DOI 10.1007/s00165-013-0289-0
BCS © 2013
Formal Aspects of Computing (2014) 26: 1077–1113

Formal Aspects of Computing

A formal framework for service modeling and prototyping

Elvinia Riccobene[1] and Patrizia Scandurra[2]

[1] Computer Science Department, Università degli Studi di Milano, via Bramante 65, 26013 Crema, CR, Italy
[2] Engineering Department, Università degli Studi di Bergamo, Dalmine, BG, Italy

Abstract. *Service-oriented Computing* is rapidly gaining importance across several application domains due to its capability of composing autonomous and loosely-coupled services. In order to support the engineering of service-oriented software applications, foundational theories, service modeling notations, evaluation techniques fully integrated in a pragmatic software engineering approach are required. This article introduces a framework for modeling and prototyping service-oriented applications. The framework consists of a precise and executable language, *SCA-ASM*, for model-based design, and of a tool for early and quick design evaluation of service assemblies. The language combines the OASIS/OSOA standard *Service Component Architecture* (SCA) capability of modeling and assembling heterogeneous service-oriented components in a technology agnostic way, with the rigor of the *Abstract State Machine* (ASM) formal method able to model notions of service behavior, interactions, orchestration, compensation and context-awareness in an abstract but executable way. The tool is based on existing execution environments for ASM models and SCA applications. An SCA-ASM model of a service-oriented component, possibly not yet implemented in code or available as off-the-shelf, can be (i) simulated and evaluated *offline*, i.e. in isolation from the other components; or (ii) executed as *abstract implementation* (or *prototype*) together with the other components implementations according to the chosen SCA assembly. As proof of concept, a case study taken from EU research projects has been considered to show the functionalities and potentialities of the proposed framework.

Keywords: Service formal modeling; Service model prototyping; Service Component Architecture; Abstract State Machines

1. Introduction

Service-oriented Computing (*SoC*) is a paradigm for distributed computing based on the unification principle that "everything is a service". Differently from component-based systems, when creating and dynamically composing applications, SoC emphasizes the *functionality*, expressed in terms of services, rather than the *structural entities* or components. Therefore, services are intended as loosely coupled autonomous and heterogeneous[1] components that are available in a distributed environment and that can be published, discovered, and composed (or orchestrated) via standard interface languages, publish/discovery protocols and composition (orchestration) languages. *Service-oriented architecture (SOA)* is a notable set of principles and methodologies for designing and developing software in the form of interoperable services, and Web Services [ACKM04] is the most popular example of service-oriented technology. Moreover, the emerging *Cloud computing* paradigm

Correspondence and offprint requests to: E. Riccobene, E-mail: elvinia.riccobene@unimi.it

[1] Services are in general, heterogeneous, i.e. they differ in their implementation/middleware technology.

[VRMCL08] for hosting Internet-based services in virtualized environments, can be seen as an evolution of the SoC with the goal of facilitating the creation of innovative Internet scale services without worrying about the computational infrastructure needed to support them.

To support the engineering of software systems in the SoC domain, foundational theories, modeling notations, evaluation techniques fully integrated in a pragmatic software engineering approach are required.

This article addresses the problem of model-based designing and prototyping service oriented applications in an assembly-oriented manner, i.e., by assembling service-oriented components already available at code level together with new created ones. For this purpose, we present a framework which integrates an high level formal but intuitive modeling language and simulation environments. It makes possible the specification and the simulation of an entire service-oriented application.

We complement the *Service Component Architecture* (SCA) [SCAa]—an open OASIS/OSOA standard model for heterogeneous service assembly—with an executable formalism based on the *Abstract State Machines* (ASM) [BS03] formal method able to model notions of service interaction, orchestration, compensation, context awareness and the service internal behavior. The result is a *formal* and *executable* language, called *SCA-ASM*, intended for the specification—and potentially for functional analysis (validation and verification)—of service-oriented applications 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. In the SCA-ASM language, SCA design primitives provide graphical representation of components structure and of components assemblies, while the ASM formalism allows formal specification of *intra*- and *inter*-behavioral aspects of services. SCA-ASM models of services are also machine-processable: their XML-based representation makes models processable by an SCA-compliant run-time platform, as well as by the ASM toolset ASMETA (ASM mETA-modeling) [Asm11, AGRS11] for functional analysis.

We developed an SCA-ASM design environment by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany [Tus], and the simulator AsmetaS/ASMETA [GRS08] for ASM models. For evaluation purposes, the combined use of the simulator AsmetaS with the Tuscany makes possible the assessment, at the moment only in terms of function validation, of a single service-oriented component, either in isolation from the other components or in an integrated manner with the other ones. Indeed, an SCA-ASM model of a single service-oriented component (or even of the entire application) can be simulated *offline* (i.e., in isolation) by means of the AsmetaS simulator, and even analyzed as an ASM specification by using the wide range of tools supported by the ASMETA framework for ASMs. In addition, for an early and quick design evaluation of the entire application, SCA-ASM models of service-oriented components (possibly not yet implemented in code or available as off-the-shelf) can be configured *in place* within the Tuscany platform as *abstract implementation* (or *prototypes*) of those components. They can be then executed *in-place*, together with the other components implementations, possibly available at different level of abstraction, according to the chosen SCA assembly. This allows the designer to execute integrated applications and evaluate different design solutions even when the implementation of some components—*abstract* or mock components[2]—is not yet available, but an abstract model, in terms of ASMs, is available as a running prototype.

We here mainly focus on presenting the SCA-ASM language and the supporting tool for application prototyping. This is an improved, from the formal point of view, and extended version, either in terms of application results and in terms of context-awareness, of our previous work in [RS10a, RS10b, RSA11b, BGRS11]. The work in [RS10a] was the first attempt to provide an ASM formal definition of the UML4SOA communication primitives by defining specific wrappers of the high-level communication patterns presented in [BB05]. However, these wrappers leaved aside the model of the communicator, which, instead, has been made here explicit and has implied a refinement of the wrappers defined in [RS10a]. [RS10b] presented a first approach to formalize structural constructs of SCA by using ASMs. This formalization left, however, abstract the communication problem by referring it to the use of the service interaction patterns in [BB05]. A first definition of the SCA-ASM modeling language can be found in [RSA11b]. This work did not deal, however, with the problem of context awareness and the semantics of the language was only sketched in natural language. The contribution in [BGRS11] was mainly devoted to show the effectiveness of our framework in the context of service-oriented robots.

[2] Mock components are simulated components that mimic the behavior of real components in controlled ways. A designer typically creates mock components to validate the behavior of some other components or of the entire integrated application.

We achieved the goal of developing a coordination model for robotic activities, as strongly required in this domain of applications [BS10], by modeling and prototype a Robotics Task Coordination, a case study coming from the EU project BRICS [BRI]. The work in [RSA11a], mainly focused on showing the development of a framework based on the Eclipse environment, presented the integration of the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany, and the Eclipse-based simulator ASMETA/AsmetaS for ASM models.

We here give a complete, consistent and integrated view of this piece of research, filling those parts left abstract and incomplete in other previous contributions. We present the complete syntax of the SCA-ASM constructs (*i*) for describing service interfaces, components and assemblies; (*ii*) for modeling service coordination, communication and internal computation; (*iii*) for handling service faults and compensation; (*iv*) for expressing service context awareness. We provide a complete formal semantics of the language in terms of ASMs. We present how the language has been implemented as a new SCA component implementation type. Illustrating results of model formal analysis is out of the scope of this paper.

The remainder of this article is organized as follows. Section 2 describes some related work along this direction. Section 3 sketches some basic notions concerning the SCA standard and the ASM formal method. The SCA-ASM language is presented in Sect. 4, while the SCA-ASM formal semantics is given in Sect. 5. The supporting prototyping tool is presented in Sect. 6. We here also draw some lesson learned from the tool development and its application on tackled case studies. Finally, Sect. 7 concludes the paper and outlines some future directions of our work.

2. Related work

Some lightweight visual notations for service modeling have been proposed, such as the OMG SoaML UML profile [matb]. SoaML, like the SCA initiative, is more focused on architectural aspects of services and relies on the standard UML 2 activity diagrams for behavioral aspects without further specialization. There are also some other popular approaches for service modeling that have been proposed by commercial modeling tools, including the Enterprise Architect Service-oriented Modeling Framework (SOMF) [SOM] and IBM SOMA [BLJM08]. They cover a broader scope by including concerns related to SOA and to cloud computing environments. UML4SOA [MSKK09] is another UML extension defined within the EU project SENSORIA [SENa]. UML4SOA is focused on modeling service orchestrations as an extension of UML2 activity diagrams. In order to make UML4SOA models executable, code generators for low-level target orchestration languages (such as BPEL/WSDL, Jolie, and Java) have been developed [MSK08]; however, these target languages are used in circumscribed application domains, and they do not have the same semantic rigor and abstraction mechanisms, necessary for early design and analysis, of a formal method. Indeed, the fact that a modeling framework is equipped with a formal semantics makes it possible to support the analysis of services, service compositions and activities.

Some works devoted to provide software developers with formal methods and techniques tailored to the SoC domain also exist (see, e.g., the survey in [BBG07] for the service composition problem), mostly developed within the SENSORIA and S-Cube [S-c] EU projects. Several process calculi for the specification of SOA systems have been designed (see, e.g., [LPT07, GLG+06, LMVR07, BBNL08, Bru09]). They provide linguistic primitives supported by mathematical semantics, and verification techniques for qualitative and quantitative properties [SENb]. In particular, in [BLPT09] an encoding of UML4SOA in COWS (Calculus for the Orchestration of Web Services), a recently proposed process calculus for specifying services and their dynamic behavior, is presented. Still within the SENSORIA project, a declarative modeling language for service-oriented applications, named SRML [FLBA11], has been developed. A formal computation and coordination model was developed for SRML over which SRML supports qualitative and quantitative analysis techniques using the UMC model checker [AMFG09] and the PEPA stochastic analyzer [PEP]. An algebraic semantics [FLB11] was also developed for the run-time discovery, selection and binding mechanisms. Similarly to SCA-ASM, SRML borrowed concepts and notations from SCA and aims for a modeling framework supported by a formal semantics in which business activities and services can be defined in a way that is independent of the languages and technologies used for programming and deploying the components that will execute them. Compared to all the formal notations mentioned above, SCA-ASM adopts the ASM method that has the advantage to be executable and formal without mathematical overkill. In general, formal methods providing executable specifications are particularly useful when validation techniques are applied at very high level of the system development before applying more complex and heavy verification techniques.

The architectural approach to SoC in [vdABvH+06] follows SCA very closely. Its purpose is to offer a metamodel that covers service-oriented modelling aspects such as interfaces, wires, processes and data.

Within the ASM community, ASMs have been used in the SoC domain for the purpose of formalizing business process modeling languages and middleware technologies related to web services, such as [BH06, FR05] to model workflow descriptions in the BPEL language [WS-07], and [BB05, BT08, BST09, Bör07, AFL08] for modeling workflow and interaction patterns, and specifically, processes in the standard BPMN [BPM10]. Still concerning to formal approaches used for the formal modeling and verification of web services composition described in BPEL and BPMN, various approaches have been proposed that involve other formal methods such as Petri Nets [HSS05, vdAMSW09, Ver05], Event-B [AAS12, LGK+11] and the LOTOS process algebra [SBS04], to name a few. A comparison of such works is out of the scope of this article. See [tBBG07] for a survey and a comparison of some of these approaches. Though the works mentioned above focus mainly on formal modeling and verification of web services and services compositions described by the standards BPEL and BPMN, some of these previous formalization efforts, especially the work in [BB05] and in [Bör07], are at the basis of our framework and helped us to make decisions about semantic issues.

On the formalization of the SCA component model, some previous works, like [DCL08, DLC08] to name a few, exist. However, they do not rely on a practical and executable formal method like ASMs. In [MM06], an analysis tool, *Wombat*, for SCA applications is presented; this approach is similar to our as the tool is used for simulation and verification tasks by transforming SCA modules into composed Petri nets. There is no proven evidence, however, that this methodology scales effectively to large systems.

An abstract service-oriented component model, named *Kmelia*, is formally defined in [AAA06, AAA08] and is supported by a prototype tool COSTO. In the Kmelia model a component has an interface made of provided services and required services. Services are used as composition units and serviced behaviour are captured with labelled transition systems. Kmelia makes it possible to specify abstract components, to compose them and to check various properties. Our proposal is similar to the Kmelia approach; however, we have the advantage of having integrated our SCA-ASM component model with an SCA runtime platform for a more practical use and an easier adoption by developers.

3. Background concepts on SCA and ASMs

3.1. Service Component Architecture (SCA)

The OASIS/OSOA standard SCA [SCAa] is an XML-based component model used to develop service-oriented applications independently from SOA platforms and middleware programming APIs (like Java, C++, Spring, PHP, BPEL, Web services, etc.). SCA is also supported by a visual notation (a metamodel-based language developed with the Eclipse-EMF) and runtime environments (like Apache Tuscany, FRAscaTI, IBM WebSphere Application Server V7, to name a few) to create service components, assemble them into a composite application, provide an implementation for them, and then run/debug the resulting composite application.

According to the principles of service-oriented computing, loosely coupled service components are used as atomic units or building blocks to build an application. Figure 1 shows an *SCA composite* (or *assembly*) as a collection of SCA components. 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 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 operations (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. Message exchange may be synchronous or asynchronous.

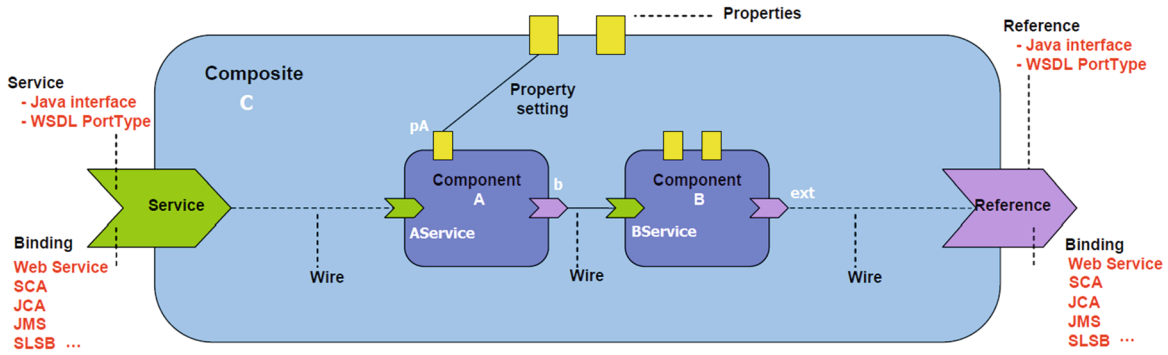

Fig. 1. An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)

As unit of composition and hierarchical design, an assembly of components deployed together is called *composite component* (or simply *composite*). A composite may expose properties, services organized as sub-components, required services as references, and contain suitable *wires* between sub-components and between the composite component itself and sub-components to connect services and references.

