Modeling and refinement SOA design patterns with Event-B method

Imen Tounsi, Mohamed Hadj Kacem, Ahmed Hadj Kacem, Khalil Drira

Modeling and refinement SOA design patterns with Event-B method

Imen Tounsi[1], Mohamed Hadj Kacem[1], Ahmed Hadj Kacem[1], and Khalil Drira[2,3]

[1] ReDCAD-Research unit, University of Sfax, Sfax, Tunisia,
{Imen.Tounsi, mohamed.hadjkacem}@isimsf.rnu.tn,
ahmed.hadjkacem@fsegs.rnu.tn
[2] CNRS, LAAS, 7 avenue du colonel Roche,F-31400 Toulouse, France
[3] Univ de Toulouse, LAAS, F-31400 Toulouse, France
khalil@lass.fr

Abstract. Using design patterns has become increasingly popular. Most design patterns are proposed in an informal way, which can give rise to ambiguity and may lead to incorrect usage. Patterns, proposed by the SOA design pattern community, are described with an appropriate notation. So they require modeling with a standard notation and then formalization. In this paper, we propose a formal architecture-centric approach that aims first to model message-oriented SOA design patterns with the SoaML standard language and second to formally specify these patterns at a high level of abstraction using the Event-B language. These two steps are performed before undertaking the effective coding of a design pattern providing correct by design solutions. Our approach is experimented through an example we present in this paper. We implemented our approach under the Rodin platform which we use to prove model consistency.

1 Introduction

The communication and the integration between heterogeneous applications are great challenges of computing science research works. Several researches have tried to solve them by various methods and technologies (message-oriented middleware, EAI, etc.). They have tried to bring a response to these problems but without leading to real decisive success. The stack of applications led to an unbearable situation. The lack of an efficient architectural solution led information systems to a deadlock with respect to trade requirements.

Service-oriented architectures (SOA) is a technology that offers a model and an opportunity to solve these problems [9]. Nevertheless these architectures are subject to some quality attribute failures (e.g., reliability, availability, and performance problems). *Design patterns*, as proven solutions to specific problems, have been widely used to solve this weakness.

Most design patterns are proposed in an informal way that can raise ambiguity and may lead to incorrect usage. Patterns, proposed by the SOA design

pattern community, are described with an appropriate notation [9]. So they require modeling with a standard notation and then formalization. The intent of our approach is to model and formalize message-oriented SOA design patterns. These two steps are performed before undertaking the effective coding of a design pattern, so that the pattern in question will be correct by construction. Our approach allows developers to reuse correct SOA design patterns, hence we can save effort on proving pattern correctness.

In this paper, we propose a formal architecture-centric approach. The key idea is to model SOA design patterns with the semi-formal Service oriented architecture Modeling Language (SoaML) and to formally specify these design patterns with the formal language Event-B. We illustrate our approach through an example. We proceed by modeling the "Asynchronous Queuing" pattern proposed by the SOA design pattern community using the SoaML language. This modeling step is proposed in order to attribute a standard notation to SOA design patterns. Then we propose a formal specification of this design pattern using the Event-B formal language. We implement these specifications under the Rodin platform which we use to prove model consistency. We provide both structural and behavioral features of SOA design patterns in the modeling phase as well as in the specification phase. Structural features of a design pattern are generally specified by assertions on the existence of types of components in the pattern. The configuration of the elements is also described, in terms of the static relationships between them. Behavioral features are defined by assertions on the temporal orders of the messages exchanged between the components.

The rest of this paper is organized as follows. Section 2 gives background information of some concepts used in this paper. Section 3 discusses related work. Section 4 focuses on the modeling of the "Asynchronous Queuing" pattern using the SoaML language. Section 5 describes how to formally specify an SOA design pattern using the Event-B language. Section 6 concludes and gives some perspectives of our work.

2 Basic Concepts

In this section, we provide some background information on the patterns, the SoaML modeling language [15] and the Event-B formal language [1].

2.1 Pattern

The concept of *patterns* is not new, the first definition was announced by Alexander in 1977 in the field of architecture of buildings and towns. He declares that: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem" [2]. In the field of information systems, a pattern is defined as a model that provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection [9].

Patterns can be classified relatively to their level of abstraction into three categories: *architectural patterns* (or architectural styles) that provide the skeleton or template for the overall shape and the structure of software applications at a high-level design [11], *design patterns* that encode a proven solution to a recurring design common problem of automated systems [8, 16], and *implementation patterns* that provide a solution to a given problem in programming [4]. It is used to generate code.

2.2 Service oriented architecture Modeling Language (SoaML)

Service-Oriented Architecture is an architectural style for building systems based on interacting services. Each service exposes processes and behaviors through contracts, which are composed of messages at discoverable addresses called endpoints [3]. SoaML [15] is a specification developed by the OMG that provides a standard way to architect and model SOA solutions. It consists of a UML profile and a meta-model that extends the UML 2.0 (Unified Modeling Language).

2.3 Event-B

Event-B is an evolution of B-Method developed by Jean-Raymond Abrial [1]. It is a formal modeling method for developing systems via stepwise refinement, based on first-order logic. The method is enhanced by its supporting Rodin Platform for analyzing and reasoning rigorously about Event-B models.

