An Approach for Modeling and Formalizing SOA Design Patterns

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

▶ To cite this version:

Imen Tounsi, Mohamed Hadj Kacem, Ahmed Hadj Kacem, Khalil Drira. An Approach for Modeling and Formalizing SOA Design Patterns. IEEE International Conference on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Jun 2013, Hammamet, Tunisia. 11p. hal-00801395

HAL Id: hal-00801395
https://hal.science/hal-00801395

Submitted on 15 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Approach for Modeling and Formalizing SOA Design Patterns

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,
[2] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[3] Univ de Toulouse, LAAS, F-31400 Toulouse, France,
{imen.tounsi,mohamed.hadjkacem}@redcad.org, ahmed.hadjkacem@fsegs.rnu.tn, khalil@lass.fr

Abstract. Although design patterns has become increasingly popular, most of them are presented in an informal way, which can give rise to ambiguity and may lead to their incorrect usage. Patterns proposed by the SOA design pattern community are described with informal visual notations. Modeling SOA design patterns with a standard formal notation contributes to avoid misunderstanding by software architects and helps endowing design methods with refinement approaches for mastering system architectures complexity. In this paper, we present 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 method. These two steps are performed before undertaking the effective coding of a design pattern providing correct by construction pattern-based software architectures. 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.

Keywords: SOA Design Patterns: SoaML modeling: Formal methods: Event-B method

1 Introduction

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

Service-oriented architectures (SOA) is a technology that offers a model and an opportunity to solve these problems [Erl, 2009]. Nevertheless these architectures are subject to some quality attribute failures (e.g., reliability, availability, and performance problems). *Design patterns*, as tested solutions to common design problems within a context, have been widely used to solve this weakness.

Most design patterns are presented in an informal way that can raise ambiguity and may lead to their incorrect usage. Patterns, proposed by the SOA design pattern community, are described with informal visual notations [Erl, 2009]. Modeling SOA design patterns with a standard formal notation contributes to avoid misunderstanding by software architects and helps endowing design methods with refinement approaches for mastering system architectures complexity. 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 to reuse correct SOA design patterns, hence we can save effort on proving pattern correctness.

In this paper, we present a formal architecture-centric design approach. The key idea is to model SOA design patterns with the semi-formal Service oriented architecture Modeling Language (SoaML) and to formally specify them with the Event-B method. We illustrate our approach through a pattern example. We proceed by modeling the *Asynchronous Queuing* pattern, proposed by the SOA design pattern community, with the SoaML language. This modeling step is proposed in order to attribute a standard notation to SOA design patterns. Then, we propose a generic formalization of these patterns using the Event-B method. Next, we illustrate the formalization step with the same pattern example used in the modeling step. We implement the 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 step as well as in the formalization step. Structural features of a design pattern are generally specified by assertions on the existence of entities types in the pattern. The configuration of the entities 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 entities [Zhu and Bayley, 2010].

This paper is organized as follows. Section 2 gives background information of some used concepts. Section 3 focuses on modeling SOA design patterns with the SoaML language. Section 4 describes how to formally specify SOA design patterns with the Event-B method. Section 5 discusses related work. Section 6 concludes and gives future work directions.

2 Event-B method

Event-B [Abrial, 2010] is a formal method for developing systems via stepwise refinement, based on first-order logic. The method is enhanced by its supporting Rodin Platform [Abrial et al., 2010] for analyzing and reasoning rigorously about Event-B models. The basic concept in the Event-B development is the model which 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 inter-related: 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 like "Extends", "Constants", "Sets" to declare a new data type and "Axioms" that denotes the type of the constants and the various predicates which the constants obey. It is a predicate that is assumed to be true in the rest of the model. Like a context, a machine has an identification name, variables that constitute the state of the machine (their values are determined by an initialization and can be changed by events), invariants and events.

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 a relation 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 denote that by $a \mapsto b$ or (a,b).

