A framework for early design and prototyping of service-oriented applications with design patterns

Steven Capelli, Patrizia Scandurra*

Università degli Studi di Bergamo, Dept. of Managment, Information and Production Engineering, Dalmine (BG), Italy

Abstract

Service-oriented computing is playing an important role in several domains. Today the biggest shift in mainstream design and programming is toward service-oriented applications. However, the service paradigm raises a bundle of problems that did not exist in traditional component-based development where abstraction, encapsulation, and modularity were the only main concerns. Due to their distributed, dynamic, and heterogeneous nature, service-oriented software applications require us to discover, document, and share new design patterns at the service- and architecture- level. Moreover, service-oriented applications are hard to design and validate, and demand for new foundational theories, modeling notations and analysis techniques.

In line to such a vision, this article presents a framework, called SCA-PatternBox, to design and prototype service-oriented applications with design patterns. The framework relies on the OASIS standard *Service Component Architecture* (SCA) and on SCA component implementation types, such as SCA-Java, for supporting an "implementation-oriented" approach to service-oriented architecture modeling and to the definition and instantiation of design patterns. Moreover, in order to provide formally verified design patterns, SCA-PatternBox allows the formal specification and analysis of the functional behavioral aspects of a design pattern using a formal service specification language called SCA-ASM (Service Component Architecture - Abstract State Machine). As major evaluation of the framework, two case studies and lessons learned are presented. A final comparison of existing design pattern languages is also reported.

Keywords: service modeling and prototyping, design pattern languages, service component architectures, formal pattern specification

*Corresponding author.

Email addresses: steven.capelli@unibg.it (Steven Capelli), patrizia.scandurra@unibg.it (Patrizia Scandurra)

1. Introduction

Today the biggest shift in mainstream programming and design is toward service-oriented applications. Service-oriented applications are playing so far an important role in several domains (e.g., information technology, health care, robotics, defense and aerospace, to name a few). Cloud service providers, in particular, are expanding their offerings to include the entire traditional IT stack, ranging from foundational hardware and platforms to application components, software services, and whole software applications.

Service-oriented Computing (SoC) is a paradigm for developing loosely-coupled, interoperable, dynamic systems relying on the basic unification principle that "*everything is a service*". *Services* are intended as loosely-coupled autonomous and heterogeneous[1] computational components that are offered by service providers in a distributed environment via publish/discovery protocols. The architectural foundation for SOC is provided by the *Service-Oriented Architecture* (SOA), which states that applications expose their functionality as services in a uniform and technology-independent way such that they can be discovered and invoked over a network and clouds.

Such a paradigm shift relies on interface-based design, composition, and reuse; but, differently from traditional component-based design where abstraction, encapsulation, and modularity were the only main concerns, the service paradigm raises a bundle of problems which did not exist previously. Early designing, prototyping, and testing of the functionality of such assembled service-oriented applications is hardly feasible since services are discoverable, loosely-coupled, and heterogeneous components that can only interact with others on compatible interfaces. Moreover, this paradigm shift requires us to discover, formally define, document, and share new *design patterns*. A SOA design pattern provides a solution in support of successfully applying service orientation and establishing a quality service-oriented architecture. SOA patterns describe common architectures, implementations, and their areas of application to help in the planning, implementation, deployment, management, and maintenance of complex systems. According to the SOA principles[1], design patterns play a fundamental role to support the engineering of service-oriented applications and the attainment of the strategic goals of SoC. In particular, a special emphasis has been put so far on the development of a catalog of *SOA design patterns*[2].

In line to such a vision, this article proposes a framework, SCA-PatternBox, for modeling service-oriented applications with design patterns. The framework relies on the OASIS open standard *Service Component Architecture* (SCA) [2] as modeling language for heterogeneous service assembly in a technology agnostic way, and on the Java implementation for SCA. A supporting prototype tool based on the Eclipse environment is also available at [34]. It was developed by extending the Eclipse plug-in *PatternBox* [3], an existing design pat-

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

[2] http://www.soapatterns.org/

tern editor for Java code, and by integrating it with the Eclipse-based SCA Composite Designer and the SCA runtime platform Tuscany [4]. Through an XML template-based mechanism, SCA-PatternBox allows the definition of design patterns and their instantiation from a scratch SCA design to be further customized depending on the application needs, or from an SCA component assembly of an existing service-oriented application to generate automatically the corresponding compound SCA component assembly and the SCA-Java skeleton code (indeed, Java classes and interfaces with appropriate annotations for SCA). The template-based approach for the pattern instantiation and code generation makes SCA-Patterbox higher usable than wizard-based approaches where you have to complete the whole design pattern instance at once. Moreover, SCA-PatternBox can be easily extended. New design patterns and code generators can be introduced by defining new XML templates depending on the target implementation platform and application domain.

Since patterns have two complementary aspects (structural and behavioral), for patterns that have a significant behavioral aspect, it is necessary to understand how service components collaborate to achieve the expected behavior. To this purpose, in addition to a Java-like implementation of design patterns, SCA-PatternBox allows also the formal specification and analysis of the functional behavioral aspect of design patterns using a formal service specification language called SCA-ASM [6, 7]. SCA-ASM is based on the formal method *Abstract State Machine* [8] that allows the definition of executable and state-based specifications of systems behavior. The main goal is to provide formally verified design patterns by producing precise and unambiguous functional descriptions of design patterns in SCA-ASM and then to validate and verify these formal specifications (e.g. through simulation and model checking).

The proposed framework offers several advantages with respect to the current state of art (see Sect. 6 for a detailed comparison with related works). In the literature, design patterns are typically described using a combination of natural language, UML class and sequence diagrams, and program code (see related work in Sect. 6.1). Such descriptions lack design pattern-specific visual formalisms, leading to pattern descriptions that are hard to understand, hard to incorporate into tool support, and therefore hard to be machine-processable in order to automate their instantiation and application in the current design and then in the software code. The proposed framework was conceived instead with automation in mind and to this purpose a template-based approach was adopted. The purpose of this work is to facilitate the construction of a service-oriented application and to help engineers understand alternative means for achieving alternative architectures in the applications design and code. Moreover, existing approaches are specific to object-oriented system design and do not address SOA design patterns. Instead, our approach addresses also design patterns related to the SOA domain. Finally, some existing approaches that use a mathematical formalism for defining design patterns formally require strong mathematical background to the user and lack of good tool support. Based on the practical and scientifically well-founded ASM formal method, SCA-ASM models are instead executable and without mathematical overkill. SCA-ASM

allows modeling both *structure* and *behavior* of service components in a unique framework integrating architectural and behavioral views. By exploiting the prototyping/validation environment for SCA-ASM [7], patterns and components can be executed already at high level of formalization, without caring about implementation details. Early validation by model simulation is a great means for evaluating architectural choices and alternative designs with limited implementation effort. The mathematical foundation of the method also facilitates reasoning about component behavior in order to guarantee their correctness.

A first prototype of the SCA-PatternBox environment was presented as a tool demo at the Eclipse Italian workshop [9], while a preliminary overview of the framework was presented in [10]. This article extends these preliminary works in several aspects. First, this article provides a more accurate description of the pattern definition language adopted by the SCAPatternBox framework and its use through concrete patterns examples. Formal analysis techniques supported by SCAPatternBox for validating and verifying the functional behavioral aspects of patterns and applications are presented. The article introduces also a supporting methodology for a general and agile prototyping of a service-oriented application with the SCA-PatternBox framework, and illustrates the methodology through two case studies – the Order system and the Stock Trading System (STS). Lessons learned that we gained through our experience in developing the case studies and that should be retained for future use are also reported. Moreover, a comparison of existing design pattern languages is also presented according to some specific criteria.

The remainder of this article is organized as follows. Section 2 provides background concepts on SCA, SCA-Java and SCA-ASM. Section 3 introduces the SCA-PatternBox language and alternative notations for editing design patterns and instantiating them automatically. Section 4 describes the SCA-PatternBox framework and a supporting design/development methodology for prototyping service-oriented applications with design patterns. Section 5 illustrates the methodology with two case studies and provides some lessons learned during the development of the framework and its use for the case studies. Section 6 surveys the main existing design pattern languages and provides a comparison of them and of our proposed SCA-PatternBox language. Finally, Section 7 concludes the article and sketches some future directions of our work.

2. Background concepts

Service Component Architecture (SCA) [2] is an XML-based component model used to develop service-oriented applications independently from SOA platforms and middleware programming APIs. SCA is also endowed with a visual notation and supported by an Eclipse-based design tool and runtime platforms (like Apache Tuscany, FRAscaTI, IBM WebSphere Application Server V7, etc.) for the development and deployment of service-oriented applications.

Fig. 1 shows an *SCA composite* (or *assembly*) as a composition of SCA components. An *SCA component* is a piece of configured software that provides

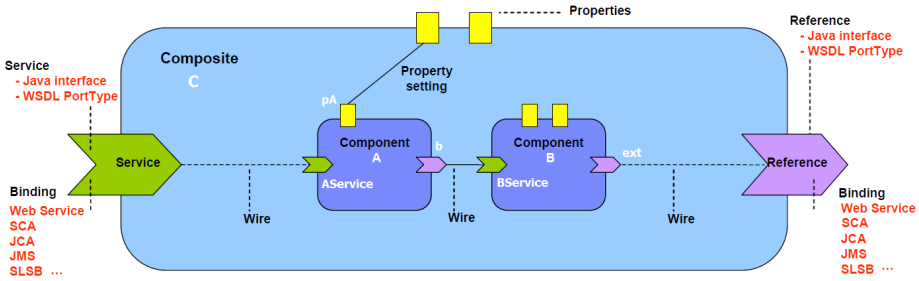

Figure 1: An SCA composite (adapted from the SCA Assembly Model V1.00 spec.)

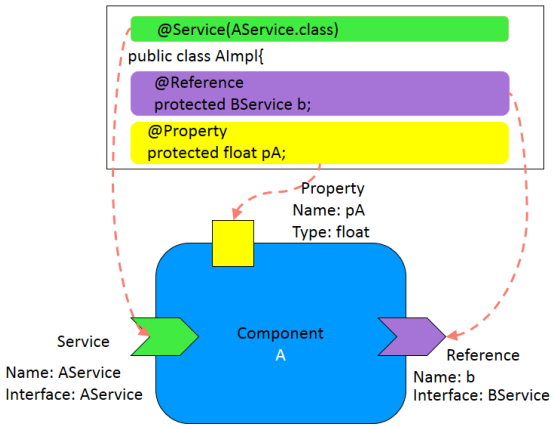

Figure 2: SCA-Java component shape

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 of an operation with zero or more messages. Message exchange may be synchronous or asynchronous.

The *SCA-Java Component Implementation* [2] defines how to implement an SCA component using Java. Fig. 2 shows the SCA component `A` of Fig. 1 and its Java implementation class `AImpl`. Java annotations (`@Property`,`@service`,etc.) are used to augment Java classes with SCA concepts.

The *SCA-ASM specification type* [6, 7] complements the SCA component model with the ASM model of computation to provide ASM-based formal and executable description of services *internal behavior*, *orchestration* and *interactions*. An open framework, the ASM toolset ASMETA (ASM mETA-modeling) [11, 12], based on the Eclipse/EMF platform and integrated with the SCA runtime Tuscany, is also available for editing, simulating, validating, and potentially model checking SCA-ASM models [6, 7].

ASMs [8] are an extension of FSMs where states are arbitrary complex data (multi-sorted first-order structures) and the transition relation is specified by rules describing how functions change from one state to the next. The basic rule has the form of *guarded update* "**if** *Cond* **then** *Updates*" where *Updates* is a set of function updates of the form $f(t_1, \ldots, t_n) := t$ which are simultaneously executed when *Cond* is true. Rule constructors express parallel actions (*par*), sequential actions (*seq*), iterations (*iterate*, *while*, *recwhile*), non-determinism (existential quantification *choose*) and unrestricted synchronous parallelism (universal quantification *forall*). Distributed computation is modeled by means of *multi-agent ASMs*: multiple agents interact in a synchronous/asynchronous way, each executing a program specified by an ASM rule.

In SCA-ASM, a service-oriented component is an ASM endowed with (at least) one agent (a business partner or role). Components' agents interact with other agents by providing and requiring services. The service behaviors encapsulated in an SCA-ASM component are captured by ASM transition rules. Fig. 3 shows the shape of the SCA-ASM component `A` of Fig. 1 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 notation ASMETA/AsmetaL and the `@annotations` to denote SCA concepts. ASM rule constructors and predefined ASM rules (i.e., named ASM rules in a model library) are used as SCA-ASM behavioral primitives. These rules 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*[3]. Rules for fault/compensation handling are also supported [7].