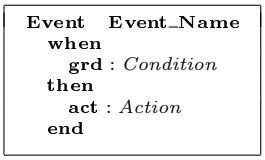
The Event-B modeling notation The basic concept in the Event-B development is the model. A model is made of two types of components: *contexts* and *machines*. A *context* describes the static part of a model, whereas a *machine* describes the dynamic behavior of a model. Machines and contexts can be interrelated: a machine can be *refined* by another one, a context can be *extended* by another one and a machine can *see* one or several contexts.

Each context has a name and other clauses:

- "Extends": The current context can extend other ones by adding their names in this clause. The resulting context consists of the context itself and all constants and axioms of all extended ones.
- "Sets": Declares a new data type. To specify a set S with elements $e1$,...,en, we declare S as a set and $e1$,...,en as constants then we add the axiom $partition(S, e1, ..., en)$
- "Constants": Declares the various constants introduced in the context. Names of constants must be all distinct.
- "Axioms": Denotes the type of the constants and the various predicates which the constants obey. It is a statement that is assumed to be true in the rest of the model and it consists of a label and a predicate.

Like a context, a machine has an identification name and several clauses. In the following, we present in detail the clauses that we will use:

- "Refines": The current machine can optionally refine another one by adding its name in this clause.
- "Sees": Lists the contexts referenced by the machine in order to use sets and constants defined in them.
- "Variables": Lists the variables introduced in this machine. They constitute the state of the machine. Their values are determined by an initialization and can be changed by events.
- "Invariants": Lists the predicates that should be true for every reachable state.
- "Events": Lists the events that change the state of the model and assign new values to variables. Each event is composed of one or several guards *grd* and one or several actions *act*. The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the following form:



A **relation** is used to describe ways in which elements of two distinct sets are related. If A and B are two distinct sets, then $R \in A \leftrightarrow B$ denotes the set of all relations between A and B. The domain of R is the set of elements in A related to something in B: $dom(R)$. The range of R is the set of elements of B to which some element of A is related: $ran(R)$. We also say that A and B are the source and target sets of R, respectively.

Given two elements a and b belonging to A and B respectively, we call **ordered pair** a to b, the pair having the first element a (start element) and the last element b (arrival element). We note: $a \mapsto b$ or (a,b).

An important special case of relations are functions. A **partial functions** is a relation where each element of the domain is uniquely related to one element of the range. If A and B are two distinct sets, then $A \nrightarrow B$ denotes the set of all partial functions between A and B.

3 Related work

This section surveys related researches to patterns used in the field of software architecture. As it is represented in Figure 1, these researches can be classified into three axes according to their abstraction level. The first axis deals with architectural patterns, the second axis deals with implementation patterns and the third axis deals with design patterns. Compared to architectural patterns, design patterns address smaller reusable designs such as the structure of subsystems within a system [10]. In this paper, we focus only on the third axis since it belongs to our research activities.

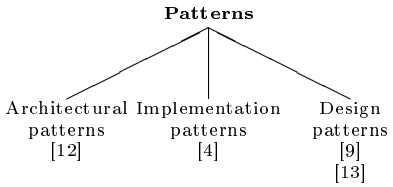


Fig. 1. Classification of Patterns

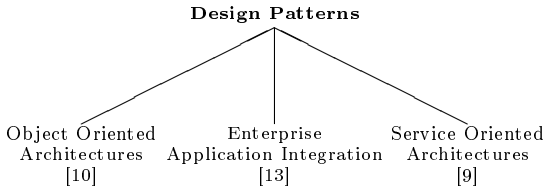


Fig. 2. Classification of Design Patterns

Researches connected to design patterns axis, are mainly classified into three branches of work according to their architectural style (Figure 2).

Among researches related to design patterns for Object Oriented Architectures, we present the work of Gamma et al.. They have proposed a set of design patterns in the field of object-oriented software design [10]. These patterns are described with graphical notations by using three diagrams based principally on the OMT (Object Modeling Technique) notation. These diagrams are the *class diagram*, the *object diagram* and the *interaction diagram*. For each design pattern, they include at least one class diagram and the other notations are used as needed to supplement the discussion.

Several researches have proposed the formalization of object-oriented design patterns. Since the most famous one are those proposed by Gamma [10] (hereafter referred to as GoF), most researches refer to these patterns. Several approaches have been proposed in the literature, we quote:

Zhu et al. [19] specify design patterns and pattern composition formally. They specify 23 GoF patterns. Zhu et al. use the first order logic induced from the abstract syntax of UML defined in GEBNF to define both structural and behavioral features of design patterns.

Taibi et al. [17, 18] develop a language called Balanced Pattern Specification Language (BPSL) to formally specify patterns, pattern composition and instances of patterns. This language is used as a formal basis to specify structural features of design patterns in the First-Order Logic (FOL) and behavioral features in the Temporal Logic of Action (TLA). Taibi et al. use as a case study the Observer-Mediator pattern composition proposed by GoF.

Dong et al. [7, 6] focus on the structural and behavioral fetures of a design pattern component. They use the First-Order Logic theories to specify the struc-

tural features of patterns by means of Object-Z and Temporal Logic of Action (TLA) to specify their behavioral features. As examples, they use GoF patterns.

