A formal framework for coordinated simulation of heterogeneous service-oriented applications[*]

D. Brugali[2] L. Gherardi[2] E. Riccobene[1] P. Scandurra[2]

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

Abstract. Early design and validation of service-oriented applications is hardly feasible due to their distributed, dynamic, and heterogeneous nature. In order to support the engineering of such applications and discover faults early, foundational theories, modeling notations and analysis techniques for component-based development should be revisited. This paper presents a formal framework for coordinated simulation of service-oriented applications based on the OSOA open standard *Service Component Architecture* (SCA) for heterogeneous service assembly and on the formal method *Abstract State Machine* (ASM) for modeling notions of service behavior, interactions, and orchestration in an abstract but executable way. The proposed framework is exemplified through a Robotics Task Coordination case study of the EU project BRICS.

1 Introduction

Service-oriented applications are playing so far an important role in several application domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few) since they offer complex and flexible functionalities in widely distributed environments by composing, possibly dynamically "on demand", different types of services. Web Services is the most notable example of technology for implementing such components. On top of these service-oriented components, business processes and workflows can be (re-)implemented as composition of services – *service orchestration* or *service coordination*[3]. Examples of composition languages are WS-BPEL[4], XLANG[5], and Jolie [29], to name a few.

This emerging paradigm raises a bundle of problems, which did not exist in traditional component-based design, where abstraction, encapsulation, and modularity were the main concerns. Early designing, prototyping, and testing of the

[*] The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics)
[3] Thorough this paper, the terms coordination and orchestration are interchangeable.
[4] www.oasis-open.org
[5] www.ebpml.org/xlang.htm

functionality of such assembled service-oriented applications is hardly feasible since services are discoverable, loosely-coupled, and heterogeneous (i.e. they differ in their implementation/middleware technology) components that can only interact with others on compatible interfaces. Concurrency and coordination aspects [4] that are already difficult to address in component-based system design (though extensively studied), are even more exacerbated in service-oriented system design. Components encapsulate and hide to the rest of the system how computations are ordered in sequential threads and how and when computations alter the system state. The consequence of improper management of the order and containment relationships or the total absence of an explicit coordination model in a complex, concurrent system leads to deadlock and starvation [17].

In order to support the engineering of service-oriented applications, to discover faults early, and to improve the service quality (such as efficiency and reliability), foundational theories and high-level formal notations and analysis techniques traditionally used for component-based systems should be revisited and integrated with emerging service development technologies. In the Robotics context, in particular, as the Internet is leveraged to connect humans to robots and robots to the physical world, there is a strong requirement to investigate service-oriented engineering approaches and knowledge representations to effectively distribute the capabilities offered by robots: *service-oriented robots* [**?**].

This paper proposes a formal framework for coordinated simulation of heterogeneous service-oriented applications. It relies on the *SCA-ASM* language [31] that combines the OSOA open standard model *Service Component Architecture* (SCA) [28] for heterogeneous service assembly in a technology agnostic way, with the formal method *Abstract State Machine* (ASM) [11] able to model notions of service behavior, interactions, and orchestration [9, 7, 10] in an abstract but executable way. A designer may use the proposed framework to provide abstract implementations in SCA-ASM of (i) *mock components* (possibly not yet implemented in code or available as off-the-shelf) or of (ii) *core components* containg the main service composition or process that coordinates the execution of other components (possibly implemented using different technologies) providing the real computation. He/she can then validate the behavior of the overall assembled application, by configuring these SCA-ASM models *in place* within an SCA-compliant runtime platform as implementation of (mock or core) components and then execute them together with the other (local or remote) components implementations according to the chosen SCA assembly.

We, in particular, show the usage of our framework through a Robotics Task Coordination scenario from a case study [26] of the EU project BRICS [12]. 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 [14].

ASMs provide a general method to combine specifications on any desired level of abstraction, ground modeling (requirements capture) techniques and stepwise refinement to executable code providing the basis for experimental validation and mathematical verification [11]. ASM rigorousness, expressiveness, and executability allow for the definition and analysis of complex structured services interaction protocols in a formal way but without overkill. Moreover, the ASM design method is supported by several tools [21, 5], useful for validation and verification of ASM-based models of services.

This paper is organized as follows. Section 2 provides background on SCA and ASMs. Section 3 presents the Robotics Task Coordination case study that will be used throughout the paper. Section 4 describes the proposed framework for coordinated simulation of service-oriented applications. Section 5 describes some related works, while Section 6 reports our lesson learned in developing the case study. Finally, Section 7 concludes the paper and sketches some future work.

2 Background on SCA and ASMs

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

Fig. 1 shows an *SCA composite* (or *SCA assembly*) as a collection of SCA components. Following the principles of SOA, loosely coupled service components are used as atomic units or building blocks to build an application.