[3]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.

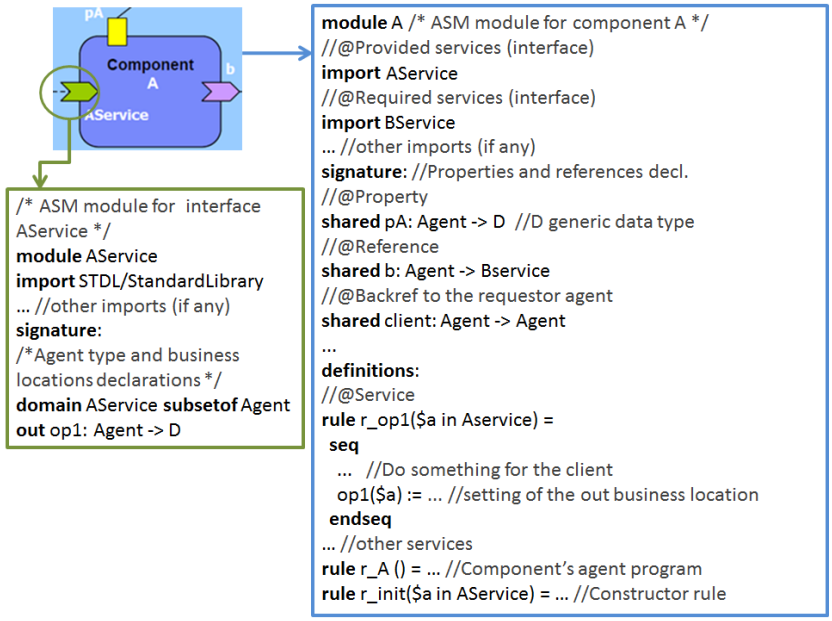

Figure 3: SCA-ASM component shape

3. The SCA-PatternBox pattern language

This section describes the SCA-PatternBox language supporting the specification of architectural design patterns and their instantiation into SCA assembly models of service-oriented applications. The section also provides concrete examples of design patterns.

The SCA-PatternBox language consists of a metamodel (the *abstract syntax*) providing a set of modeling constructs to define and reuse design patterns, and of XML-based notations (the *concrete syntax*) for modeling design pattern solutions and solution instances within SCA models. The XML-based concrete notations are both human- and machine- comprehensible.

The SCA-PatternBox language can be used as a stand-alone modeling notation for design patterns or in conjunction with SCA to model design pattern instances within SCA assembly models of service-oriented applications. A design pattern instance describes the relationships between the design pattern elements modeled in the SCA-PatternBox language and the design elements (service-oriented components and interfaces) in the SCA assembly of a specific application.

The proposed language is intended to be used only to model the generalized solutions proposed by design patterns and facilitate their application and reuse. Further details, such as when the solution should be applied and consequences of using the pattern, are not included. The SCA-PatternBox language has been

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	**wreply**[*lnk,R,snd*]	returns data *snd* to *lnk*, as response of R request received from *lnk* (no ack)

designed with automated tool support in mind. It is relatively easy to learn, particularly in conjunction with SCA.

3.1. Design pattern specification

Typically, an architectural design pattern describes components and details their roles and interactions. All components together solve the problem that the pattern addresses.

Fig. 4 shows the SCA-PatternBox language's metamodel capturing the very essence of a design pattern independently of its domain. A design pattern in SCA-PatternBox is to be intended as an instance of this metamodel. Essentially, a design pattern (the class `Pattern`) defines roles (the class `Role`), role dependencies (the class `Dependence`), and an informal description (the attribute `comment`) in natural language. Roles are played by the participants of a pattern solution. A role has, among other things, an attribute `type` to tailor the type of architectural design elements (a component or interface) that can play the role, a multiplicity (the attributes `min` and `max`) that constrains the number of elements that can play the role, an attribute `property` to denote a set of mandatory properties, and an attribute `operation` to denote a set of mandatory service operations. The default multiplicity interval `(min,max)` in the class `Role` is 1..*, specifying that there must be at least one element playing the role. Dependencies are typically between a component role and a provided/required

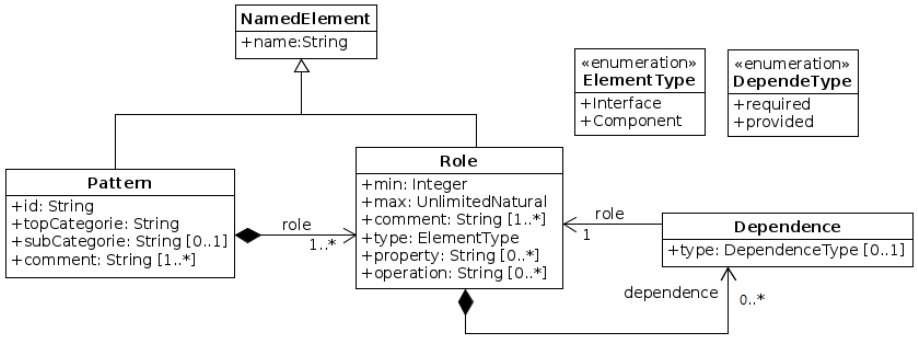

Figure 4: SCA-PatternBox design pattern metamodel

interface role, but not necessarily. There can be also dependencies between components themselves to denote, for example, wires connecting a composite component with a sub-component. In this last case the attribute type of the dependence is not present. This is enforced by the following OCL invariant (i.e., a constraint that must always be met by all instances of a class) introduced within the context of the class `Dependence`:

context Dependence

inv targetRoleType: **not** self.type.isNull **implies** self.role.type=interface

Some other OCL constraints not reported here have been defined to constrain the number of valid instances of this metamodel. These constraints are embedded in the SCAPatternBox editor and checked during pattern instantiation.

As concrete textual syntax associated to the metamodel, a pattern is defined in an XML file conforming to a DTD `manifest.dtd`. Code 1 reports the XML file for the definition of a service interaction micro-pattern[4] called `Request-Response`. In this interaction schema, a component `ClientRequest-Response` invokes a service of the component `Server` and waits for the result to be returned before continuing with its processing. Request-response is the default mode of invoking a service in a synchronous way.

As complementary concrete syntax, a pattern can be also defined in terms of an SCA assembly model thus exploiting the SCA graphical notation. This SCA assembly consists of service components and reference-to-service wires corresponding, respectively, to roles and role dependencies. Exploiting the SCA (graphical) notation provides a much higher expressive power, but it is tied to a specific design language. Fig. 5 shows the SCA assembly for the Request-Response micro-pattern.

[4]Micro-patterns are prime candidates of "units of design" to look for and are the "basis" of more complex SOA patterns.

Code 1: Request-Response micro-pattern

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="requestresponse" name="RequestResponse"
         topCategorie="SCA" subCategorie="Micro Pattern">
<role name="RequestResponseService" min="1" max="1" type="Interface" operation="request">
 <comment>
  <li>defines a service interface.</li>
 </comment></role>
<role name="Server" min="1" max="1" type="Component">
 <comment>
  <li>implements the RequestResponseService interface.</li></comment>
 <dependence role="RequestResponseService"/ type="provided"></role>
<role name="ClientRequestResponse" min="1" max="1" type="Component">
 <comment>
  <li> maintains a reference to a RequestResponseService;</li>
  <li> makes a request to a RequestResponseService and waits for the result. </li></comment>
 <dependence role="RequestResponseService" type="required"/></role>
</pattern>
```

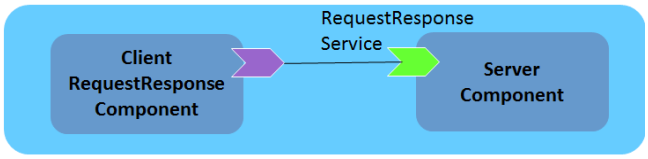

Figure 5: Request-Response micro-pattern in SCA

3.2. Design pattern instantiation

Instantiating a design pattern means creating a new piece of design called "design pattern instance" by mapping the design elements and relationships of the design pattern with elements and relationships of the domain (application) knowledge. After applying (instantiating) a pattern into an existing design, the resulting software architecture should include a particular structure that provides for the roles specified by the pattern, but adjusted and tailored to the specific needs of the problem at hand.

SCA-PatternBox instantiates a pattern by generating from scratch an SCA-assembly and a skeleton SCA-Java code to be further re-factored and refined to the specific application needs. The generation process is *template-based*: there must exist a template for each design pattern supported. Such a template is an XML file conforming to a DTD called `templates.dtd` in SCA-PatternBox. Code 2 shows an example of SCA-Java template for the Request-Response micro-pattern.

Similarly, to support the generation of SCA-ASM formal specifications from patterns, we defined a grammar module `ASM templates.dtd` for the definition of SCA-ASM pattern templates. The goal is to precisely define the intended execution semantics of a pattern (which is typically expressed in a quite informal way by graphical notations like SCA or UML) using the ASM formalism. SCA-ASM offers a more accurate description of the principles involved in a design

Code 2: SCA-Java template for RequestResponse

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE templates SYSTEM "templates.dtd">
<templates id="Request-Response" version="1.0">
 <role name="RequestResponseService" type="interface" modifiers="public">
  <import type="org.osoa.sca.annotations.*"/>
    <method modifiers="public" return="java.lang.Object" name="request">
      <comment> The request service operation </comment>
      <param type="java.lang.Object" name="item"/>
    </method> </role>
 <role name="Server" type="class" modifiers="public" >
  <import type="org.osoa.sca.annotations.*"/>
  <annotation>@Service(RequestResponseService.class)</annotation>
    <interface type="$RequestResponseService$"/>
      <constructor modifiers="public">
       <comment>Default constructor</comment>
       <code> super();</code> </constructor>
    <method modifiers="public" return="java.lang.Object" name="request">
      <comment>Method implementing the service operation
                of the RequestResponseService interface. </comment>
      <param type="java.lang.Object" name="item"/>
      <code> // TODO Write your code here ...
              return item; </code> </method> </role>
 <role name="ClientRequestResponse" type="class" modifiers="public" >
  <import type="org.osoa.sca.annotations.*"/>
  <annotation>@Service(Runnable.class)</annotation>
  <interface type="java.lang.Runnable"/>
    <field modifiers="protected" type="$RequestResponseService$"
          name="fRequestResponseService">
      <comment>reference to a RequestResponseService</comment>
      <annotation>@Reference</annotation>
    </field>
    <constructor modifiers="public">
      <comment>Constructor</comment>
      <param type="$ClientRequestResponse$" name="requestresponse" />
      <code> super(); </code>
    </constructor>
    <method modifiers="public" return="void" name="run">
      <code> Object item = fRequestResponse.request(item);
              // TODO Write your code here ... </code> </method> </role>
</templates>
```

pattern and offers a more sophisticated insight into the pattern behavior for those familiar with formal notations like ASM.

Code 3 shows the SCA-ASM template for the request-response micro pattern, namely the SCA-ASM definition of the interface `RequestResponse` and of the `ClientRequestResponse` and `Server` component roles.

Code 4 shows the resulting ASM specification after instantiating the micro pattern from a scratch design. The ASM module `RequestResponseService` corresponds to the `RequestResponseService` interface. It contains only declarations of the business agent type `RequestResponseService` and of the business function (ASM out function) `request`. The ASM module `ClientRequestResponseComponent` imports the ASM module of the required service interface `RequestResponseService` of the component, annotated with `@Required`. The signature of the component contains declarations for a reference (shared function annotated with @Reference) as abstract access endpoint to the `RequestResponse` service, and declarations of ASM functions used by the component for internal computation only. In particular, the function `items` represents data associated to the request made to the server. The ASM module `ServerComponent` imports the ASM module of the provided service interface `RequestResponseService` of the component, annotated with `@Provided`. The annotation `@MainService` on the import clause for the `RequestResponseService` interface denotes the main service (read: main component's agent) that is responsible for initializing the component's state (in the predefined `r_init` rule). The signature of the component contains declarations for: a back reference to requester agent (the shared function `client` annotated with @Backref), and declarations of ASM functions used by the component for internal computation only. This resulting SCA-ASM specification is directly executable within the SCA-ASM execution environment (see Sect. 4) for formal validation.