A **partial function** is a relation where each element of the domain is uniquely related to one element of the range. If A and B are two sets, then $A \nrightarrow B$ denotes the set of partial functions from A to B.

Partitions are used in two different manners. The first one is *partition*(S, A,B). It means that A and B partition the set S, i.e. $S=A \cup B \land A \cap B = \varnothing$. The second one is *partition*(S, $\{A\}$,$\{B\}$,$\{C\}$) which is a specialized use for enumerated sets. It means that $S=\{A,B,C\} \land A \neq B \land B \neq C \land C \neq A$.

3 Modeling SOA design patterns

We provide a modeling solution for describing SOA design patterns using a visual notation based on the graphical SoaML language [OMG, 2012]. Three main reasons lead to use SoaML. First, it is a standard modeling language defined by OMG. Second, it is used to describe SOA. Third, diagrams used in SoaML, allow to represent structural features as well as behavioral features of design patterns.

In this paper, we model as example the Asynchronous Queuing pattern proposed by Erl [Erl, 2009]. This pattern example is also used in the formalizing step as a case study. *Asynchronous Queuing* pattern [4] is an SOA design pattern for inter-service message exchange [Erl, 2009]. 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 this 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

[4] http://soapatterns.org/design_patterns/asynchronous_queuing

both consumer and service. The proposed solution by this pattern is to introduce an intermediate queuing technology into the architecture. The behavior of this pattern is described in detail in section 3.2.

3.1 Structural features

In the structural modeling step, we specify entities of the pattern and their dependencies (connections) in the Participant diagram (Figure 1) and we specify their interfaces and exchanged messages in the ServiceInterface and MessageType diagrams respectively (Figure 2).

ServiceA, *ServiceB* and the *Queue* are defined as participants because they provide and use services. As shown in Figure 1, *ServiceB* provides a ServiceX used by *ServiceA* and the *Queue* provides a storage service. We did not 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. Both *ServiceA* and *ServiceB* have a port typed with "ServiceX". *ServiceB* is the provider of the service and has a Service port. *ServiceA* is a consumer of the service and uses a Request port. We note that *ServiceB*'s port provides the "`ProviderServiceX`" interface and requires the "`OrderServiceX`" interface. Since *ServiceA* uses a Request port preceded with a tilde (∼), the conjugate interfaces are used. So, *ServiceA*'s port provides the "`OrderServiceX`" interface and uses the "`ProviderServiceX`" interface. In this diagram, ServiceChannels are explicitly represented, they enables communication between the different participants.

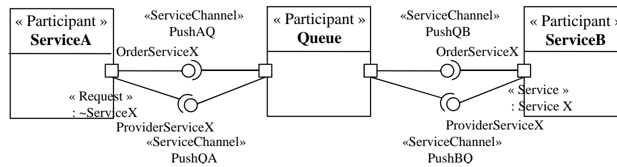


Fig. 1. Participant diagram

Figure 2 shows a couple of MessageType that are used to define the information exchanged between *ServiceA* and *ServiceB*. These messages are "`RequestMessage`" and "`ResponseMessage`", they are used as types for operation parameters of the service interfaces. The type of the *ServiceB*'s port is the UML interface "`ProviderServiceX`" that has the operation "`processServiceXProvider`". This operation has a message style parameter where the type of the parameter is the MessageType "`ResponseMessage`". *ServiceA* expresses its request for the "ServiceX" using its request port. The type of this request port is the UML interface "`OrderServiceX`". This interface has an operation "`ProcessServiceXOrder`" and the type of parameter of this operation is the MessageType "`RequestMessage`".