Kim et al. [14] present an approach to describe design patterns based on role concepts. First, they develop an initial role meta-model using an existing modeling framework, Eclipse Modeling Framework (EMF), then they transform the meta-model to Object-Z using model transformation techniques in order to specify structural features. Behavioral features of patterns are also specified using Object-Z and integrated in the pattern role models. Kim et al. also use GoF patterns as examples to represent their approach.

Blazy et al. [5] propose an approach for specifying design patterns and how to reuse them formally. They use the B method to specify structural features of design patterns but they do not consider the specification of their behavioral features.

Among researches related to design patterns for Enterprise Application Integration, we present the work of Hohpe and Woolf. They have proposed a set of design patterns which are dealing with enterprise integration using messaging [13]. These design patterns are represented with a visual notation using their appropriate notation. Hohpe and Woolf argue their choice by saying that there is no a comprehensive notation that is geared toward the description of all aspects of an integration solution. The Unified Modeling Language (UML) does not contain semantics to describe messaging solutions and the UML Profile for EAI enriches the semantics of collaboration diagrams to describe message flows between components but it does not capture all the patterns described in their pattern language.

To our knowledge, there is no research work that propose the formalization of enterprise integration design patterns and as examples they refer to Hohpe and Woolf patterns and to enterprise integration patterns in general.

In the branch of SOA design patterns, we find out the work of Erl. Erl have proposed a set of design patterns for service-oriented architecture and service-orientation [9]. Each pattern is modeled with an appropriate notation represented in a symbol legend. These patterns are modeled without any formal specification. In order to understand these patterns, the first step is to form a knowledge on the pattern-related terminology and notation. In addition to the pattern notation, Erl proposes a set of specific pattern symbols used to represent a design pattern, a compound design pattern and a group of related design patterns. Erl argues his choice by saying that there is a lack of abstract definitions, architectural models, and vocabularies for SOA but there are several efforts underway by different standards and research organizations.

In our research work we are interested in *SOA design patterns* defined by Erl [9]. Erl presents SOA design patterns with an appropriate notation because there is no a standard modeling notation for SOA, but now OMG announces the publication of a Service oriented architecture Modeling Language (SoaML), it is a specification for the UML Profile and a Meta-model for Services (UPMS). So, in our work, we propose to model SOA design patterns with the SoaML standard language and we focus on the structural and behavioral features of SOA design

patterns. After the modeling step, we propose to specify SOA design patterns formally. We use the Event-B language, which is an extension to the B method, to define both structural and behavioral features of design patterns.

In this paper we concentrate on a specific category of patterns called “Service messaging patterns", it is a collection of patterns witch are message oriented. It is focused on inter-service message exchange, and provides design solutions for a wide range of messaging concerns. From this collection, we use the “Asynchronous Queuing" pattern.

In conclusion, most proposed patterns are described using a combination of textual description and a graphical appropriate notations in order to make them easy to read and understand. However, using these descriptions makes patterns ambiguous and may lack details. There have been many researches that define pattern specifications using formal techniques but researches that model design patterns with semi-formal languages are few. We find a number of approaches that formally specify different sorts of fetures of patterns: structural, behavioral, or both. Table 1 is a recapitulation of related works that contains a comparison between the above-mentioned approaches and our approach.

Approach	**Object Oriented Design Patterns**						**EAI Design Patterns**	**SOA Design Patterns**	
	Gamma et al. 1995	**Zhu et al. 2010**	**Taibi et al. 2006**	**Dong et al. 2007**	**Kim et al. 2009**	**Blazy et al. 2006**	**Hope et al. 2003**	**Erl 2009**	**Ours 2012**
Pattern modeling	OMT	GoF (OMT)	GoF (OMT)	GoF (OMT)	GoF (OMT)	GoF (OMT)	Appropriate Notation	Appropriate Notation	SoaML
Structural formal specification	–	GEBNF (FOL)	BPSL (FOL)	Object Z (FOL)	Object Z (FOL)	B Method	–	–	Event-B
Behavioral formal specification	–	GEBNF (FOL)	BPSL (TLA)	TLA	Object Z (FOL)	–	–	–	Event-B

Table 1. Summary table of related works

4 Modeling SOA design patterns with the SoaML language

We provide a modeling solution for describing SOA design patterns using a visual notation based on the graphical SoaML notation. Two main reasons lead to use SoaML for modeling these patterns. First, SoaML is a standard modeling language defined by OMG to describe service oriented architectures. Second, diagrams used in the SoaML language, allow to represent structural features as well as behavioral features of SOA design patterns.

To model an SOA architecture, we can represent many levels of description. The highest level is described as a *Services Architectures* where participants are working together using services. Services Architectures is modeled using UML collaborations diagram stereotyped « ServicesArchitecture ». The next level is described as *Participants* using UML class diagram stereotyped « Participant ». The *Service Contract* is at the middle of the SoaML set of SOA architecture constructs, it describes the services mentioned above and it is modeled using UML collaborations stereotyped « ServiceContract ». In the next level, we find the specification of *Interfaces* and *Message Types* using respectively UML class diagram stereotyped « ServiceInterface » and UML class diagram stereotyped « MessageType ». For both the service contract and the interface levels we can specify behavioral features of services using any UML behavior (e.g sequence or activity diagrams).

In this paper, we model the Asynchronous Queuing pattern proposed by Erl [9]. This pattern is described in detail within the next section.