An *SCA component* is a piece of software that has been configured to provide its business functions (operations) for interaction with the outside world. This interaction is accomplished through: *services* that are externally visible functions provided by the component; *references* (functions required by the component)

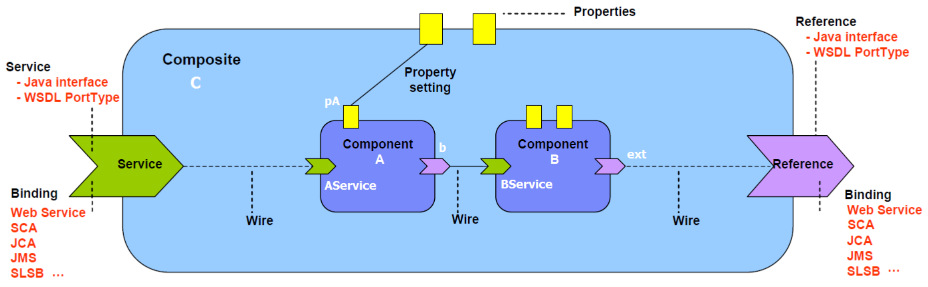


Fig. 1. An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)

wired to services provided by other components; *properties* allowing for the configuration of a component implementation with externally set data values; and *bindings* that specify access mechanisms used by services and references according to some technology/protocol (e.g. WSDL binding to consume/expose web services, JMS binding to receive/send Java Message Service, etc.). Services and references are typed by *interfaces*. An interface describes a set of related 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. These messages may be returned synchronously or asynchronously.

Assemblies of service components deployed together are supported in terms of *composite* components consisting of: properties, services, service implementations organized as sub-components, required services as references, and wires connecting sub-components.

Abstract State Machines ASMs are an extension of FSMs [11] where unstructured control states are replaced by states comprising arbitrary complex data. The *states* of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them. The *transition relation* is specified by rules describing how functions change from one state to the next. There is a limited but powerful set of ASM *rule constructors*, but the basic transition rule has the form of *guarded update* "**if** *Condition* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed[6] when *Condition* is true.

Dynamic functions are those changing as a consequence of agent actions (or *updates*). They are classified as: *monitored* (only read, as events provided by the environment), *controlled* (read and write), *shared* (read and write by an agent and by the environment or by another agent) and *out* (only write) functions.

Distributed computation can be modeled by means of *multi-agent ASMs*: multiple agents interact in parallel in a synchronous/asynchronous way. Each agent's behavior is specified by a basic ASM. The predefined variable (or 0-ary function) *self* can occur in the model and is interpreted by each agent as itself.

Besides ASMs comes with a rigorous mathematical foundation [11], ASMs can be read as pseudocode on arbitrary data structures, and can be defined as the tuple (*header*, *body*, *main rule*, *initialization*): *header* contains the *signature*[7] (i.e. domain, function and predicate declarations); *body* consists of domain and function definitions, state invariants declarations, and transition rules; *main rule* represents the starting point of the machine program (i.e. it calls all the other ASM transition rules defined in the body); *initialization* defines initial values for domains and functions declared in the signature.

[6] f is an arbitrary n-ary function and $t_1, \ldots, t_n, t$ are first-order terms. To fire this rule in a state S_i, $i \geq 0$, 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.

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

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

A lightweight notion of module is also supported. An *ASM module* is an ASM (*header*, *body*) without a main rule, without a characterization of the set of initial states, and the body may have no rule declarations.

An open framework, the *ASMETA tool set* [5], based on the Eclipse/EMF platform and developed around the *ASM Metamodel*, is also available for editing, exchanging, simulating, testing, and model checking models. *AsmetaL* is the textual notation to write ASM models within the ASMETA tool-set.

The SCA-ASM modeling language By adopting a suitable subset of the SCA standard for modeling service-oriented components assemblies and exploiting the notion of *distributed multi-agent ASMs*, the SCA-ASM modeling language [31] complements the SCA component model with the ASM model of computation to provide ASM-based formal and executable description of the services internal behavior, services orchestration and interactions. According to this implementation type, a service-oriented component is an ASM endowed with (at least) one agent (a business partner or role instance) able to be engaged in conversational interactions with other agents by providing and requiring services to/from other service-oriented components' agents. The service behaviors encapsulated in an SCA-ASM component are captured by ASM transition rules.

The ASM rule constructors and predefined ASM rules (i.e. named ASM rules made available as model library) used as basic SCA-ASM behavioral primitives are recalled in Table 1 by separating them according to the separation of concerns *computation*, *communication* and *coordination*. In particular, communication primitives provide both synchronous and asynchronous interaction styles (corresponding, respectively, to the *request-response* and *one-way* interaction patterns of the SCA standard). Communication relies on a dynamic domain *Message* that represents message instances managed by an *abstract message-passing* mechanism: components communicate over wires according to the semantics of the communication commands reported above and a message encapsulates information about the partner link and the referenced service name and data transferred. We abstract, therefore, from the SCA notion of *binding*[8].

Fault/compensation handling is also supported (see [31]), but their SCA-ASM constructs are not reported here since they are not used in the case study.

[8] Indeed, we adopt the default SCA binding (`binding.sca`) for message delivering, i.e. the SOAP/HTTP or the Java method invocations (via a Java proxy) depending if the invoked services are remote or local, respectively.