A wire is a connector that passes control and data from a component to a target. Specifically, components can be wired together in a composite in two ways: by *promotion* and *by target*. By promotion, a composite service is wired by promoting a component service that is defined on one of the sub-components of the composite. Similarly, a composite reference can promote a sub-component reference, and a composite property can promote a sub-component property. See, for example, the promotion wires in Fig. 1 between components `C` and `A` for the service `AService` and the property `pA`, and the promotion wire between `B` and `C` for the reference `ext`. If a binding type is added to a service or reference, the binding attributes on a promoted composite service or reference take precedence over attributes on the component service or reference. It is not possible to promote an internal service if the external service is using a different type of binding (e.g., if an external service has a JMS binding, the internal service cannot have a WSDL service binding).

By target, a reference of a component (atomic or composite) is wired directly to a service of another component. See, for example, the wire in Fig. 1 between components `A` and `B`. Both the wired reference and the service must be typed by the same interface. A service cannot be wired to another service, and a reference/property cannot be wired to a reference/property.

A composite component can be also used as a complete component implementation within other composites, allowing for a hierarchical construction of business solutions, where high-level services are implemented internally by sets of lower-level services. A *top level composite* describes the overall assembly of a service-oriented application. Composites can therefore group and wire components built from different implementation technologies, allowing appropriate technologies to be used for each business task and appropriate binding attributes to services or references.

3.2. Abstract State Machines (ASMs)

ASMs are an extension of FSMs [BS03] where unstructured control states are replaced by states comprising arbitrary complex data, and transitions are expressed by rules describing how data change from one state to the next.

A complete presentation of the ASMs can be found in [BS03]. We here report only those essential concepts that are useful to understand the formal bases of the SCA-ASM language, either in terms of syntax and in terms of semantics.

Definition 1 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.

Definition 2 A *location* of a state s of an ASM is a pair $(f, (a_1 \dots a_n))$, where f is an n-ary function name and $a_1 \dots a_n$ are elements of the base set (or superuniverse) of s. The value $f(a_1, \dots, a_n)$ is called the *content* of the location in s.

Definition 3 Location-value pairs (l, v) are called (function) *updates* and represent the basic units of state change.

The meaning of an update (l, v) is that the content of a location l in the state has to be changed to a value v. Function values change from one state to the next by effect of the execution of *transition rules*.

Definition 4 A transition rule has the basic form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ and *Condition* a first order formula. *Updates* are simultaneously executed when *Condition* is true.

To fire this rule in 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.

Due to the parallelism of updates execution, we require such updates to be consistent.

Definition 5 An update set U is *consistent*, if it contains no pair of updates with the same location, i.e., no two elements (l, v), (l, w) with $v = w$.

Two updates clash, if they are are not consistent.

Functions changing as a consequence of *updates* are *dynamic* and they are further classified in: *monitored* (only read, as events provided by the environment), *controlled* (read and written by the machine), *shared* (read and written by the machine and by the environment) and *out* (only written by the machine) functions.

There is a finite set of *rule constructors* to model 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`).

Based on [BS03], an ASM has a working definition as follows:

Definition 6 An ASM is the tuple (*header*, *body*, *main rule*, *initialization*), where

header is the sequence (*name*, *import*, *export*, *signature*), being *name* the ASM denomination, *import* and *export* (optional) clauses specifying domains, functions and rules imported/exported from/to other ASMs/modules (see Definition 7), if any; *signature* the declarations of all domains, functions, predicates of the ASM.

body is the sequence (*domain_defs*, *function_defs*, *rules*, *invariants*), being *domain/function_defs* (static) domain and (static/derived) function definitions according to domain and function declarations in the signature of the ASM, *rules* declarations and definitions of transition rules, *invariants* may be present to specify constraints over domains and functions of the ASM.

main rule is a transition rule and represents the (unique) starting point of the machine program, i.e., it calls all the other ASM rules defined in the body. The main rule is *closed*, i.e., it does not have parameters.

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

Definition 7 An ASM with no main rule and no initialization is called *module*.

Definition 8 A *move* of an ASM from a state S_{i-1} to the state S_i is a single computation step executed by the machine. It consists into firing the updates produced by the main rule of the machine, if they do not clash.

Due to the closure of the main rule and to the absence of free global variables in the rule declarations of an ASM, the notion of a move does not depend on variable assignment, but on the machine state.

Definition 9 A *run* 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 the main rule.

Because of the non-determinism of the choose rule and of moves of the environment, an ASM can have several different runs starting from the same initial state.

Distributed computation can be modeled by *multi-agent ASMs* where multiple agents interact in parallel in a synchronous/asynchronous way.

Definition 10 A *multi-agent ASM* is given by a family of pairs $(a, ASM(a))$ of pairwise different agents, elements of a possibly dynamic finite set *Agent*, each executing its own (possibly the same but differently instantiated) machine $ASM(a)$ specifying the agent's behavior.

A predefined function *program* on *Agent* indicates the ASM associated to an agent, i.e, a (un/)named transition rule working as the agent main rule; it is used to dynamically associate behavior to agents. Within transition rules, each agent can identify itself by means of a special 0-ary function $self : Agent$ which is interpreted by each agent a as itself.

In a *synchronous multi-agent ASM*, the set of agents execute their own ASMs in parallel, synchronized using an implicit global system clock. In case of *asynchronous multi-agent ASM*, each agent reacts at its own speed without any global clock. Therefore, the runs of a multi-agent ASM differ in case of synchronous or asynchronous behavior.

Definition 11 A multi-agent ASM with synchronous agents has *quasi-sequential runs*, namely a sequence of states where each state is obtained from the previous state by firing in parallel the rules of all agents.

Definition 12 A multi-agent ASM with asynchronous agents has *partially ordered runs*, namely a partially ordered set $(M, <)$ of moves m (read: rule applications) of its agents satisfying the following conditions: (a) *finite history*: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' \mid m' < m\}$ is finite; (b) *sequentiality of agents*: the set of moves $\{m \mid m \in M, a \text{ performs } m\}$ of every agent $a \in Agent$ is linearly ordered by $<$; (c) *coherence*: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$—think of it as the result of all moves in X with m executed before m' if $m < m'$—which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

Besides ASMs comes with a rigorous mathematical foundation, 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. An open framework, the *ASMETA toolset* [Asm11, GRS08, AGRS11], is also available for editing, exchanging, simulating, testing, and model checking models. It is based on the Eclipse Modeling Framework (EMF) [EMF08]. *AsmetaL* is the textual notation to write ASM models within the ASMETA toolset. *AsmetaS* is the simulator of ASMs. It supports the synchronous computation of multi-agent ASM, and this feature has been exploited to develop the SCA Tuscany implementation type (see Sect. 6.1).

4. The SCA-ASM modeling language

The SCA-ASM modeling language complements the SCA component model with the "model of computation" of the ASM formal method. The aim is to define and implement a new *SCA component implementation type* (see Sect. 6) to provide ASM-based formal stateful and executable descriptions of the services *internal behavior*, *orchestration* and *interaction*. The SCA-ASM component implementation type will result in a distributed multi agent ASM where a service-oriented component is an ASM endowed with (at least[3]) one agent (a business partner or role) able to interact with other agents by providing and requiring services to/from other service-oriented components' agents.

This section defines an SCA-ASM component (Sect. 4.2) and its exposed interface (Sect. 4.1) as ASM modules that are assigned at runtime to an active ASM agent to perform the services the component provides. The language primitives (Sect. 4.4) to express service internal computation, interaction and orchestration are introduced, as well as those for fault and compensation handling. The mechanism for context awareness is presented in Sect. 4.5. The language semantics and the model of computation is given in Sect. 5.

4.1. Interface description

An *interface* is a collection of business functions. It types services (as provided interface) and references (as required interface) of a component (see next subsection). As *interface definition language* (IDL), SCA-ASM exploits the ASM notion of module, and module signature, in particular, for declaring domains and functions symbols characterizing an ASM state.

[3] In Sect. 5 we make precise the collection of predefined ASM agents running around an SCA-ASM component.

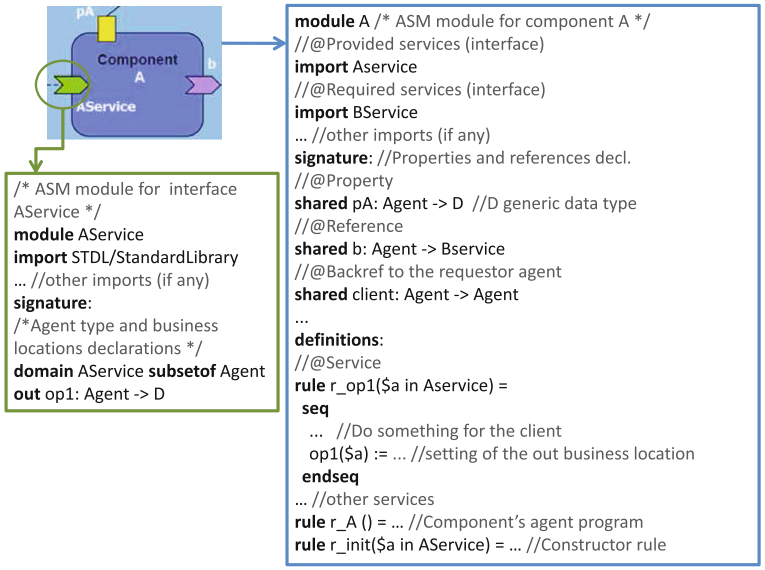

Fig. 2. SCA-ASM component shape

Definition 13 An *interface* of an SCA-ASM component is an ASM module containing *only* the header part (*name*, *import*, *export*, *signature*)

where:

name is the interface name;

import specifies module libraries to be imported;

export denotes signature symbols to be exported;

signature consists of the sequence (*bus_agent_types_decl*, *bus_functions_decl*) and is a collection of declarations of business agent types, declared in terms of subdomains of the predefined ASM domain `Agent`, and of business functions declared as parameterized ASM *out* functions;

As an example of SCA-ASM interface, see the ASM module `AService` (on the left) in Fig. 2. If refers to the SCA-ASM component `A` depicted above by using the visual SCA notation. The interface module is expressed using the textual notation ASMETA/AsmetaL for ASM models.

As additional IDL, Java interfaces are also supported for the execution of an SCA-ASM component within an heterogeneous SCA assembly.

4.2. Component description

An SCA-ASM component is an ASM module that may provide interfaces (called services), require interfaces (called references) and expose properties. The services behaviours encapsulated in an SCA-ASM component are captured by ASM transition rules. References and services are connected through wires in an SCA-ASM composite component to configure and assemble components.

Figure 2 shows (on the right) the skeleton (written in ASMETA/AsmetaL[4]) of the component `A`. The `@annotations` are used to denote SCA concepts such as references, properties, etc.. The formal definition of a component follows.

[4] Two grammatical conventions must be recalled: a variable identifier starts with $; a rule identifier begins with "*r_*".

Definition 14 An *SCA-ASM component* is an ASM module (*header*, *body*) characterized as follows:

- for the header part (see Definition 7):

  name is the component name;

  import is the sequence (*prov_services*, *req_services*, *other_import*), being *prov_services* and *req_services* import clauses annotated, respectively, with `@Provided` and `@Required`, to include the ASM modules of the service interfaces provided/required by the component; *other_import* further module libraries to be imported;

  export specifies component elements to be exported (*export** exports all);

  signature is the sequence (*prop_decl*, *ref_decl*, *backref_decl*, *dom_and_funct_decl*) and contains declarations for externally settable *property* values, i.e., ASM monitored functions—or shared functions when promoted as a composite property[5]—annotated with `@Property`, declarations for *references* and *back references*, i.e., ASM shared functions annotated with `@Reference` and `@Backref`, and declarations of other ASM *domains* and *functions* to be used by the component for internal computation only;

- for the body part (see Definition 7):

  domain_defs and *function_defs* are definitions of domains and functions (static concrete-domains and static/derived functions) already declared in the signature;

  rules is the sequence (*services*, *int_rules*, *prog*, *init_rule*, *handlers*), where

  services are definitions of services, i.e., definition of transition rules annotated with `@Service`;

  int_rules are definitions of (utility) transition rules for internal computation;

  prog is the definition of a *transition rule* working as main rule of the ASM component when the "program" is assigned to the component's agent during the component initialization (see Sect. 5.1)—by convention the rule name for *prog* is the same name of the component's module–;

  init_rule is the definition of a transition rule with predefined name `r_init` that is invoked during initialization to set the internal state (controlled functions) of the SCA-ASM component;

  handlers are definitions of transition rules, annotated with `@ExceptionHandler` and `@CompensationHandler`, fired as, respectively, exception and compensation handlers in case of faults;

  invariants are definitions of state invariants (eg., first-order formulas over some functions of the ASM which must hold in every state of the ASM).

Figure 2 shows on the right the ASM module for the component `A`. This module provides definitions for the business functions declared in the imported ASM module `AService` corresponding to the provided interface `AService`. The module `A` also provides declarations for the property `pA`, the reference `b` to an agent `BService` (see Fig. 1), a back reference `client` to the requestor agent, and other functions. The agent domain `AService` declared in the interface module `AService` and the named rule `r_A` (having the same name of the component) characterize the agent associated to the component `A`. Note that, in SCA-ASM, references are represented as shared functions (annotated with `@Reference`) having 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). This domain is declared in the ASM module corresponding to the reference's typing interface; the ASM module corresponding to the component exposing the interface has also to import the ASM module for the interface. Thus, we identify the partner's business role, i.e., the agent type, even if it is not known at design time. Back references to requester agents are modeled as shared functions in the same way by using the annotation `@Backref`, but the agent codomain is the most generic one, i.e., the `Agent` domain.

For computational purposes, for each service provided by a component, it is useful to associate the service interface and its behavior in terms of ASM transition rules. We provide the following

Definition 15 Given a service s provided by a component C, the *service operation* (*so*) of s is the pair (I_s, R_s), where I_s is the ASM module imported by the component C as provided interface and working as the service interface, and R_s is the named ASM transition rule occurring in the module of C to perform s and annotated with `@Service`.

[5] A property value can be supplied directly as the content of the property element or by referencing a property value of the composite which contains the component.

```
module C
import A,B //import of subcomponents
signature:
//Agents of the sub-components
static compA: AService
static compB: Bservice
//@Property
monitored pA: D
//@Backref to the requestor agent
shared client: Agent-> Agent
...
definitions:
   rule r_init = //Initialization (constructor) rule
            par client(compA) := client //wires setting
                b(compA):= compB
                pA(compA):= pA //sub-component's property setting
                program(compA) := r_A()
                program(compA) := r_B() //Agents' program  assignment
                r_init(compA) //sub-components initialization routines
                r_init(compB)
            endpar
```

Fig. 3. SCA-ASM composite shape

Note that, by convention, R_s takes the same name of the out business function declared in I_s. In case of a return value, the body of such a rule must contain, among other things, an update of such out business function; the value of such function denotes the value to be returned to the client. See, e.g., the rule `r_op1` in the ASM module `A` in Fig. 2, and the occurrence within it of the business function `op1`, declared in the module `AService`, on the left-side of an update-rule.