3.3. Design patterns examples

We defined and collected different types of design patterns. First, we considered *micro-patterns* for service interaction. Table 2 summarizes the SCA micro-patterns for service interactions. Further semantics details can be found in the SCA assembly specification model [2].We inspired to the SCA standard and defined all these service oriented micro-patterns in SCA-PatternBox.

We then started to work with conventional architectural patterns and with SOA patterns (like FIFO, Replicator, Ping-Echo, Publish-Subscribe, Router-

Table 2: Service Interaction Micro-patterns

Micro-Pattern	**Communication**	**Annotation**
Local	*pass-by-reference*	–
Remote	*pass-by-value*	@Remotable
Request-response	*Synchronous*	–
Oneway	*Asynchronous*	@oneway
CallBack	*Asynchronous Bidirectional*	@Callback

Code 3: SCA-ASM template for RequestResponse

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE templates SYSTEM "templates.dtd">
<templates id="Request-Response" version="1.0">
<role name="RequestResponseService" type="module"/>
 <import type="STDL/StandardLibrary"/>
 <delimiter type="signature"/>
 <domain name="RequestResponseService"/> <subsetof name="Agent"/>
 <function modifier="out" name="response">
  <domain>Agent</domain> <codomain>D</codomain>
 </function>
</role>
<role name="ClientRequestResponseComponent" type="module"/>
 <comment> <li>//@Required interface</li> </comment>
 <import type="RequestResponseService"/>
 <comment> <li>//@Reference to the agent RequestResponseService</li></comment>
 <function modifier="shared" name="fRequestResponse">
  <domain>Agent</domain> <codomain>RequestResponseService</codomain>
 </function>
 <function modifier="controlled" name="items">
  <domain>Agent</domain> <codomain>D</codomain>
 </function>
 <delimiter type="definitions"/>
 <rule name="r_ClientRequestResponseComponent">
  <body> //Make the request in a synchronous manner by send-receive
r_wsendreceive[fRequestResponse(self),"r_request(Agent,D)",items(self),result(self)]</body>
 </rule>
 <rule name="r_init">
  <param type="ClientRequestResponseComponent" name="$a" />
  <body> //Complete this rule body for the startup of the component
    status($a) := READY </body>
 </rule>
</role>
<role name="ServerComponent" type="module"/>
<comment> <li>//@Provided interface</li> </comment>
 <import type="RequestResponseService"/>
 <comment> <li>//@Backref to the client agent</li> </comment>
 <function modifier="shared" name="client">
  <domain>Agent</domain> <codomain>Agent</codomain>
 </function>
 <function modifier="controlled" name="params">
  <domain>Agent</domain> <codomain>D</codomain>
 </function>
 <delimiter type="definitions"/>
 <comment> <li>//@Service</li> </comment>
 <rule name="r_request">
  <param type="Agent" name="$a" /> <param type="D" name="$params" />
  <body> skip //Replace this rule body with your ASM rule scheme </body>
 </rule>
 <rule name="r_ServerComponent">
  <body>
        let($r = nextRequest(self)) in//Select the next request (if any)
         if isDef($r) then seq //Handle the request $r
          r_wreceive[client(self),"r_request(Agent,D)",params(self)]
                if (isDef(params(self))) then r_request[self,params(self)] endif
                r_wreply(client(self),"r_request(Agent,D)",response(self))
         endseq endif endlet
        </body>
 </rule>
 <rule name="r_init">
  <param type="RequestResponseService" name="$a" />
  <body> //Complete this rule body for the startup of the component
    status($a) := READY </body>
 </rule>
</role>
</templates>
```

Code 4: SCA-ASM specification of the request-response micropattern

```
1  //@Remotable
2  module RequestResponseService
3  import STDL/StandardLibrary
4  import STDL/CommonBehavior
5  export *
6  signature:
7  domain RequestResponseService subsetof Agent
8  out request: Prod(Agent,D) -> Rule
9
10 module ClientRequestResponseComponent
11 import STDL/StandardLibrary
12 import STDL/CommonBehavior
13 //@Required service
14 import RequestResponseService
15 export *
16 signature:
17 //@Reference
18 shared fRequestResponse : Agent -> RequestResponseService
19 controlled items: Agent -> D
20 controlled result: Agent -> D
21 definitions:
22 rule r_ClientRequestResponseComponent =
23   //Make the request in a synchronous manner by send-receive
24   r_wsendreceive[fRequestResponse(self),"r_request(Agent,D)",items(self),result(self)]
25 rule r_init($a in ClientRequestResponseComponent) =
26  //Complete this rule body for the startup of the component
27     status($a) := READY
28
29 module ServerComponent
30 import STDL/StandardLibrary
31 import STDL/CommonBehavior
32 //@MainService
33 import RequestResponseService
34 export *
35 signature:
36 //@Backref to the client agent
37 shared client : Agent -> Agent
38 controlled params: Agent -> D
39 definitions:
40 //@Service
41 rule r_request($a in Agent,$params in D)=
42   skip //Replace this rule body with your ASM rule scheme
43 rule r_ServerComponent =
44   let($r = nextRequest(self)) in//Select the next request (if any)
45           if isDef($r) then seq //Handle the request $r
46             r_wreceive[client(self),"r_request(Agent,D)",params(self)]
47                   if (isDef(params(self))) then r_request[self,params(self)] endif
48                   r_wreply(client(self),"r_request(Agent,D)",response(self))
49           endseq endif endlet
50 rule r_init($a in RequestResponseService) =
51 //Complete this rule body for the startup of the component
52     status($a) := READY
```

Code 5: Ping-echo tactic

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="pingecho" name="PingEcho" topCategorie="SCA">
<comment>
One component issues a ping and expects to receive back an
echo, within a predefined time, from the component under
scrutiny.
</comment>
<role name="PingService" min="1" max="1" type="Interface" operation="ping">
<comment>
<li>Defines an interface for the Ping sending service.</li>
</comment></role>
<role name="EchoService" min="1" max="1" type="Interface" operation="echo">
<comment>
<li>Defines a callback interface for the Echo sending service.</li>
</comment></role>
<role name="PingSender" min="1" max="1" type="Component"
      property= "timeInterval maxWaitingTime ">
<comment>
<li>Maintains a reference to one or more Receiver services.</li>
<li>Sends a Ping to a Receiver component. </li>
</comment>
<dependence role="PingService" type="required"/></role>
<dependence role="EchoService" type="provided"/></role>
<role name="PingReceiver" min="1" max="1" type="Component">
<comment>
<li>Maintains a reference to a Sender service.</li>
<li>Receives a Ping from a sender and reply to it with an Echo.</li>
</comment>
<dependence role="PingService" type="provided"/></role>
<dependence role="EchoService" type="required"/></role>
</role>
</pattern>
```

Filtering, etc.) related to the service level of abstraction (rather than the business process or orchestration level). Some of them are to be considered as *tactics*, i.e., architectural patterns providing a generic solution to issues pertaining to extra-functional quality attributes (such as performance, availability, etc.). For example, a design concern for availability is "Fault Detection". Since services and their service providers can be discovered at run time, new service providers may be brought into existence at any time and existing service providers may fail or stop operating entirely. To make the application more robust to these kind of faults, two well-known tactics for fault detection are Ping/echo and Heartbeat.

As an example of design pattern, Fig. 6 shows the SCA diagram for the Ping/Echo tactic. Ping-Echo is a tactic for monitoring and checking the availability of a component by sending ping messages to the component (the receiver) regularly every `timeInterval` units. If the receiver component does not send back an echo to the sender component within the maximum waiting time (property `maxWaitingTime`), the sender considers the receiver component failed. Code 5 shows the XML file defining the Ping/Echo tactic in more general terms for people not familiar with SCA. The Ping/Echo tactic can be intended as a refinement of the call-back micro-pattern.

Fig. 7 shows the *Heartbeat* design pattern in SCA as another example of

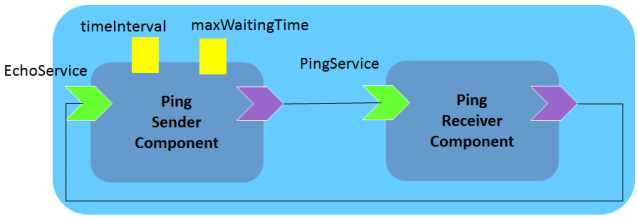

Figure 6: Ping/Echo tactic in SCA

tactic for availability. It detects a fault by listening to heartbeat messages from monitored components periodically. A sender sends a heartbeat message to a receiver (operation `imAlive`) every specified time interval (property `timeInterval`). The receiver updates the current time when it receives the heartbeat message. The aliveness of the sender is checked by the receiver regularly every specified time interval (property `timeInterval`) by comparing the latency time between the current time and the last time of a heartbeat message is received from the server. If the heartbeat message is not received within a certain time (property `maxWaitingTime`) the sender is considered to be unavailable. Code 6 shows the XML file defining the Heartbeat tactic.

The Ping/Echo tactic and the Heartbeat tactics can be combined to make more efficient and bidirectional the fault detection mechanism. The result of this tactic composition is shown in Fig. 8 using SCA. Technically it has been obtained by applying the Heartbeat tactic to the SCA asssembly of the Ping-Echo tactic. Property re-naming is necessary to avoid ambiguities. For example, property `timeInterval` of the role *HeartBeatSender* of the Heartbeat tactic is renamed in `beatTimeInterval`. The result of this pattern composition can be saved and reused as new compound design pattern.

4. SCA-PatternBox framework

SCA is supported by an Eclipse-based design tool and runtime platforms for the development of service-based applications. Based on the ideas of model-based development, the proposed framework SCA-PatternBox is Eclipse-based and complements the SCA design environment by the use of a collection of models to specify and configure architectural design patterns and the automated application of them in the development of SCA models of service-oriented applications. SCA aims to encompass a wide range of target implementation technologies for SCA service components and for the access methods which are used to connect them. In our work, we focused on the SCA-Java implementation type and on the SCA-ASM formal specification type.

In the following, we provide an overview of the SCA-PatternBox framework, a general methodology for application designers to develop service-oriented component architectures using SCA-PatternBox and design patterns, and a description of the supported formal analysis techniques.

Code 6: Heartbeat tactic

```
<!DOCTYPE pattern SYSTEM "manifest.dtd">
<pattern id="heartbeat" name="Heartbeat" topCategorie="SCA">
<comment> One sender component sends a heart beat message to a receiver periodically.
The aliveness of the sender is checked by the receiver regularly every specified time interval.
If the heartbeat message is not received within a certain maximum waiting, the sender is
considered to be unavailable.
</comment>
<role name="HeartbeatService" min="1" max="1" type="Interface" operation="imAlive">
<comment>
<li>Defines an interface for the heart beat sending service.</li>
<role name="HeartbeatSender" min="1" max="1" type="Component" property= "timeInterval">
<comment> <li>Maintains a reference to one or more Receiver services.</li>
<li>Sends an heartbeat message to a Receiver component. </li> </comment>
<dependence role="HeartBeatService" type="required"/></role>
<role name="HeartbeatReceiver" min="1" max="1" type="Component"
      property= "timeInterval maxWaitingTime">
<comment> <li>Maintains a reference to a Sender service.</li>
<li>Receives an heart beat from a sender periodically.</li></comment>
<dependence role="HeartbeatService" type="provided"/></role>
</role>
</pattern>
```

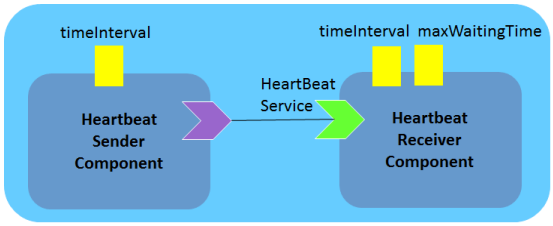

Figure 7: Heartbeat tactic in SCA

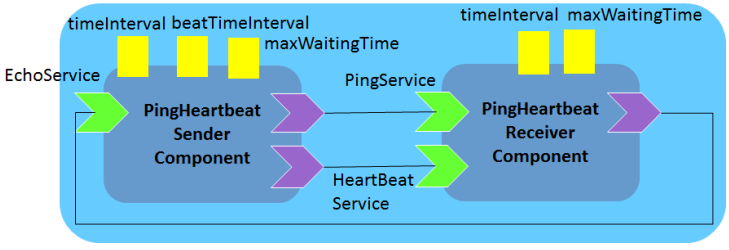

Figure 8: SCA composition of the Ping/Echo and HeartBeat tactics

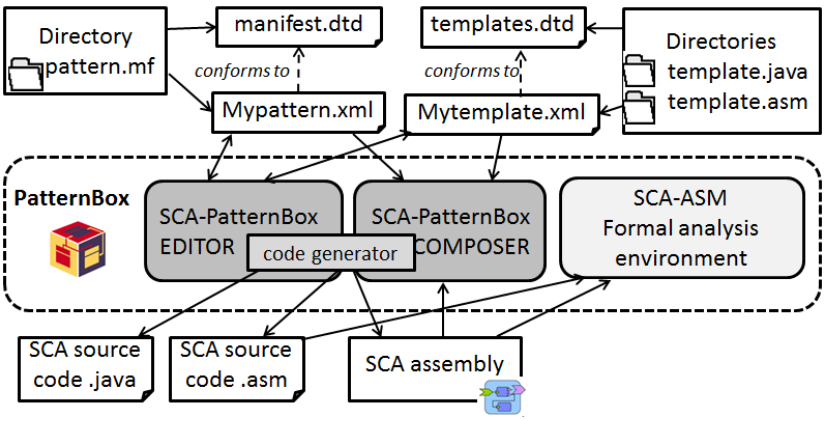

Figure 9: SCA-PatternBox architecture

4.1. Framework architecture

Fig. 9 shows the SCA-PatternBox architecture using a free-style notation, while Fig. 10 shows a screenshot of the Eclipse view of SCA-PatternBox. SCA-PatternBox includes a design *pattern editor*, *code generators* for producing SCA-Java code and SCA-ASM specifications, and a *pattern composer*. It also exploits an external *formal analysis environment* for SCA-ASM specifications.