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

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 **endif**	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
COMMUNICATION		
Send rule	**wsend**[*lnk,R,snd*]	send data *snd* to *lnk* in reference to rule R (no blocking, no acknowledgment)
Receive rule	**wreceive**[*lnk,R,rcv*]	receive data *rcv* from *lnk* in reference to R (blocks until data are received, no ack)
SendReceive rule	**wsendreceive** [*lnk,R,snd,rcv*]	send data *snd* to *lnk* in reference to R waits for data *rcv* to be sent back (no ack)
Reply rule	**wreplay**[*lnk,R,snd*]	returns data *snd* to *lnk*, as response of R request received from *lnk* (no ack)

3 Running case study: a robotics tasks coordination

We propose a simple scenario where a laser scanner offers its scan service to different clients, which compete for the use of this shared resource. The scenario is defined by three participants:

– A *Laser Scanner*, which executes scans of the environment on demand and writes the acquired values on a data buffer. A scan is a sequence of measures executed in a single task (for example 360 values, one for each degree). We suppose that the Laser Scanner allows its client to request a scan from an initial angle (start) to a finale one (end) defined as the number of steps between start and end.
– A *3D Perception application*, which requests the measures to the Laser Scanner in order to generate a set of meshes that describe the surface of the objects present in the environment.
– An *Obstacle Avoidance application*, which requests the measures to the Laser Scanner in order to detect the obstacles along the robot path.

The proposed scenario is subjected to the following requirements:

1. The laser scan activity requires a certain amount of time to be completed. This time is not fixed, and depends on the number of measures requested by the client. During this time the client could have the need of executing other activities and so it does not have to wait for the scan termination.
2. A client could request a single scan or multiple scans (for example 4 scans composed each one by 20 measures).
3. While the Laser Scanner is executing a scan requested by a client A, a client B could require another scan. These requests have to be managed according to one of the following policies:
   – Policy 1: Discard the scan request.
   – Policy 2: Queue the scan request.

Moreover, it is assumed that different clients could simultaneously access to the services offered by the Laser Scanner and that client requests are asynchronous, i.e. the client requests a scan to the Laser Scanner and then it continues to execute its work. In this case the interactions between the clients and the Laser Scanner have to be managed by a third part: a coordinator. This coordinator, *Sensor 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.

High-level solution In order to keep the example simple to expose, we assume in this paper[9] to address only the request management policy 1, i.e. if a request is received while the laser is already scanning the new request will be discarded. With this assumption, the Sensor Coordinator behavior can be captured, as first high-level model, by the finite state machine shown in Fig. 2.

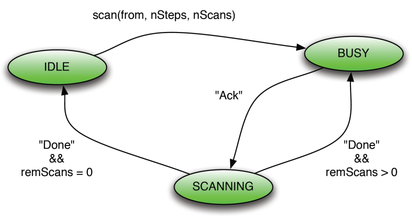


Fig. 2. Sensor Coordinator FSM

Essentially, the Sensor Coordinator receives a request of one or n scans from a client. According to the followed policy (see above) the new request could be discarded, or queued or forwarded (the normal case) to the Laser Scanner. When the request is forwarded, the Laser Scanner starts the scanning work and sends a notification (*Ack*) to the Sensor Coordinator in order to inform it that the scan has started. Depending on the number of scan requested, the Sensor Coordinator will forward to the Laser Scanner one or more single scans. In case of multiple scans, the Sensor Coordinator will forward n single scan requests to the Laser Scanner (to this purpose, the count variable *remScans*, initially set to n, is used and decremented at each forward). The Laser Scanner then writes each measure on the Measures Buffer until the final angle is reached, and it finally sends a notification (*Done*) to the Sensor Coordinator in order to inform it that the scan is finished. At this point, if there are not remaining scans to execute (*remScans* is equal to 0) it sends a notification to the client in order to inform it that the new measures are available on the Buffer. The client then can access the Measures Buffer to read the measures.

[9] Details on different variants of this scenario can be found in [26].

SCA modeling The application is heterogeneous: by the icons attached to components, the Sensor Coordinator is implemented in ASM, while the other two components in Java. The clients are considered external entities interacting with the Sensor Coordinator and with the Measures Buffer through the services offered (promoted) by the composite. More precisely, a client could request a scan by means of the service *SensorCoordinating* and could access the Measures Buffer by means of the service *MeasuresBufferReading*.

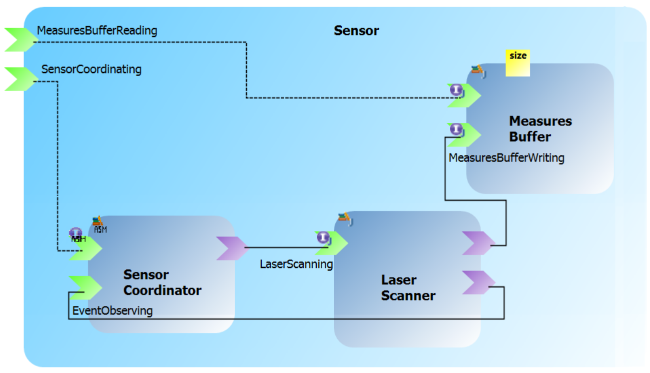


Fig. 3. The Sensor Composite

The definition of the service interfaces is reported in the listing 1.1 using the Java interface construct as IDL (Interface Definition Language). Note that, the interface *EventObserving* is implemented by the Sensor Coordinator to manage the notification received from the Laser Scanner[10].