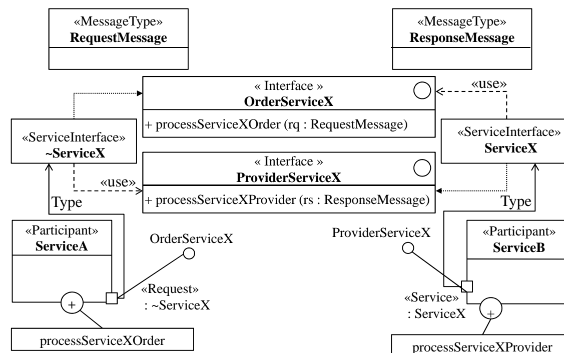


Fig. 2. ServiceInterface and MessageType diagrams

3.2 Behavioral features

We use UML2.0 sequence diagram (Figure 3) to specify behavioral features. During a course of exchanging messages, the first service (*ServiceA*) sends a request message to the second one (*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.

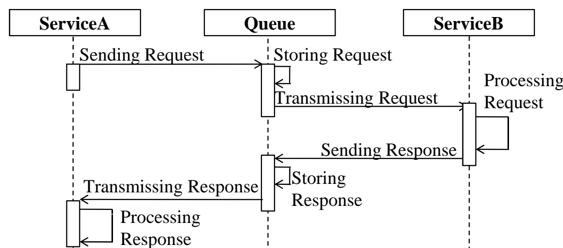


Fig. 3. Sequence diagram

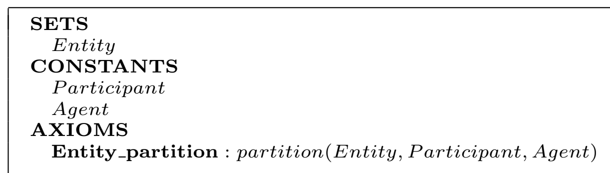
4 Formalizing SOA Design Patterns

In this section, we present an overview of the generic formalization of SOA design patterns with the *Event-B* method [Abrial, 2010]. We use the Rodin Platform [Abrial et al., 2010] in order to prove the correctness of the pattern specification.

A pattern P is described with structural features and behavioral features. Structural features are specified with one or several contexts PCi and behavioral features are specified with one or several machines PMi.

4.1 Structural features

Structural features are generally specified by assertions on the existence of types of entities in the pattern. Entities, that compose the architecture of an SOA design pattern, can be either *Participants* or *Agents*. Using Event-B, we specify in a context PCi the two entities as constants. The set *Entity* is composed of the set of all *Participants* and the set of all *Agents* ($Entity = Participant \cup Agent \wedge Participant \cap Agent = \varnothing$). This is specified by using a partition (section 2) in the `AXIOMS` clause (*Entity_partition*).



SETS
Entity
CONSTANTS
Participant
Agent
AXIOMS
Entity_partition : $partition(Entity, Participant, Agent)$

Participants name P_i are specified as constants in the `CONSTANTS` clause. The set of participants is composed of all participants name. Formally, this is specified by a partition (*Participant_partition*) i.e. $Participant = \{\mathrm{P}_1,...,\mathrm{P}_n\} \wedge \mathrm{P}_1 \neq \mathrm{P}_2 \wedge ... \wedge \mathrm{P}_{n-1} \neq \mathrm{P}_n$.

Agents name A_i are also specified as constants. The set of agents is specified using a partition in the `AXIOMS` clause (*Agent_partition*), that is $Agent = \{\mathrm{A}_1,...,\mathrm{A}_n\} \wedge \mathrm{A}_1 \neq \mathrm{A}_2 \wedge ... \wedge \mathrm{A}_{n-1} \neq \mathrm{A}_n$.

In the SoaML modeling a ServiceChannel $\mathrm{PushE}_i\mathrm{E}_j$ is a connection between two entities. It can be between two participants ($\mathrm{PushP}_i\mathrm{P}_j$), two agents ($\mathrm{PushA}_i\mathrm{A}_j$) and between a participant and an agent. When the direction of the connection is from a participant to an agent, it is named $\mathrm{PushP}_i\mathrm{A}_j$ and if

CONSTANTS
$P_1, ..., P_n$
AXIOMS
Participant_partition : $partition(Participant, \{P_1\}, ..., \{P_n\})$

CONSTANTS
$A_1, ..., A_n$
AXIOMS
Agent_partition : $partition(Agent, \{A_1\}, ..., \{A_n\})$

it is from an agent to a participant, it is named Push$A_i P_j$. Formally, ServiceChannels are specified with an Event-B relation between two entities. ServiceChannel's name Push$E_i E_j$ are specified with constants in the `CONSTANTS` clause. The set of ServiceChannels is composed of all ServiceChannel's name. This is specified formally with a partition (*ServiceChannel_partition*).

