A modeling and executable language for designing and prototyping service-oriented applications

Elvinia Riccobene
DTI - Università degli Studi di Milano, IT
Email: elvinia.riccobene@unimi.it

Patrizia Scandurra Fabio Albani
DIIMM - Università degli Studi di Bergamo, Italy
Email: patrizia.scandurra@unibg.it

Abstract—This paper presents an intuitive, precise and executable language, SCA-ASM, for model-based design and prototyping of service-oriented applications. The language combines the SCA (Service Component Architecture) capability of modeling and assembling heterogeneous service-oriented components in a technology agnostic way, with the rigor of the Abstract State Machines formal method able to model notions of service behavior, interactions, orchestration, compensation in an abstract but executable way. For an early and quick design evaluation of a composite software application, 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; (ii) configured in place within an SCA-compliant runtime platform as abstract implementation (or prototype) of a component and then executed together with the other components implementations according to the chosen SCA assembly.

I. INTRODUCTION

Service-Oriented Computing (SOC) is a paradigm for distributed computing based on the principle that “Everything is a service”. Services are intended as loosely coupled autonomous and heterogeneous1 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. Web Services is the most notable example of service oriented technology.

In order 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 paper addresses the problem of designing, prototyping and evaluating service oriented systems in an assembly-oriented manner (i.e. by assembling already available service-oriented components) by means of high level modeling languages. We complement the Service Component Architecture (SCA)[1] – the open standard model for heterogeneous service assembly – with an executable formalism based on the Abstract State Machines (ASM) [2] formal method able to model notions of service interactions, orchestrations, compensations, and the services internal behavior. The result, and this is the novelty of our approach, is a formal and executable language, called SCA-ASM, intended for the specification and 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 [3] for functional analysis. An SCA-ASM model of a service-oriented component (or even of the entire system) can be simulated and analyzed off line, i.e. in isolation by means of the ASMETA toolset. 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 an SCA-compliant runtime platform (like Tuscany) as abstract implementation (or prototypes) of those (mock) 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 components2 – is not yet available, but an abstract model, in terms of ASMs, is available as a prototype specifying their desired behavior.

We here mainly focus on presenting the SCA-ASM language and the supporting tool for application prototyping. Illustrating results of model formal analysis is

The second author has been supported in part by the European project FP7-ICT-231940-BRICS (Best Practice in Robotics).

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

2Mock 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.
out of the scope of this paper. Moreover, we assume the reader familiar with the basic notions concerning the SCA standard and the ASM formal method, which are here not provided for the sake of space. The remainder of this paper is organized as follows. Section II describes some related works along this direction. The SCA-ASM language is presented in Section III, while Section IV presents the supporting design tool. Finally, Section V concludes the paper and outlines some future directions of our work.

II. RELATED WORK

Some visual notations for service modeling have been proposed, such as the OMG SoaML UML profile [4]. SoaML, like the SCA initiative, is more focused on architectural aspects of services. UML4SOA [5] is another UML extension defined within the EU project SENSORIA [7]. UML4SOA is focused on modeling service orchestrations as an extension of UML 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 [8]; 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.

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 [9] for the service composition problem), mostly developed within the SENSORIA and S-Cube [10] EU projects. Several process calculi for the specification of SOA systems have been designed (see, e.g., [11], [12], [13]). They provide linguistic primitives supported by mathematical semantics, and verification techniques for qualitative and quantitative properties [14]. In particular, in [15] 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 systems, named SRML [16], has been developed. SRML supports qualitative and quantitative analysis techniques using the UMC model checker [17] and the PEPA stochastic analyzer. Compared to the formal notations mentioned above, the ASM method has the advantage to be executable.

Within the ASM community, the 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 [18], [19], [20], [21] to name a few. Some of these previous formalization efforts, as better explained later, are at the basis of our work.

On the formalization of the SCA component model, some previous works, like [22], [24] to name a few, exist. However, they do not rely on a practical and executable formal method like ASMs. In [25], 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 not proven evidence, however, that this methodology scales effectively to large systems.