Pattern editor. The pattern editor adopts a template-based approach that allows you to insert new pattern definitions (file `Mypattern.xml` in Fig. 9, and appropriate templates (file `Mytemplate.xml` in Fig. 9) for the generation of SCA-Java code and of SCA-ASM formal specifications. A design pattern is defined in an XML file within a predefined directory `pattern.mf` and must be conforming to the DTD `manifest.dtd` within the same directory.

Code generator. There exists an XML-based template files within the predefined directory `template.java` and within `template.asm` of SCA-PatternBox for each design pattern to support the generation of the corresponding SCA-Java code and SCA-ASM specification, respectively. For the SCA-Java code generation, we extended the existing PatternBox XML grammar `templates.dtd` to allow the use of Java annotations for SCA into the code templates, and the source code of PatternBox (`CodeTemplateXmlHandler.java` and `MemberCodeGenerator.java`) to generate Java code from the pattern code templates with the appropriate SCA annotations. The code generator makes intensive use of Eclipse's Java development tooling (JDT) and the Plug-in Development Environment (PDE). It creates Java classes and interfaces according to the code templates and generates also an SCA XML assembly file corresponding to it. Fragments of SCA assembly files and the associated Java skeleton code can be therefore produced from scratch from a pattern definition.

Pattern composer. The composer supports the composition and application of design patterns on existing SCA assemblies (and on the corresponding Java or ASM implementation). Currently, the composition is carried out in an interactive manner, indeed the designer has to specify the names of the elements (components or interfaces) from the SCA assembly that play specific roles of the design pattern. The composition strategy is *incremental*, i.e., the application of design patterns to an existing SCA assembly is made through a sequential chain of adaptation actions of the SCA assembly and of the associated components implementation according to the pattern definitions.

Formal specification and analysis environment. The formal analysis environment allows for an early and formal validation of the design of a service-oriented application. It consists of the Eclipse-based SCA-ASM design and execution framework for the simulation of SCA-ASM components within an SCA assembly (exploiting the SCA Tuscany runtime platform), and of the ASM analysis toolset ASMETA [11, 12]. An SCA-ASM specification of a service-oriented component (or of a component assembly), possibly not yet implemented in code or available as off-the-shelf, can be: (i) simulated and possibly formally verified (by model checking techniques) offline, i.e., in isolation from the other components, by the use of the ASM toolset ASMETA; (ii) configured in place within the SCA Tuscany runtime platform as abstract implementation (or specification) of a "mock" component and then executed with the other components implementations according to the chosen SCA assembly.

This toolkit allows to study the behavior of some "critical" components in a formal way before the final transformation into a specific implementation code. The same consideration also apply to component assemblies representing design patterns that can be therefore specified formally in SCA-ASM as abstract and partially complete specifications. Moreover, once these patterns are instantiated into the current SCA design and their specification in SCA-ASM is completed, the structural and behavioral conformance of the component assemblies to design patterns can be (potentially) checked formally.

4.2. Framework methodology

A general and agile development of an SCA service-oriented component architecture (or SCA assembly) with SCA-PatternBox can be organized in the following steps (see also Figure 11).

Step 1: Sketch an outline of the application's architecture. Define an SCA assembly of the service-oriented application with the SCA Composite Designer by determining the ground services and components required according to the main requirements and use cases from the user perspective.

Step 2: Refine the architecture. Identify the main ways in which the components will interact and the interfaces between them. Decide how each piece of data and functionality will be distributed among the various components. Choose among the various service-oriented design patterns. Instantiate and compose them within the SCA assembly.

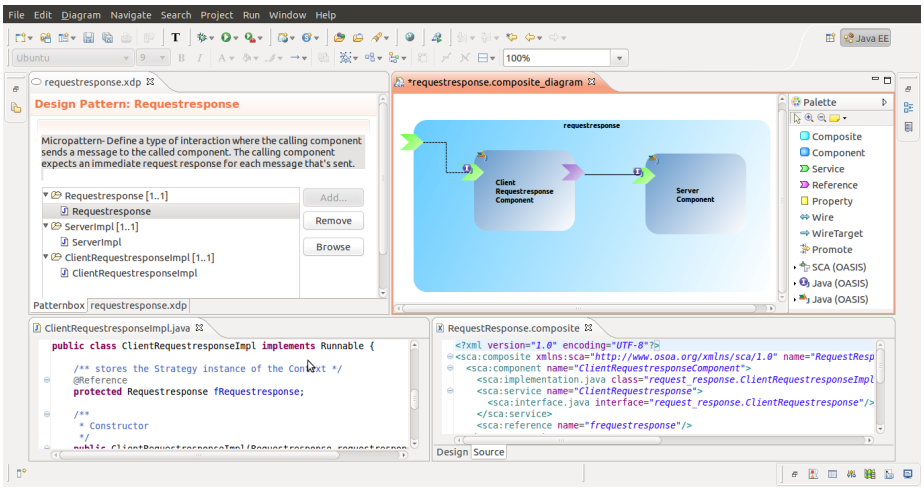

Figure 10: A screenshot of the SCA-PatternBox framework

Step 3: Finalize the interfaces. Consider each use case and adjust the architecture to make it realizable trying to finalize the interface of each component. Refine components properties in the SCA assembly.

Step 4: Map design to implementation. Finalize the architecture as you define the final implementation classes for components and interaction protocols according to the (possibly many and different) target implementation technologies. Determine if you can re-use existing components implementations before implementing them from scratch. You may implement, for example, some SCA components in SCA-Java and adopt Java-based standard communication bindings – such the Java API for RESTful Web Services (JAX-RS) or the Java Message Service (JMS) – for specifying how SCA services and references enable a component to communicate with other components/applications.

Step 4ı: Map design to formal specification. Optionally, you can specify formally the behavior of some "critical" components using the SCA-ASM implementation type.

Step 5: Formal validation. Optionally, once specified formally the behavior of some components or of a component assembly in SCA-ASM (Step 4ı), you may validate such components or assembly separately (in an offline manner) using the ASM analysis toolset ASMETA.

Step 6: Overall design validation. Execute and validate the overall SCA assembly of the application within an SCA runtime platform (like Tuscany).

More sophisticated development processes can be adopted as well. SCA-PatternBox has been used, for example, to support a design exploration process [15] involving more automation and combining meta-heuristic search techniques with design patterns to produce and evaluate different design alternatives.

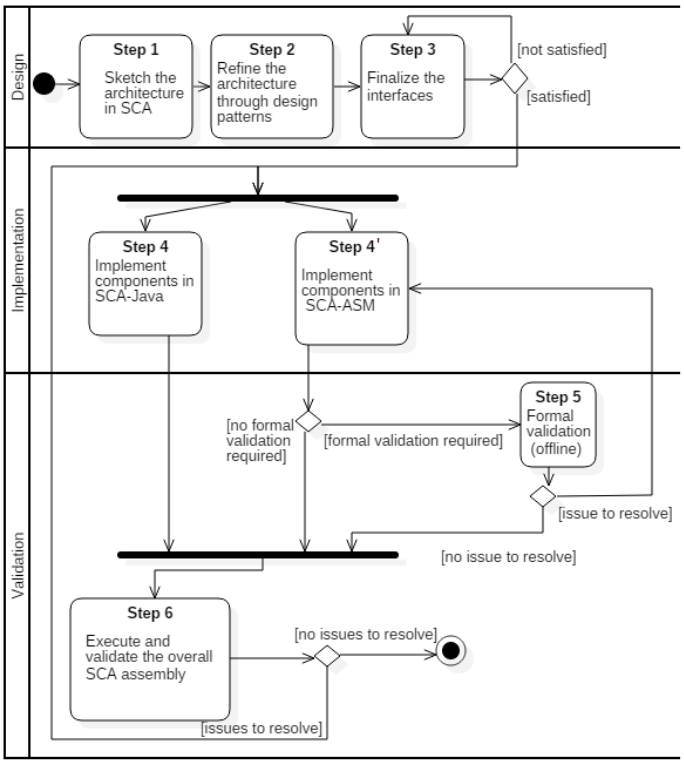

Figure 11: Model development steps with SCA-PatternBox

4.3. Formal analysis techniques

We here describe the SCA-ASM model analysis activities we can perform offline for validating and verifying the functional behavior of design patterns, and getting early feedback (already at system design time) of their functional correctness before applying them in concrete systems implementations.

Validation is the process of investigating a model with respect to the user perceptions in order to ensure that the model really reflects the user needs and statements about the application. On the SCA-ASM models of design patterns and their concrete instantiations, formal validation can be carried out in terms of *model simulation* and construction of execution scenarios – *scenario-based validation* – through the execution platform for SCA-ASM [7] and the ASM simulator `AsmetaS` [14] and validator `AsmetaV` [13] provided by the ASMETA framework [11, 12]. Fig. 12 shows the validation process: the user can directly simulate an ASM-based specification in an interactive way or write a scenario that automatizes the simulation and the checking of the produced output. Early validation by model simulation is a great means for evaluating architectural

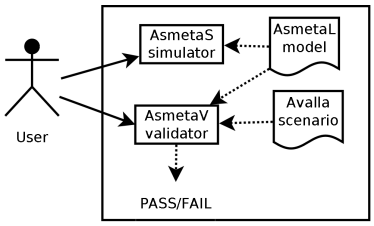

Figure 12: Validation in the ASMETA framework

choices and alternative designs with limited implementation effort, and usually, though not in an exhaustive manner, it permits to detect faults in the specification with limited effort w.r.t. more sophisticated analysis techniques such as property verification through model checking.

Model simulation and Scenario-based validation. `AsmetaS` permits to perform either *interactive simulation*, where required inputs are provided interactively by the user during simulation, and *random simulation*, where inputs values are chosen randomly by the simulator itself. The simulator, at each step, performs *consistent updates checking* to check that all the updates are consistent: in an ASM, two updates are inconsistent if they update the same location to two different values at the same time [8]. In preliminary versions of our patterns specifications, by simulation we found some consistency violations due to a wrong order scheduling of the send-receive operations of the participants agents. Moreover, the `AsmetaS` simulator also permits to check if some invariants are satisfied during simulation. Obviously, by simulation we can verify only the states covered by the executed runs, whereas model checking (see next paragraph) gives the assurance that the invariants hold in each model state.

A more advanced way to simulate and inspect ASMs is by specifying a scenario representing a description of the actions of an external actor and the corresponding reactions of the system. There are two kinds of external actors:

- a *user* interacts with the system in a black box manner, by setting the values of the external environment (e.g., asking for a particular service), waiting for a step of the machine as reaction to his/her request, and checking the output values;
- an *observer*, instead, can also inspect the internal state of the system (i.e., values of machine functions) and check the validity of possible invariants of a certain scenario.