4.1 Asynchronous Queuing

Asynchronous Queuing pattern is an SOA design pattern for inter-service message exchange [9]. It belongs to the category "Service Messaging Patterns". It establishes an intermediate queuing mechanism that enables asynchronous message exchanges and increases the reliability of message transmissions when service availability is uncertain.

The problem addressed by *Asynchronous Queuing* pattern is that when services interact synchronously, it can inhibit performance and compromise reliability when one of services cannot guarantee its availability to receive the message.

Synchronous message exchanges can impose processing overhead, because the service consumer needs to wait until it receives a response from its original request before proceeding to its next action. Responses can introduce latency by temporally locking both consumer and service.

The proposed solution by this pattern is to introduce an intermediate queuing technology into the architecture (Figure 3). The queue receives request messages sent by the *ServiceA* and then forwards them on behalf of the *ServiceB*. If the target service is unavailable, the queue acts as temporary storage and retains the message. It then periodically attempts retransmission. Similarly, if there is a response, it can be issued through the same queue that will forward it back to the *ServiceA* when it is available. While either *ServiceA* or *ServiceB* is processing message contents, the other can deactivate itself in order to minimize memory consumption.

4.2 Structural features of the Asynchronous Queueing pattern

In the structural modeling phase, we specify components of the pattern and their dependencies or connections in the « Participant » diagram (Figure 4) and we specify their interfaces and exchanged messages in the « ServiceInterface » and « MessageType » diagrams respectively (Figure 5).

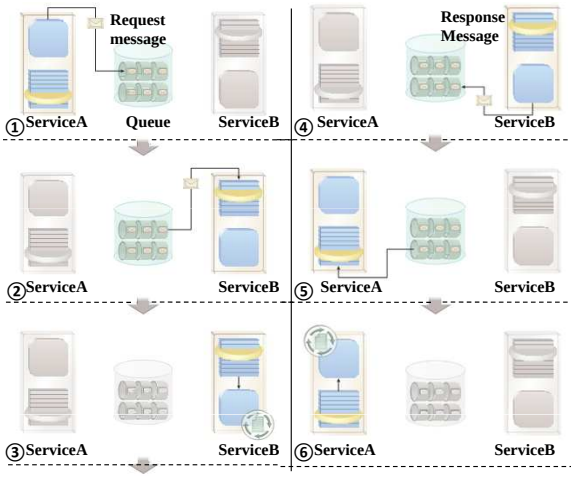


Fig. 3. Asynchronous Queuing solution

ServiceA, *ServiceB* and the *Queue* are defined as participants because they provide and use services. As shown in Figure 4, the *ServiceB* provides a ServiceX service used by the *ServiceA* and the *Queue* provides a storage service. We didn't represent the storage service provided by the *Queue* in order to concentrate principally on the communication between *ServiceA* and *ServiceB* and to not complicate the presented diagrams.

Participants provide capabilities through service ports typed by UML interfaces that define their provided capabilities. Both *ServiceA* and *ServiceB* have a port typed with the "ServiceX". The *ServiceB* is the provider of the ServiceX and has a « Service » port. The *ServiceA* is a consumer of the ServiceX and uses a « Request » port. We note that the *ServiceB*'s port provides the "`ProviderServiceX`" interface and requires the "`OrderServiceX`" interface.

Since the *ServiceA* uses a « Request » the *conjugate* interfaces are used, so the *ServiceA*'s port provides the "`OrderServiceX`" interfaces and uses the "`ProviderServiceX`". Since they are conjugate, ports on *ServiceA* and *ServiceB* can be connected to enact the service. The « Request » port is preceded with a tilde (∼) to show that the conjugate type is being used. In this diagram, « ServiceChannels » are explicitly represented, they enables communication between the different participants.

Figure 5 shows a couple of « MessageType » that are used to define the information exchanged between ServiceA (consumer) and ServiceB (provider). These « MessageType » are "`RequestMessage`" and "`ResponseMessage`", they will be used as types for operation parameters of the service interfaces.

Figure 5 depicts a *ServiceB* participant providing a "serviceX" service. The type of the service port is the UML interface "`ProviderServiceX`" that has the operation "`processServiceXProvider`". This operation has a message style pa-

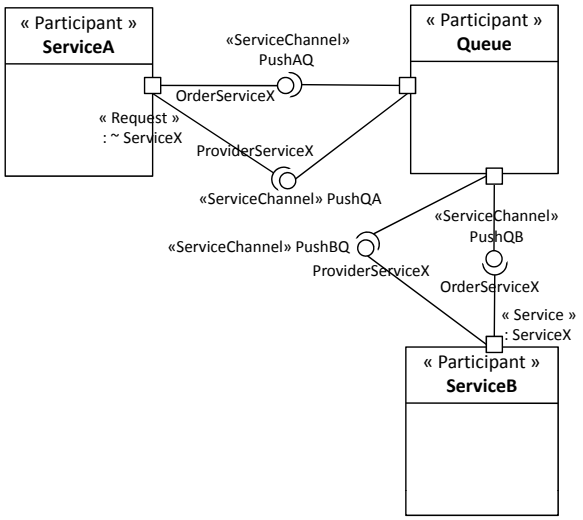


Fig. 4. « Participant » diagram

rameter where the type of the parameter is the MessageType "`ResponseMessage`". The *ServiceA* participant expresses its request for the "serviceX" service using its request port. The type of this service port is the UML interface "`OrderServiceX`". This interface has an operation "`ProcessServiceXOrder`" and the type of parameter of this operation is the MessageType "`RequestMessage`".