The ASM (abstract) implementation of the SCA Sensor coordinator's behavior will be provided later in Sect. 4.1. For the sake of space, the Java implementation code of the other components is not reported.

Listing 1.1. Service interfaces definition in Java

```
public interface MeasuresBufferReading { public LaserScan getScan(); }
public interface MeasuresBufferWriting { public void writeMeasure(LaserMeasure measure); }
public interface LaserScanning {
    /**@param from: point from which the laser starts the scan
     * @param numOfSteps: number of steps of the scan */
    @OneWay public void scan(int from, int numOfSteps); }
public interface SensorCoordinating {
    /**@param from: point from which the laser starts the scan
     * @param numOfSteps: number of steps of the scan
     * @param numOfScans: number of scans required */
    @OneWay public void request(int from, int numOfSteps, int numOfScans); }
public interface EventObserving {
    /**@param event: it describe the type of event.
     * For the laser scanner valid values are "Ack" and "Done"
     */
    public void update(String event); }
```

[10] So far it is used as a service to resemble a callback (not yet supported in SCA-ASM).

4 Coordinated simulation framework

The proposed framework relies on the SCA-ASM language originally presented in [31] as a formal and abstract *component implementation type* to cover *computation*, *communication*, and *coordination* aspects during early execution (or simulation) of an SCA assembly of an heterogeneous service-oriented application. ASMs 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. The framework relies also on other SCA component implementation types (such as Java, Spring, C++, etc., see [28]) to include components providing the real computation services used by the core component(s) and these components can themselves require services provided by other local or remote components.

The framework was developed by integrating the Eclipse-based SCA Composite Designer, the SCA runtime platform Tuscany [34], and the simulator AsmetaS of the ASM toolset ASMETA [5]. This environment[11] allows us to graphically model, compose, analyze, deploy, and execute heterogeneous service-oriented applications in a technologically agnostic way. As described and exemplified below, an heterogeneous SCA assembly (or composition) of service-oriented components (implemented in ASM or in another implementation language) can be graphically produced using the SCA Composite Designer and also stored or exchanged in terms of an XML-based configuration file. This last file is then used by the SCA runtime to instantiate and execute the system by instrumenting AsmetaS and other execution infrastructures in an unique environment (see Fig. 5).

4.1 Service component implementation and configuration

Through the considered case study, we here show the use of the *ASM implementation type* (i.e. of the SCA-ASM language) for SCA components. This implementation type provides a behavioral specification that is essential for performing early prototyping, functional simulation and formal verification to prove (verifiable) correctness and reliability of both single services and the overall assembly.

Service component implementation The following listings report the ASM (abstract) implementation of the Sensor Coordinator component (request management policy 1). To this purpose, the AsmetaL textual notation to write ASM models within the ASMETA tool-set is used. Two grammatical conventions must be recalled: a variable identifier starts with $; a rule identifier begins with "*r_*".

Listing 1.2 shows the header of the ASM. The import clauses include the ASM modules of the provided service interfaces (SensorCoordinating and EventObserving) and required interfaces (the LaserScanning interface) of the component, annotated, respectively, with `@Provided` and `@Required`. The `@MainService` annotation, when importing the SensorCoordinating interface, denotes the main service (read: main component's agent) that is responsible for initializing the

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

component's state (in the predefined `r_init` rule). The signature of the machine contains declarations for: references (shared functions annotated with *@Reference*), which are abstract access endpoints to services, back references to requester agents (shared functions annotated with *@Backref*), and declarations of ASM domains and functions, which are used by the component for internal computation only. In particular, the variable (a controlled 0-ary function) `ctl_state` stores the current control state of the ASM.

Listing 1.2. ASM header of the Sensor Coordinator component

```
module SensorCoordinator
import STDL/StandardLibrary
import STDL/CommonBehavior
//@MainService
import SensorCoordinating
//@Provided
import EventObserving
//@Required
import LaserScanning
export *
signature:
//@Reference
shared laserScanning : Agent −> LaserScanning
//@Backref
shared clientSensorCoordinating : Agent −> Agent
//@Backref
shared clientEventObserving : Agent −> Agent
enum domain State = {IDLE | BUSY | SCANNING}
//Internal properties
controlled ctl_state : Agent −> State //stores the current control state
controlled paramScan : Agent −> Prod(Integer,Integer,Integer) //arguments of an scan request
controlled from : Agent −> Integer //stores the start position of an scan request
controlled steps : Agent −> Integer //stores the number of measures of an scan request
controlled remScans : Agent −> Integer //stores the number of scans requested by a client
controlled event : Agent −> String //stores the argument of an update request.
```

The body of the ASM (see Listing 1.3) includes definitions of the services (transition rules annotated with `@Service`) `r_request` and `r_update`, the main transition rule `r_SensorCoordinator` (that takes by convention the same name of the component), the transition rule with the predefined name `r_init` that is invoked in the initialization to set up the internal component state (i.e. values of controlled functions), and another utility rule named `r_acceptRequest`.