Scenarios are described in an algorithmic way using the textual language `Avalla` [13]. A scenario is as interaction sequence consisting of actions of the external actor (user or observer) and activities of the machine as reaction to the actor actions. The `Avalla` language provides constructs to **set** the environment (i.e., the values of input/shared functions), to **check** the machine state, to ask for the **exec**ution of certain transition rules, and to force the machine itself to make one **step** (or a sequence of steps by **step until**).

```
1  scenario request−response
2  load mainRequestResponse.asm
3  ...
4  //for the startup of the client and server agents
5  set status(c) := READY;
6  set fRequestResponse(c) := s;
7  set items(c):= ...;
8  set ...
9  exec r_wsendreceive[fRequestResponse(c),"r_request(Agent,D)",items(c),result(c)];
10 step
11 check isDef(response(fRequestResponse(c))) and isDef(result(c));
12 ...
```

Code 7: Request-response validation scenario in Avalla

The tool `AsmetaV` reads scenarios written in `Avalla` and executes them using the simulator `AsmetaS`; during simulation, `AsmetaV` captures any check violation and, if none occurs, it finishes with a *PASS* verdict (see Fig. 12).

As an example, the excerpt of the scenario reported in Code 7 describes the interactions among the client c and server agents s in the pattern request-response from the client side. Appropriate assertions control that the result message is sent. This scenario has to be used as a template and therefore it has then to be instantiated according to the real services and components involved in the pattern instance. An example of its instantiation will be given for the case studies presented in Sect. 5.

Model verification. Model checking is an automated formal verification technique based on state exploration of the system to be checked. `AsmetaSMV` [5] is a tool of the ASMETA framework that translates ASM specifications into models of the NuSMV model checker. It allows the verification of *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL) formulae expressing desired behavioral properties of the system under verification.

Behavioral properties related to interactions among the participants (agents) can be obtained from a pattern definition and then expressed and configured in terms of CTL formulae within the SCA-ASM model itself of the pattern instance. This last is then validated against the behavioral properties by using the `AsmetaSMV` model checker. Thereafter the state space of the SCA-ASM model to be checked is searched by the model checker to verify or falsify (by generating counterexamples) the CTL properties.

For our purposes, CTL can be used to express a temporal ordering between send and receive messages in a sequence of interactions among agents or also general properties such as *reactivity* and *liveness*. For example, the following CTL property can be checked to validate the *reactivity* of the behavior of an instance of the Request-Response micro-pattern:

```
ag(isDef(nextRequest(server)) implies af(isDef(result(client(server)))))
```

It specifies that whenever there is a request, eventually the server (agent *server* of type *RequestResponseService*) will reply producing a result for the client.

The formula has then to be instantiated according to the real services of the components involved in the pattern instance.

Liveness properties (informally, liveliness means that some actions will be executed infinitely) can be expressed and verified by giving explicit fairness requirements. For example, for the Ping-echo tactic, used to test the reachability of a host, the following property can be checked for a timed confirmed sender. Property: A ping message is always followed by an echo confirmation message or timeout:

```
ag( ping(sender,receiver) implies af( (maxWaitingTime(sender) and not(echo(receiver,sender)))
or (not(maxWaitingTime(sender)) and echo(receiver,sender))) )
```

where we assumed the following atomic propositions: *ping(sender,receiver)* ≡ a ping request has been sent from the sender to the receiver, *echo(receiver,sender)* ≡ an echo request has been sent back from the receiver to the sender, and *maxWaitingTime(sender)* ≡ the ping request is not answered with the echo reply within the given time.

Invariants, i.e. properties that must hold in all the states, can be also checked. For example, for a Request-Response micro-pattern we can verify that, after making the request in a synchronous manner by a send-receive action, the client effectively remains blocked until it receives back the result:

```
ag( isDef(awaitingRespMsg(client)) implies status(client) = BLOCKED)
```

where *client* is the client component's agent and the controlled function *awaitingRespMsg*, that is set within the predefined rule send-receive [7], stores the message for which the client agent is waiting to receive the corresponding response message. Invariants are useful, for example, for guaranteeing certain *safety* properties (informally, that nothing will go wrong with the system) by verifying that invariants' formulae are effectively true at all states of the system.

Currently, the derivation of CTL formulae from a pattern definition is carried out by hand. We postpone as future work the automatic generation of CTL formulae and their configuration with information from concrete pattern instances. Moreover, verification of properties on large SCA-ASM models is possible, but it would require the use of some *model slicing* and *model abstraction* techniques in order to avoid the well-known problem of state explosion of the model checking, and make the verification feasible.

5. Illustrative case studies and lessons learned

This section presents two case studies as illustrative examples of the proposed framework. In the first example, we designed and validated a service-oriented architecture model for the Order system case study [24]. As second example, we considered the Stock Trading system originally presented in [16]. Finally, the section reports our lessons learned as gained by our experience in developing these case studies.

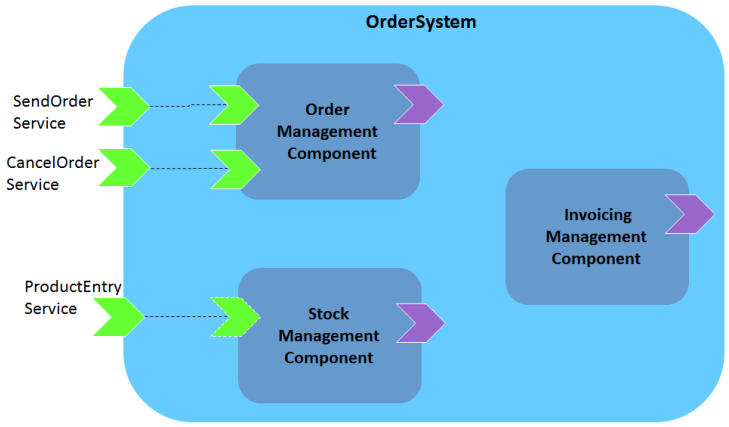

Figure 13: SCA assembly of the Order System

5.1. The Order System

The Order system is essentially an exercise of requirements capture, notoriously a difficult and error prone activity that requires a formalization task. To this purpose, we show how the SCA-ASM method allows one to capture informal functional requirements of a system architecture, including both the structural and behavioral aspects of services.

The main service of the system is that of invoicing orders. Every order refers to a product for a certain quantity (greater than zero). The same product can be referenced by several different orders. Every product is in the stock in different quantity. Invoicing requires to check if the order can be satisfied, i.e. if the ordered quantity of products is less than or equal to the quantity in stock. If so, the stock is updated and the order state changes from the state pending to the state invoiced. If the order cannot be satisfied, it is left pending.

Two additional services offered by the system are: cancel orders and add new quantities of products in the stock. A customer can only cancel his or her personal orders. The order is cancelled if it is still pending. When the order is already invoiced, the conflict must be resolved manually by the user. For the third functionality of adding a quantity of product in the stock, the product must be already registered in the system because the system does not consider entries of unreferenced products. So for each product entry, the supplier must specify the product provided and its quantity.

5.1.1. Step 1 – Sketch an outline of the application's architecture

According to Step 1 of the design methodology presented in Sect. 4.2, we modeled the initial SCA architecture of the system based (see Fig. 13) on the main service of invoicing orders, and the additional services for order cancellation and supply new products quantity. Essentially, we consider the following application scenarios.

Order management (order entry/cancellation). For an order entry scenario, the user requirements specify that an order is made by sending (the service operation `sendOrder(ref, qty, customerID)`) a reference to the desired product, a quantity, and a customer identifier to the system. So the methodology leads us to define a service-oriented component namely `OrderManagement`, which takes into account this entry and is in charge of saving internal orders. For an order cancellation, we assume that a customer must identify the order he wants to cancel and invokes the service operation `cancelOrder(orderID, customerID)` also offered by the component `OrderManagement`. This last checks that the customer can cancel the order (a customer can cancel only one of his or her orders, not the order of another customer) and if it is the case, cancels the order.

Stock Management. We assume that all products are already referenced in the system. So, when a supplier sends a new quantity of product to the `StockManagement` component (by invoking the service operation `productEntry (productID, qty)`), the product quantity is updated.

Invoicing Management (order invoicing). The `OrderManagement` component only registers in the database that there are new orders to invoice (i.e., orders initially pending), while the component `InvoicingManagement` is in charge of effectively invoicing orders. This last component does not expone any puplic service. It executes the order invoicing functionality in background. Essentially, it selects a set of orders which are invoicable, i.e. they are pending and refer to a product in the stock in enough quantity, it simultaneously changes the state of each order in this set from pending to invoiced, and updates the stock by subtracting the total product quantity in orders to invoice. The system keeps to invoice orders as long as there are orders which can be invoiced. The system guarantees that the state of an order is always defined and the stock quantity is always greater or equal to zero.

5.1.2. Steps 2 and 3 – Refine the application's architecture and finalize the interfaces

The second step of the methodology suggests to refine the architecture by introducing new components and using design patterns.

First, we decided to organize the overall system architecture according to the *three-layer architectural pattern*[5] by introducing the components `GUI`, `Application`, and `Data`. The component `Data` represents the data layer of a classical three-layer-architecture so it hides details of the database and provides data access to the application layer represented by the component `Application`. The component `Application` contains the application logic. It uses the services `QueryService` and `PersistenceService` defined by the component `Data` in order to send queries or changes to the database, and provides the interface `StoreService` to deliver results of database queries to the component `GUI`. This last acts like an interface for the user and the `Application` component.

[5]The templates definition and SCA assembly of such patterns are available online at [34].

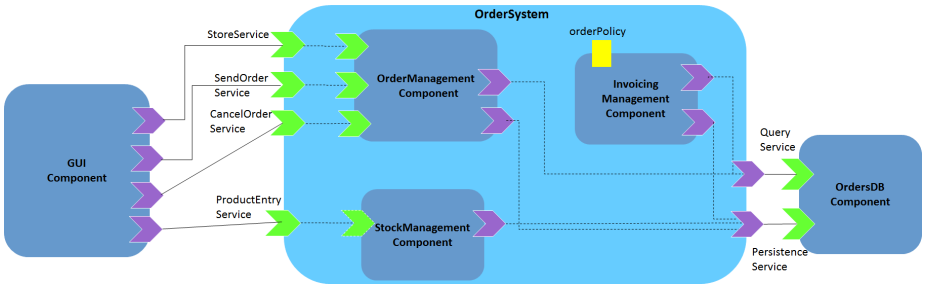

Figure 14: SCA Order system refined (Step 2 - `GUIApplicationData` design pattern)

We identify the `OrderSystem` composite component introduced in the previous step as the component `Application`. So according to the three-layer architectural pattern, we have to refine the `OrderSystem` component to allow the interaction with two other external components: the `OrdersDBComponent` (the `Data` component) and the `GUIComponent` (the `GUI`). These last are left in abstract. The result of such refinement is shown in Fig. 14.

According to the user requirements, the `InvoicingManagment` component can adopt different selection strategies of orders to invoice: *single-order*, *all-or-none*, *max-orders*, and *default*. Single-order strategy means that per step at most one order is invoiced, with an unspecified schedule (thus also not taking into account any arrival time of orders). In case all orders for one product are simultaneously invoiced or none if the stock cannot satisfy the request, a all-or-none strategy can be expressed. To further maximize a product quantity invoiced at the time, a new strategy (strategy max-orders) consists in choosing a maximal invoicable subset of simultaneously invoiced pending orders for the same product. If the user requests a selection strategy which is not driven by a first choice of a product, another possible strategy consists in choosing a set of pending orders, with enough referenced products in the stock, to be simultaneously invoiced. This last strategy matches the intended behavior of the system better than the previous ones, so it is the default strategy. For supporting one of this mode of operation, we added to the `InvocingManagement` component the property `OrderPolicy` (see Fig. 14) whose value range in the set {single-order, all-or-none, max-orders, or default}. By default, this property is initialized to the value *default*.