In case of multiple services provided by the same component, i.e., multiple `@Provided` interfaces, one is elected as *main service* by specifying the annotation `@MainService` when importing the corresponding service interface (see Sect. 6.1 for simulation implications).

4.2.1. Assembly (or composite component) description

According to the SCA composite component (see Sect. 3.1), an *SCA-ASM composite component* (or assembly) can contain SCA-ASM components as sub-components, expose services, references and properties, and contain suitable target wires between sub-components to connect services and references together and promotion wires between the composite component itself and sub-components. A composite can also be used as groupings of components which contribute by inclusion into higher-level compositions. A top level composite describes the overall assembly of the application.

A definition similar to the one provided in the previous paragraph can be given for an SCA-ASM composite component. An SCA-ASM composite is essentially an ASM module that embeds (through import clauses) the ASM modules corresponding to its sub-components. In particular, communication links between components, that are denoted in SCA by appropriated *wires* as configured by the SCA composite, are created in the initialization rule of the ASM module 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 initialization rule `r_init` the wires setting, properties setting, agents' program assignment, and initialization of the sub-components.

Note that, 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, RMI 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, the referenced service name, and data.

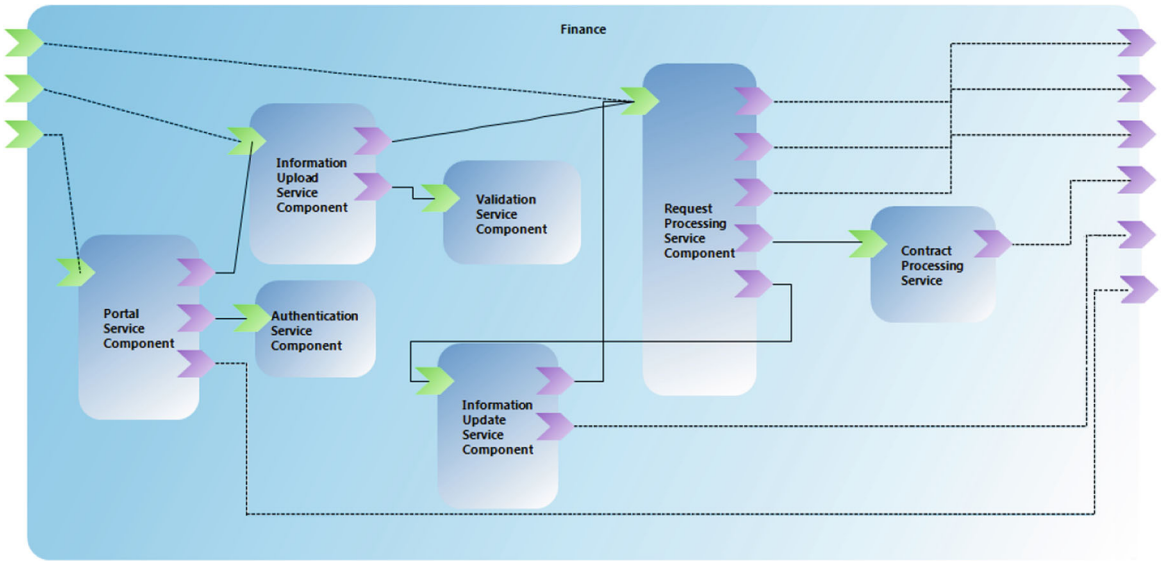

Fig. 4. SCA assembly of the Finance application

4.3. Running example

As running case study, we consider a credit (web) portal application of a credit institute that allows customer companies to ask for a loan to a bank. Figure 4 shows the SCA assembly of the finance application. It consists of the following SCA components: `Portal`, `InformationUpload`, `Authentication`, `Validation`, `InformationUpdate`, `RequestProcessing` and `ContractProcessing`. Actors supervisor, employee and the customer itself that starts the overall scenario appear as external partners (see the promoted services and references of the SCA composite `Finance` in Fig. 4). The considered scenario was taken from [BLPT09] and is related to the orchestration of the necessary steps for processing the credit request, involving a preliminary evaluation by an employee, and subsequent evaluation by a supervisor before a contract proposal is sent to the customer. At any moment the customer may require to abort the process and the system has to rollback the partially executed actions, thus preventing an employee or a supervisor from examining an already aborted request. More details and functional requirements on this scenario can be found in the informal description reported in [BLPT09].

As concrete instantiation of the component `A` in Fig. 2, Listing 1 and Listing 2 report the SCA-ASM specification of the SCA `PortalServiceComponent` (or simply `Portal`) and its provided service interface, respectively, of the SCA `Finance` composite in Fig. 4. Consider the interaction between the customer and the service `Portal` when this last receives a login request from the customer (see the rule `r_PortalServiceComponent`). The customer ID is sent to the `Portal` that invokes the login service (the rule `r_login`). `Portal` synchronously exchanges messages with the service `Authentication`, sending the customer ID and receiving back the boolean `valid`. If `valid` is true, then the service generates a new session ID (by incrementing the current ID number) and sends it back to the customer. If `valid` is false, then the service sends a message back to the customer signaling the failure of the login and it raises the exception `failedLogin` (see the simulation snapshot in Fig. 8 in Sect. 6) that terminates the process as denoted by the status of the Portal'agent set to `EXCEPTION`. `Portal` also receives the customer's choice about the desired service and invokes the service `InformationUpload` by sending it a message with the `requestID`. From then on, the customer communicates with the `InformationUpload`. Here we only consider the service `CREDIT_REQUEST`.

Listing 1: SCA-ASM implementation of the SCA Portal component

```
module PortalServiceComponent
import STDL/StandardLibrary, STDL/CommonBehavior
//@Provided
import PortalService
//@Required
import InformationUploadService, AuthenticationService,
       CustomerService
export *
signature:
//@Property
shared statusWord : Agent -> String
//@Reference
shared authenticationService : Agent -> AuthenticationService
//@Reference
shared informationUploadService : Agent -> InformationUploadService
//@Reference
shared customerService : Agent -> CustomerService
controlled valid : Boolean
controlled inputPortal : Agent -> Prod(String,String)
controlled inputService : Agent -> Prod(Integer,String)
controlled sessionId : Integer
definitions:
//@ExceptionHandler
rule r_failedLogin($a in Agent) = statusWord(self):= ''Exception_caused_by_Login_failed!''

//@pre statusWord=''''
//@Service
rule r_login($user in String, $pwd in String) =
 seq
  r_sendreceive[authenticationService(self),
                ''r_authentication(Agent,String,String)'',
                ($user,$pwd),valid]
  if (valid)
  then seq
       sessionId:=sessionId+1
       r_send[customerService(self),''r_logged(Agent,String,Integer)'',
              ($user,sessionId)]
        endseq
  else if (not(valid)) //valid can still be undef
       then seq
            r_send[customerService(self),
                   ''r_failedLogin(Agent,String)'',$user]
            r_raiseException[self,''r_login'',''Login_failed!'']
            endseq
 endseq

//@Service
rule r_selectService($sessionId in Integer, $service in String) =
 if ($service=''CREDIT_REQUEST'')
 then r_send[informationUploadService(self),
             ''r_createInst(Agent,Integer)'',$sessionId]

rule r_PortalServiceComponent = //Portal's agent program
 par
  if nextRequest(self)=''r_login(String,String)''
  then seq
       r_receive[customerService(self),
                 ''r_login(String,String)'',inputPortal(self)]
       if isDef(inputPortal(self))
       then r_login[first(inputPortal(self)),second(inputPortal(self))]
       endseq

  if nextRequest(self)=''r_selectService(Integer,String)''
  then seq
       r_receive[customerService(self),
                 ''r_selectService(Integer,String)'',inputService(self)]
       if isDef(inputService(self))
    then r_selectService[first(inputService(self)),
                         second(inputService(self))]
    endseq
 endpar

rule r_init($a in PortalService) = //Constructor rule
par
 sessionId:=0
 status($a):=READY
 exceptionHandler($a,''r_login''):= <<r_failedLogin(Agent)>>
endpar
```

Listing 2: SCA-ASM definition of the `PortalService` interface

```
module PortalService
export *
signature:
domain PortalService subsetof Agent
out login: Prod(Agent,String,String) -> Rule
out selectService : Prod(Agent,Integer,String) -> Rule
```

4.4. Service behavior

ASM rule constructors and predefined ASM rules (i.e., named ASM rules made available as model library) are used as SCA-ASM behavioral primitives. They are here described and exemplified through the running case study by grouping them according to the separation of concerns *computation* and *coordination*, *communication* (or *interaction*), and *fault/compensation handling*.

The formal semantics of the commands corresponding to predefined ASM rules has been precisely defined in terms of ASMs, and it is described in Sect. 5. The corresponding AsmetaL implementation is provided as external library `CommonBehavior` available in [SCAb]. It has to be imported as part of an SCA-ASM implementation.

4.4.1. Service computation and coordination

Service tasks are modeled as named ASM transition rules. Table 1 reports all the language constructs that can be used to model the service internal behavior. Apart the last two rules split and spawn, all the other rules are adopted from the ASM rule constructors [BS03].

An ASM rule invocation R represents the invocation of a service. These services can be orchestrated (or coordinated) in accordance with a workflow expressible by still all the constructs in Table 1:[6]

The running example contains some instances of the rule constructors for computation and coordination—such as, function updates, conditional rules, sequential rules, etc.—in the rule bodies of the `PortalServiceComponent`'s program and of the service operations `r_login` and `r_selectService`.

4.4.2. Service communication (or interaction)

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 an *abstract message-passing* mechanism by adopting the default SCA binding (`binding.sca`) for message delivering. SCA-ASM rule constructors can be combined to model specific interaction and orchestration patterns in well structured and modularized entities.

Services are invoked through the primitives reported in Table 2. These primitives, mainly inspired by the UML profile UML4SOA [MSKK09], correspond to the invocation of predefined ASM rules defined in [RS10a] as "wrappers" of high-level communication patterns, originally presented in [BB05], which model in terms of 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. These communication rules rely on a dynamic domain *Message* that represents messages managed by the abstract message passing mechanism.

Examples of the primitives `receive`, `send` and `sendreceive` are shown in the Listing 1 within the bodies of the `PortalServiceComponent`'s program and of the service operations `r_login` and `r_selectService`.

The language can be easily enriched with additional communication patterns (e.g., for *multi-party interactions* already supported in ASM as specializations of the more abstract patterns formalized in [BB05]). They will be considered for future extension.

[6] These language constructs provide the same expressiveness of the control-flow commands of WS-BPEL [WS-07], leaving out aspects as termination and event handlers within scope activities, synchronization dependencies within flow activities, *wait* activities, which will be considered for future extension. However, our notation has a broader scope: it provides, in an unique formalism, modeling primitives for orchestration, communication and computation aspects.

Table 1. SCA-ASM rule constructors for 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
Conditional rule	**it** ϕ **then** R_1 **else** R_2 if ϕ is true, then execute rule 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
Spawn rule	**spawn** child **with** R create a child agent with program R

Table 2. SCA-ASM rule constructors for Communication

Send rule	**send**[*lnk,R,snd*] sends data *snd* to the partner *lnk* in reference to the service operation *R* (no blocking, no acknowledgment expected)
Receive rule	**receive**[*lnk,R,rcv*] receives data *rcv* from the partner *lnk* in reference to the service operation *R* (blocks until data are received, no acknowledgment expected)
SendReceive rule	**sendreceive**[*lnk,R,snd,rcv*] sends data *snd* to the partner *lnk* in reference to the service operation *R* and waits for data *rcv* to be sent back (no acknowledgment expected)
Reply rule	**replay**[*lnk,R,snd*] returns data *snd* to the partner *lnk* as response of a request of the service *R* received from the same partner *lnk* (no acknowledgment expected)

4.4.3. Fault/compensation handling

Fault and compensation handling are strictly related. They require the execution of specific activities (attempting) to reverse the effects of previously executed activities. The language primitives for fault and compensation handling are reported in Table 3.

The mechanism described here is mainly inspired by the UML4SOA. The behavior of an exception handler for a service operation R is specified by an ASM rule to be executed in case of fault. The annotation `@ExceptionHandler` denotes the rule's role as exception handler. The function *exceptionHandler*(R) is used, within the `init`ialization rule for a given component, to associate a component service operation R_A with its exception handler. To raise an exception when a fault occurs, the predefined rule `raiseException`[a, R, msg] is invoked to put the agent a in status *exception* (see Sect. 5.1 for more explanation), expose a possible error message *msg* (if any), and lunch the rule *exceptionHandler*(R).

Table 3. SCA-ASM rule constructors for Fault/Compensation Handling

Raise rule	**raiseException**[*a, R,msg*] puts the agent *a* in exception mode, exposes a possible error message *msg* (if any), and lunchs the rule *exceptionHandler(R)*
Compensate rule	**compensate**[*a, R*] puts the agent *a* in a compensation mode, and lunches the rule *compensationHandler(R)*
CompensateAll rule	**compensateAll**[*a, R*] puts the agent *a* in a compensation mode, and invokes sequentially all compensation handlers nested in the activity *R* in reverse order of their completion

As exemplification of the exception handling mechanism, consider the rule `failedLogin` in the Listing 2 annotated with `@ExceptionHandler`. It acts as exception handler for the service operation `login`—according to the value of the function `exceptionHandler` in the rule `init`. The exception is raised by executing the rule `raiseException` inside the `Portal`'s service `login` when the boolean value `valid` that the service `Portal` receives back from the service `Authentication` is false. The handler `failedLogin` of such exception sets explicitly the property `statusWord` and terminates—implicitly, as effect of its execution as better described in Sect. 5—the agent processing in an exception state (see the simulation snapshot in Fig. 8 in Sect. 6). A message back to the customer signaling the failure of the login is sent as part of the service `login` before raising the exception.

The mechanism for compensation handling is similar. The annotation `@CompensationHandler` is used to mark a rule acting as compensation handler of a given service operation R. This last is associated with its handler by the function *compensationHandler*(R) settled in the component `init`ialization rule. When a compensation for a service operation R, already completed successfully, must be activated, the predefined rule `compensate`[a, R] is invoked to put the agent a in status *compensation* (see Sect. 5.1 for more explanation) and lunch the rule *compensationHandler*(R).

The predefined rule `compensateAll`[a, R] can be used, instead, to invoke all compensation handlers that are nested in the current service activity R. This rule invokes, in a sequential order, all compensation handlers rules for all service actions inner in the scope of R, in reverse order of their completion. It has the same semantics of the «compensateAll» actions of the UML4SOA.