The `r_request` service is in charge of requesting a scan to the laser scanner. When the rule is called, it executes in parallel the following actions: sets the state of the ASM to *BUSY*, stores the arguments of the requested scan, invokes (by a send action) the service *scan*, which is provided by the service Laser Scanning.

The `r_update` service is in charge of receiving the notification from the laser scanner and updating the control state by resembling the FSM shown in Fig. 2.

The `r_acceptRequest` rule advances the control state of the machine properly according to the arriving service requests (the input parameter `$r`). In case of a new scan request (*r_request*), this is removed from the requests queue (by the `r_wreceive` action) and the input is stored in the variable `paramScan`. A direct invocation of the `r_request` service then follows if the input is defined. In case, instead, of a notification (`r_update`) from the laser scanner, the request is removed from the requests queue (by the `r_wreceive` action) and in case the input (stored in the variable `event`) is defined the service `r_update` is invoked. Note that all the scan requests received while the scanner is already scanning are discarded (what the policy 1 defines).

Listing 1.3. ASM body of the Sensor Coordinator component

```
definitions:
//State invariant: Number of scans required by a client must be non negative
invariant inv_neverNeg over remScans(): not(remScans < 0)
//@Service
rule r_request($a in Agent,$from in Integer,$steps in Integer, $nScans in Integer)=
  par
    ctl_state($a) := BUSY
    from($a) := $from
    steps($a) := $steps
    remScans($a) := $nScans − 1
    r_wsend[laserScanning($a),"r_scan(Agent,Integer,Integer)",($from,$steps)]
  endpar

//@Service
rule r_update($a in Agent, $event in String) =
  if (ctl_state($a)=BUSY and $event="Ack")
  then ctl_state($a) := SCANNING
  else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)>0)
    //continue with next scan
    then par
      ctl_state($a) := BUSY
      remScans($a) := remScans($a)−1
      r_wsend[laserScanning($a),"r_scan(Agent,Integer,Integer)",(from($a),steps($a))]
    endpar
  else if (ctl_state($a)=SCANNING and $event="Done" and remScans($a)=0)
  then ctl_state($a) := IDLE endif endif endif

rule r_acceptRequest ($a in Agent, $r in String) =
  if (ctl_state($a)=IDLE and $r="r_request(Agent,Integer,Integer,Integer)")
  then
   seq //first scan
    r_wreceive[clientSensorCoordinating($a),"r_request(Agent,Integer,Integer,Integer)",paramScan(
         $a)]
    if (isDef(paramScan($a))) then
        r_request[$a,first(paramScan($a)),second(paramScan($a)),third(paramScan($a))] endif
   endseq
  else if (not ctl_state($a)=IDLE and $r="r_update(Agent,String)")
    then
     seq
      r_wreceive[clientEventObserving($a),"r_update(Agent,String)",event($a)]
      if (isDef(event($a))) then r_update[self,event($a)] endif
     endseq
  endif endif

//Main agent's program
rule r_SensorCoordinator =
  let($r = nextRequest(self)) //Select the next request(if any)
  in if isDef($r) //Handle the request $r
      then r_acceptRequest[self,$r] endif
  endlet

rule r_init($a in SensorCoordinating) = //for the startup of the component
  par
    status($a) := READY
    ctl_state($a) := IDLE
    from($a) := 0
    steps($a) := 0
    remScans($a) := 0
  endpar
```

The `r_SensorCoordinator` rule is the program of the main component's agent and is invoked every times a client requests a service offered by the Sensor Coordinator. This rule simply forwards the request to the `r_acceptRequest` rule.

Finally, the `r_init` rule is called during initialization of the component execution. This rule simply sets the status of the agent to *READY*, the control state to *IDLE* and initializes the scan parameters to 0.

The ASM definitions of the sensor coordinator's provided interfaces are reported in the listing 1.4 using the AsmetaL notation. They are ASM modules containing only declarations of business agent types (subdomains of the predefined ASM domain `Agent`), and of business functions (ASM out functions).

Listing 1.4. ASM definition of the Sensor Coordinating interface

```
//@Remotable
module SensorCoordinating
import STDL/StandardLibrary
import STDL/CommonBehavior
export *
signature:
// the domain defines the type of this agent
domain SensorCoordinating subsetof Agent
// out is a function that implements the provided service
out request: Prod(Agent,Integer,Integer,Integer) −> Rule
definitions:

//@Remotable
module EventObserving
import STDL/StandardLibrary
import STDL/CommonBehavior
export *
signature:
domain EventObserving subsetof Agent
out update: Prod(Agent,String) −> Rule
definitions:
```