As further refinement, we applied the *router pattern*[6] to enable different invoicing strategies as implemented by different business components: `SingleOrderBComponent`, `AllOrNoneBComponent`, `MaxOrdersBComponent`, and `DefaultBComponent` (see Fig. 15). The router component, i.e. the `InvoicingManagment` component, is responsible for sending the pending orders to a certain business component for invoicing them according to a specific strategy (the routing cri-

[6]The templates definition and SCA assembly of such patterns are available online at [34]

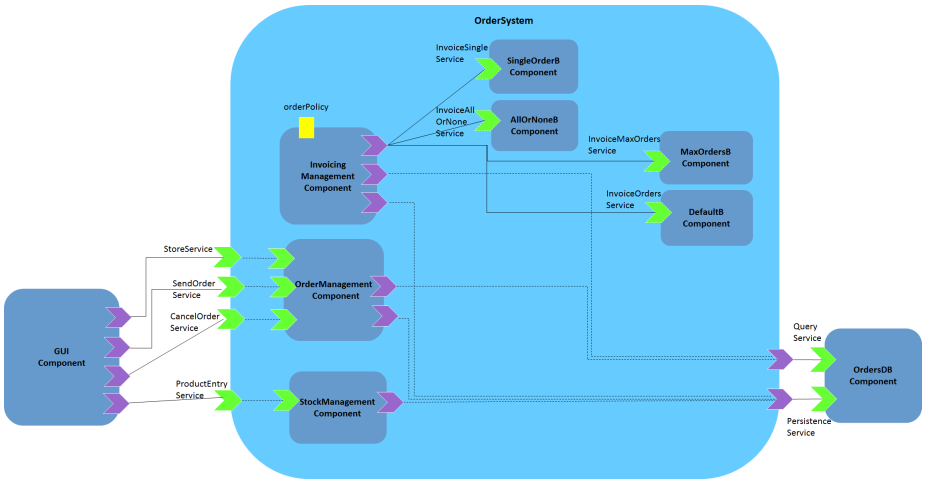

Figure 15: SCA Order system refined (Step 2 - `Router` design pattern)

Code 8: ASM module of the `InvoiceOrdersService` interface

```
1 module InvoiceOrdersService
2 import STDL/StandardLibrary
3 import Order //Interface of the Order data type
4 signature:
5 // the domain defines the type of the provider component's agent
6 domain InvoiceOrdersService subsetof Agent
7 out invoiceOrders: Prod(Agent, Powerset(Order)) -> Powerset(Order)
```

teria) as reflected by the current value of the property `OrderPolicy`.

The third step of the methodology is to finalize the interfaces and properties of each component in the SCA assembly. So we added the service operations to the interfaces and properties types.

5.1.3. Steps 4′ and 5 – Map design to formal specification and formal validation

Being a design from scratch, before implementing components in Java we preferred specifying the behavior of the business components for invoicing orders formally using SCA-ASM. The complete SCA-ASM implementation of such components is also available at [34].

The default invoice strategy (the `InvoiceOrdersService` service) is provided by the business component `DefaultBComponent`. The ASM definition for the interface `InvoiceOrdersService` is reported in Code 8. It is an ASM module containing only declarations of a business agent type (the subdomain `InvoiceOrdersService` of the predefined ASM `Agent` domain) and of a business function used as temporary location to store service computation results (i.e., the orders to invoice) to return back to the service caller.

The behavior of the `DefaultBComponent` is formalized in SCA-ASM as reported in Code 9. The service rule `invoiceOrders` uses a predicate `invoicable`

Code 9: The behavior of the `DefaultBComponent`

```
asm DefaultBComponent
import STDL/StandardLibrary
import STDL/CommonBehavior
import InvoiceOrdersService
signature:
//@Backref
shared client: Agent -> Agent
// orders to invoice
controlled orders: Agent -> Powerset(Order)
definitions:

rule r_DeleteStock($p in Product ,$q in Natural) = stockQuantity($p):= stockQuantity($p) - $q

//@Service Choose subset of orders
rule r_invoiceOrders($a in Agent, $orders in Powerset(Order)) =
 choose $orderSet in Powerset($orders) with invoicable($orderSet) do
        par
                forall $order in $orderSet with true do orderState($order) := INVOICED
                forall $product in referencedProducts($orderSet) with true do
                        r_DeleteStock[$product, totalQuantity($orderSet,$product)]
                invoiceOrders(self,orders(self)) := $orderSet //setting of the out business function
        endpar

rule r_DefaultBComponent =
if nextRequest(self)="r_invoiceOrders(Agent, Powerset(Order))" then
 seq
        r_wreceive[client(self),"r_invoiceOrders(Agent, Powerset(Order))",orders(self)]
        if (isDef(orders(self))) then r_invoiceOrders[self,orders(self)] endif
        r_wreply[client(self),"r_invoiceOrders(Agent, Powerset(Order))",invoiceOrders(self,orders(self))]
 endseq
endif

rule r_init($a in InvoiceOrdersService) = status($a):=READY
```

that is true on a set of pending orders with enough quantity of requested products in the stack, and a function `refProducts` which yields the set of all products referenced in a set of orders. Note that the non-deterministic selection of the orders to invoice could be performed by a input function which would formalize the user selection of a set of orders or the results of a particular scheduling.

A single-order strategy is realized by the service `invoiceSingleOrder` as provided by the `SingleOrderBComponent`. This service's behavior is formalized in SCA-ASM as reported in Code 10. Per step at most one order is invoiced, with an unspecified schedule (by the `choose` rule contructor).

The `InvoiceAllOrNone` service of the `AllOrNoneBComponent` for the all-or-none strategy can be specified in SCA-ASM as reported in code 11. The service rule makes use of a function `pendingOrders` yielding the set of pending orders for a certain product, and of a (static) function `totalQuantity` returning the total quantity of a set of orders.

Finally, the service `InvoiceMaxOrders` can be formalized in SCA-ASM as shown in code 12. For this rule we need to define a static function `maxQuantitySubsets` which, given a set of set of orders, returns the set of all the sets having a maximum quantity.

The Order system case study was essentially a requirements formalization

Code 10: The behavior of the `invoiceSingleOrder` service

```
1  //@Service Invoice an order at a time.
2  rule r_invoiceSingleOrder($a in Agent, $orders in Powerset(Order)) =
3          choose $order in orders with orderState($order) = PENDING do
4           if(orderQuantity($order) <= stockQuantity(referencedProduct($order))) then
5                   par
6                    orderState($order) := INVOICED
7                    r_DeleteStock[referencedProduct($order),orderQuantity($order)]
8                    invoiceSingleOrder(self,orders(self)) := $order //setting of the out business function
9                   endpar
10          endif
```

Code 11: The behavior of the `InvoiceAllOrNone` service

```
1  //@Service All orders for one product are simultaneously invoiced or none.
2  rule r_invoiceAllOrNone($a in Agent, $product in Product) =
3  let ( $pending = pendingOrders($product) ) in
4   let ( $total = totalQuantity($pending) ) in
5          seq
6           if $total > 0 and $total <= stockQuantity($product) then
7            par
8                   forall $order in $pending do orderState($order) := INVOICED
9                           r_DeleteStock[$product, $total]
10                  endpar
11          endif
12          invoiceAllOrNone(self,product(self)) := $pending //setting of the out business function
13         endseq
14  endlet
15 endlet
```

Code 12: The behavior of the `InvoiceMaxOrders` service

```
1  //@Service Invoice maximum orders for one product.
2  rule r_InvoiceMaxOrders($a in Agent, $product in Product) =
3   let ($pending = pendingOrders($product)) in
4    let ($invoicablePending = {$o in $pending | totalQuantity($o) <= stockQuantity($product) : $o}) in
5           choose $orderSet in maxQuantitySubsets($invoicablePending) do
6                   par
7                    forall $order in $orderSet do orderState($order) := INVOICED
8                           r_DeleteStock[$product, totalQuantity($orderSet)]
9                           invoiceMaxOrders(self,product(self)) := $orderSet //setting of the out business function
10                  endpar
11         endlet
12 endlet
```

```
1  scenario DefaultInvoicingManagement
2  load main.asm
3  ...
4  //for the startup of the client and server agents
5  set status(c) := READY;
6  set fRequestResponse(c) := s;
7  set orders(c):= ...;
8  set ...
9  exec r_wsendreceive[fRequestResponse(c),"r_invoiceOrders(Agent,Powerset(Order))",orders(c),result(c)];
10 step
11 check isDef(invoiceOrders(fRequestResponse(c),orders(c))) and isDef(result(c));
```

Code 13: Validation scenario in Avalla for the `InvoicingManagementComponent` – default strategy

task and we have shown how the SCA-ASM method allowed us to capture informal functional requirements of systems services by constructing a consistent and unambiguous, simple and concise, abstract and complete models of service-oriented components, including behavioral aspects of services. These models can be understood and checked (for correctness and completeness) by both domain experts and system architects/designers. For example, Code 13 reports a simulation scenario in Avalla (instantiated from the one reported in Code 7) for checking the orders invoicing. It checks the interactions among the `OrderDeliveryComponent` (the client c) and the default business component (the server agent s) in the pattern request-response from the client side. Appropriate assertions control that the result message (the set of orders to invoice as chosen by the default strategy) is sent.

Through the requirements capture we have introduced several assumptions to fill missing information. We introduced some assumptions directly in the specification by means of invariants and used the `AsmetaS` simulator to check them during simulation. For example, the assumption that the quantity in every order must be greater than 0 is formalized as:

```
invariant over orderQuantity:
forall $o in Order with orderQuantity($o) > 0
```

We have also stated the following desired properties which express state invariants and correctness conditions. The first one states that the stock quantity is always greater than 0.

```
invariant over stockQuantity:
forall $p in Product with stockQuantity($p)>=0
```

The second property is that the state of every order is either pending or invoiced, but never undefined.

```
invariant over orderState:
forall $o in Order with orderState($o) != undef
```

For other more complex properties, which are not state invariants but they refer to execution paths, the model checker can be used, although assumptions about the finiteness of the domains are necessary and uninterpreted domains are not allowed. For example, one may want to express that an order o is eventually

invoiced if it refers to a product available in the stock in enough quantity. In CTL, this can be expressed as:

```
AF( AG( orderState(o) = INVOICED or
orderQuantity(o) > stockQuantity(referencedProduct(o)))
```

5.1.4. Steps 5 and 6 – Mapping design to implementation and overall design validation

In a further iteration of the methodology, we implemented in Java (Step 4) the components `OrderManagement`, `InvoicingManagement`, and `StockManagement` as mock components, and then execute and validate the overall SCA assembly of the application within the SCA runtime platform (Step 6). We did not effectively implemented all the system in Java since, being an example of requirements elicitation, our main goal was to develop a ground model showing how the SCA-ASM formal method allows one to capture informal behavioral requirements of components' services. So, for validation we used orders collections and customers identifiers with fixed values to avoid at this design phase the implementation of a real database component.

5.2. The Stock Trading System

The goal of such a case study is to embody non-functional requirements (NFRs) using architectural tactics. We chose such a case study as major evaluation of the SCA-PatternBox language and tool and as a comparative benchmark because our work is on the spirit of the approach in [16] for design pattern specification and application (see related work in Sect. 6.1), but instead of extending UML diagrams like the approach in [16] does, we preferred just to specify what is really needed to express a design pattern for facilitating pattern instantiation and code generation.

Our case study was adopted in the adaptation exploration process presented in [15]. It consists of an optimization process with respect to different architectural configurations of the system for multiple adaptation scenarios (e.g., a user claims a new level of reliability and response time, or the monitor raises the violation of the minimum level of required availability).

Figure 16 shows the initial SCA assembly of the STS obtained by carrying out Step 1 of the design methodology presented in Sect. 4.2. Briefly, an STS user, through the `OrderWebComponent` interacting with the `OrderDelivery-Component`, can check the current price of stocks, placing buy or sell orders and reviewing traded stock volume. Moreover, he/she can know stock quote information through the `StockQuoteComponent`. STS interacts also with an external Stock Exchange system.

