if the invoked services are remote or local, respectively.

3 The SCA-ASM design framework

The SCA-ASM framework[5] allows to design, assembly, and execute SCA-ASM models of components in an unique Eclipse-based environment.

The framework consists of a graphical modeling *front-end* and of a run-time platform as *back-end*. The graphical front-end is the *SCA Composite Designer* that is an Eclipse-based graphical development environment for the construction of SCA composite assemblies. An SCA metamodel (based on the Eclipse Modeling Framework (EMF) [11] – a platform for Model-driven Engineering) is at the core of such a graphical editor. We extended the SCA Composite Designer and the SCA metamodel to support ASM elements like component and interface implementation. Fig. 3 shows a screenshot of the tool. Appropriate ASM icons (see the right side of Fig. 3) may be used to specify ASM modules as (abstract) implementation of components and interfaces of the considered SCA assembly; alternatively, ASM modules files can be selected from the explorer view (on the left side of Fig. 3) and then dragged and dropped on the components and interfaces of the SCA assembly diagram.

The back-end is the open *Apache Tuscany SCA runtime* [25] – to run and test SCA assemblies of components developed with different implementation technologies and spread across a distributed environment (cloud and enterprise infrastructures) – combined with the ASMETA toolset to support various forms of high-level functional analysis via ASMs. In particular, we extended (as better described in the next Section) the Tuscany runtime platform to allow the execution of ASM models of SCA components through the simulator ASMETA/AsmetaS (as shown by the console output in Fig. 3) within Tuscany.

SCA-ASM makes it possible to specify abstract components, to compose them, and to simulate them and check various functional properties with the help of the ASMETA analysis toolset and of the Tuscany platform. Basically, the following two functional analysis scenarios are supported. *Offline analysis*: First, designers are able to exploit the ASMETA analysis toolset (also based on the Eclipse environment) to validate and verify SCA-ASM models of components in an off line manner, i.e. ASM models of such abstract (or mock) components may be analyzed in isolation. As analysis techniques, the ASMETA toolset includes simulation, scenario-based simulation, model-based testing and model checking. *In-place simulation*: Then, an in-place simulation scenario may be also carried out to execute early the behavior of the overall composite application. In this case, the AsmetaS simulator is directly invoked within the SCA runtime platform to execute the ASM specifications (intended as *abstract implementations*) of mock components together with the other real and heterogeneous (non ASM-implemented) components according to the chosen SCA assembly.

Several case studies of varying sizes and covering different uses of the SCA-ASM constructs have been developed. These include application examples taken from the SCA Tuscany distribution, a *Robotics task coordination* case study [15] of the EU project BRICS [10], and a scenario of the *Finance* case study of the

[5] https://asmeta.svn.sourceforge.net/svnroot/asmeta/code/experimental/SCAASM

EU project SENSORIA [23] related to a credit (web) portal application of a credit institute that allows customer companies to ask for a loan to a bank. Fig. 3 shows the SCA assembly of this last finance application. More details and functional requirements on this scenario can be found in [4].

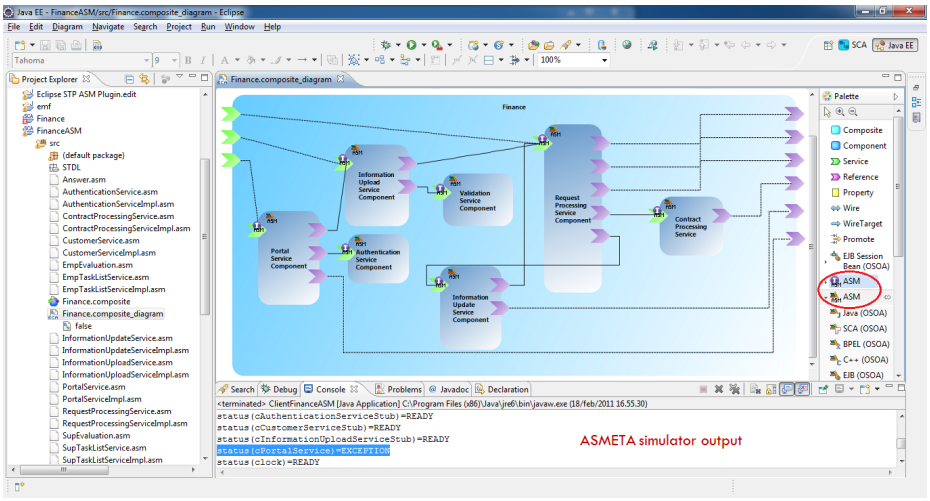

Fig. 3. SCA-ASM tool screenshot

3.1 Framework implementation

This subsection provides some details on how we extended the Eclipse-based SCA composite designer (the frontend) and the SCA Tuscany runtime (the backend) to support the SCA-ASM component implementation type.