Listing 1.5. XML configuration file

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<sca:composite xmlns:sca="http://www.osoa.org/xmlns/sca/1.0" xmlns:asm="http://asm"
name="Sensor" targetNamespace="http://eclipse.org/CaseStudy/src/Sensor">
  ...
  <sca:component name="SensorCoordinator">
    <asm:implementation.asm location="SensorCoordinator.asm"/>
    <sca:reference name="laserScanning"/>
    <sca:service name="SensorCoordinating">
      <asm:interface.asm location="SensorCoordinating.asm"/>
    </sca:service>
  </sca:component>
  ...
</sca:composite>
```

Service component configuration Component metadata, describing which services are required and provided by a component, and information that allow the SCA runtime to locate (locally or remotely) the component implementation, must be provided in the SCA XML composite file. Listing 1.5 shows a fragment of the SCA XML composite file regarding the metadata of the Sensor Coordinator component that is implemented (by the tag `implementation.asm`) in ASM.

4.2 In-place simulation of SCA-ASM models

SCA-ASM components use annotations to denote services, references, properties, etc. With this information, as better described below, an SCA runtime platform (Tuscany in our case) can create a composition (an application) by tracking service references (i.e. required services) at runtime and injecting required services into the component when they become available.

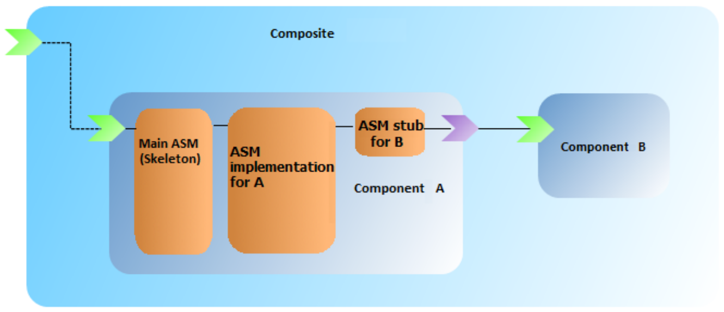


Fig. 4. Instantiating and invoking ASM implementation instances within Tuscany

In-place ASM execution mechanism. Fig. 4 illustrates how the *ASM implementation provider*[12] sets up the environment (the container) within Tuscany for instantiating and handle incoming/outgoing service requests to/from an ASM component implementation instance (like component `A` in the figure) by instrumenting the ASM simulator AsmetaS. Currently, the implementation scope of an SCA-ASM component is *composite*, i.e. a single component instance – a single *main ASM* instance (see the main ASM for component `A` in Fig. 4) – is created within AsmetaS for all service calls of the component[13]. This main ASM is automatically created during the setting up of the connections and it is responsible for instantiating the component agent and related resources, and for listening for service requests incoming from the protocol layer and forward them to the component'agent instance (see component `A` in Fig. 4). Executing an ASM component implementation means executing its main ASM. For each reference, another entity (i.e. another ASM module) is automatically created (and instantiated as ASM agent within the main ASM of the component) as "proxy" for a remote component (see the ASM proxy for component `B` in Fig. 4) for making an outbound service call from the component. Using a terminology adopted in

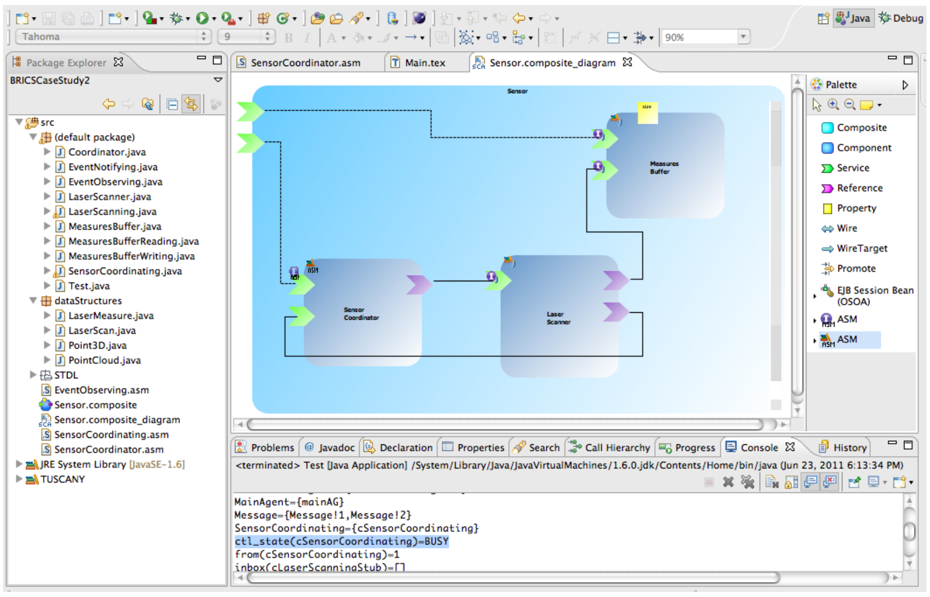
[12] The Tuscany core delegates the start/stop of component implementation instances and related resources, and the service/reference invocations, to specific *implementation providers* that typically respond to these life-cycle events.

[13] 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).

the Java Remote Method Invocation (RMI) API, this proxy ASM plays the role of *stub* to forward a service invocation (and their associated arguments) to an external component's agent, and to send back (through the ASM rule `r_replay`) the result (if any) to the invoker component'agent (the agent of the component `A` in Fig. 4). The main ASM, instead, plays the role of *skeleton*, i.e. a proxy for a remote entity that runs on the provider and forward (through the ASM rule `r_sendreceive`) client's remote service requests (and their associated arguments) to the appropriate component's agent (usually the main agent of the component), and then the result (if any) of the invoked service is returned to the client component'agent (via stubs). For the sake of space, the ASM implementation of the stub and skeleton (as generated by the runtime) for the component Sensor Coordinator is not reported.

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

Fig. 5 shows a simulation snapshot of the considered case study where the Sensor Coordinator changes state from `IDLE` to `BUSY` (see also the rule `r_request` in the Listing 1.3) after receiving a first scan request from a client.