As an example of a global compensation scenario, Listing 3 shows a fragment of the SCA-ASM component `RequestProcessing` of the Finance running example. In the initialization rule, two compensation handlers and an exception handler are installed. Notably, at any time after the login, the customer can require the cancellation of the credit request processing by invoking the service operation `abortExecution`. This causes the rising of an (internal) fault signal `abortException`. To deal with such a fault, a specific fault handler `abortException` catches the fault and forces the termination of all activities by a `compensateAll` action.

4.5. Context awareness

Context-awareness is a key property enabling service-oriented applications, where services are designed independently by different service providers and are dynamically composed to provide a specific functionality, to cope with the dynamics of the continuously changing environment in which they operate.

According to the general reference framework for change and adaptation in [FGT12], let D be the (descriptive) formal statements specifying the application domain assumptions captured by domain knowledge. The behavior of the environment may diverge from the domain assumptions D made when the specification of the application was devised. A response to these changes is traditionally handled by modifying the application offline during a maintenance phase. An alternative is *adaptive maintenance*, that is, the application has autonomous capabilities through which it tries to self-adapt to satisfy the requirements under the newly discovered domain properties.

Here, we only focus on endowing an SCA-ASM component with the capacity of being sensitive to changes to D, that is the capacity of being "context-aware",[7] Hence, in SCA-ASM, components' agents execution is subject to assumptions on the application *context* (or *environment*). To this purpose, we rely on the framework presented in [FGT12] and, therefore, we partition the relevant set of domain properties D into three disjoint subsets, D_f, D_u, and D_s.

[7] We postpone as future work the extension of the SCA-ASM formalism to support self-adaptive features.

Listing 3: ASM implementation of `RequestProcessingService`

```
module RequestProcessingServiceComponent
...
definitions:
//@Service
rule r_abortExecution($a in Agent, $sessionId in Integer) =
 seq
  sessionId($a):=$sessionId
  r_raiseException[$a,''abortExecution'',''abort_by_user'']
endseq

//@ExceptionHandler
rule r_abortException($a in Agent) =
  r_compensateAll[$a,''requestProcessing'']
...
//@Service
rule r_requestProcessing($a in Agent,$sessionId in Integer, $balance in Real, $amount in Real, $securities in String) =
...

rule r_RequestProcessingServiceComponent =
...
if nextRequest(self)=''r_abortExecution(Agent,Integer)'' then
seq
 r_receive[clientRequestProcessingService(self),''r_abortExecution(Agent,Integer)'',inputAbortExecution(self)]
 if (isDef(inputAbortExecution(self)))
 then r_abortExecution[self,inputAbortExecution(self)]
endseq
...
rule r_init($a in RequestProcessingService) =
par
 status($a):=READY
 compensationHandler($a,''creditRequestEvaluation''):= <<r_compensate_creditRequestEvaluation(Agent)>>
 compensationHandler($a,''requestEvaluation''):= <<r_compensate_requestEvaluation(Agent)>>
 exceptionHandler($a,''abortExecution''):= <<r_abortException(Agent)>>
 ...
endpar
```

D_f is the fixed part: it captures the set of stable assumptions, which will not change later (e.g., it may include the known physical laws that regulate certain environment phenomena). D_u and D_s, instead, capture the assumptions that are likely to change over time. Specifically, D_u denotes the assumptions on *usage profiles*. They consist of properties that characterize how the application being designed is expected to be used by its clients, and may change dynamically. For example, in an e-commerce application, the usage profile may concern the average number of purchase transactions allowed per registered user per day. $D_s = D_{s'} \bigcup D_{s''}$, instead, denotes the set of assumptions on the *external services*, respectively, invoking the application ($D_{s'}$) and invoked by the application ($D_{s''}$).[8] These assumptions must of course be matched by the corresponding specifications of the external services to be composed in the application implementation. These specifications may be viewed as the dependability contracts that constrain service providers through *Service Level Agreement* (SLA).

The application context of an SCA-ASM component's service is defined through a set of *context properties*, each describing a particular aspect of the application domain. Such aspects may be related to the *usage profile* of the component or to assumptions on *external services* invoking/invoked-by the component's service. Moreover, a context property may evolve as an effect of the execution of the application itself, which corresponds to the normal behavior, but also as a result of an action or event coming from outside an application.

Since like services and references, properties are the configurable aspects of an SCA component implementation, a context property related to the usage profile is represented in SCA-ASM as a component's property. A component's property contains a value that can be read by that component when it is instantiated (configured).

[8] D_s has been slightly extended from [FGT12] to include also the external services that invoke the service.

For example, a component might rely on a property to tell it what part of the world it is running in, letting it customize its behavior appropriately. A component implementation may define the property type and a default value, and it also may be able to set values for the property during execution. Context properties related to assumptions on external invoking services are represented by the input parameters values of the service, while those related to external services invoked by the component service are represented by controlled/shared locations of the component used to store the results returned back from the invoked services.

A *context configuration* is a snapshot of the context at a specific time, capturing the current status of all context properties of an SCA assembly.

Each SCA-ASM component may exhibits a *context-aware behavior* relating its execution to the context by annotating its services with *pre-conditions* and *post-conditions*. Pre-conditions constrain a service execution to specific context configurations and are used to catch violations in the expected behavior. In our framework SCA-ASM, they are represented as annotated boolean formulas `@pre` $cond_u$ & $cond_{s'}$ & $cond_{s''}$ over component properties or service parameters or controlled/shared locations, before the annotation `@Service` of a service rule. Consider for instance the `PortalServiceComponent` in Listing 1. A precondition of the `r_login` service is that the property `statusWord` is empty denoting that the previous execution of the `r_login` operation (if any) was successful.

Similarly, post-conditions denote conditions of the application context that should be satisfied after a component service is executed. They are represented as annotated boolean formulas `@post` *cond* over component properties or out business functions, in between the annotations `@pre` and `@Service` of a service rule. They are used by the component to return back the output values of the computed service to the external invoking services.

5. The SCA-ASM language formal semantics

This section presents the operational semantics enabling composition of the SCA-ASM components into an executable SCA application. Sect. 5.1 presents the computational semantics and the life-cycle of an SCA-ASM component. Sect. 5.2 describes the abstract message-based communication infrastructure. Sect. 5.3 gives the formalization of the SCA-ASM behavioral primitives in terms of ASMs rules. These ASM rules are imported as model library (the ASM module named `CommonBehavior`) in each SCA-ASM module and are the result of a prior formalization in ASM of the behavioral semantics of SCA-ASM actions. Finally, Sect. 5.4 provides the operational semantics of context-aware behavior in terms of SCA-ASM rules.

5.1. SCA-ASM computational model

The semantic model we introduce to capture the behavior of an SCA-ASM component is a distributed multi-agent ASMs. At runtime, there is a family of agents cooperating together: (at least) an agent for each SCA-ASM component of the assembly, and a special agent, the *communicator*, which is always alive and operates differently from the others being responsible for managing the communication mechanism. The communicator's behavior is described in Sect. 5.2.

The agent a of an SCA-ASM component `A` has for $ASM(a)$ (or SCA-ASM machine) the `module A` with the module's element *prog* as main rule. At implementation level, this instantiation is guaranteed by the *in-place simulation* mechanism described in Sect. 6.1. Agent a provides the `A`'s services by executing the corresponding service rules of the `module A`.

The concepts of *location*, *move* and *run* of an SCA-ASM component, or SCA-ASM machine, within an SCA assembly are the same as for an ASM in the multi-agent ASM.

On the distributed model of computation, we do not impose any synchronicity among components agents, but we need to impose the constraint about the alternation between a move of an SCA-ASM component's agent and the move of the communicator.[9]

[9] At implementation level, this constraint is guaranteed, as described in Sect. 6.1, by the sequential execution of the communicator and of the component (see Listing 4).

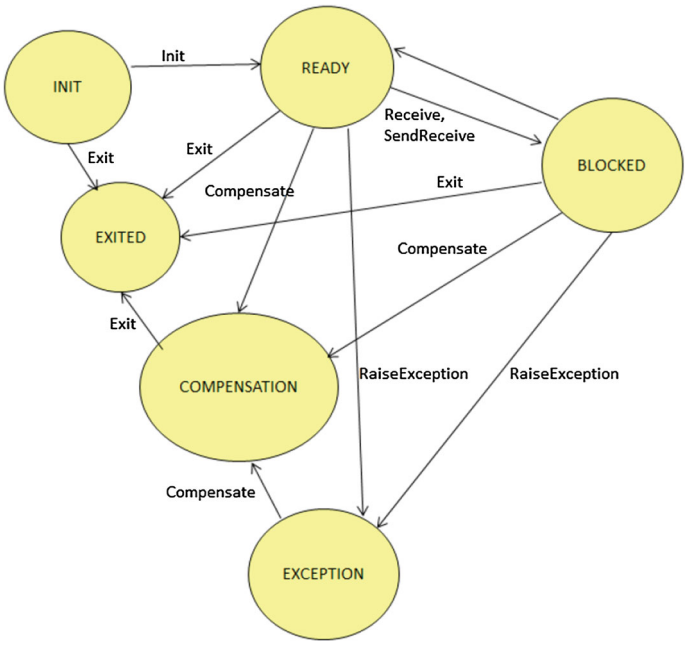

Fig. 5. SCA-ASM component's life cycle

As long as the SCA-ASM machine can make a move, the run proceeds, requiring only that the interspersed moves of the *environment*,[10] namely updating monitored or shared functions (essentially, SCA properties of the component), produce a consistent state for the next machine move. If in a state the machine cannot produce a consistent update set or no update set at all, then the state is considered to be the last state in the run.

Component Life Cycle The component's agent deployed and instantiated in an assembly, follows a simple *life cycle* among the states in $Status = \{INIT, READY, BLOCKED, EXITED, COMPENSATION, EXCEPTION\}$ (see the finite state automaton in Fig. 5). A controlled function $status : Agent \longrightarrow Status$ keeps value of the current state of the agent.

A component's agent has initial status *INIT*. The status becomes *READY* when the component is initialized by executing the rule `r_init` of the component module—the rule initializes the controlled portion (functions and domains) of the SCA-ASM component's signature. A ready agent is available to interact with other service components. It is *BLOCKED* when data are expected upon service invocation, and returns back to *READY* when the interaction with the invoked service terminates. *COMPENSATION* and *EXCEPTION* refer to the agent's modes of compensation (or rollback) and exception. Finally, the status of a component's agent a is set to *EXITED* upon deferred termination through the execution of the action `exit[a]`. After entering in the state *EXITED*, the component is frozen and no longer activated for the execution.

In general, the status of a component's agent is updated by predefined ASM transition rules specifying the dynamic semantics of the SCA-ASM behavioral commands (see next subsection). Therefore, when engaged in service interactions, the life cycle of a component's agent results into a *control-state ASM* [BS03], a class of ASMs that is a natural extension of Finite State Machines. This embedded control-state ASM is used to model the overall status or mode of an SCA-ASM component's agent, guiding the execution of guarded synchronous parallel updates of the underlying component's state. The function *status* describes the different control states (or modes) of such a control-state ASM.

[10] Read: by some other (say an unknown) agent representing the context in which the SCA-ASM machine computes, namely the "container" of the component according to the ASM implementation technology explained in Sect. 6.1.

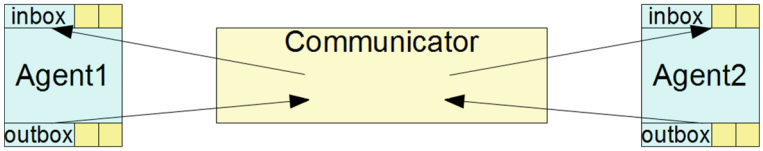

Fig. 6. Abstract message mailboxes

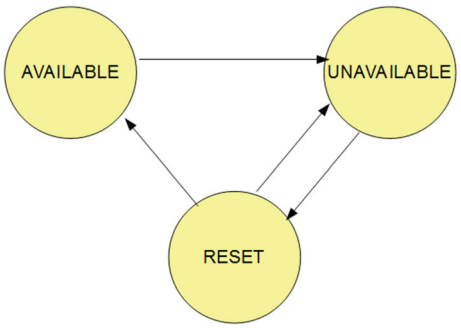

Fig. 7. Communicator FSM

5.2. Communication infrastructure

The communication infrastructure permits to keep the SCA-ASM components separated from communication related problematics. We adopt an abstract communication model. The purpose is not to design a new kind of network, rather, it is to provide a basic infrastructure for a message-based communication in distributed architectures. We implemented it in AsmetaL in the `CommonBehavior` library and tested within the SCA runtime platform Tuscany. Its modularized design permits to implement easily support for new protocols and communication mediums.

The communication infrastructure of an SCA-ASM component is managed by a special ASM agent, called *communicator*, who runs in background to handle incoming connections and internal message routing to the various components.

A dynamic domain *Message* is also introduced to represent message instances. Each message is characterized by dynamic context-dependent information provided by the following dynamic controlled 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.

Moreover, a set of abstract mailboxes (see Fig. 6) are adopted and represented by the dynamic controlled functions:

inbox: Agent → *Seq(Message)*
outbox: Agent → *Seq(Message)*

Essentially, every component's agent has an input mailbox *inbox* containing the received messages and an output mailbox *outbox* containing the messages to deliver.[11] The communicator transfers messages between components. To this purpose it takes the messages to deliver from the agents' outboxes, and forwards them into the inboxes of the respective recipients of the messages afterwards.[12]

The communicator's behavior is modeled as a control-state ASM (see the finite state automaton in Fig. 7) based on the control variable *communicatorStatus* : {*AVAILABLE, UNAVAILABLE, RESET*} denoting the communicator's operation mode.

[11] In general, ASM agents are not required to have mailboxes.
[12] The proposed communication model is ideal. We postpone as future work the definition of a model for a more realistic communication infrastructure where messages can be unexpectedly lost and the communicator agent may contain an internal buffer to temporarily store the messages to deliver.

The initial state of a communicator agent is *RESET*. In the state *RESET*, the communicator sets its status to *AVAILABLE* and in parallel initializes the mailboxes of the assembly components by invoking a predefined rule INIT. In the state *AVAILABLE*, the communicator retrieves and forwards the messages from/to the mailboxes accordingly by invoking the rule MSGDELIVERY, and at the same time monitors the correct functioning of the underlying low-level network by invoking the rule CHECKCHANNEL. The state *UNAVAILABLE* denotes a situation where the communicator cannot deliver messages due to a connection problem of the underlying network; for instance, external actions and events may affect the communication load of the network, or otherwise have an impact on message delays. The state of the communicator is set to *UNAVAILABLE* within the rule CHECKCHANNEL that we no further specify—by defining CHECKCHANNEL ≡ *skip*—since we abstract here from lower-level network layers.

```
rule COMMUNICATORPROGRAM =
if communicatorStatus = RESET
then par
        communicatorStatus := AVAILABLE
        INIT()
     endpar
if communicatorStatus = AVAILABLE
then par
        MSGDELIVERY()
        CHECKCHANNEL()
     endpar
```

The rule INIT initializes all mailboxes to empty as follows:

```
rule INIT() =
forall a ∈ Agent with a ≠ communicator do
     par inbox(a):=[]; outbox(a):= [] endpar
```

The rule MSGDELIVERY is executed by the assembly communicator to retrieve the messages ready to deliver from the outboxes of the sender agents and put them into the inboxes of the recipients. It is defined as follows:

```
rule MSGDELIVERY() =
loop through x ∈ [ a ∈ Agent | outbox(a) ≠ [] ] do
     loop through m ∈ outbox(x) do
          inbox(recipient(m)):=append(inbox(recipient(m)),m)
```

Note that the outboxes of the sender agents are treated and empty sequentially by the constructs **loop through**.[13] The outermost loop iterates over all outboxes of the agents to avoid inconsistent updates of the inbox of a recipient agent in case there are two messages in the outboxes of two different agents with the same recipient.