By executing steps 2 and 3 of the methodology, architectural tactics are selected, composed and instantiated based on a given set of NFRs to refine the initial architecture of the application into one that meets the desired NFRs. We here show, in particular, how availability and performance tactics can be used to embody NFRs into the SCA architecture of the STS application. Let us assume the following NFRs (as taken from [16]):

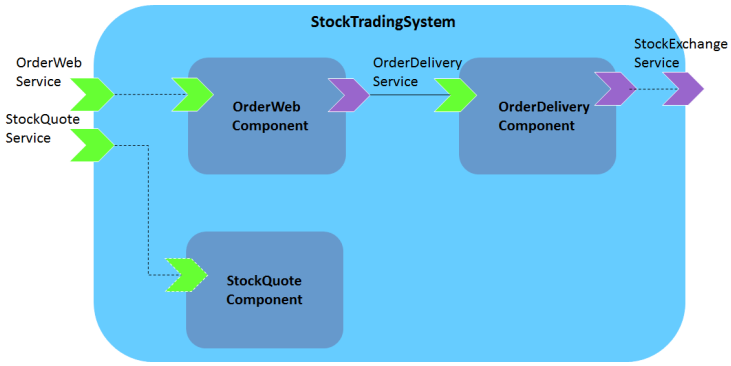

Figure 16: SCA assembly of the Stock Trading System

– NFR1. *The STS should be available during the trading time (7:30 AM – 6:00 PM) from Monday through Friday. If there is no response from the system for 30 s, the STS should notify the administrator.*
– NFR2. *The system should be able to process 300 transactions per second, 400,000 transactions per day. A client may place multiple orders of different kinds (e.g., stocks, options, futures), and the orders should be sent to the system within 1 s in the order they were placed.*

We support NFR1 by applying the Fault Detection tactics *Ping/Echo* and *Heartbeat* through a monitoring component playing the role of `PingHeartbeatReceiver`. NFR2 is supported by combining the tactics *FIFO* and *Introduce Concurrency*. The *FIFO* tactic allows clients to place each type of orders (e.g., stocks, options, futures) to a dedicated queue for immediate processing. The *Introduce Concurrency* tactic allows the concurrent dispatching of the same kind of orders thus reducing the blocking time of transactions on I/O. Figure 17 shows the new SCA assembly obtained by composing these tactics: the assembly is extended to add the new component `Monitoring` (for the Fault Detection tactics) and to refine the existing components `OrderWebComponent` and `OrderDeliveryComponent`. These last two components are to be intended as subsystems, indeed they are composite components with a hierarchical design not further reported here. In particular, the `OrderDeliveryComponent` is refined for adding a queue sub-component for the FIFO tactic, a sub-component for the functionality of a `PingHeartbeatReceiver` role and for the concurrent consuming of different kinds of orders placed into the queue sub-component. Similarly, the `OrderWebComponent` is refined by adding a sub-component for concurrently producing orders to place into the queue of `OrderDeliveryComponent`. Of course, this refinement implies a change of the components shape (i.e., in the required/provided interfaces) and of their behavior.

Some components were implemented in SCA-ASM (Step 4′). As an example, Code 14 shows a fragment of an SCA-ASM specification of the `OrderDeliveryComponent`. The main service of this component (the rule `r_place` annotated with `@service`) is to place buy or sell orders when requested (see the blocking

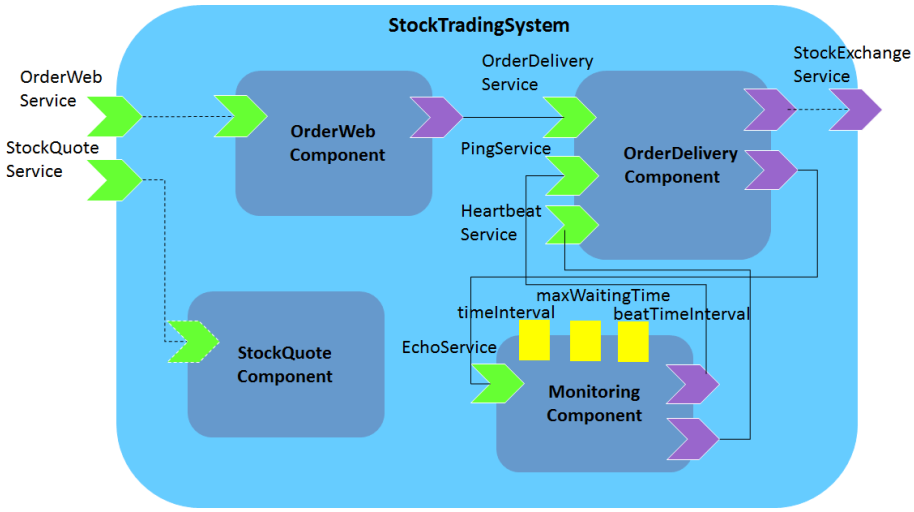

Figure 17: SCA assembly of the STS after applying tactics for NFR1 and NFR2

Code 14: The behavior of the `OrderDeliveryComponent`

```
1  module OrderDeliveryComponent
2  ...
3  @Service
4  rule r_place($client in Agent,$o in Order)= ... //to place buy or sell orders
5  ...
6  rule r_OrderDeliveryComponent=
7   seq
8    r_wreceive(client(self),"place",order(self))
9    r_place(client(self),order(self))
10   r_wreply(client(self),"place",place(self,order(self)))
11  endseq
12 ...
```

receive action and the reply action preceding and following, respectively, the service invocation within the component's main rule `r_OrderDeliveryComponent`). The ASM definition for the provided and required interfaces of the `OrderDeliveryComponent` are reported in Code 15. They are ASM modules containing only declarations of business agent types (the subdomains `OrderDelivery` and `StockExchange` of the predefined ASM `Agent` domain) and of business functions (parameterized ASM out functions) used as temporary locations to store service computation results.

Code 16 reports a simplified fragment of a simulation scenario (instantiated from the one reported in Code 7) for checking the order placement executed by the `OrderDeliveryComponent`.

The SCA-ASM specification of some components behavior were further refined to reflect at behavioral level the Introduce Concurrency tactic. The behavior, for example, of the `OrderDeliveryComponent` was further refined in ASM as shown in the fragment reported in Code 17: the consuming and sending of

Code 15: ASM modules of the `OrderDeliveryComponent` interfaces

```
module OrderDeliveryService
import ... //Other module imports
signature:
// the domain defines the type of the provider component's agent
domain OrderDelivery subsetof Agent
// business function value
out place: Prod(Agent,Order) −> Order

//@Remotable
module StockExchangeService
import ... //Other module imports
signature:
domain StockExchange subsetof Agent
out sendOrder: Prod(Agent,Order) −> Rule
```

```
scenario OrderPlacement
load main.asm
...
//for the startup of the client and server agents
set status(c) := READY;
set fRequestResponse(c) := s;
set order(c):= ...;
set ...
exec r_wsendreceive[fRequestResponse(c),"r_place(Agent,Order)",order(c),result(c)];
step
check isDef(place(fRequestResponse(c),order(c))) and isDef(result(c));
```

Code 16: Validation scenario in Avalla for `OrderDeliveryComponent`

different kind of orders (stock, option, or future) are executed concurrently by the `par` rule. The availability of such pattern specified in abstract terms by ASM allowed us to formally validate early in the design the benefits provided by such a tactic (Steps 5 and 6).

5.3. Lessons Learned

We found the modeling approach to the definition of design patterns and their instantiation in SCAPatternBox particularly useful. First, exploiting the existing PatternBox tool for Java allowed us to develop a large and sophisticated tool for SCA in a very short period of time. The ability to work with design patterns in conjunction with SCA and supporting implementation types was the major benefit. We found it very sympathetic to the implementation of the skeleton code of the application that is obtained automatically from an SCA assembly model and through pattern instantiation.

SCA-ASM allows modeling both structure and behavior of service components in a unique framework integrating architectural and behavioral views. Moreover, SCA-ASM gives us an avenue for the formal validation of the pattern and of the application model to which the pattern is applied. ASM rigorousness, expressiveness, and executability allowed us to reason and validate some services interaction patterns in a formal way but without mathematical overkill. Formal validation can be carried out on SCA-ASM specification fragments re-

Code 17: The refined behavior of the `OrderDeliveryComponent`

```
1  module OrderDeliveryComponent
2  ...
3  rule r_OrderDeliveryComponent=
4  ... seq
5        par //Queue consuming
6          r_wsendreceive[queue(self),"dequeue",("Stock",stockorder(self))]
7          r_wsendreceive[queue(self),"dequeue",("Option",optionorder(self))]
8          r_wsendreceive[queue(self),"dequeue",("Future",futureorder(self))]
9        endpar
10       par //Order sending to the Stock Exchange system
11         r_wsend(stockExchange(self),"sendOrder",(self,stockorder(self)))
12         r_wsend(stockExchange(self),"sendOrder",(self,optionorder(self)))
13         r_wsend(stockExchange(self),"sendOrder",(self,futureorder(self)))
14       endpar
15     endseq ...
```

sulting from the instantiation from scratch of single design patterns or from the composition of two or more patterns, or on an entire SCA-ASM specification resulting from the application of one or more design patterns on an existing SCA assembly (partially) implemented in SCA-ASM.

During the development of the case studies, for example, we implemented some components in SCA-ASM in order to have an ASM (abstract) formal specification of their behavior. We then validated the corresponding SCA-ASM specification fragments before and after the application of some formally-validated design patterns, (such as the application of the three-layer and Router patterns to the SCA assembly of the Order system, and of the the Ping-echo and Heartbeat tactics to the STS SCA assembly). We experienced that ASM is a good formalism for prototyping and simulation purposes. We have carried out model validation through the SCA-ASM simulation environment. Model validation is a model analysis activity to be executed from the earlier stages of the model development before other more demanding but more sophisticated and complex analysis techniques such as formal verification by model checking. Formal high-level ASM specifications of SCA components can be assembled together with other SCA components implemented in a different technology (e.g., in Java) and the resulting heterogeneous SCA assembly can be managed and executed within the SCA Tuscany runtime platform as. So SCA components can be early validated at high level of formalization, without caring about implementation details.

During the formalization task of the business components in the Order system case study, we discovered how requirements are often incomplete and assumptions must be stated in order to complete the specification. To this purpose we noted how the SCA architecture (including design patterns) and the SCA-ASM behavioral specification of a service are easy to adapt when different interpretations of same requirements are possible (for example, to support different selection strategies of orders to invoice), and how the rigor of the ASM ground model allows formal validation and verification of properties. Model validation, in general, allowed us to reproduce component configuration scenarios

with different pattern instantiations and be confident that the model behaved as expected. We found this early validation is a great means for evaluating architectural choices and alternative designs with limited implementation effort, but it requires to be skilled in the ASM formal method. On this last point, it is widely recognized that ASMs is a lightweight formal method since ASMs are a precise abstract form of pseudo-code, generalizing (the familiar) Finite State Machines to operate over arbitrary data structures [8].

6. Evaluation of software design pattern languages

Before starting our work, we evaluated the current state of the art. We, in particular, analyzed existing design pattern languages and their supporting tools. This section reports the outcome of such evaluation. In Subsection 6.1 we describe the main approaches existing in the literature, while in Subsection 6.2 we briefly compare them (including our proposal, i.e., the SCA-PatternBox pattern language) with respect to some specific criteria.

6.1. Related work

In the literature, there are several notations and tools supporting the process of pattern-based system design and development. Some of these languages are based on formal mathematical notations (such as [17–21]) or ontology (such as [22]). Other languages are based on the OMG standard UML notation [23, 25–27]. Finally, some other languages are based on XML or general purpose programming languages [28–30]. Below, we describe some of these approaches.

The Language for Pattern Uniform Specification (LePUS) [17] is based on mathematics and formal logic. It describes only the structure of design patterns and provides a weak basis for integrated tool support. The language eLeLePUS [18] tries rectify the shortcomings of LePUS.

In [19] another formal language is proposed for the behavioral specification of the GOF design patterns. It is based on the language of temporal ordering specification (LOTOS). This LOTOS adaptation to patterns did not yield simple and clear specifications. Distributed Co-operation (DisCo) [20] is another specification language for design patterns based on an action system to specify the behavioral aspects of a pattern.

The Balanced Pattern Specification Language (BPSL) [21] formally specifies the structural as well as behavioral aspects of patterns. It was derived from LePUS and DisCo, and therefore shares many of the advantages and disadvantages of the two languages.

Conceptual Ontology Design Pattern (CODeP) [22] is based on the Web Ontology Language (OWL) and Resource Description Framework (RDF) for engineering ontology content over the Semantic Web.

Many semi-formal UML-based notations exist. Among the most notable ones are the following. *ArchInst* [31] is a tool developed as a plug-in of the IBM Rational Software Architect (RSA) for UML to support quality-driven development of software architectures. It allows to configure architectural tactics

based on quality requirements and compose the configured tactics to produce an initial architecture of the system under development. The tool uses the Role-Based Metamodeling Language (RBML), a UML-based pattern specification notation presented in a previous work [23], to specify tactics. This UML extension defines a design pattern in terms of roles and role dependencies. The concept of *ClassifierRole* that RBML uses has been, however, superseded in UML 2.x. In [16, 32], a prototype tool, called RBML Conformance Checker, has been also presented for verifying the conformance of UML models to design patterns. This work demonstrates how instantiated elements conform to their corresponding metamodel elements. *PerOpteryx* [25] is an Eclipse-based optimization framework to improve component-based software architectures for performance, reliability, and cost through model-based quality prediction techniques and architectural tactics. It is based on the Palladio Component Model (PCM) and a UML-like modeling notation that uses annotated UML models as software design models. In [26] a prototype tool named *DPTool* is presented. It is based on the DPML (Design Pattern Modeling Language) notation for the specification of design patterns and their instantiation into UML design models. However, the proposed modeling constructs are more complex than other similar UML-based modeling notations. In [27] is presented the problem to prevent defect injection during design-patterns maintenance. A design method called Pattern Instance Changes with UML Profiles (PICUP) has been developed to this purpose as an improved design method for a corrective UML pattern-based design maintenance and assessment after variations.