CONSTANTS
$ServiceChannel$
$PushE_iE_j, ..., PushE_nE_m$
AXIOMS
ServiceChannel_Relation : $ServiceChannel \in Entity \leftrightarrow Entity$
ServiceChannel_partition : $partition(ServiceChannel, \{PushE_iE_j\}, ..., \{PushE_nE_m\})$

To define the source and the target of a service channel, two axioms must be added, namely the domain and the range.

PushE_iE$_j$_Domain : $dom(\{PushE_iE_j\}) = \{E_i\}$
PushE_iE$_j$_Range : $ran(\{PushE_iE_j\}) = \{E_j\}$

MessageType is the type of messages exchanged between different entities, it is declared in the `SETS` clause. Messages name M_i are specified in the `CONSTANTS` clause. They are attributed with their type with a partition in the `AXIOMS` clause (*Message_partition*).

SETS
$MessageType$
CONSTANTS
$M_1, ..., M_n$
AXIOMS
Message_partition : $partition(MessageType, \{M_1\}, ..., \{M_n\})$

In some SOA design patterns, entities are organized in various ways across many orthogonal dimensions. For example they can be organized by service layers or by physical boundaries. In the SoaML modeling Catalogs provide a means of classifying and organizing elements by Categories . A collection of related entities are characterized by a Category . Applying a Category to an entity by using a *Categorization* places that entity in the Catalog . Formally Catalogs are specified with an Event-B catalog type and catalogs name C_i are specified with constants in the `CONSTANTS` clause. The set of Catalogs is composed of all Catalogs name. This is specified formally with a partition (*Catalog_partition*). Like Catalogs , Categories are specified with an Event-B category type and categories name C_i are specified with constants in the `CONSTANTS` clause. The set of Categories is composed of all Categories name. This is specified formally with a partition (*Category_partition*). The containment relation of a Catalog with Categories is specified with the relation *Belongs_to* and the link of *Categorization* is specified with a relation between a Category and an Entity.

SETS
$Catalog$
$Category$
CONSTANTS
$C_1, ..., C_n$
$Ca_1, ..., Ca_n$
$Belongs_to$
$Categorization$
AXIOMS
Catalog_partition : $partition(Catalog, \{C_1\}, ..., \{C_n\})$
Category_partition : $partition(Category, \{Ca_1\}, ..., \{Ca_n\})$
Belongs_to_Relation : $Belongs_to \in Catalog \leftrightarrow Category$
Categorization : $Categorization \in Category \leftrightarrow Entity$
Belongs_to_init : $Belongs_to = \{C_n \mapsto Ca_1, ..., C_n \mapsto Ca_n\}$
Categorization_init : $Categorization = \{Ca_1 \mapsto P_i, ..., Ca_n \mapsto A_j\}$

4.2 Behavioral features

A machine of a pattern specification PMi has a state defined by means of a number of variables and invariants. Some of variables can be general as the variable *Send*, which denotes the sent message and the variable *Process*, which denotes the message process. The variable *Send* is defined with the invariant *Send_Relation* which specify that *Send* is a relation between a $ServiceChannel$ and a $MessageType$ so we know the sender, the receiver and the sent message. The variable $Process$ is defined with the invariant *Process_Function* which specify that *Process* is a function between a $Participant$ and a $MessageType$ so we know which participant is processing which message.

```
VARIABLES
  Send
  Process
INVARIANTS
  Send_Relation : Send ∈ ServiceChannel ↔ MessageType
  Process_Function : Process ∈ Participant ⇸ MessageType
```