An abstract service-oriented component model, named Kmelia, is formally defined in [26], [27] 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 and the ASM-related tools with the standard SCA and its runtime platform for a more practical use and an easier adoption by developers.

III. THE SCA-ASM LANGUAGE OVERVIEW

We adopt a suitable subset of the SCA standard for modeling service-oriented components assemblies, and we complement such models with an ASM-based formal and executable description of the services internal behavior, services orchestration and interactions. To this purpose, we exploit the notion of distributed multi-agent ASMs. Each service-oriented component is thus modeled by an ASM endowed with (at least) one agent (a business partner or role instance) able to be engaged in conversational interactions with other agents by providing and requiring services to/from other service-oriented components’ agents.

A. SCA-ASM COMPONENTS AND ASSEMBLIES

An SCA-ASM component is an ASM module that may provide interfaces (called services), require interfaces (called references) and expose properties. The services behaviors 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.

Fig. 2 shows the shape of an SCA-ASM component A using the graphical SCA notation, and the corresponding ASM modules for the provided interface AService (on the left) and the skeleton of the component itself (on the right) using the textual AsmetaL notation of the ASMETA toolset. Similarly, Fig. 3 shows the shape of an SCA-ASM composite component and the resulting ASM module C corresponding to the SCA composite C in Fig. 1. Details on these concepts follows.

a) 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 paragraph). As interface definition language (IDL), SCA-ASM exploits the ASM notion of signature for declaring domains and functions symbols

\(^3\)http://www.dcs.ed.ac.uk/pepa/
characterizing an ASM state. An interface of an SCA-ASM component is therefore an ASM module containing only an header of the form \((\text{name}, \text{signature}, \text{import}, \text{export})\): \text{name} is the interface name, \text{signature} is defined as \((\text{bus Agent types decl}, \text{bus functions decl})\) and denotes a collection of declarations of business agent types (declared in terms of subdomains of the predefined ASM Agent domain) and of business functions (declared as parameterized ASM out functions), \text{import} denotes other imported module libraries, and \text{export} exposes signature symbols to be imported from other modules.

As additional IDL, Java interfaces are also supported.

b) **Component description:** We maintain the vision that service-oriented components are configured instances of implementations. An SCA-ASM component is therefore an ASM instance with an associated ASM agent that executes a specific program (a named ASM transition rule) as its behavior. To this purpose, an SCA-ASM component implementation is an extension of an ASM module of form \((\text{header}, \text{body})\). The header has shape \((\text{name}, \text{prov_services}, \text{req_services}, \text{signature}, \text{import}, \text{export})\), where: \text{name} is the component name; \text{prov_services} and \text{req_services} are import clauses annotated, respectively, with \@Provided and \@Required, to include the ASM modules of the service interfaces provided/required by the component; the \text{signature} is defined as \((\text{pro_decl, ref_decl, dom_and_funct_decl})\) and contains declarations for externally settable property values (i.e. ASM monitored functions – or shared functions when promoted as a composite property – annotated with \@Property), declarations for references (ASM controlled functions annotated with \@Reference) that are abstract access endpoints to services (as better explained below), and declarations of other ASM domains and functions to be used by the component for internal computation only; finally, import and export of other module libraries may be also included as well.

In SCA, references are abstract access endpoints to services that will be possibly discovered at runtime. In SCA-ASM, references are represented as 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 in the ASM module \(A\) in Fig.1). 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 (even if it is not known at design time) the partner’s business role (i.e. the agent type). Back references to requester agents are modeled as functions in the same way (using the annotation \@Backref), but the agent codomain is the most generic one (i.e. the Agent domain).

The body of an SCA-ASM component has the shape \((\text{dom_and_funct_def, inv_def, rule_def, service_def, prog_def, inst_def, handler_def})\), and consists of definitions of domains and functions (static concrete-domains and static/derived functions) already declared in the signature, definitions of state invariants, definitions of (utility) transition rules for internal computation, definitions of services (i.e. definition of transition rules annotated with \@Service), the transition rule definition (that takes by convention the same name of the component’s module) to assign as “program” to the component’s agent created during the initialization of the top composite ASM, and the transition rule with the predefined name \(r\_init\) that is in turn invoked in the initialization rule of the container
composite to initialize the internal state (controlled functions). In addition, named transition rules, annotated respectively, with @ExceptionHandler and @Compensation-Handler can be defined as exception and compensation handlers (see paragraph III-B4 below).