[13] **The loop-through**-construct semantics is captured by the following ASM rule:

```
loop through x ∈ L do R(x) ≡
    while L ≠ [] do
        let x = first(L) in
            par
                R(x)
                L := tail(L)
            endpar
```

The innermost loop iterates over all messages of an agent's outbox to avoid inconsistent updates of the inbox of a recipient agent in case there are two messages in the agent's outbox with the same recipient. Moreover, note that in such a model conflicts between the communicator and the agents in updating the agents' mailboxes are avoided because it is assumed there is a strict alternation between the execution of the communicator and of the components agents in the main rule of the main ASM. Note that this is different from the abstract communication model in [GGV04]—though we were inspired by such a work—, where several agents may simultaneously insert messages into the mailbox of an agent and conflicts in updating the mailbox are avoided by adopting the ASM algebraic framework of *partial updates* [GT05].

5.3. Formalization of the SCA-ASM behavioral primitives

A detailed description follows on the formal semantics, in terms of ASM transition rules, of the SCA-ASM behavioral constructs.

5.3.1. Service computation and coordination

Computation constructs do not require a special treatment as they correspond to the well-defined ASM rule constructors. So, the semantics of the computation constructs is the same of the ASM rule constructors. A similar consideration also apply to the coordination constructs.

The construct for the spawn of sub-threads as agents deserves a special emphasis. Each of these threads executes a different program and is independent of other threads. Moreover, there is no need to synchronize these threads. The semantics of such a construct can be intuitively expressed in ASM according to the following ASM rule pattern (or schema):

spawn *child* **with** $R \equiv$ **extend** *Agent* **with** *child* **do**
  par
    program(*child*) := *R*
    status(*child*):= *READY*
  endpar

In addition to this form of *asynchronous parallel split* of control flow, further control flow patterns for *merging*, *interleaving*, and *trigger* can be also easily supported (though not yet added to the SCA-ASM) and expressed in terms of ASMs (see [Bör07]).

A further construct is the rule exit for the deferred termination of an agent *a* by setting its status to *exited*:
rule *EXIT*(*a*) = *status*(*a*):= *EXITED*

5.3.2. Service communication (or interaction)

To give the semantics of SCA-ASM primitives for communication, we take advantage of the high-level models for fundamental bilateral service interaction patterns specified by Barros and Boerger in [BB05] in terms of 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 $s \in \mathit{SendType} = \{\mathit{noAck}, \mathit{ackNonBlocking}, \mathit{ackblocking}\} \cup \{\mathit{noAckResend}, \mathit{ackNonBlockingResend}, \mathit{ackBlockingResend}\}$ and $t \in \mathit{ReceiveType} = \{\mathit{blocking}, \mathit{buffer}, \mathit{discard}\} \cup \{\mathit{noAckBlocking}, \mathit{noAckBuffer}, \mathit{ackBlocking}, \mathit{ackBuffer}\}$.

The semantics of the SCA-ASM constructors for communication in Table 2 can be captured by ASM submachines defined as *wrappers* of the turbo rules already formalized in [BB05]. We report below the definition of these wrapper rules, each of which describes one side of the interaction and relies on the dynamic domain *Message*. 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 [BB05]) by two dynamic predicates[14] *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*.

[14] We identify sets with unary predicates.

The adaptation from [BB05] required, as expected, to refine those rules involving the communication model which was left abstract in [BB05]. This is also the reason why the wrappers specification here is an improved version compared to that presented in [RS10a].

The wrapper rule send. The rule **send**[*lnk, R, snd*] is invoked by an agent to send, without blocking, the data *snd* to the partner link *lnk* in reference to the service operation *R*. No acknowledgment is expected.

```
rule send[lnk, R, snd] =
if status(self)=READY then
  extend Message with m do
  seq
      par
          recipient(m):= lnk
          sender(m):= self
          serviceOp(m):= R
          serviceData(m):= snd
          RequestMsg(m):= true
          SendMode(m):= true
      endpar
      SEND_noAck(m)
  endseq
```

The first sequential block of function updates set the message parameters and prepares the work of the communicator. The submachine

```
rule SEND_noAck(m) =
par
    FIRSTSEND(m)
    HANDLESENDFAULT(m)
endpar
```

is a simplified version of the pattern SEND_s in [BB05] with $s = noAck$ to denote a non-blocking action with no ack. The submachine:

```
rule FIRSTSEND(m) = if SendMode(m) and OkSend(m) then BASICSEND(m)
```

makes the first send without further resending. The function *OkSend*(*m*), denoting the existence of a channel connecting the sender to the recipient, which is open to send m, and the rule BASICSEND(m), having the intended interpretation that the message m is sent to *recipient*(*m*), are left abstract in [BB05]. According to the communicator model presented in Sect. 5.2, they are here refined as expected:

$$OkSend(m) \equiv communicatorStatus = AVAILABLE$$

```
rule BASICSEND(m) =
par
    outbox(sender(m)):=append(outbox(sender(m)),m)
    SendMode(m) := false
endpar
```

Possible faults at the sender's side during an attempt to send message *m* are captured by an abstract rule HANDLESENDFAULT(m) typically triggered by a condition **not** *OkSend*(*m*).

```
rule HANDLESENDFAULT(m) = if SendFaultMode(m) then SENDFAULTHANDLER(m)
```

were $SendFaultMode(m) \equiv SendMode(m)$ **and not** *OkSend*(*m*).

We still no further specify the SENDFAULTHANDLER—by defining $\text{SENDFAULTHANDLER}(m) \equiv skip$—since in the communicator model adopted here we abstract from lower-level network problems. In any case, as in [BB05], we assume the firing of this rule is preemptive, namely the guard $SendFaultMode(m)$ automatically becomes false after firing the rule.

The wrapper rule receive. The rule **receive**[*lnk, R, rcv*] is invoked by an agent to receive data in the location *rcv* from the partner link *lnk* in reference to the service operation *R*. The agent blocks until data are received. No acknowledgment is expected.

rule receive[*lnk, R, rcv*] = RECEIVE$_{noAckBlocking}$(*m,rcv*)
where *recipient*(*m*) = *self* **and** *sender*(*m*) = *lnk* **and** *serviceOp*(*m*) = *R*

This wrapper uses a simplified version of the pattern RECEIVE$_t$ in [BB05] with t = *noAckBlocking* to denote a blocking action with no ack.[15]

rule RECEIVE$_{noAckBlocking}$(*m,rcv*) =
  if *Arriving*(*m*) **then** CONSUME(*m,rcv*)
  else *status*(*self*) := *BLOCKED*

The intended interpretation of *Arriving*(*m*) is that m is in the agent's inbox [BB05]. Thus, it yields *Arriving*(*m*) ≡ *m* ∈ *inbox*(*recipient*((*m*))).

The rule CONSUME(m), left abstract in [BB05], is here refined in order to store in the location *rcv* the received data embedded in the received message and delete this last from the inbox of the recipient.

rule CONSUME(*m,rcv*) =
par
  rcv := *serviceData*(*m*)
  status(*recipient*(*m*)) := *READY*
  inbox(*recipient*(*m*)) := *excluding*(*inbox*(*recipient*(*m*)),*m*)
endpar

The wrapper rule replay. The rule **replay**[*lnk, R, snd*] is invoked by an agent to return some data *snd* to the partner link *lnk*, as response of a previous *R* request received from the same partner link; no acknowledgment is expected.

rule replay[*lnk, R, snd*] =
  if *status*(*self*)=*READY* **then**
    choose $m \in$ *Message* **with** (*RequestMsg*(m) **and** *serviceOp*(m) = *R* **and**
    recipient(m) = *self* **and** *responseMsg*(m) = *undef*) **do**
    extend *Message* **with** m' **do**
      seq
        par
          recipient(m') := *lnk*
          sender(m') := *self*
          serviceOp(m') := *R*
          serviceData(m') := *snd*
          ResponseMsg(m') := *true*
          requestMsg(m') := *m*
          SendMode(m') := *true*
        endpar
        responseMsg(m) := m'
        SEND$_{noAck}$(m')
      endseq

[15] The predicate *ReadyToReceive*(*m*), left monitored in [BB05], conditioning the execution on the action CONSUME(m) yields *ReadyToReceive*(*m,lnk,R*$_A$) ≡ *recipient*(*m*) = *self* **and** *sender*(*m*) = *lnk* **and** *serviceOp*(*m*) = R_A.

The wrapper rule sendreceive. The rule **sendreceive**[*lnk, R, snd, rcv*] is invoked when, in reference to the service operation *R*, some data *snd* are sent to the partner link *lnk*, and the agent waits for data to be sent back, which are stored in the receive location *rcv*; no acknowledgment is expected for send and receive.

rule sendreceive[*lnk, R, snd, rcv*] =
if *status*(*self*)=*READY* **then**
extend *Message* **with** *m* **do**
seq
par
recipient(*m*):= *lnk*
sender(*m*):= *self*
serviceOp(*m*):= *R*
serviceData(*m*):= *snd*
RequestMsg(*m*):= *true*
SendMode(*m*):= *true*
endpar
awaitingRespMsg(*self*):= *m*
SENDRECEIVE$_{noAck,noAckBlocking}$(*m*,*rcv*)
endseq
if *status*(*self*)=*BLOCKED* **then** RECEIVE$_{noAckBlocking}$(m', *rcv*)
where $m' \in$ *Message* **and** *ResponseMsg*(m') **and** *recipient*(m') = *self* **and** *serviceOp*(m') = R **and**
requestMsg(m') = *awaitingRespMsg*(*self*)

At first send, when the agent is ready, the rule uses a simplified version of the pattern SEND$_s$RECEIVE$_t$ in [BB05] with $s = noAck$ and $t = noAckBlocking$. It is a combination of the machines for the send and the receive patterns and specifies 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 SENDRECEIVE$_{noAck,noAckBlocking}$(*m*,*rcv*) =
par
SEND$_{noAck}$(*m*)
RECEIVE$_{noAckBlocking}$(m', *rcv*)
endpar
where $m' \in$ *Message* **and** *ResponseMsg*(m') **and** *recipient*(m') = *self* **and** *serviceOp*(m') = R **and**
requestMsg(m') = *m*

When, instead, the request message has been already sent and the agent is still blocked waiting to receive the response message, the action is described by the receive pattern only. In this case, the value of the controlled function *awaitingRespMsg*—here introduced to store the message m for which the agent is waiting to receive the corresponding response message—is used as parameter m.

The subrule CONSUME(m) invoked by RECEIVE$_{noAckBlocking}$ is the same as that introduced for the wrapper receive but the update *awaitingRespMsg*(*self*):= *undef* must be added to reset the location *awaitingRespMsg* when the response message has been effectively delivered and consumed.[16]

5.3.3. *Fault/compensation handling*

Exception and compensation handlers to be executed in case of exception/compensation for a component's service operation R_A may be specified in terms of user-defined rules annotated, respectively, with `@ExceptionHandler` and `@CompensationHandler`. The predefined functions *exceptionHandler*(R_A) and *compensationHandler*(R_A) are used, within the `init`ialization rule for a given component, to associate R_A, respectively, with its exception handler or compensation handler. They are formally declared as dynamic controlled functions by specifying the underlying agent and the name of the service operation (R_A) as parameters:

exceptionHandler: Agent × *String* → *Rule*(*Agent*)
compensationHandler: Agent × *String* → *Rule*(*Agent*)

[16] In case of a simple receive action, the update of the location *awaitingRespMsg* to the undef value has no effect since the location is already undefined.

The predefined rule RAISEEXCEPTION to rise an exception is defined as follows:

```
rule RAISEEXCEPTION(a, R_A, exceptionSbj) =
par
    status(a):=EXCEPTION
    EXCEPTIONHANDLER(a,R_A)(a)
    exceptionSubject(a):= exceptionSbj
endpar
```

where the dynamic controlled function *exceptionSubject: Agent* → *String* is a placeholder for a message motivating the exception rising.

The predefined rule COMPENSATE to compensate a service activity R_A is defined as follows:

```
rule COMPENSATE(a, R_A) =
par
    status(a):=COMPENSATION
    COMPENSATIONHANDLER(a,R_A)(a)
endpar
```

The predefined rule COMPENSATEALL to compensate all activities nested within a completed service activity R_A is defined as follows:

```
rule COMPENSATEALL(a, R_A) =
    loop through act ∈ toCompensate(a,R_A) do COMPENSATE(a,act)
```

It invokes sequentially the installed compensation handlers of the sub-activities in reverse order of their completion. To this end, the dynamic controlled function *toCompensate: Agent* × *String* → *Seq(String)*

takes an agent and the name of an activity as parameters and returns a sequence (a queue) of actions that have been executed and can be compensated. This function is to be initialized to the empty sequence [] in the `init`ialization rule of the component, and then populated by appending the sub-activities's names after their completion during the execution of the activity R_A (as part of its behavior specification).