Extending the Eclipse-based SCA composite designer First, we extended the SCA metamodel [22], an extensible EMF Ecore-compliant metamodel that represents concepts of the Open SOA SCA specifications 1.0 [18] plus different extending concepts to support open SCA runtimes (Apache Tuscany and Frascati). Extending the SCA metamodel to add new concepts to SCA and extend the tools to include them is straightforward. Fig. 4 shows the two basic concepts, *implementation* and *interface*, that we added to the SCA metamodel to support the editing of SCA-ASM components within standard SCA assembly files.

Then, we extended the SCA Composite Designer (see Figure 3), a graphical (GMF) development environment for the construction of SCA composite applications. This required us to develop Eclipse plug-ins to allow the use of the `ASMInterface` and the `ASMImplementation` creation tools from the palette or the contextual menu, to allow the setting of properties values in the `Properties` view for each created element, etc..

Extending the Tuscany SCA runtime SCA-ASM components use annotations to denote services, references, properties, etc. With this information, as

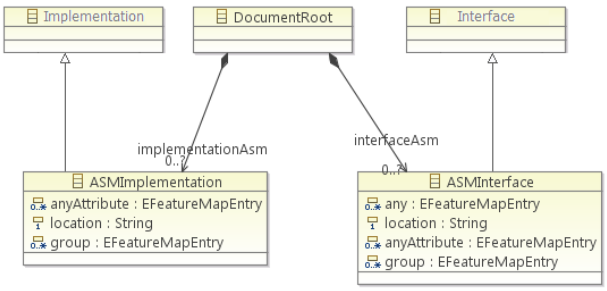

Fig. 4. A fragment of the SCA metamodel extension to support SCA-ASM

better described below, an SCA runtime platform (Tuscany in our case) can create a composition (an application) by tracking service references (i.e. required services) at runtime and injecting required services into the component when they become available.

Creating a new extension in Tuscany required to us two distinct steps. First, we developed the extension code (using the Java programming language) to handle the new technology `implementation.asm`. The UML package diagram in Fig. 5 shows the high-level structure and classes of this extension code. In the second step, the Tuscany runtime was configured to load, invoke, and manage the new extension through the Tuscany extension point mechanism. An extension point is the place where the Tuscany runtime collects the information required for handling an extension. Specifically, we had to do the following: (i) define how the extension can be used and configured in an SCA composite (assembly) file, by defining an XML schema `implementation-asm.xsd` that defines the XML syntax for the extension `implementation.asm` of the SCA implementation type[6] – `XML schema validation extension point`; (ii) define how to create an Java model that represents the in-memory version of the configured ASM extension by providing the code for a processor (the Java class `ASMImplementationProcessor` in Fig. 5) that knows how to transform the XML representation in the composite file into an in-memory Java model and vice versa – `XML processor extension point`; (iii) enable the Tuscany runtime to invoke and manage the ASM extension by adding the code, the *ASM extension provider*[7], that the Tuscany runtime will use to locate, invoke, and manage the extension at runtime – `Provider factory extension point`.

The ASM extension provider delegates the handling of the ASM component implementation to AsmetaS. To this purpose, the Tuscany runtime calls the `ASMImplementationProviderFactory` (see Fig. 5) to create an instance of the `ASMImplementationProvider` for each component implementation in-

[6] For example, `implementation.asm` adds the `location` attribute for the pathname of the ASM file (an AsmetaL file) that implements the underlying component.

[7] The Tuscany core delegates the start/stop of component implementation instances and related resources, and the service/reference invocations, to specific *implementation providers* that typically respond to these life-cycle events.

stance. The `ASMImplementationProvider`'s `start()` method is invoked to set up the implementation instance when the component is started. Tuscany also calls the `ASMImplementationProvider`'s `createInvoker()` method to create an `ASMInvoker` for each service operation and add it to the incoming invocation chain. When the request comes in, the `ASMImplementationInvoker` will be called to dispatch the request to the ASM instance and get the response back to the caller. When the component is stopped, the `ASMImplementationProvider`'s `stop()` method is triggered to clean up the resources associated with the implementation instance. An SCA service can offer multiple operations. Below, we describe the mechanism (technology dependent) adopted by the invoker specific to ASM (the `ASMInvoker`) for dispatching (through the `invoke` method) requests to the appropriate operation.