4.3 Behavioral features of the Asynchronous Queueing pattern

During a course of exchanging messages, the first service (*ServiceA*) sends a request message to the second service (*ServiceB*), at that time, its resources are locked and consumes memory. This message is intercepted and stored by an intermediary queue. *ServiceB* receives the message forwarded by the *Queue* and *ServiceA* releases its resources and memory. While *ServiceB* is processing the message, *ServiceA* consumes no resources. After completing its processing, *ServiceB* issues a response message back to *ServiceA* (this response is also received and stored by the intermediary *Queue*). *ServiceA* receives the response and completes the processing of the response while *ServiceB* is deactivated.

To specify behavioral features of design patterns we use the UML2.0 sequence diagram. As depicted in Figure 6, this diagram specifies the valid interactions between participants.

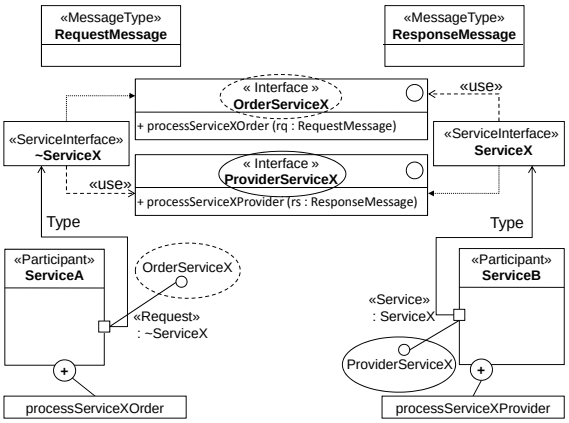


Fig. 5. « ServiceInterface » and « MessageType » diagrams

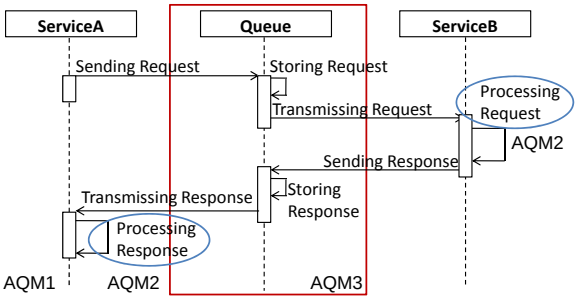


Fig. 6. Sequence diagram

5 Formal semantics of SOA Design Patterns

In this section, we describe semantics of design patterns with the *Event-B* notation. In order to prove the correctness of the pattern specification we use the Rodin Platform.

As we have mentioned in section 2.3, *contexts* are used to model static properties of a model, so we specify structural features of design patterns with a context (AQC). Whereas, with machines we model the dynamic properties, so we specify behavioral features of design patterns with machines. Our model is composed of three machines named respectively $AQM1$, $AQM2$ and $AQM3$ (AQM denotes Asynchronous Queuing Machine). In the first machine ($AQM1$), we specify the pattern at a hight level of abstraction, i.e. we suppose that the communication happens only between *ServiceA* and *ServiceB*. In the second machine ($AQM2$), we add the process function to the model. Finally, in the third machine($AQM3$),

we add the *Queue* and all its behavior to the model. We use the refinement techniques to gradually introduce details and complexity into our model. Machines and context relationships are illustrated in Figure 7.

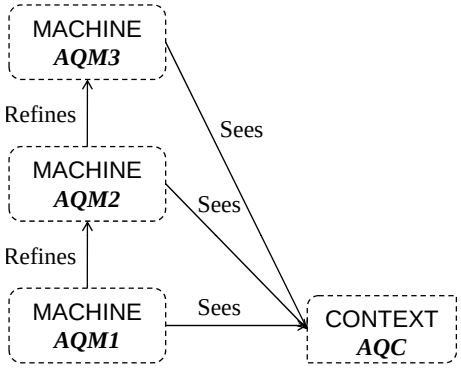


Fig. 7. Context and machines relationship

5.1 Structural features of SOA Design Patterns

In the Asynchronous Queueing pattern, we have three Participants: *ServiceA*, *ServiceB* and the *Queue*. Using Event-B, we specify in the context *AQC* the three participants as constants (one constant for each participant). These constants or participants are part of a set *Participant*. We model this by creating a partition (section 2.3) in the AXIOMS section:

CONSTANTS
ServiceA
ServiceB
Queue

SETS
Participant

AXIOMS
Participant_partition : $partition(Participant, \{Queue\}, \{ServiceA\}, \{ServiceB\})$

We create four more constants to specify relations between participants, modeled as « ServiceChannel » in the SoaML modeling. These constants are specified as relations in the AXIOMS clause and they are named *PushAQ*, *PushQB*, *PushBQ* and *PushQA*.