5.4. Context awareness

The expected behavior of an SCA-ASM component in presence of pre/post conditions annotating a service R can be formally refined in terms of the SCA-ASM behavioral primitives according to the following equivalence schema:

```
//@pre cond_u & cond_s' & cond_s''
//@post cond_post
//@service
rule R = ... ≡
//@service
rule R' =
if cond_u & cond_s'
then
    seq
        R[cond_s'']
        if not cond_post then RAISEEXCEPTION(self, R, "post-violation")
    endseq
else RAISEEXCEPTION(self, R, "pre-violation")
```

where R[$\text{cond}_{s''}$] is the service R whose *receive* and *send&receive* action occurrences are immediately followed by "assert" actions (through the conditional rule constructor IF) to check that the condition over values returned (if any) by invoked external services are fulfilled and in case of an assert violation, a raise condition due to a pre violation of R must be risen. In case of a post violation, the exception handler of the service R has by default to

compensate R:

//@ExceptionHandler
rule EXCEPTIONHANDLERR(a, R) =
if *exceptionSubject*(a) = "*post-violation*" **then** *compensateAll*(a,R)

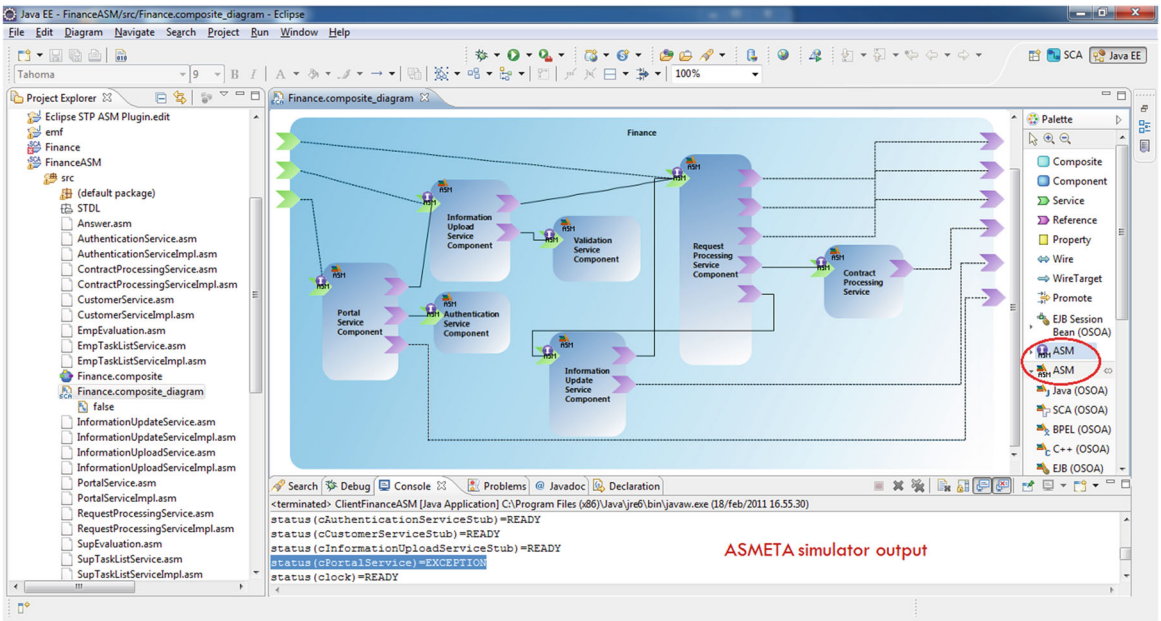

Fig. 8. SCA-ASM tool screenshot

5.5. Executable semantic model

The complete SCA-ASM computational model presented in this section, which consists of the transition rules for the communicator, the transition rules capturing the semantics of all the primitives for service computation, coordination, communication, fault/compensation handling, and the transition rules expressing context awareness, has been encoded in AsmetaL and is executable by using the simulator AsmetaS.

In the following section, we present the tool that we developed by integrating the simulator AsmetaS into the SCA runtime platform Tuscany. The availability of this executable SCA-ASM computational model has made possible the composition of SCA-ASM components into an executable SCA application.

In Sect. 6.3, in the context of a wider presentation of the developed applications by using the SCA-ASM, we discus our experimentation regarding the SCA-ASM model of the finance case study.

6. Tool support

As a proof of concept, we developed a tool [SCAb] that allows modelers to design, assembly, and execute SCA-ASM models of components in an unique integrated environment (see Fig. 8).

The tool 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)—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 SCA-ASM elements like component and interface implementation. Figure 8 shows a screenshot of the tool. Appropriate ASM icons (see the right side of Fig. 8) 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. 8) and then dragged and dropped on the components and interfaces of the SCA assembly diagram.

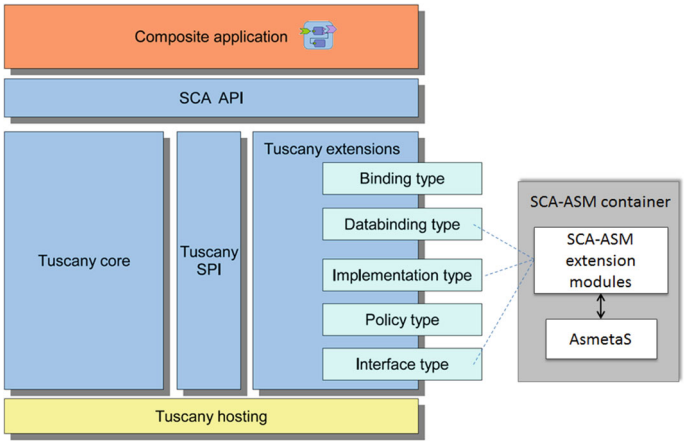

Fig. 9. The main building blocks of the Tuscany SCA runtime with SCA-ASM (adapted from [NAS11])

The back-end is the *Apache Tuscany SCA runtime* [Tus]—to run and test SCA assemblies of components developed with different implementation technologies and spread across a distributed environment (cloud and enterprise infrastructures). We extended it to allow the interaction with the simulator ASMETA/AsmetaS [Asm11, GRS08] for the execution of SCA-ASM components. The extended Tuscany runtime allows therefore the execution of ASM models of SCA components through the simulator ASMETA/AsmetaS (as shown by the AsmetaS console output in Fig. 8) within Tuscany. An application developer or designer may rely on the SCA-ASM language as a formal and abstract component implementation type to cover *computation*, *communication*, *coordination* and *fault/compensation* aspects during early execution (or simulation) of an SCA assembly of an heterogeneous service-oriented application. The developer/designer can still adopt other SCA component implementation types (such as Java, Spring, C++, etc., see [SCAa]) to include components providing real computation services and these components can themselves require services provided by other local or remote components.

The Tuscany runtime has a modular and pluggable architecture (see Fig. 9) developed in the Java programming language. At a high level the Tuscany runtime can be divided into a core infrastructure and a set of extensions that extend the core to work with various technologies. The SCA *composite application* is shown in the top box in Fig. 9 and represents the service application being built. Technically, it is an XML file used by the runtime to instantiate and execute the resulting application by instrumenting AsmetaS and other execution infrastructures in an unique environment. The *SCA API* allows component implementations in the composite application to interact with the runtime. The *Tuscany core* supports construction of components and their services, the assembly of components into usable composite applications, and the management of the resulting applications. The basic plug points are shown on the right-hand side of Fig. 9 and consist of *binding*, *data binding*, *implementation type*, *policy*, and *interface*. Bindings provide support for different kinds of communication protocols, such as SOAP/HTTP web services, JSON-RPC, and RMI. Databindings provide support for different data formats that can pass between services, such as SDO, JAXB, and AXIOM. The implementation type extension provides support for different programming languages and container models, such as the Java language, BPEL, Spring, etc, and SCA-ASM itself. Tuscany users can develop or use services written with different languages in their composite applications. The policy extension separates infrastructure setup concerns from the development of services. This provides flexibility to adjust infrastructure-related policies such as security and transactions without impacting the code. Finally, the interface extension allows service interfaces to be described using a variety of technologies.

In general, a third party can extend an SCA runtime like Tuscany by creating a "container" that plugs specific extension code into that runtime to support a particular implementation technology.[17] As shown in Fig. 9, we extended the Tuscany platform by developing a container for the SCA-ASM technology. Essentially, the *SCA-ASM container* includes the AsmetaS simulator and the extension code to Tuscany for instantiating and handle incoming/outgoing service requests to/from an ASM component implementation instance. We implemented (in Java) the extension code for the SCA-ASM implementation type, Interface type (to allow the specification of interfaces through ASM modules) and databinding (a data format for ASM data types). 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. The interaction between the Tuscany core and the simulator AsmetaS is carried out through an ASM implementation provider. This provider is therefore responsible for handling the lifecycle of an SCA-ASM component implementation and delegates to the simulator AsmetaS incoming/outgoing service requests to/from an SCA-ASM component. More implementation details are provided in [RSA11a] and reported also in Sect. 6.2.

SCA-ASM makes it possible to specify abstract components, to compose them, and to simulate them with the help of the Tuscany platform and the simulator AsmetaS. This tools integration allows *in-place simulation* to execute the ASM specification (intended as *abstract implementation*) of SCA-ASM components together with other heterogeneous (non ASM-implemented) components according to the chosen SCA assembly. The designer can exploit the functionality of the AsmetaS simulator directly within the SCA runtime platform to execute early the behavior of the overall SCA composite application. Therefore, SCA-ASM can be adopted to provide abstract implementations (or prototypes) of mock components, or to implement "core" components that contain the main service composition or coordination process that guides the application's execution.

Potentially, SCA-ASM components can be also *functionally analyzed offline*, i.e., ASM models of such abstract (or mock) components may be analyzed in isolation to determine if they are fit for use. A variety of techniques exist to this purpose by exploiting the ASMETA analysis toolset, thus providing increasing degrees of confidence in functional model correctness. Indeed, in addition to simulation through the simulator ASMETA/AsmetaS, the ASMETA toolset supports other model *validation*[18] and verification techniques useful for SCA-ASM components: *Scenario-based validation*[CGRS08] to run execution scenarios and report any violation of the expected behavior; *Model inspection and review* [AGR10b] to critically examine ASM models and determine if they not only fulfill the intended requirements, but also, by the violation of *meta-properties*, if they are of sufficient quality to be easy to develop, maintain, and enhance; and *Property verification* [AGR10a] to verify properties written in the temporal logics Computational Tree Logic (CTL) and Linear Temporal Logic (LTL), using the capabilities of the NuSMV model checker [NuS]. In addition, the validated and verified SCA-ASM models can be eventually reused as *oracles*, when the real implementation of those components is available, to perform *model-based testing* [GR01, GRR03], *conformance analysis* [AGR11], and *run-time monitoring* [AGR13] of the behavior of such components. We postpone such forms of analysis and the extension of the supporting ASM tools for the SCA-ASM framework as future work.

6.1. In-place simulation of SCA-ASM components

SCA-ASM components use annotations to denote services, references, properties, etc. With this information, as better described below, the SCA runtime platform Tuscany can create and execute an SCA assembly containing SCA-ASM components by tracking service references, i.e., required services, at runtime and injecting required services into a component when they become available. In SCA-ASM annotations appear within a comment in the form //@annotation to o improve readability and allow AsmetaS to interpret an SCA-ASM specification as a conventional ASM specification. These annotations are then extracted from the comments and used by the SCA Tuscany runtime to enable service components and service clients to be built in the ASM language.

[17] The SCA approach to components is different from other component technologies such as EJB, Microsoft .NET, and Spring. In SCA, any container-specific dependencies are encapsulated in the implementation type instead of being part of the component definitions [NAS11].

[18] Model validation is intended as the process of investigating a model (intended as formal specification) with respect to its user perceptions, in order to ensure that the specification really reflects the user needs and statements about the application, and to detect faults in the specification as early as possible with limited effort.

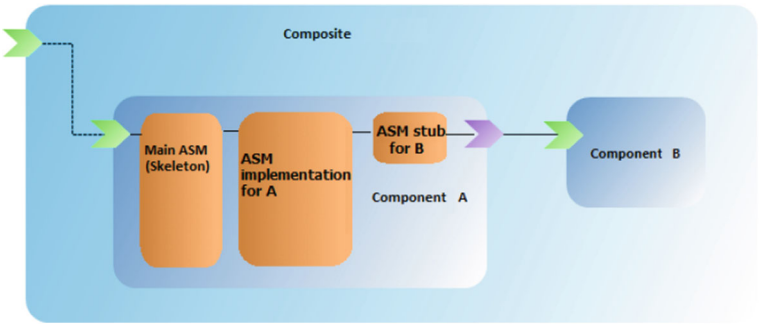

Fig. 10. Instantiating and invoking ASM implementation instances within Tuscany

Figure 10 illustrates how the *ASM implementation provider* sets up the *ASM container* within Tuscany in which the SCA-ASM component (like component `A` in the Figure) run. 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. 10)—is created within AsmetaS for all service calls of the component.[19] 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. Listing 4 reports a simplified version of such a main ASM for the component `A` in Fig. 10. Executing an ASM component implementation means executing its main ASM.

For each reference of the SCA-ASM component, another entity, i.e., another ASM module, is automatically created and instantiated as ASM agent within the main ASM of the component (see the ASM proxy for the reference `b` of `A` to the service component `B` in Fig. 10). This further agent acts as "proxy" for a remote component to allow outbound service calls from the component. Using a terminology adopted in the Java Remote Method Invocation (RMI) API, this proxy ASM agent 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's agent (the agent of the component `A` in Fig. 10). The main ASM, instead, plays the role of *skeleton*, i.e. a proxy for a remote entity that runs on the provider and forwards (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). Listing 5 reports the ASM module (including its providing interface) generated automatically by the framework as stub for the reference of `B` to the service component `B` in Fig. 10. Note that (though not shown) the framework also re-write the ASM module for the SCA-ASM component `A` by adding the suffix "Stub" to the name of the required interface `BService` and to the name of the agent codomain `BService` of the reference `b` for avoiding interface type mismatching with the stub interface.

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 also the Java interface as IDL for SCA interfaces.

In case the component provides multiple services, the annotation `@MainService` is used to mark that elected as *main service* (see Sect. 4.2). Indeed, in case of multiple `@Provided` interfaces, and, therefore, multiple agent types declarations, one must be elected as main active agent (the one providing the *main service*). This allows a component to contain more than one active agent within it, but only one, the *main agent*, is responsible for initializing the component's state (in the rule `r_init`) and, eventually, for the startup of the other agents by assigning programs to them.

[19] We postpone as future work the implementation of the other two SCA implementation scopes, *stateless* (to create a new component instance on each service call) and *conversation* (to create a component instance for each conversation).

Listing 4: Main ASM of the component A in Fig. 10

```
asm MainASMforA
import STDL/StandardLibrary
import STDL/CommonBehavior
import AComponent
import BServiceStubComponent
signature:
domain Skeleton subsetof Agent

//agents declarations
static skeleton : Skeleton
static communicator : CommunicationChannel
static compA : AService
static stubB : BServiceStub

//Name of the requested service
controlled serviceOp : String
//default location for an input parameter (if any) as part of the requested service
monitored inValue : D
//default location for a result (if any) to return back to the client
controlled outValue : D
definitions:

rule r_init =
par
 //the skeleton's status is set to READY
 status(self):=READY
 //wires setting
 clientAService(compA):=self
 b(compA) := stubB
 //component's agents initialization
 r_init[compA]
 r_init[stubB]
endpar

//skeleton's program
rule r_skeleton =
 if status(self) = INIT then r_init[]
 else //The skeleton forwards a client service request incoming from the Tuscany protocol layer
        //to the A component'agent
   r_sendreceive[compA,serviceOp,inValue,outValue]

//main rule
main rule r_main =
seq
    program(communicator)
    program(skeleton)
    par program(compA)
        program(stubB)
    endpar
endseq

default init s0:
//initial state
function status($a in Agent)= INIT
function communicatorStatus(communicator) = RESET
function program(communicator):= r_CommunicatorProgram[]
function program(skeleton):= r_skeleton[]
function program(compA):= r_A[]
function program(stubB):= r_BStub[]
```

Listing 5: ASM stub for the reference `b` of component `A` in Fig. 10

```
module BServiceStubComponent
import STDL/StandardLibrary
import STDL/CommonBehavior
import BServiceStub
signature:
//@Backref to the client
shared clientBServiceStub: Agent −> Agent
//location to store the input parameter of the requested service
controlled inputb : D
definitions:

//@Service
rule r_opB=
  seq
      r_receive[clientBServiceStub(self),"r_opB(Agent,D)",inputb]
      r_replay[clientBService(self),"r_opB(Agent,D)",opB(self)]
      opB(self):=undef //reset of the business location for the return value
  endseq

rule r_BServiceStubComponent =
if nextRequest(self)="r_opB(Agent,D)" then r_opB[]

rule r_init($a in BServiceStub) = status($a):=READY


module PingServerServiceStub
import STDL/StandardLibrary
import STDL/CommonBehavior
signature:
domain BServiceStub subsetof Agent
dynamic out opB : Agent −> Any
definitions :
```

Other simulation features Useful features are currently supported by the AsmetaS simulator when running within the SCA Tuscany platform.