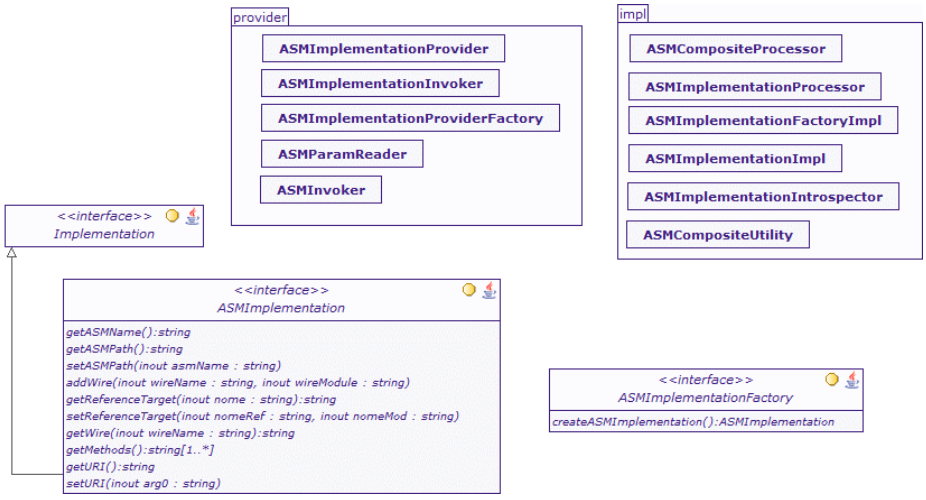

Fig. 5. The classes that define the `implementation.asm` extension

In-place ASM execution mechanism. Fig. 6 illustrates how the ASM implementation provider sets up the environment (the ASM container) within Tuscany for instantiating and handle incoming/outgoing service requests to/from an ASM component implementation instance (like component `A` in the figure) by instrumenting the ASM simulator AsmetaS. Currently, the implementation scope of an SCA-ASM component is *composite*, i.e. a single component instance – a single *main ASM* instance (see the main ASM for component `A` in Fig. 6) – is created within AsmetaS for all service calls of the component. This main ASM is automatically created during the setting up of the connections and it is responsible for instantiating the component agent and related resources, and for listening for service requests incoming from the protocol layer and forward them to the component'agent instance (see component `A` in Fig. 6). Executing an ASM component implementation means executing its main ASM.

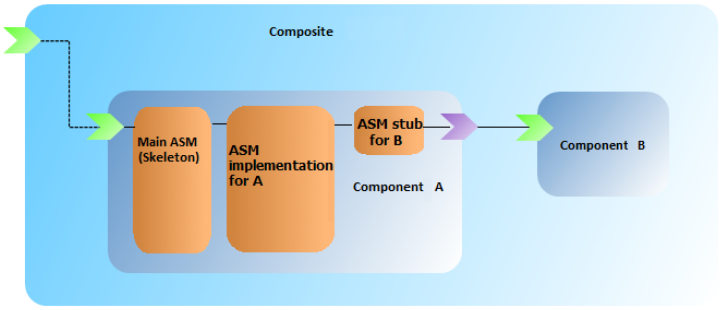

Fig. 6. Instantiating and invoking ASM implementation instances within Tuscany

For each reference, another entity (i.e. another ASM module) is automatically created (and instantiated as ASM agent within the main ASM of the component) as "proxy" for a remote component (see the ASM proxy for component `B` in Fig. 6) for making an outbound service call from the component. Using a terminology adopted in the Java Remote Method Invocation (RMI) API, this proxy ASM plays the role of *stub* to forward a service invocation (and their associated arguments) to an external component's agent, and to send back (through the ASM rule `r_replay`) the result (if any) to the invoker component'agent (the agent of the component `A` in Fig. 6). The main ASM, instead, plays the role of *skeleton*, i.e. a proxy for a remote entity that runs on the provider and forward (through the ASM rule `r_sendreceive`) client's remote service requests (and their associated arguments) to the appropriate component's agent (usually the main agent of the component), and then the result (if any) of the invoked service is returned to the client component'agent (via stubs).

When an ASM implementation component is instantiated, the Tuscany runtime also creates a value for each (if any) externally settable property (i.e. ASM monitored functions, or shared functions when promoted as a composite property, annotated with `@Property`). Such values or proxies are then injected into the component implementation instance. A data binding mechanism also guarantees a matching between ASM data types and Java data types, including structured data, since we assume the Java interface as IDL for SCA interfaces.

4 Related work

Some visual notations for service modeling exist, such as the OMG SoaML UML profile [17] and the UML4SOA [20] defined within the EU project SENSORIA [23]. SoaML, like SCA, is focused on architectural aspects of services. UML4SOA is more focused on modeling service orchestrations as an extension of UML2 activity diagrams. In order to make UML4SOA models executable, code generators towards low-level orchestration languages (such as BPEL/WSDL, Jolie, and Java) were developed [19]; however these target languages do not provide the same preciseness of a formal method necessary for early design and analysis.