```
<terminated> Test [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jun 23, 2011 6:13:34 PM)
MainAgent={mainAG}
Message={Message!1,Message!2}
SensorCoordinating={cSensorCoordinating}
ctl_state(cSensorCoordinating)=BUSY
from(cSensorCoordinating)=1
inbox(cLaserScanningStub)=[]
```

Fig. 5. Simulation of the Sensor Composite application

Other ASM execution 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 `InvalidInvariant-Exception`, which keeps track of the violated invariant. Listing 1.3 shows an example of state invariant (`inv_neverNeg`) for the Sensor Coordinator. It states that the number of scans required by a client must be non negative.

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 also inspect how the simulator performs particular tasks (including the evaluation of terms, the building of update set for rules, and the variables substitution) by providing a log4j configuration file.

5 Related work

Some works devoted to provide software developers with formal methods and techniques tailored to the service domain also exist (see, e.g., the survey in [8] for the service composition problem), mostly developed within the EU projects SENSORIA [32] and S-Cube [27]. Several process calculi for the specification of SOA systems have been designed (see, e.g., [22, 24, 15]). They provide linguistic primitives supported by mathematical semantics, and verification techniques for qualitative and quantitative properties [32]. Still within the SENSORIA project, a declarative modeling language for service-oriented systems, named SRML [33], has been developed. SRML supports qualitative and quantitative analysis techniques using the UMC model checker[1] and the PEPA stochastic analyzer[14].

Compared to the formal notations mentioned above, the ASM method has the advantage to be executable. On the formalization of the SCA component model, some previous works, like [18, 19] 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 [6, 3] and is supported by a prototype tool (COSTO). In the Kmelia model, services are used as composition units and services behavior is modeled by labeled transition systems. Our proposal is similar to the Kmelia approach;

[14] http://www.dcs.ed.ac.uk/pepa/

however, we have the advantage of having integrated our SCA-ASM component model and the ASM-related tools with an SCA runtime platform for a more practical use and an easier adoption by developers.

Within the ASM community, the ASMs have been used for the purpose of formalizing business process notations and middleware technologies related to web services, such as [16, 9, 20, 2] to name a few. Some of these previous formalization efforts, as explained in [31], are at the basis of our work.

Concerning the Robotics domain, [23] proposes a new approach for coordinating the behavior of Orocos RTT (Open Robot Control Software Real Time Toolkit) [30] components. Orocos RTT is a C++ framework, which allows the design and the deployment of component-based robotics control systems. The proposed approach defines the behavior of single components and of entire systems by means of a minimal variant of the UML hierarchical state-charts, which is called *reduced FSM* (rFSM). The main advantages of the rFSM are their hierarchical composability and their applicability in hard-real time applications. Furthermore, despite they are currently used only with Orocos, rFSM are totally framework independent. The main differences between the ASMs and the rFSMs are that rFSMs do not allow the execution of parallels agent actions and parallel states; moreover, they do not have the universality and broad application of the ASMs, and do not offer the same flexibility and tools provided by the ASMs.

6 Lesson learned

We have shown how formal high-level ASM models of service-oriented components can be assembled together with real components through the SCA framework and how we manage the coordination of the overall resulting application by means of the ASM formalism for prototyping and simulation purposes. We experienced that the use of two different frameworks for modeling two different concerns (SCA and its various implementation types for computation, and ASM for coordination) improves the level of flexibility and reusability.

We have shown this by means of a use case in the Robotics field, where flexibility and reusability are very challenging issues [12–14]. In general, robotics software require and provide a number of different functionalities, which are typically encapsulated in components that cooperate and compete in order to control the behavior of a robot. Cooperation and competition are forms of interaction among concurrent activities and so they have to be coordinated. In order to achieve a good level of reusability and flexibility the coordination and the computation (how the component provides the service) need to be managed separately. So by our experience, the service paradigm seems promising also in the Robotics domain. In particular, we appreciated the possibility to change the coordination policies (see [26]) without modifying the implementation of the services provided by components merely dedicated to computation (such as sophisticated algorithms), thus improving the level of flexibility and reusability.

7 Conclusion and future directions