Fig. 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 $BService$, 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 $r_A$ characterize the agent associated to the component $A$.

Note that the notion of service operation (so) provided by a component is characterized by the pair $(I_s,R_s)$: $I_s$ is the service interface (an ASM module) imported by the component as provided interface, $R_s$ is the named ASM transition rule annotated with $@Service$ (by convention it 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 (location); the value of such location 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 and, therefore, multiple agent types declarations), one is elected as main service (read: main active agent) by specifying the annotation $@MainService$ when importing the corresponding service interface. 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.

c) Assembly description: SCA describes the content and linkage of an application in assemblies called composites. Composites can contain components, services, references, property declarations, plus suitable wires to establish connections. Composites can be used as complete component implementations within other composites, allowing for a hierarchical construction of business solutions. A top level composite describes the overall assembly.

Definitions similar to the ones 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 the sub-components of the SCA composite. 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 (constructor) rule of the composite ASM in terms of function (reference) assignments. The top composite SCA-ASM describing the overall assembly is the main ASM endowed with an initial state and a main rule to provide the necessary initialization and the initial startup of all agents’ programs to make the system model executable. The resulting system is an asynchronous multi-agent ASM that will behave accordingly to the behavior of each service (ASM agent) involved in.

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 sub-components. It also carries out in the constructor rule $r_{init}$ the wires setting, properties setting, agents’ program assignment, and initialization of the sub-components.

We abstract from the SCA notion of binding, i.e. from several access mechanisms used by services and references (e.g. WSDL binding, JMS binding, 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 and the referenced service name and data.

B. Service behavior

Commands of the SCA-ASM language to model behavior include constructs to express the control flow of component’s tasks, as well as primitive for services orchestration. Some of these commands correspond to predefined ASM rules whose semantics have been precisely defined in terms of ASMs [28] and whose AsmetaL implementation is provided as external library CommonBehavior to be imported as part of an SCA-ASM module (see the import section in listing 1).

1) Service internal behavior: Service tasks are modeled as ASM rules [2].
2) Service interaction: External services are invoked in a synchronous and asynchronous manner through the following primitives:

- **wsend**[lnk, R, snd]: sends data snd without blocking to the partner link lnk in reference to the service operation R (no acknowledgment is expected).
- **wrecv**[lnk, R, rcv]: receives data in the location rcv from the partner link lnk in reference to the service operation R; it blocks until data are received. No acknowledgment is expected.
- **wsendrecv**[lnk, R, snd, rcv]: in reference to the service operation R, some data snd are sent to the partner link lnk, then the action waits for data to be sent back, which are stored in the receive location rcv; no acknowledgment is expected for send and receive.
- **wreplay**[lnk, R, snd]: returns 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.

These primitives, mainly inspired by the UML4SOA, correspond to the invocation of predefined ASM rules defined in [28] as “wrappers” of high-level communication patterns, originally presented in [29], 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 an abstract message passing mechanism.

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 [29]). They will be considered for future extension.

3) Workflow management: Service activities (i.e. ASM rules invocations) can be orchestrated in accordance with a workflow expressible by the following constructs⁴:

- Conditional behavior: **if** cond **then** R1 **else** R2 to select exactly one activity for execution from alternative choices.
- Repetitive execution: **while** cond **do** R to repeat execution of an activity R as long as the Boolean condition cond evaluates to true at the beginning of each iteration.
- Sequential processing: **seq** R1 R2 ... Rn **enseq** to perform a collection of activities R1, R2, ... Rn in sequential order.
- Parallel processing: **par** R1 R2 ... Rn **endpar** to perform a collection of activities in a synchronous parallel way⁵.

- Multiple branch processing: **forall** n ∈ N **do** R(n) to split N times the execution of the same activity R.
- Spawn of sub-threads: **spawn** child with R to create a child agent having activity R as program to execute.