AXIOMS
PushAQ_Relation : $PushAQ \in Partcipant \leftrightarrow Partcipant$
PushQB_Relation : $PushQB \in Partcipant \leftrightarrow Partcipant$
PushBQ_Relation : $PushBQ \in Partcipant \leftrightarrow Partcipant$
PushQA_Relation : $PushQA \in Partcipant \leftrightarrow Partcipant$

For each relation, we add two axioms in order to define the domain and the rang. For example, for the *PushAQ* relation we add the following two axioms to denote that the source of the relation *PushAQ* is *ServiceA* and its target is the *Queue*:

PushAQ_Domain : $dom(PushAQ) = \{ServiceA\}$
PushAQ_Range : $ran(PushAQ) = \{Queue\}$

In this context AQC, we didn't specify ports and interfaces because they are fine details that we will not use them in machines. Whereas, we specify messages to know what message is being exchanged. So, we define another SET named *MesaageType*, two constants *RequestMessage* and *ResponseMessage* and then the Message_Partition.

SETS
MessageType
CONSTANTS
RequestMessage
ResponseMessage
AXIOMS
Message_Partition : $partition(MessageType, \{RequestMessage\}, \{ResponseMessage\})$

This part of specification belongs to the « Participant » diagram and « MessageType » diagram represented respectively in Figure 4 and Figure 5.

5.2 Behavioral features of SOA Design Patterns

In order to specify behavioral features of the pattern, we have three steps: in the first step we specify the pattern with a machine at a hight level of abstraction. In the second step, we add more details to the first machine by using the refinement technique. In the third and the final step we add all necessary details to the second machine by using the refinement technique too.

Specifying the pattern at a high level of abstraction As already mentioned above, in the Asynchronous Queueing pattern there are three parties participating in it namely the *ServiceA*, the *ServiceB* and the *Queue*. In this first machine $AQM1$, we only specify the communication between *ServiceA* and *ServiceB*, i.e. the queue is completely transparent, meaning that neither *ServiceA* nor *ServiceB* may know that a queue was involved in the data exchange. So, the behavior is described as follows:

- The *ServiceA* sends a *RequestMessage* to the *ServiceB* and then remains released from resources and memory (becomes unavailable).

– When the *ServiceB* becomes available, it receives the *RequestMessage* and sends the *ResponseMessage*.
– When the *ServiceA* becomes available, it receives the *ResponseMessage* and then returns deactivated.

Formally, we can use two variables to represent the state of the pattern: *Dispo* to denote the state of the participant either available or not, and *Send* to indicate who sends what message. The first invariant *Dispo_Function* specifies the availability feature of participants. This feature is specified with a partial function which is a special kind of relation (each domain element has at most one range element associated with it) i.e. the function *Dispo* relates *Participants* to a *Boolean* value indicating that it is either available or not. We use the partial function because a participant can't be available and not available at the same time. The second invariant, i.e. *Send_Relation*, specifies what is the message sent and who is the sender.

INVARIANTS
Dispo_Function : $Dispo \in Partcipant \nrightarrow BOOL$
Send_Relation : $Send \in Partcipant \leftrightarrow MessageType$

Initially, *ServiceA* is available and *ServiceB* is not available and there are no messages sent, hence *Send* relation is initialized to the empty set.

INITIALISATION
begin
act1 : $Dispo := \{ServiceA \mapsto TRUE, ServiceB \mapsto FALSE\}$
act2 : $Send := \varnothing$
end

The dynamic system can be seen in Figure 6. Sending the request starts when there is no messages sent and the *ServiceA* is available, then *ServiceA* sends the *RequestMessage* and becomes unavailable. Sending the response starts after the questioning phase (when the request message is sent) and when the *ServiceB* is available and then *ServiceB* sends the *ResponseMessage* and becomes also unavailable. This scenario is formalized by the following two events, namely **Sending_Req** (Sending Request) and **Sending_Resp** (Sending Response).

Event Sending_Req
when
grd1 : $Send = \varnothing$
grd2 : $ServiceA \in dom(Dispo) \wedge Dispo(ServiceA) = TRUE$
then
act1 : $Send := Send \cup \{ServiceA \mapsto RequestMessage\}$
act2 : $Dispo(ServiceA) := FALSE$
end

Event Sending_Resp
when
grd1 : $RequestMessage \in ran(Send)$
grd2 : $ServiceB \in dom(Dispo) \wedge Dispo(ServiceB) = TRUE$
then
act1 : $Send := Send \cup \{ServiceB \mapsto ResponseMessage\}$
act2 : $Dispo(ServiceB) := FALSE$
end

After sending the response message and when the *ServieA* is available, receiving the response message becomes possible. After receiving the response, *ServiceA* becomes not available again. So, we assign the value FALSE to the *ServiceA* disponibility. Formally, this action is specified with the event **Receiving_Resp** (Receiving Response).

Event Receiving_Resp
  when
    grd1 : $ResponseMessage \in ran(Send)$
    grd2 : $ServiceA \in dom(Dispo) \land Dispo(ServiceA) = TRUE$
  then
    act1 : $Dispo(ServiceA) := FALSE$
  end

Each time when a service is unavailable and an event can't be triggered only if this service becomes available, we use a special event named **Activating_Participant**. This event is with a parameter of type *Participant* (represented in the clause *any*) and it has the functionality of modifying the availability of a participant. For this event, we use the function overriding operator ($\leftarrow$), this operator replaces existing mappings with new ones in the *Dispo* function, here we replace the availability of a service from FALSE to TRUE.