State invariant checker: AsmetaS implements an invariant checker, which at the end of each transition execution checks if the invariants (if any) expressed over the state of the currently executed SCA-ASM component are satisfied or not. If an invariant is not satisfied, AsmetaS throws an `InvalidInvariantException`, which keeps track of the violated invariant.

Consistent Updates checking: The simulator also includes a checker for revealing inconsistent updates. In case of inconsistent updates an `UpdateClashException` is thrown by reporting the location which is being inconsistently updated and the two different values which are assigned to that location. The user, analyzing this error, can detect the fault in the ASM component implementation.

Logging: AsmetaS produces a minimal output to show the current state and the update set. The user can inspect how AsmetaS performs some tasks (e.g. terms evaluation, building of updates set, variables substitution) by a log4j[20] file.

6.2. Implementation details

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. The section contains information useful for those interested in developing specific implementation provider for SCA. However, it can be skipped by a not interested reader.

[20] http://logging.apache.org/.

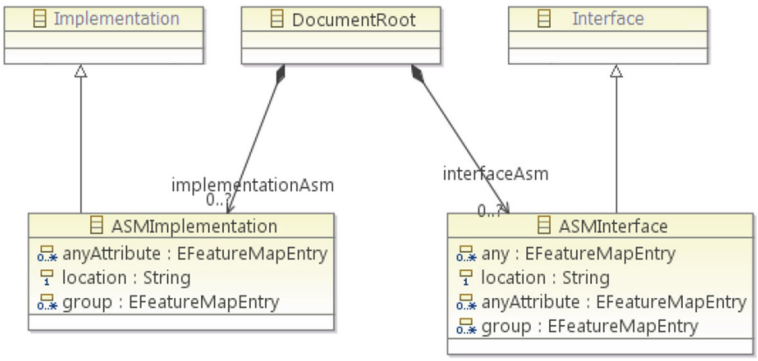

Fig. 11. A fragment of the SCA metamodel extension to support SCA-ASM

6.2.1. Extending the Eclipse-based SCA composite designer

First, we extended the SCA metamodel [SCAd], an extensible EMF Ecore-compliant metamodel that represents concepts of the Open SOA SCA specifications 1.0 [SCAa] plus different extending concepts to support open SCA runtimes (like Apache Tuscany and Frascati). Extending the SCA metamodel to add new concepts to SCA and extend the tools to include them is straightforward. Figure 11 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 Fig. 8), the graphical 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.

6.2.2. Extending Tuscany with an SCA-ASM container

Creating a new extension in the runtime Tuscany required to us two distinct steps. First, we developed the extension code (using the Java programming language) for the SCA-ASM container handling the new technology `implementation.asm`. The UML package diagram in Fig. 12 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[21]—`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. 12) that knows how to transform the XML description 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 Java class `ASMImplementationProvider`, that the Tuscany runtime uses to locate, invoke, and manage the extension at runtime.

The ASM implementation provider is responsible for handling the lifecycle of an SCA-ASM component implementation and creating operation invokers for the service operations provided by the implementation. Precisely, the ASM implementation provider delegates the handling of the ASM component implementation to AsmetaS. To this purpose, the Tuscany runtime calls the `ASMImplementationProviderFactory` (see Fig. 12) to create an instance of the `ASMImplementationProvider` for each SCA-ASM component. The `ASMImplementationProvider`'s `start()` method is invoked to set up the ASM implementation instance (i.e., the main ASM that acts as skeleton and includes the ASM modules for the stubs, the SCA-ASM component itself, etc.) when the SCA-ASM component is started.

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

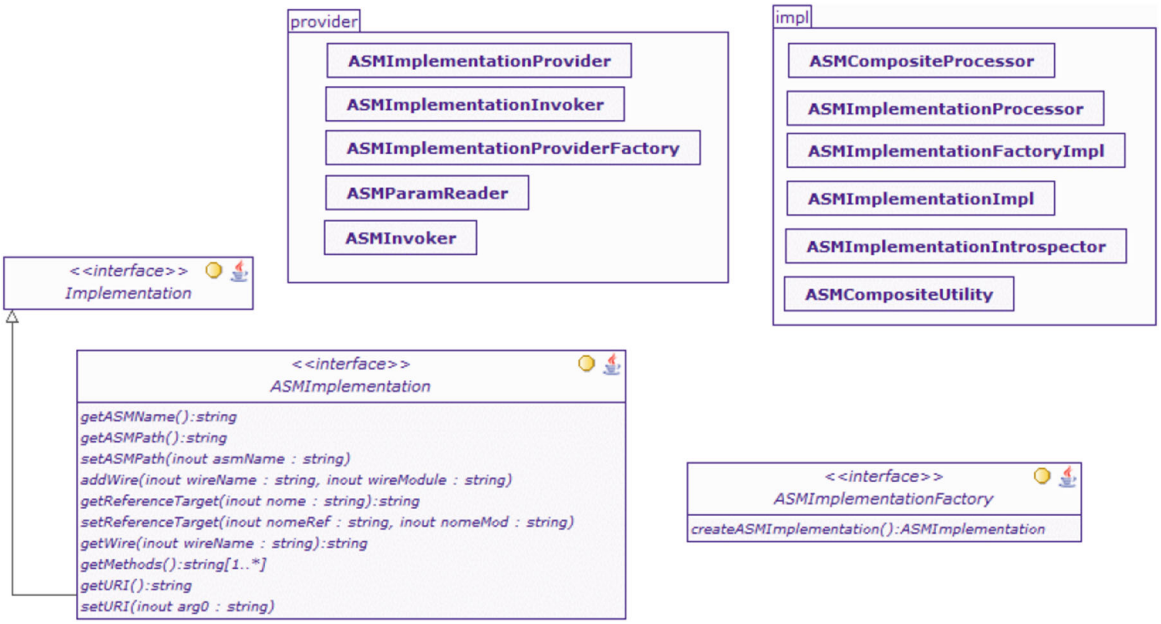

Fig. 12. Classes and interfaces for the extension `implementation.asm`

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 main ASM of the SCA-ASM component and get the response back to the caller, by simply calling the simulator AsmetaS to run the main ASM. When the SCA-ASM component is stopped, the `ASMImplementationProvider`'s `stop()` method is triggered to clean up the resources associated with the ASM implementation instance.

6.3. Case studies and lesson learned

Several case studies of varying size and covering different uses of the SCA-ASM constructs have been developed [SCAb]. These include a *Robotics task coordination* case study [BGS11] of the EU project BRICS [BRI] and the *Finance* case study of the EU project SENSORIA [SENa]. The goal was twofold: exercising and exemplifying, through more complex case studies, advanced SCA-ASM modeling constructs for service behavior such as for coordination and fault/compensation, and illustrate the in-place simulation technique of the considered scenarios in an heterogeneous SCA assembly (or composition).

In [BGRS11, BGS11], we presented a scenario of the Robotics task coordination example. In Robotics, service-oriented components embed the control logic of the application. They cooperate with each other locally or remotely through a communication network to achieve a common goal and compete for the use of shared resources, such as a robot sensors and actuators, the robot functionality, and the processing and communication resources. Cooperation and competition are forms of interactions among concurrent activities. So, in this domain, applications are very workflow-oriented and require developing coordination models explicitly [BS10]. In the considered scenario, a laser scanner offers its scan service to different clients (a 3D Perception application and an Obstacle Avoidance application), which compete for the use of this shared resource. The interactions between the clients and the Laser Scanner have to be managed by a third entity: a coordinator. This coordinator is in charge of forwarding the clients requests to the Laser Scanner and so it has to manage the concurrent access of the clients. The aim of such case study was modeling formally the coordination aspects by defining the coordinator component in SCA-ASM and executing such a component (in-place simulation) together with the other components of the application (implemented in Java) in an heterogeneous SCA assembly. More technical details on such case study can be found in the technical report [mata].

In [RSA11b], we presented a scenario of the Finance case study. This large case study from the financial domain was investigated within the EU project Sensoria [SENa] on software engineering techniques for service-oriented applications. The aim was comparing the SCA-ASM formalism with other service-oriented modeling notations developed within the Sensoria consortium (see Sect. 2). In particular, this example allowed us to experiment fault/compensation handlers. In this article, we considered as running example some components of such application. The complete SCA-ASM specification of such application is available at [SCAb].

During the development of the case studies we found that Tuscany is a lightweight architectural platform facilitating the integration of service applications based on the guidelines specified by the standard SCA. Integrated applications developed in different languages, spanning across multiple domains can be easily created and composed using this platform. An heterogeneous SCA assembly of service-oriented components implemented in SCA-ASM or in another implementation language can be produced graphically using the SCA Composite Designer and then stored and exchanged through an XML file. This last file is then used by the runtime to instantiate and execute the resulting application by instrumenting AsmetaS and other execution infrastructures in an unique environment. Developers should only manage the logic of the component, forgetting service discovery and service publication. The only drawback of such flexibility is that the SCA runtime requires the effort of developing a specific extension code to support any new implementation technology.

From an ASM point of view, a service component's model given in terms of the SCA-ASM language is more readable and modular than that expressed as plain mathematical ASM. Even if computationally there is no difference among the two models, the SCA-ASM specification allows to represent in a more direct and standard way the static view of an SCA model. This feature facilitates the use of the formal notation, and makes, therefore, easier the possibility to perform formal model analysis, which is one of the main and most difficult goals of the application of a formal framework in the development of a service application. SCA-ASM also makes the specification more reusable by making possible to embed it (or part of it) into other SCA-ASM components. Moreover, large ASM specifications can be analyzed in a modular way by analyzing single SCA-ASM components (off line analysis).

7. Conclusion and future work

We presented the SCA-ASM modeling language and its supporting framework for modeling and prototyping service-oriented applications. The language combines the standard SCA with the ASM formal method, and complements the static and architectural view of an SCA component model with the dynamic and executable view of a multi-agent ASM computational model. In the SCA-ASM language, SCA design primitives provide graphical representation of components structure and of components assemblies, while the ASM formalism allows modeling notions of service behavior, interactions, orchestration, compensation and context-awareness in an abstract and technology agnostic but executable way. Therefore, SCA-ASM makes possible to specify service components and their composition at a very high level of abstraction without worrying about implementation details and programming languages limitations.

The language has been implemented as a new *SCA component implementation type* of the SCA runtime platform Tuscany that has been extended to support the integration of the AsmetaS simulator for ASM models. Therefore, the Tuscany platform allows execution of SCA-ASM components either as component isolated from the rest of the assembly, or in combination with other components according to the chosen SCA assembly. The overall execution can be heterogeneous (i.e., by using different execution engines), since the other components can be available at different levels of abstraction and implemented in different code. The SOA designer can thus execute integrated applications and evaluate different design solutions even when the implementation of some components—abstract or mock components—is not yet available, but accessible as SCA-ASM running abstract prototype. Overall, the proposed framework supports a practical approach to tackle the complexity of service oriented applications by offering a high degree of design and validation at early development phases.

Besides simulation, by exploiting the other components of the ASMETA tool set for ASM models, the proposed framework can potentially be extended to support other, more heavier, forms of components analysis, as model checking and properties verification. It was out of the scope of this article to deal with form of functional analysis, but we want to work on this aspect. Still, on the functional analysis side, we plan to experiment the use of SCA-ASM models as oracles for reasoning and testing about real components implementations, including but not limited to, *conformance testing* and *run-time monitoring*.

As future work, we also plan to support further useful SCA concepts. In particular, we want to introduce the SCA *callback interface* for bidirectional services. The availability of such a concept (that is common in many components models specific to the Robotics domain) would have been useful especially in the *Robotics task coordination* case study [BGS11] to model asynchronous interactions among services in a more natural way. Moreover, currently the implementation scope of an SCA-ASM component is *composite*, i.e. a single component instance is created for all service calls. We postpone as future work the implementation of the other two implementation scopes, *stateless*—to create a new component instance on each service call—and *conversation*—to create a component instance for each conversation—supported by the Tuscany runtime.

Currently SCA-ASM supports a traditional SOA *request-driven* interaction style: a client requests a service from the server and waits to receive a reply from the server. The interaction is initiated by the client and completes when the server replies. We plan to enrich the notation with other interaction and workflow patterns based on the BPMN specification. In particular, we want to add specific rule constructors to support an *event-driven* interaction style, as defined in the last extensions to the SCA assembly model [SCAc] where the components of a distributed system communicate via events which are generated by some components and received by others through a publish/subscribe schema. A further interaction type, that would be interesting to support, is the *time-driven* interaction : an agent, or a group of agents, initiates an interaction with a specific timeout. The interaction completes upon reaching the specified time. Time-driven interactions are used when scheduling mechanisms are in place. In a SOA this can be achieved, for instance, in a BPEL environment using the various `OnAlarm` features on a `Pick Activity` or on a `Process/Scope` level [CS10]. We also aim at experimenting with the use of our framework by adopting more realistic models of the communication infrastructure by including message buffering and loss of messages.

Other interesting research directions we intend to investigate concern the introduction of language features to model and support *dynamic adaptation* issues, both at structural level (as addition/substitution of components) and at behavioral level (by modifying components interactions). Self-adaptability of systems has been studied in a wide range of disciplines, from biology to robotics. Only recently has the software engineering community recognized its key role in enabling the development of future software systems [CdLG+09] that are required to be *context-aware* and *self-adaptive*, i.e., the context, capturing information about the environment or its requirements, should trigger certain changes that may occur in the system in a self-manner. Service-oriented applications may require dynamic adaptation for several reasons, such as service evolution (e.g., a new version may be available), hardware volatility (e.g., network quality changes), varying user demands and new requirements (e.g., a new functionality or a different level of quality of service).