Each pattern has its own behavior but some events can be general like the event of sending a message $Sending_M_i$ and the event of processing a message $Processing_M_i$.

```
Event  Sending_M_i
  when
    grd : G(v)
  then
    act : Send := Send∪
    {PushE_iE_j ↦ M_i}
  end
```

```
Event  Processing_M_i
  when
    grd : G(v)
  then
    act : Process := Process⩤
    {P_i ↦ M_i}
  end
```

4.3 Context extension and Machine refinement

The specification of a pattern P will be too complicated and error prone if it is done in one shot. To reduce this complexity, we define specification levels. In the first level, we create an abstract model (a context $PC0$ and a machine $PM0$). In the next levels, we use the refinement techniques to gradually introduce detail and complexity into our model until obtaining the final pattern specification. Our refinement strategy is explained in Figure 4. When we move from Level(i) to Level(i+1), we add a new entity and its connections to the model. In Level(i+1), the context PCi is extended with the context $PC(i+1)$ and the machine PMi is refined with the machine $PM(i+1)$. The refined machine sees the extended context.

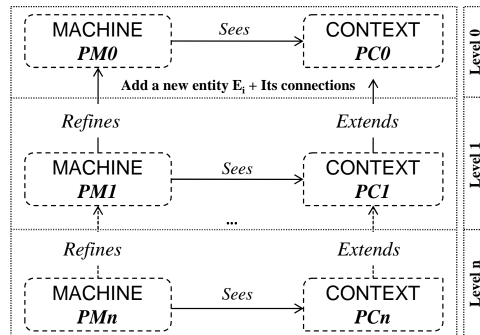


Fig. 4. Refinement strategy

4.4 Case study: Asynchronous Queuing pattern

To illustrate the formalization step of our approach, we apply it on the same pattern example used in the modeling step (*Asynchronous Queuing* pattern). The model of this pattern is composed of two contexts $AQC0$ and $AQC1$ and two machines $AQM0$ and $AQM1$ (AQC denotes Asynchronous Queuing Context and AQM denotes Asynchronous Queuing Machine). In the first level of specification, we specify the pattern at a high level of abstraction, i.e. we suppose that the communication is only between $ServiceA$ and $ServiceB$. In the second level, we add the $Queue$ and all its behavior to the model. Machines and contexts relationships are illustrated in Figure 5.

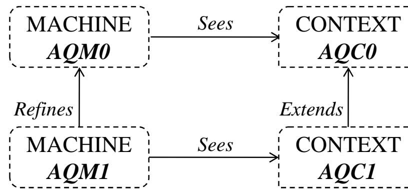


Fig. 5. Contexts and machines relationships

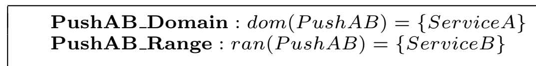
Structural features In the *Asynchronous Queuing* pattern, we have three Participants: $ServiceA$, $ServiceB$ and the $Queue$. In the context $AQC0$, we specify only two participants $ServiceA$ and $ServiceB$.

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

$ServiceA$ and $ServiceB$ are connected together through the *ServiceChannels* $PushAB$ and $PushBA$.

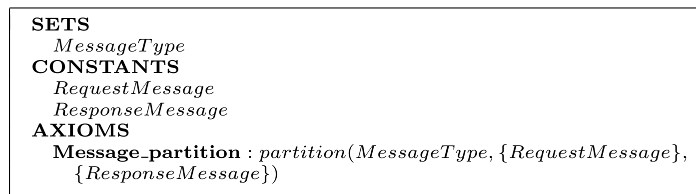
CONSTANTS
$ServiceChannel$
$PushAB$
$PushBA$
AXIOMS
ServiceChannel_Relation : $ServiceChannel \in Entity \leftrightarrow Entity$
ServiceChannel_partition : $partition(ServiceChannel, \{PushAB\}, \{PushBA\})$

For each service channel, we add two axioms in order to define the domain and the range. For example, for $PushAB$ relation we add the following two axioms to denote that its source is $ServiceA$ and its target is $ServiceB$.