Event Activating_Participant
  any
    P $Participant$
  where
    grd1 : $P \in Partcipant$
    grd2 : $P \in dom(Dispo) \land Dispo(P) = FALSE$
  then
    act1 : $Dispo := Dispo \leftarrow \{P \mapsto TRUE\}$
  end

First refinement: Adding the message processing The second machine $AQM2$ (concrete machine) refines the cited above $AQM1$ machine (abstract machine) and uses the AQC context. In this machine we introduce the notion of processing messages. So we add a new variable named *Process*. This variable is specified with a partial function that relates a *Participant* to a *MessageType* indicating who participant is processing what message. Initially, the *Process* function is initialized to the empty set.

INVARIANTS
  Process_Function : $Process \in Partcipant \nrightarrow MessageType$

The $AQM2$ machine events are now defined below. We keep the **Sending_Req** event as it is, we add a new event **Processing_Req** (refining skip), we add more details to the abstract event **Sending_Resp** and the abstract event **Receiving_Resp** is refined by the concrete event **Processing_Resp**. This is illustrated in Figure 8.

Processing_Req (Processing Request) event is triggered when the message is sent (grd1), not yet processed (grd2) and the *ServiceB* is available (grd3). In the action part, we add to the process function the pair $ServiceB \mapsto$

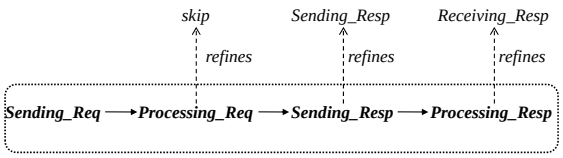


Fig. 8. Refinement of $AQM1$

RequestMessage to denote that the *ServiceB* is processing the *RequestMessage* (act1).

Event Processing_Req
when
grd1 : $RequestMessage \in ran(Send)$
grd2 : $RequestMessage \notin ran(Process)$
grd3 : $ServiceB \in dom(Dispo) \land Dispo(ServiceB) = TRUE$
then
act1 : $Process := Process \lhd\!\!- \{ServiceB \mapsto RequestMessage\}$
end

Now the event **Sending_Resp**, is triggered after processing the *RequestMessage* and when the *ResponseMessage* is not yet send. So, we refine this event by adding two new guards ($grd3$ and $grd4$).

grd3 : $RequestMessage \in ran(Process)$
grd4 : $ResponseMessage \notin ran(Send)$

For the event **Processing_Resp**, it refines the event **Receiving_Resp** by adding the action of processing the message.

act2 : $Process := Process \lhd\!\!- \{ServiceA \mapsto ResponseMessage\}$

Second refinement: Adding the storage service In the third machine ($AQM3$), we introduce the behavior of the *Queue*, so as to complete all the behavior of the Asynchronous Queuing pattern. We add two new variables named respectively *Store* and *Transmit*. *Store* is specified with a relation that relates a *Participant* to a *MessageType* and we add an invariant that restrict the domain of this relation to only the *Queue* indicating that the queue is storing what message. *Transmit* is specified with a partial function that relates a *Participant* to a *MessageType* and we add an invariant that restrict the domain of this function to only the *Queue* indicating that the *Queue* is transmitting what message. Initially the *Store* relation and *Transmit* function are both initialized to the empty set.

INVARIANTS
Store_Relation : $Store \in Partcipant \leftrightarrow MessageType$
Store_Dom_Rest : $dom(Store) = \{Queue\} \lor Store = \varnothing$
Transmit_Function : $Transmit \in Partcipant \nrightarrow MessageType$
Transmit_Dom_Rest : $dom(Transmit) = \{Queue\} \lor Transmit = \varnothing$

The $AQM3$ machine events are now defined below. We keep the **Sending_Req** and the **Sending_Resp** events as they are. We add four new events namely **Storing_Req**, **Transmissing_Req**, **Storing_Resp** and **Transmissing_Resp**. These events are related to the *Queue* behavior. We add more details to the abstract events **Processing_Req** and **Processing_Resp**. This is illustrated in Figure 9.

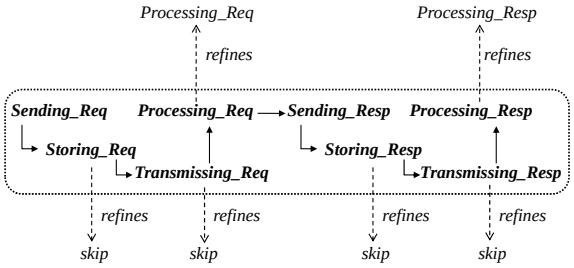


Fig. 9. Refinement of $AQM2$

Due to space restrictions, we didn't represent the four new events in this paper. We present only **Storing_Req** and **Transmissing_Req** events, the other two events, **Storing_Resp** and **Transmissing_Resp**, are similar to them. The event **Storing_Req** is triggered when the *RequestMessage* is sent and not yet processed and the *ServiceB* is available. When the message is stored, the **Transmissing_Req** event can be triggered.

Event Storing_Req
when
grd1 : $RequestMessage \in ran(Send)$
grd2 : $RequestMessage \notin ran(Process)$
grd3 : $ServiceB \in dom(Dispo) \wedge Dispo(ServiceB) = FALSE$
grd4 : $Stores = \varnothing$
then
act1 : $Stores := Stores \cup \{Queue \mapsto RequestMessage\}$
end

Event Transmissing_Req
when
grd1 : $RequestMessage \in ran(Stores)$
then
act1 : $Transmit := Transmit \leftarrow \{Queue \mapsto RequestMessage\}$
end

If a participant (*ServiceA* or *ServiceB*) receives a message, the storage of this message in the queue becomes unnecessary, so, the only modification in the processing event is to empty the *Queue*.