Another interesting area, in fact, concerns the non-functional aspects of services, namely the policies and constraints for service level agreement that have to be taken into account in the composition of services. Through the *SCA Policy Framework* [SCAa], we intend to enrich service descriptions with non-functional properties (such as availability, reliability, etc.) that jointly represent the quality of the service and that can be used to drive non-functional analysis of service assemblies. To this purpose, our preliminary work presented in [RPS12], about a reliability prediction for service components architecture with the SCA-ASM component model, is an example of such a form of analysis.

References

[AAA06] Attiogbé C, André P, Ardourel G (2006) Checking component composability. In: Löwe W, Südholt M (eds) Software composition. Lecture notes in computer science, vol 4089. Springer, Berlin, pp 18–33

[AAA08] André P, Ardourel G, Attiogbé C (2008) Composing components with shared services in the Kmelia model. In: Pautasso C, Tanter É (eds) Software composition. LNCS, vol 4954. Springer, Berlin, pp 125–140

[AAS12] Ameur YA, Aït-Sadoune I (2012) Stepwise development of formal models for web services compositions: modelling and property verification. In: Liddle SW, Schewe K-D, Tjoa AM, Zhou X (eds) DEXA (1). Lecture notes in computer science, vol 7446. Springer, Berlin, p 9

[ACKM04] Alonso G, Casati F, Kuno HA, Machiraju V (2004) Web Services—concepts, architectures and applications. Data-centric systems and applications. Springer, Berlin

[AFL08] Altenhofen M, Friesen A, Lemcke J (2008) ASMs in service oriented architectures. J Univers Comput Sci 14(12):2034–2058 http://www.jucs.org/jucs_14_12/asms_in_service_oriented

[AGR10a] Arcaini P, Gargantini A, Riccobene E (2010) Asmetasmv: A way to link high-level ASM models to low-level NuSMV specifications. In: Frappier M, Glässer U, Khurshid S, Laleau R, Reeves S (eds), ASM. Lecture notes in computer science, vol 5977. Springer, Berlin, pp 61–74

[AGR10b] Arcaini P, Gargantini A, Riccobene E (2010) Automatic review of Abstract State Machines by meta property verification. In: Muñoz C (ed) NASA formal methods. NASA conference proceedings, vol NASA/CP-2010-216215, pp 4–13

[AGR11] Arcaini P, Gargantini A, Riccobene E (2011) Coma: conformance monitoring of java programs by Abstract State Machines. In: Khurshid S, Sen K (eds) RV. Lecture notes in computer science, vol 7186. Springer, Berlin, pp 223–238

[AGR13] Arcaini P, Gargantini A, Riccobene E (2013) Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In: Proceedings of the 9th workshop on advances in model based testing (A-MOST 2013)

[AGRS11] Arcaini P, Gargantini A, Riccobene E, Scandurra P (2011) A model-driven process for engineering a toolset for a formal method. Softw Pract Exp 41(2):155–166

[AMFG09] Abreu J, Mazzanti F, Fiadeiro JL, Gnesi S (2009) A model-checking approach for service component architectures. In: Lee D, Lopes A, Poetzsch-Heffter A (eds) FMOODS/FORTE. LNCS, vol 5522. Springer, Berlin, pp 219–224

[Asm11] The ASMETA toolset website (2011) http://asmeta.sf.net/

[BB05] Barros AP, Börger E (2005) A compositional framework for service interaction patterns and interaction flows. In: Lau K-K, Banach R (eds) ICFEM. LNCS, vol 3785. Springer, Berlin, pp 5–35

[BBBB08] Börger E, Butler MJ, Bowen JP, Boca P (eds) (2008) In: Proceedings of the Abstract State Machines, B and Z, first international conference, ABZ 2008, London, UK, September 16–18, 2008. Lecture notes in computer science, vol 5238. Springer, Berlin

[BBG07] Ter Beek MH, Bucchiarone A, Gnesi S (2007) Formal methods for service composition. Ann Math Comput Teleinform 1(5):1–10

[BBNL08] Boreale M, Bruni R, De Nicola R, Loreti M (2008) Sessions and pipelines for structured service programming. In: Barthe G, de Boer FS (eds) FMOODS. LNCS, vol 5051. Springer, Berlin, pp 19–38

[BGRS11] Brugali D, Gherardi L, Riccobene E, Scandurra P (2011) Coordinated execution of heterogeneous service-oriented components by Abstract State Machines. In: Arbab F, PeterCsaba n++lveczky (eds) FACS. Lecture notes in computer science, vol 7253. Springer, Berlin, pp 331–349

[BGS11] Brugali D, Gherardi L, Scandurra P (2011) A robotics task coordination case study. In: Workshop on software development and integration in robotics (SDIR), 9 May 2011

[BH06] Bussler C, Haller A (eds) (2006) Business process management workshops, BPM 2005 international workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, vol 3812. Revised Selected Papers

[BLJM08] Bieberstein N, Laird R, Jones K, Mitra T (2008) Executing SOA: a practical guide for the service-oriented architect. Addison-Wesley, Reading

[BLPT09] Banti F, Lapadula A, Pugliese R, Tiezzi F (2009) Specification and analysis of SOC systems using COWS: a finance case study. Electr Notes Theor Comput Sci 235:71–105

[Bör07] Börger E (2007) Modeling workflow patterns from first principles. In: Parent C, Schewe K-D, Storey VC, Thalheim B (eds) ER. Lecture notes in computer science, vol 4801. Springer, Berlin, pp 1–20

[BPM10] OMG Business Process Model and Notation (BPMN) 2.0. (2010). http://www.omg.org/spec/BPMN/2.0

[BPZ09] Bernardo M, Padovani L, Zavattaro G (eds) (2009) Formal methods for web services. In: 9th International school on formal methods for the design of computer, communication, and software systems, SFM 2009, Bertinoro, Italy, June 1–6, 2009, Advanced lectures. LNCS, vol 5569. Springer, Berlin

[BRI] EU project BRICS (Best Practice in Robotics). www.best-of-robotics.org/

[Bru09] Bruni R (2009) Calculi for Service-Oriented Computing. In: Bernardo et al. [BPZ09], pp 1–41

[BS03] Börger E, Stärk R (2003) Abstract State Machines: a method for high-level system design and analysis. Springer, Berlin

[BS10] Brugali D, Shakhimardanov A (2010) Component-based robotic engineering (part II): systems and models. Robotics XX(1):1–12

[BST09] Börger E, Sörensen O, Thalheim B (2009) On defining the behavior of or-joins in business process models. J Univ Comput Sci 15(1):3–32

[BT08] Börger E, Thalheim B (2008) Modeling workflows, interaction patterns, web services and business processes: the ASM-based approach. In: Börger et al. [BBBB08], pp 24–38

[CdLG+09] Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds) (2009) Software engineering for self-adaptive systems [outcome of a Dagstuhl Seminar]. Lecture notes in computer science, vol 5525. Springer, Berlin

[CGRS08] Carioni A, Gargantini A, Riccobene E, Scandurra P (2008) A scenario-based validation language for ASMs. In: Börger et al. [BBBB08], pp 71–84

[CS10] Chandy M, Schulte R (2010) McGraw-Hill

[DCL08] Ding Z, Chen Z, Liu J (2008) A rigorous model of service component architecture. Electr Notes Theor Comput Sci 207:33–48

[DLC08] Du D, Liu J, Cao H (2008) A rigorous model of contract-based service component architecture. In: CSSE (2). IEEE Computer Society, pp 409–412

[EMF08] Eclipse Modeling Framework (2008). http://www.eclipse.org/emf/

[FGT12] Filieri A, Ghezzi C, Tamburrelli G (2012) A formal approach to adaptive software: continuous assurance of non-functional requirements. Form Asp Comput 24(2):163–186

[FLB11] Fiadeiro JL, Lopes A, Bocchi L (2011) An abstract model of service discovery and binding. Form Asp Comput 23(4):433–463

[FLBA11] Fiadeiro JL, Lopes A, Bocchi L, Abreu J (2011) The sensoria reference modelling language. In: Wirsing M, Hölzl MM (eds) Results of the SENSORIA project. Lecture notes in computer science, vol 6582. Springer, Berlin, pp 61–114

[FR05] Fahland D, Reisig W (2005) ASM-based semantics for BPEL: the negative control flow. In: Proceedings of the 12th international workshop on Abstract State Machines, pp 131–151

[GGV04] Glässer U, Gurevich Y, Veanes M (2004) Abstract communication model for distributed systems. IEEE Trans Softw Eng 30(7):458–472

[GLG+06] Guidi C, Lucchi R, Gorrieri R, Busi N, Zavattaro G (2006) : A calculus for service oriented computing. In: Dan A, Lamersdorf W (eds) ICSOC. LNCS, vol 4294. Springer, Berlin, pp 327–338

[GR01] Gargantini A, Riccobene E (2001) ASM-based testing: coverage criteria and automatic test sequence. J UCS 7(11):1050–1067

[GRR03] Gargantini A, Riccobene E, Rinzivillo S (2003) Using spin to generate tests from ASM specifications. In: Börger E, Gargantini A, Riccobene E (eds) Abstract State Machines. Lecture notes in computer science, vol 2589. Springer, Berlin, pp 263–277

[GRS08] Gargantini A, Riccobene E, Scandurra P (2008) A metamodel-based language and a simulation engine for Abstract State Machines. J UCS 14(12):1949–1983

[GT05] Gurevich Y, Tillmann N (2005) Partial updates. Theor Comput Sci 336(2–3):311–342

[HSS05] Hinz S, Schmidt K, Stahl C (2005) Transforming BPEL to Petri nets. In: Proceedings of the international conference on business process management (BPM2005). Lecture notes in computer science, vol 3649. Springer, Berlin, pp 220–235

[LGK+11] Louhichi S, Graiet M, Kmimech M, Bhiri MT, Gaaloul W, Cariou E (2011) MDE approach for the generation and verification of sca model. In: Proceedings of the 13th international conference on information integration and web-based applications and services, iiWAS '11, New York, NY, USA. ACM, pp 317–320

[LMVR07] Lanese I, Martins F, Vasconcelos VT, Ravara A (2007) Disciplining orchestration and conversation in service-oriented computing. In: SEFM'07. IEEE, pp 305–314

[LPT07] Lapadula A, Pugliese R, Tiezzi F (2007) A calculus for orchestration of web services. LNCS. Springer, Berlin, pp 33–47

[mata] EU project BRICS (2011) Technical Report. A coordination use case. www.best-of-robotics.org/wiki/images/e/e0/coordinationusecaseubergamo.pdf

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

[MM06] Martens A, Moser S (2006) Diagnosing SCA components using wombat. In: Dustdar S, Fiadeiro JL, Sheth AP (eds) Business process management. Lecture notes in computer science, vol 4102. Springer, Berlin, pp 378–388

[MSK08] Mayer P, Schroeder A, Koch N (2008) A model-driven approach to service orchestration. In: IEEE SCC (2), pp 533–536

[MSKK09] Mayer P, Schroeder A, Koch N, Knapp A (2009) The UML4SOA profile. Technical Report, LMU Muenchen

[NAS11] Laws S, Combellack M, Feng R, Mahbod H, Nash S (2011) Tuscany SCA in action. Manning Publications

[NuS] NuSMV: a new symbolic model checker. http://nusmv.fbk.eu/

[PEP] The PEPA stochastic analyzer. http://www.dcs.ed.ac.uk/pepa/

[RPS12] Riccobene E, Potena P, Scandurra P (2012) Reliability prediction for Service Component Architectures with the sca-asm component model. In: Cortellessa V, Muccini H, Demirörs O (eds) EUROMICRO-SEAA. IEEE Computer Society, pp 125–132

[RS10a] Riccobene E, Scandurra P (2010) An ASM-based executable formal model of service-oriented component interactions and orchestration. In: BM-MDA'10: workshop on behavior modeling in model-driven architecture. ACM

[RS10b] Riccobene E, Scandurra P (2010) Specifying formal executable behavioral models for structural models of service-oriented components. In: van Sinderen M, Sapkota B (eds) ACT4SOC. SciTePress, pp 29–41

[RSA11a] Riccobene E, Scandurra P, Albani F (2011) An Eclipse-based SCA design framework to support coordinated execution of services. In: Online proceedings of the 6th workshop of the Italian eclipse community (Eclipse-IT'2011)

[RSA11b] Riccobene E, Scandurra P, Albani F (2011) A modeling and executable language for designing and prototyping service-oriented applications. In: EUROMICRO-SEAA. IEEE, pp 4–11

[S-c] EU project S-Cube. http://www.s-cube-network.eu/

[SBS04] Salaün G, Bordeaux L, Schaerf M (2004) Describing and reasoning on web services using process algebra. In: Proceedings of the IEEE international conference on web services, ICWS '04, Washington, DC, USA. IEEE Computer Society, p 43

[SCAa] OASIS/OSOA. Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca

[SCAb] The SCA-ASM design framework. https://asmeta.svn.sf.net/svnroot/asmeta/code/experimental/SCAASM

[SCAc] SCA Service Component Architecture Assembly Model Specification—Extensions for Event Processing and Pub/Sub (2009). http://www.oasis-opencsa.org/sca

[SCAd] SCA Tools. http://eclipse.org/stp/sca/

[SENa] EU project SENSORIA. www.sensoria-ist.eu/

[SENb] The SENSORIA Approach: White Paper October 17th, 2007. http://www.sensoria-ist.eu/images/stories/frontpage/whitepaper_sensoria.pdf/

[SOM] The Service-Oriented Modeling Framework in Enterprise Architect. http://www.sparxsystems.com/somf

[tBBG07] ter Beek MH, Bucchiarone A, Gnesi S (2007) Web service composition approaches: From industrial standards to formal methods. In: ICIW. IEEE Computer Society, p 15

[Tus] Apache Tuscany. http://tuscany.apache.org/

[vdABvH+06] van der Aalst WMP, Beisiegel M, van Hee KM, König D, Stahl C (2006) A SOA-based architecture framework. In: Leymann F, Reisig W, Thatte SR, van der Aalst WMP (eds) The role of business processes in service oriented architectures. Dagstuhl seminar proceedings, vol 06291. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany

[vdAMSW09] van der Aalst WMP, Mooij AJ, Stahl C, Wolf K Service interaction: patterns, formalization, and analysis. In: Bernardo et al. [BPZ09], pp 42–88

[Ver05] Verbeek HMW, van der Aalst WMP (2005) Analyzing BPEL processes using Petri nets. In: Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management. Florida International University, Miami, Florida, USA, pp 59–78

[VRMCL08] Vaquero LM, Rodero-Merino L, Caceres J, Lindner M (2008) A break in the clouds: towards a cloud definition. SIGCOMM Comput Commun Rev 39(1):50–55

[WS-07] OASIS Standard WS-BPEL 2.0 (2007). http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

Received 14 February 2013
Revised 8 October 2013
Accepted 10 October 2013 by M.J. Butler
Published online 25 December 2013