Some works devoted to provide software developers with formal methods and techniques tailored to the service domain also exist for the service composition problem), mostly developed within the EU projects SENSORIA [23] and S-Cube [16]. Several process calculi for the specification of SOA systems have been designed (see, e.g., [12, 13]). They provide linguistic primitives supported by mathematical semantics, and verification techniques for qualitative and quantitative properties [23]. Still within the SENSORIA project, a declarative modeling language for service-oriented systems, named SRML [24], has been developed. Compared to the formal notations mentioned above, the ASM method has the advantage to be executable. In [14], the analysis tool *Wombat* for SCA is presented; the tool is used for simulation and verification tasks by transforming SCA modules into composed Petri nets. There is not proven evidence, however, that this methodology scales effectively to large systems.

An abstract service-oriented component model, named *Kmelia*, is formally defined in [1, 3] and is supported by a prototype tool (COSTO). Our proposal is similar to the Kmelia approach; however, we have the advantage of having integrated our SCA-ASM component model and the ASM-related tools with an SCA runtime platform for a practical use and an easy adoption by developers.

Within the ASM community, the ASMs have been used for the purpose of formalizing business process notations and middleware technologies related to web services, such as [9, 6] to name a few. Some of these previous formalization efforts, as explained in [21], are at the basis of our work.

5 Conclusion and future directions

We presented a framework for service design and prototyping that combines the SCA open standard model for service assembly and the ASM formal support to assemble service-oriented components as well as intra- and inter- service behavior. The framework is supported by a tool that exploits the SCA runtime Tuscany and the toolset ASMETA for model execution and functional analysis.

We plan to support more useful SCA concepts, such as the SCA *callback interface* for bidirectional services. We want also to enrich the SCA-ASM notation with interaction and workflow patterns based on the BPMN specification and with specific actions to support an *event-based style of interaction*. We also plan to extend the language with pre/post-conditions defined on services (transition rules) for contract correctness checking in component assemblies.

References

1. P. André, G. Ardourel, and C. Attiogbé. Composing components with shared services in the kmelia model. In C. Pautasso and É. Tanter, editors, *Software Composition*, volume 4954 of *LNCS*, pages 125–140. Springer, 2008.
2. The ASMETA toolset website. `http://asmeta.sf.net/`, 2006.
3. C. Attiogbé, P. André, and G. Ardourel. Checking component composability. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2006.

4. F. Banti, A. Lapadula, R. Pugliese, and F. Tiezzi. Specification and Analysis of SOC Systems Using COWS: A Finance Case Study. *Electr. Notes Theor. Comput. Sci.*, 235:71–105, 2009.
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. E. Börger. Modeling Workflow Patterns from First Principles. In *Proc. of the 26th Int. Conf. on Conceptual Modeling - ER 2007*, pages 1–20, 2007.
7. E. Börger, O. Sörensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *J. UCS*, 15(1):3–32, 2009.
8. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
9. E. Brger, O. Srensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *Journal of Universal Computer Science*, 15(1):3–32, 2009.
10. EU project BRICS (Best Practice in Robotics), `www.best-of-robotics.org/`.
11. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`, 2008.
12. 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.
13. I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM'07*, pages 305–314. IEEE, 2007.
14. A. Martens and S. Moser. Diagnosing SCA Components Using Wombat. In *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proc.*, pages 378–388, 2006.
15. EU project BRICS, Tech. Rep. A Coordination Use Case. March 24, 2011. `www.best-of-robotics.org/wiki/images/e/e0/coordinationusecaseubergamo.pdf`.
16. EU project S-Cube `http://www.s-cube-network.eu/`.
17. OMG. Service oriented architecture Modeling Language (SoaML), ptc/2009-04-01, april 2009 `http://www.omg.org/spec/soaml/1.0/beta1/`.
18. Service Component Architecture (SCA) `www.osoa.org`.
19. P. Mayer, A. Schroeder, and N. Koch. A model-driven approach to service orchestration. In *IEEE SCC (2)*, pages 533–536. IEEE, 2008.
20. P. Mayer, A. Schroeder, N. Koch, and A. Knapp. The UML4SOA Profile. In *Technical Report, LMU Muenchen*, 2009.
21. E. Riccobene and P. Scandurra. A modeling and executable language for designing and prototyping service-oriented applications. In *EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA 2011)*.
22. SCA Tools. `http://eclipse.org/stp/sca/`.
23. EU project SENSORIA, `www.sensoria-ist.eu/`.
24. SRML: A Service Modeling Language. `http://www.cs.le.ac.uk/srml/`, 2009.
25. Apache Tuscany. `http://tuscany.apache.org/`.