5.3 Proof obligations

The proof obligations define what is to be proved to show the consistency of an Event-B model. They are automatically generated by the Rodin Platform. In this section, we give an overview about proof obligations belonging to our whole specification. Each proof obligation is identified by its label. The proof statistics belonging to our specification is given in Table 2.

- Well-definedness of an axiom (*axiomLabel/WD*): This proof obligation rule ensures that an axiom is Well-defined. In our model we have 14 well-definedness axiom proof obligations.
- Well-definedness of a guard (*guardLabel/WD*): This proof obligation rule ensures that a guard is Well-defined. Some expressions, especially function applications, may not be defined everywhere. For example, $Dispo(ServiceB)$ is only defined if *ServiceB* is in the domain of *Dispo*, i.e. $ServiceB \in dom(Dispo)$. In our model we have 7 well-definedness guard proof obligations.
- Invariant preservation proof obligation rule (*invariantLabel/INV*): This proof obligation rule ensures that each invariant in a machine is preserved whenever variable values change by each event. In our model we have 19 invariant preservation proof obligations.

Proof obligations	**Number**
Generated in the context	
-Well-definedness of an axiom	14
Generated for machine consistency	
-Well-definedness of a guard	7
-Invariant preservation	19

Table 2. Proof statistics

These proof obligation rules ensure that the specified SOA design pattern is correct by construction. Our approach allows developers to reuse correct SOA design patterns, hence we can save effort on proving pattern correctness.

6 Conclusions

In this paper, we presented a formal architecture-centric approach supporting the modeling and the formalization of message-oriented SOA design patterns. The modeling phase allows us to represent SOA design patterns with a graphical standard notation using the SoaML language proposed by the OMG. The formalization phase allows us to formally characterize both structural and behavioral features of these patterns at a high level of abstraction, so that they will be correct by construction. We implemented these specifications under the Rodin platform. We outlined many categories of SOA design patterns. We illustrated

our approach through a pattern example ("Asynchronous Queuing pattern") under the "Service messaging patterns" category. Currently, we are working on generalizing our approach in order to examine the other categories and formally specifying pattern compositions. Currently, the passage from the SoaML modeling to the formal specification is done manually; in the future, we will work on automating this phase by using transformation rules.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, August 1977.
3. Eric Bruno Arnon Rotem-Gal-Oz and Udi Dahan. *SOA Patterns.* MEAP (Manning Early Access Program), June 2007.
4. Kent Beck. *Implementation Patterns.* Addison Wesley; 1 edition (23 Oct 2007), 2007.
5. Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of specification patterns with the b method. In Didier Bert, Jonathan Bowen, Steve King, and Marina Waldén, editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 626–626. Springer Berlin / Heidelberg, 2003.
6. Jing Dong, Paulo S. C. Alencar, and Donald D. Cowan. A behavioral analysis and verification approach to pattern-based design composition. *Software and System Modeling*, 3(4):262–272, 2004.
7. Jing Dong, Paulo S. C. Alencar, Donald D. Cowan, and Sheng Yang. Composing pattern-based components and verifying correctness. *J. Syst. Softw.*, 80:1755–1769, November 2007.
8. Ghizlane El-Boussaidi and Hafedh Mili. A model-driven framework for representing and applying design patterns. In *COMPSAC (1)*, pages 97–100, 2007.
9. Thomas (with additional contributors) Erl. *SOA Design Patterns (The Prentice Hall Service-Oriented Computing Series from Thomas Erl).* Prentice Hall PTR, 1 edition, January 2009.
10. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
11. Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures (The Addison-Wesley Object Technology Series).* Addison-Wesley Professional, July 2004.
12. Hassan Gomaa, Koji Hashimoto, Minseong Kim, Sam Malek, and Daniel A. Menascé. Software adaptation patterns for service-oriented architectures. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 462–469, New York, NY, USA, 2010. ACM.
13. Bobby Woolf Gregor Hohpe. *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions.* Addison Wesley, October 2003.
14. Soon-Kyeong Kim and David A. Carrington. A formalism to describe design patterns based on role concepts. *Formal Asp. Comput.*, 21(5):397–420, 2009.

15. O. M. G. Object Management Group. Service oriented architecture Modeling Language (SoaML)- Specification UML Profile and metamodel for services (UPMS). August 2008.
16. Andres J. Ramirez and Betty H.C. Cheng. Developing and applying design patterns for dynamically adaptive systems. Technical Report MSU-CSE-09-8, Department of Computer Science, Michigan State University, East Lansing, Michigan, March 2009.
17. T Taibi. Formalising design patterns composition. *Software IEE Proceedings*, 153(3):127–136, 2006.
18. Toufik Taibi and David Chek Ling Ngo. Formal specification of design pattern combination using bpsl. *Information and Software Technology*, 45(3):157 – 170, 2003.
19. Hong Zhu and Ian Bayley. Laws of pattern composition. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 630–645, Berlin, Heidelberg, 2010. Springer-Verlag.