PushAB_Domain : $dom(PushAB) = \{ServiceA\}$
PushAB_Range : $ran(PushAB) = \{ServiceB\}$

We did not specify ports and interfaces because they are fine details.Whereas, we specify messages to know what message is being exchanged. So, we define the $MessageType$ set, two constants $RequestMessage$ and $ResponseMessage$ and then the message partition.



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

The second context $AQC1$ is an extension of the context $AQC0$. In this context we add a new constant *Queue* and we redefine the *Participant_partition* by adding the *Queue*. Also we add four constants $PushAQ$, $PushQB$, $PushBQ$ and $PushQA$ to define the new $ServiceChannels$. Axioms that restrict the domain and the range of these $ServiceChannels$ are also added to the context. This part of specification belongs to the Participant diagram (Figure 1) and MessageType diagram (Figure 2).

Behavioral features To specify behavioral features, we have two steps. First, we specify the pattern with a machine at a high level of abstraction. Second, we add all necessary details to the first machine by using the refinement technique.

In the first machine $AQM0$, 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: $ServiceA$ sends a $RequestMessage$ to $ServiceB$ and then remains released from resources and memory (*unavailable*). When $ServiceB$ becomes available, it receives the $Request\ Message$, process it and sends the $Response\ Message$. When $ServiceA$ becomes available, it receives the $Response\ Message$, process it and then becomes deactivated.

Formally, we can use three variables to represent the state of the pattern; *Dispo* to denote the state of the participant either available or not, *Send* to indicate who sends what message and *Process* to indicate which participant is processing 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 cannot be available and not available at the same time. The second invariant, i.e. *Send_Relation*, specifies what is the sent message, who is the sender and the receiver. The third invariant, i.e. *Process_Function*, specifies the message process with a partial function that relates a *Participant* to a $MessageType$.

INVARIANTS
  Dispo_Function : $Dispo \in Participant \rightarrow\!\!\!\!\!+ BOOL$
  Send_Relation : $Send \in ServiceChannel \leftrightarrow MessageType$
  Process_Function : $Process \in Participant \rightarrow\!\!\!\!\!+ MessageType$

As presented in the pattern, initially $ServiceA$ is available and $ServiceB$ is not available. Also, there are no messages sent and no message is processed. Hence, both *Send* relation and *Process* function are initialized to the empty set.

INITIALISATION
  begin
    init1 : $Dispo := \{ServiceA \mapsto TRUE, ServiceB \mapsto FALSE\}$
    init2 : $Send := \varnothing$
    init3 : $Process := \varnothing$
  end

The dynamic system can be seen in Figure 3. It is formalized by the following events; **Sending_Req**, **Processing_Req**, **Sending_Resp** and **Processing_Resp** (Req denotes Request and Resp denotes Response). Sending the request message starts when there is no messages sent and $ServiceA$ is available. This is formally specified with the event *Sending_Req*.

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

The event of processing the request is triggered when the message is sent, not yet processed and $ServiceB$ is available. In the action part, we add, to the process function, the pair ($ServiceB \mapsto RequestMessage$) to denote that $ServiceB$ is processing the request.

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

$ServiceB$ sends the $ResponseMessage$ when the request message is processed and when $ServiceB$ is available. After that $ServiceB$ becomes unavailable.

Event Sending_Resp
when
grd1 : $ServiceB \in dom(Dispo) \land Dispo(ServiceB) = TRUE$
grd2 : $RequestMessage \in ran(Process)$
grd3 : $ResponseMessage \notin ran(Send)$
then
act1 : $Send := Send \cup \{PushBA \mapsto ResponseMessage\}$
act2 : $Dispo(ServiceB) := FALSE$
end

After sending the response, $ServiceA$ precess the received message and becomes unavailable.