4) Error and compensation handling: Fault and compensation handlings are strictly related. They require the execution of specific activities (attempting) to reverse the effects of previously executed activities. The mechanism described here is mainly inspired by the UML4SOA.

The behavior of an exception handler for an activity RA 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(RA) is used, within the initialization rule for a given component, to associate a component service operation RA with its exception handler. To raise an exception when a fault occurs, the predefined rule raiseException[a, RA, msg] is invoked to put the agent a in exception mode, expose a possible error message msg (if any), and lunch the rule exceptionHandler(RA).

As exemplification of the error handling mechanism, consider the rule failedLogin, in listing 1, acting as error handler for the activity login – according to the value of the function exceptionHandler in the rule init. The exception is raised by executing the rule raiseException inside the service rule login.

The mechanism for compensation handling is treated similarly. The annotation **@Compensate** is used to mark a rule acting as compensation handler of a given activity RA. This last is associated with its handler by the function compensationHandler(RA) settled in the component initialization rule. When a compensation for a service activity RA, already completed successfully, must be activated, the predefined rule compensate[a, RA] is invoked to put the agent a in compensation mode and lunch the rule compensationHandler(RA).

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

5) Component Life Cycle: SCA-ASM supports a simple component life cycle. A component (agent) deployed and instantiated in an assembly may be in a state ranging in the set {init, ready, blocked, exited, compensation, exception}. The initial state of the component is init. The agent becomes ready when available to interact with other

⁴These language constructs provide the same expressiveness of the control-flow commands of WS-BPEL, 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.

⁵This parallel processing corresponds to the fork/join construct of other languages, which can be used to spawn finitely many sub-agents and merge the control flow again when all the parallel activities end. Asynchronous parallel split is not yet supported, although it can be provided by using the concept of asynchronous ASMs.
service components. It is blocked when data are expected upon service invocation. Compensation and exception modes refer to the agent’s activity of compensation and rollback. An agent puts itself at exited mode upon deferred termination. The functionality needed to manipulate the state of a component is implemented through those ASM rules specifying the semantics of the predefined commands of the SCA-ASM language regarding service invocation, components interaction, error and compensation handling.

**IV. Tool support and evaluation**

We implemented a tool\(^6\) that allows modelers to design, assemble, and execute SCA-ASM models of components in an unique integrated environment (see Fig. 4).

a) SCA-ASM tool overview: 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 ASM elements like component and interface implementation. Fig. 4 shows a screenshot of the tool. Appropriate ASM icons (see the right side of Fig. 4) 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. 4) and then dragged and dropped on the components and interfaces of the SCA assembly diagram.

The back-end is the Apache Tuscany SCA runtime \(^7\) – to run and test SCA assemblies of components developed with different implementation technologies and spread across a distributed environment (cloud and enterprise infrastructures) – combined with the ASMETA toolset to support various forms of high-level functional analysis. In particular, we extended the Tuscany platform to allow the execution of ASM models of SCA components through the simulator ASMETA/AsmetaS (as shown by the console output shown in Fig. 4) within Tuscany.

b) Formal functional analysis scenarios and case studies: SCA-ASM makes it possible to specify abstract components, to compose them, and to simulate them and check various functional properties with the help of the ASMETA analysis toolset and of the Tuscany platform.

The following functional analysis scenarios are supported. Offline analysis: First, designers are able to exploit first the functionality of the ASMETA analysis toolset (also based on the Eclipse environment) to validate and verify SCA-ASM models of components in an off line manner, i.e. ASM models of such abstract (or mock) components may be analyzed in isolation to determine if they are fit for use. As analysis techniques, the ASMETA toolset includes simulation, scenario-based simulation, model-based testing and model checking.

In-place simulation: Then, an in-place simulation scenario may be also carried out to execute early the behavior of the overall composite application. In this case, the designer can exploit the functionality of the AsmetaS simulator directly within the SCA runtime platform to execute the ASM specification (intended as abstract implementation) of mock components together with the other real and heterogeneous (non ASM-implemented) components according to the chosen SCA assembly.