We presented a practical framework for early service design and prototyping that combines the SCA open standard and the ASM formal support to assemble service-oriented components as well as intra- and inter- service behavior. The framework is supported by a tool that exploits the SCA runtime Tuscany and the toolset ASMETA for model execution and functional analysis. The effectiveness of our framework was experimented through various case studies of different complexity and heterogeneity. These include examples taken from the SCA Tuscany distribution, the case study of the EU project BRICS [12] presented here, and also a scenario of the *Finance* case study of the EU project SENSORIA [32].

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

On the functional analysis side, we plan to experiment the framework on large distributed application scenarios involving SCA assemblies deployed on various local and remote hosting nodes. We want also to try 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.

References

1. J. Abreu, F. Mazzanti, J. L. Fiadeiro, and S. Gnesi. A model-checking approach for service component architectures. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 219–224. Springer, 2009.
2. M. Altenhofen, A. Friesen, and J. Lemcke. ASMs in Service Oriented Architectures. *Journal of Universal Computer Science*, 14(12):2034–2058, 2008.
3. P. André, G. Ardourel, and C. Attiogbé. Composing components with shared services in the kmelia model. In C. Pautasso and É. Tanter, editors, *Software Composition*, volume 4954 of *LNCS*, pages 125–140. Springer, 2008.
4. F. Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science*, page 1122, Mar. 1998.
5. The ASMETA tooset website. `http://asmeta.sf.net/`, 2006.
6. C. Attiogbé, P. André, and G. Ardourel. Checking component composability. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2006.
7. A. P. Barros and E. Börger. A compositional framework for service interaction patterns and interaction flows. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.
8. M. T. Beek, A. Bucchiarone, and S. Gnesi. Formal Methods for Service Composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007.
9. E. Börger. Modeling Workflow Patterns from First Principles. In C. Parent, K.-D. Schewe, V. C. Storey, and B. Thalheim, editors, *ER*, volume 4801 of *LNCS*, pages 1–20. Springer, 2007.

10. E. Börger, O. Sörensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *J. UCS*, 15(1):3–32, 2009.
11. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer Verlag, 2003.
12. EU project BRICS (Best Practice in Robotics), `www.best-of-robotics.org/`.
13. D. Brugali and P. Scandurra. Component-based robotic engineering (Part I) [Tutorial. *IEEE Robotics & Automation Magazine*, 16(4):84–96, Dec. 2009.
14. D. Brugali and A. Shakhimardanov. Component-based robotic engineering part ii: Systems and models. *Robotics*, XX(1):1–12, 2010.
15. R. Bruni. Calculi for service-oriented computing. In M. Bernardo, L. Padovani, and G. Zavattaro, editors, *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 1–41. Springer, 2009.
16. E. Brger, O. Srensen, and B. Thalheim. On defining the behavior of or-joins in business process models. *Journal of Universal Computer Science*, 15(1):3–32, 2009.
17. J. S. Davis. *Order and containment in concurrent system design.* PhD thesis, Univ. of California, Berkeley, 2000.
18. Z. Ding, Z. Chen, and J. Liu. A rigorous model of service component architecture. *Electr. Notes Theor. Comput. Sci.*, 207:33–48, 2008.
19. D. Du, J. Liu, and H. Cao. A rigorous model of contract-based service component architecture. In *CSSE (2)*, pages 409–412. IEEE Computer Society, 2008.
20. R. Farahbod, U. Glässer, and M. Vajihollahi. A formal semantics for the business process execution language for web services. In S. Bevinakoppa, L. F. Pires, and S. Hammoudi, editors, *WSMDEIS*, pages 122–133. INSTICC Press, 2005.
21. ASMs web site. `http://www.eecs.umich.edu/gasm/`, 2008.
22. C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. : A calculus for service oriented computing. In A. Dan and W. Lamersdorf, editors, *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.
23. M. Klotzbuecher, P. Soetens, and H. Bruyninckx. OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages. In *International Workshop on Dynamic languages for RObotic and Sensors*, pages 284–289, 2010.
24. I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *SEFM'07*, pages 305–314. IEEE, 2007.
25. A. Martens and S. Moser. Diagnosing sca components using wombat. In S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2006.
26. EU project BRICS, Tech. Rep. A Coordination Use Case. March 24, 2011. `www.best-of-robotics.org/wiki/images/e/e0/coordinationusecaseubergamo.pdf`.
27. EU project S-Cube `http://www.s-cube-network.eu/`.
28. Service Component Architecture (SCA) `www.osoa.org`.
29. F. Montesi, C. Guidi, and G. Zavattaro. Composing services with jolie. In *ECOWS*, pages 13–22. IEEE Computer Society, 2007.
30. The Orocos Project. `http://www.orocos.org`.
31. E. Riccobene and P. Scandurra. A modeling and executable language for designing and prototyping service-oriented applications. In *EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA 2011)*.
32. EU project SENSORIA, `www.sensoria-ist.eu/`.
33. SRML: A Service Modeling Language. `http://www.cs.le.ac.uk/srml/`, 2009.
34. Apache Tuscany. `http://tuscany.apache.org/`.