Event Processing_Resp
when
grd1 : $RequestMessage \in ran(Send)$
grd2 : $ServiceA \in dom(Dispo) \land Dispo(ServiceA) = TRUE$
then
act1 : $Process := Process \lhd\!\!\!- \{ServiceA \mapsto ResponseMessage\}$
act2 : $Dispo(ServiceA) := FALSE$
end

The second machine $AQM1$ refines the cited above $AQM0$ machine and uses the $AQC1$ context. In the $AQM1$ machine, we introduce the behavior of the $Queue$, so as to complete all the behavior of the pattern. We add two new variables named $Store$ and $Transmit$. $Store$ is specified with a relation that relates a $Participant$ to a $MessageType$. 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 $MessageType$. We add an invariant that restrict the domain of this function to only the $Queue$ indicating that the $Queue$ is transmitting what message. Initially $Store$ relation and $Transmit$ function are both initialized to the empty set.

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

The $AQM1$ machine events are defined in Figure 6. We keep the **Sending_Req** and the **Sending_Resp** events. 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**.

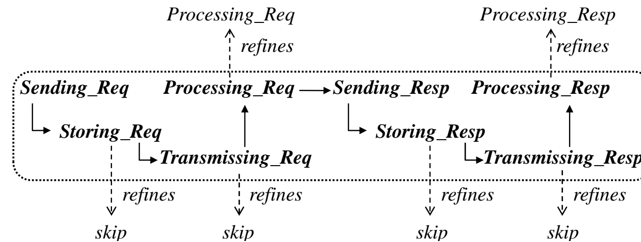


Fig. 6. $AQM1$ events

Due to space restrictions, we did not present the four new events. We present only **Storing_Req** and **Transmissing_Req** events, the other two events are similar to them. The event **Storing_Req** is

triggered when the $RequestMessage$ is sent, not yet processed and when $ServiceB$ is not available. When the message is stored, the **Transmissing_Req** event can be triggered.

Event **Storing_Req**
when
grd1 : $RequestMessage \in ran(Send)$
...
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

The two events of processing the messages are refined by adding in the guards clause the condition of transmitting the message. If a participant ($ServiceA$ or $ServiceB$) receives a message, the storage of this message in the $Queue$ becomes unnecessary, so in the processing event we empty the $Queue$.

Proof obligations Proof obligations define what is to be proved to ensure the consistency of an Event-B model and there are no deadlocks present in it. More over, when we enrich the pattern model by using refinement techniques, we make sure that refined models are not contradictory. These proofs are automatically generated by the Rodin Platform. They 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.

5 Related work

Research connected to design patterns in the field of software architecture, are mainly classified into three branches of work according to their architectural style. The first is about design patterns for Object-Oriented Architectures, the second is about design patterns for Enterprise Application Integration (EAI), and the third is for SOA.

Among research related to design patterns for Object-Oriented Architectures, we present the work of Gamma et al [Gamma et al., 1995]. They have proposed a set of design patterns in the field of object-oriented software design. These patterns are described with graphical notations based on the OMT (Object Modeling Technique) notation. There is no formal semantics associated with these patterns, hence their meanings can be imprecise. Several research have proposed the formalization of these patterns [Gamma et al., 1995] (hereafter referred to as GoF) using different formal notations. We quote: Zhu et al. [Zhu and Bayley, 2010] specify 23 GoF patterns formally. They use the First-Order Logic (FOL) induced from the abstract syntax of UML defined in the Graphic Extension of BNF (GEBNF) to define both structural and behavioral features of design patterns. Kim et al. [Kim and Carrington, 2009] present an approach to describe design patterns based on role concepts. First, they develop an initial role meta-model using Eclipse Modeling Framework (EMF), then they transform the meta-model to Object-Z in order to specify structural features. Behavioral features of patterns are also specified using Object-Z. Kim et al. also use GoF patterns as examples. Blazy et al. [Blazy et al., 2003] propose an approach for specifying design patterns and how to reuse them formally. They use B-method to specify structural features of design patterns but they do not consider the specification of their behavioral features.