In addition, the validated and verified SCA-ASM models can be eventually reused in the future as oracles, when the real implementation of those components is available, to perform conformance analysis (or model-based testing) and run-time monitoring.

Several case studies of varying sizes and covering different uses of the SCA-ASM constructs have been developed. These include a Robotics task coordination case study [23] of the EU project BRICS [6] and a scenario of the Finance case study of the EU project SENSORIA [7]. This last is a credit (web) portal application of a credit institute that allows customer companies to ask for a loan to a bank. Fig. 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 [15] 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 [15]. It should be also noted that the functional analysis of this case study is out of the scope of this paper.

Listings 1 and 2 report the ASM specification of the SCA PortalServiceComponent (or simply Portal) and its provided service interface, respectively. 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

---

\(^6\)https://asmeta.svn.sourceforge.net/svnroot/asmeta/code/experimental/SCAASMS

\(^7\)http://tuscany.apache.org/
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. 4) that terminates the process (as denoted by the status of the Portal agent that is set to exception). Portal also receives the customer’s choice about the desired service (here we only consider the service CREDIT_REQUEST) and invokes the service InformationUpload by sending it a message with the requestID. From then on, the customer communicates with the InformationUpload.

V. CONCLUSION AND FUTURE WORK

We presented a practical approach that combines the SCA open standard model for service assembly and the ASM formal support to tackle the complexity of service-oriented applications by offering a high degree of design and validation at early development phases. The language permits to express service-oriented components assemblies, as well as internal service and service orchestration behavior. The language is supported by a tool that exploits the SCA runtime Tuscany and the toolset ASMETA for system model execution and analysis. The effectiveness of the language was experimented through various case studies of different complexity and heterogeneity.

We plan to support more useful SCA concepts, such as the SCA callback interface for bidirectional services. 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. We want also to enrich the notation with interaction and workflow patterns based on the BPMN specification and with specific actions to support an event-based style of interaction where the components of a distributed system communicate via events which are generated by one component and received by others through a publish/subscribe schema, including service publication, discovery and negotiation. Moreover, we aim at addressing self-adaptation issues, both at structural level (as addition/substitution of components) and at behavioral level (by modifying components interactions).

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. We also plan to extend the language with pre/post-conditions defined on services (transition rules) for contract correctness checking in component assemblies. Through the SCA Policy Framework, we want also to enrich service descriptions with non-functional properties (such as availability, reliability, etc.) that jointly represent the quality of the service.

REFERENCES

Listing 1. ASM implementation of the SCA Portal component

```java
module PortalServiceComponent
import STDL/StandardLibrary, STDL/CommonBehavior
//@Provided
import PortalService
//@Required
import InformationUploadService, AuthenticationService,
                 CustomerService
export *
signature:
//@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) = skip

//@Service
rule r_login($user in String, $pwd in String) =
  seq
  r_sendReceive[authenticationService(self), "$authentication(AGENT,STRING,String)", ($user,$pwd,valid)] if (valid)
  then seq
  sessionId:=sessionId+1
  r_send[customerService(self), "$r_login(AGENT,STRING,Integer)", ($user,$pwd)]
  r_raiseException[false,"r_login","Login_failed"]
  endif
endseq
endif
else if (not(valid)) // valid can still be undef
  then seq
  r_send[customerService(self), "$r_failedLogin(AGENT,STRING,Integer)", $user]
  endif
endseq
endpar

//@Service
rule r_selectService($sessionId in Integer, $service in String) =
  seq
  if ($service="CREDIT"
    then seq
    if isDef(inputPortal(self))
      then seq
      r_request[inputPortal(self), "r_login(String,Integer)", inputPortal(self)]
      r_wait[yield[sessionId]]
      endif
    endif
    else if (not(isDef(inputPortal(self))))
      then seq
      r_request[inputPortal(self), "r_selectService(String,Integer)", inputPortal(self)]
      endif
    endif
  endif
endseq
endpar
```

Listing 2. ASM definition of the PortalService interface

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