These are design pattern languages that are based on general purpose programming languages. For example, a Prolog-like language called SPINE [28] allows patterns to be defined in terms of constraints on their implementation in Java. It makes addition of patterns and variants easier for those who have programmed in PROLOG before, rather than creating an entirely new syntax. These patterns can then be processed using a proof engine called Hedgehog. Hedgehog reads the SPINE definitions, along with Java source code, and attempts to automatically prove whether or not the class correctly realizes the design pattern. In [29], AspectJ (aspect-oriented extension to the Java programming language) is used to provide some improvements to the GoF OO design patterns. These improvements allow several advantages like better code locality, reusability, and composability, but are at code level. No abstract modeling notation for pattern specification is supported.

Finally, the Design Pattern Definition Language (DPDL) [30] allows to specify both the structural and behavioral aspects of design patterns. It is a text-based pattern description language purely based on XML, but it is endowed also with a graphical representation through an automated transformation of DPDL descriptions into UML class and sequence diagrams.

6.2. A comparison of design pattern languages

To compare existing design pattern languages, we revised and extended the comparison framework originally presented in [30]. The criteria adopted for comparing the languages reflect the main and generally agreed objectives in

proposing a language for defining, sharing and applying design patterns into concrete system designs. Essentially, a design pattern language should be:

- Easy to understand and use: The language should be easily understandable by the designers/developers and thus easy to use.
- Unambiguous: The language should have a clear semantics enabling its sharing among design/development teams without any ambiguity.
- Extensible: Because technology is progressing, the language should be capable of being extended.
- Based on existing technology/languages: The language should be based on existing and common technology/languages so that it gains wider and faster acceptance.
- Able to support also a graphical notation: A visual representation of a design pattern and/or a pattern instance provides an intuitive and lightweight overview for the user.

Table 3 below summarizes the results of such a comparison by reporting the main features of the design pattern languages described in Subsection 6.1 and of our proposed language. The main difference with respect to our proposal, is that all the existing approaches that we considered are specific to object-oriented system design. They do not address design patterns related to SOA. Moreover, most of these approaches are not *implementation-oriented*; indeed, they provide the needed formality in the pattern specification, for example at UML level, but often at the expense of usability and programmaticity of the approach because they do not aim at generating the corresponding implementation code. In particular, languages that purely use a mathematical formalism for modeling design patterns allows to specify the behavioral aspects of a design pattern in a rigorous and unambiguous way, but often they are considered not easy to use because they require strong mathematical background to the user and lack of good tool support. In conclusion, to the best of our knowledge, there is a lack of tools supporting an "implementation-oriented" definition and use of design patterns, especially in the SOA domain. So the experience gained with the development of SCA-PatternBox makes us optimistic.

Table 3: Design pattern languages' features comparison (extended from [30])

	Basis	Integration in IDEs	Platform independence	Template support	UML support	Learning curve	Graphical	Target	Service oriented
LePUS	Mathematical logic	No	N/A	Yes	No	Required strong mathematical background (high)	No	Verification of design pattern on first order logic basis	No
eLePUS	Mathematical logic	No	N/A	Yes	No	Required strong mathematical background (high)	No	Design pattern designing through mathematical formalism	No
DPML	UML based	No (separate IDE)	Yes	Yes	Yes	UML knowledge required (medium)	UML	Creating design patterns in UML	No
RBML	UML based	No	Yes	No	Yes	UML knowledge required (medium)	UML	Adding support of design pattern in UML	No
DisCo	Temporal logic of action (TLA)	No	N/A	Yes	Yes	Based on rigid formal apparatus (high)	No	Capturing behavioral aspect of design patterns	No
SPINE	Prolog	No	N/A	Yes	No	Required Prolog understanding (hard)	No	Verification of design pattern implementation in application	No
CODeP	OWL (Web Ontology)	No	Yes	No	No	Knowledge of RDF and OWL required (hard)		Creation of knowledge artifacts based on RDF	No
LOTOS	TLA	No	N/A	No	No	Based on rigid formal apparatus (high)	No	Verifying the behavioral aspect of design patterns	No
BPSL	TLA	No	No	Yes	No	Based on rigid formal apparatus (high)	No	Capturing behavioral aspect of design patterns	No
DPDL	XML	Yes (using XML)	Yes	Yes	Yes	XML knowledge is required (Low)	Yes - Optional	Easy initiation and implementation in software development	No
PCM	UML based	Yes	Yes	Yes	Yes	UML knowledge required (medium)	Yes	Prediction of QoS	No
PICUP	UML based	Yes	Yes	Yes	Yes	UML knowledge required (medium)	Yes	Pattern design mantainer	No
SCA-PatternBox Language (our approach)	XML, SCA, ASM	Yes	Yes	Yes	No	Scalable: – Knowledge of XML, SCA (low effort) – Knowledge of ASMs (high effort)	Yes - Optional	Prototyping and formal validation of service-oriented software	Yes

7. Conclusion and future work

We presented a methodology and a supporting framework SCA-PatternBox for the design and prototyping of service-oriented applications with design patterns. The framework allows the definition and the semi-automated application of design patterns into the design of a service-oriented software architecture. We evaluated the usability and usefulness of the framework on the Order system case study and on a quality-driven adaptation scenario of the Stock Trading System [15] where patterns and tactics required would have been difficult to apply and combine manually without the availability of a tool like SCA-PatternBox. We also provided a comparison of the SCA-PatternBox's language with existing design pattern languages.

There are a number of issues that we want to address in the future:

– Automation support for design pattern composition to create complex pattern hierarchies through composition strategies.

– Formal strategies and techniques to check the structural and behavioral conformance of the component assemblies to the applied design patterns.

– Transformation support for compatibility with UML component diagrams, since UML is the standard language for system modeling.

– Definition of more sophisticated SOA patterns.

– Support for more SCA component implementation types.

Moreover, a software engineering area that is gaining importance for the maintenance and evolution of software systems is reverse engineering [33]. A real challenge in this context is to obtain representations of a software system at a higher level of abstraction and to identify the fundamental components, its constituent structures and design patterns. As future work, we want to extend the proposed framework with a methodology for the detection of design patterns and the reconstruction of service architecture models from the source code.

Finally, since Cloud service providers are expanding their offerings to include cloud-native software services (other than foundational hardware and platforms to application components), we would also like to extend our framework to support the new standard TOSCA [35] for modeling cloud-based applications and design patterns related to the Cloud domain.

References

[1] Thomas Erl, SOA Design Patterns. Dec 31, 2008 by Prentice Hall

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

[3] PatternBox website, —www.patternbox.com/—.

[4] Apache Tuscany, —http://tuscany.apache.org/—.

[5] P. Arcaini, A. Gargantini, E. Riccobene, AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In *Abstract State*

Machines, Alloy, B and Z (LNCS) Vol. 5977. Springer Berlin Heidelberg, 61–74, 2010.

[6] E. Riccobene, P. Scandurra, A modeling and executable language for designing and prototyping service-oriented applications, in: EUROMICRO Conf. SEAA 2011.

[7] E. Riccobene, P. Scandurra, A formal framework for service modeling and prototyping, Formal Asp. Comput. 26 (6) (2014) 1077–1113.

[8] E. Börger, R. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer Verlag, 2003.

[9] P. Scandurra, B. Nodari, S. Capelli, SCA-PatternBox: an eclipse-based design pattern editor for service component architectures, in: ECLIPSE-IT'2012, Sept. 20-21, 2012, Naples, Italy, 2012.

[10] P. Scandurra, S. Capelli, A practical and automated approach for engineering service-oriented applications with design patterns, COMPSAC Workshops 2014, Vasteras, Sweden, July 21-25, 2014, IEEE, 2014, pp. 684–689.

[11] Asmeta website. —asmeta.sf.net/—

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

[13] A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A Scenario-Based Validation Language for ASMs. In E. Börger, M. Butler, J. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z*, *LNCS* vol. 5238, pages 71–84. Springer Berlin Heidelberg, 2008.

[14] A. Gargantini, E. Riccobene, and P. Scandurra. A Metamodel-based Language and a Simulation Engine for Abstract State Machines. *J. of Universal Computer Science*, 14(12):1949–1983, 2008.

[15] R. Mirandola, P. Potena, P. Scandurra, Adaptation space exploration for service-oriented applications, Sci. Comput. Program., 80 (2014) 356–384.

[16] S. Kim, D. Kim, L. Lu, S. Park, Quality-driven architecture development using architectural tactics, JSS, 82 (8) (2009) 1211–1231.

[17] J. Nicholson, E. Gasparis, A. H. Eden, R. Kazman, Verification of design patterns with lepus3, in: First NASA Formal Methods Symposium Proceedings - NFM 2009, Moffett Field, California, USA, April 6-8, 2009., Vol. NASA/CP-2009-215407 2009, pp. 76–85.

[18] R. R. Raje, S. Chinnasamy, eLeLePUS - a language for specification of software design patterns, in: Proc. of the ACM Symposium on Applied Computing, SAC '01, ACM, New York, USA, 2001, pp. 600–604.

[19] M. Saeki, Behavioral specification of GOF design patterns with LOTOS, in: 7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5-8 December 2000, Singapore, IEEE Computer Society, 2000, pp. 408–415.

[20] T. Mikkonen, Formalizing design patterns, in: Proceedings of the 20th International Conference on Software Engineering, ICSE '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 115–124.

[21] T. Taibi, D. Ngo, Formal specification of design pattern combination using BPSL, Information and Software Technology 45 (3) (2003) 157–170.

[22] A. Gangemi, Ontology design patterns for semantic web content, in: 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, Nov. 6-10, 2005, Proc., LNCS Vol. 3729, Springer, 2005, pp. 262–276.

[23] R. B. France, D. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Trans. Software Eng. 30 (3) (2004) 193–206.

[24] Marc Frappier, Henri Habrias, Software Specification Methods: An Overview Using a Case Study, Springer London, 2001.

[25] A. Koziolek, H. Koziolek, R. Reussner, Peropteryx: automated application of tactics in multi-objective software architecture optimization, in: I. Crnkovic, J. A. Stafford, D. C. Petriu, J. Happe, P. Inverardi (Eds.), QoSA/ISARCS, ACM, 2011, pp. 33–42.

[26] D. Mapelsden, J. Hosking, J. Grundy, Design pattern modelling and instantiation using DPML, in: Proc. of the Fortieth Int. Conf. on Tools Pacific, CRPIT '02, Australian Computer Society, Darlinghurst, 2002, pp. 3–11.

[27] Jaeyong Park, Seok-Won Lee, and David C. Rine, UML design pattern metamodel-level constraints for the maintenance of software evolution. Software: Practice and Experience Special Issue: Pattern Languages: Addressing the Challenges Volume 43, Issue 7, pages 835866, July 2013

[28] A. Blewitt, A. Bundy, I. Stark, Automatic verification of design patterns in java, in: 20th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2005), Nov. 7-11, 2005, Long Beach, CA, USA, ACM, 2005, pp. 224–232.

[29] J. Hannemann, G. Kiczales, Design pattern implementation in java and aspectj, SIGPLAN Not. 37 (11) (2002) 161–173.

[30] S. Khwaja, M. Alshayeb, Towards design pattern definition language, Softw., Pract. Exper. 43 (7) (2013) 747–757.

[31] S. Kim, D.-K. Kim, S. Park, Tool support for quality-driven development of software architectures, in: C. Pecheur, J. Andrews, E. D. Nitto (Eds.), ASE, ACM, 2010, pp. 127–130.

[32] D.-K. Kim, W. Shen, Evaluating pattern conformance of uml models: A divide-and-conquer approach and case studies, Software Quality Control 16 (3) (2008) 329–359.

[33] E. Chikofsky, I. Cross, J.H., Reverse engineering and design recovery: a taxonomy, Software, IEEE 7 (1) (1990) 13–17. doi:10.1109/52.43044.

[34] SCA-PatternBox repository —https://github.com/stevencapelli/SCA-PatternBOX—.

[35] Topology and Orchestration Specification for Cloud Applications (TOSCA) —http://docs.oasis-open.org/tosca/tosca/v1.0/os/tosca-v1.0-os.html—.