Among research related to design patterns for EAI, we present the work of Gregor et al. [Gregor and Bobby, 2003]. They have proposed a set of design patterns dealing with EAI using messaging. These patterns are presented with a visual proprietary notation. To our knowledge, there is no research work that propose the formalization of EAI design patterns and as examples it refer to Gregor et al. patterns and to EAI patterns in general.

In the branch of SOA design patterns, we find out the work of Erl. Erl has proposed a set of design patterns for SOA [Erl, 2009]. Each pattern is presented with a proprietary informal notation presented in a symbol legend. In order to understand them, the first step is to form a knowledge on the

pattern-related terminology and notation. In addition, Erl proposes a set of specific pattern symbols used to represent a design pattern.

In our research work we are interested in *SOA design patterns* defined by Erl [Erl, 2009]. For these patterns, there are no work that model or formally specify them. Erl presents his patterns with an informal proprietary notation because there is no standard modeling notation for SOA, but now OMG announces the publication of the SoaML language [OMG, 2012], it is a specification for the UML profile and a metamodel for services. So, in our work, we propose to model SOA design patterns with the SoaML standard language.After the modeling step, we propose to specify these patterns formally. Similar to [Zhu and Bayley, 2010,Kim and Carrington, 2009] we define both structural and behavioral features of design patterns using FOL, but we use a different formal method which is Event-B.

In conclusion, most proposed patterns are described with a combination of textual description and a graphical presentation [Gamma et al., 1995], some times using proprietary notations [Gregor and Bobby, 2003], [Erl, 2009], 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 research that specify patterns using formal techniques [Zhu and Bayley, 2010,Blazy et al., 2003] but research that model design patterns with semi-formal languages are few [Mapelsden et al., 2002]. We find a number of approaches that formally specify different sorts of features of patterns: structural, behavioral, or both.

6 Conclusions

In this paper, we presented a formal architecture-centric design approach supporting the modeling and the formalization of message-oriented SOA design patterns. The modeling phase allows to represent SOA design patterns with a graphical standard notation using the SoaML language. The formalization phase allows to formally specify both structural and behavioral features of these patterns at a high level of abstraction using the Event-B method.We implemented the elaborated specifications under the Rodin platform. We illustrated our approach through a pattern example within the "Service messaging patterns" category. Currently, the transition from the SoaML modeling to the formal specification is achieved manually, we are working on automating this phase by implementing transformation rules. Also, we are working on formally specifying pattern composition to make design tasks easier for complex software system architects.

References

[Abrial, 2010] Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition.

[Abrial et al., 2010] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., and Voisin, L. (2010). Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466.

[Blazy et al., 2003] Blazy, S., Gervais, F., and Laleau, R. (2003). Reuse of specification patterns with the B method. In *Proceedings of the 3rd international conference on Formal specification and development in Z and B*, ZB'03, pages 40–57, Berlin, Heidelberg. Springer-Verlag.

[Erl, 2009] Erl, T. w. a. c. (2009). *SOA Design Patterns (The Prentice Hall Service-Oriented Computing Series from Thomas Erl).* Prentice Hall PTR, 1 edition.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[Gregor and Bobby, 2003] Gregor, H. and Bobby, W. (2003). *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions.* Addison Wesley.

[Kim and Carrington, 2009] Kim, S.-K. and Carrington, D. A. (2009). A formalism to describe design patterns based on role concepts. *Formal Asp. Comput.*, 21(5):397–420.

[Mapelsden et al., 2002] Mapelsden, D., Hosking, J., and Grundy, J. (2002). Design pattern modelling and instantiation using DPML. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT'02, pages 3–11. Australian Computer Society, Inc.

[OMG, 2012] OMG (2012). Service oriented architecture Modeling Language (SoaML) Specification. Technical report.

[Zhu and Bayley, 2010] Zhu, H. and Bayley, I. (2010). 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. Springer-Verlag.