A Refinement-Based Approach for Building Valid SOA Design Patterns

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

A Refinement-Based Approach for Building Valid SOA Design Patterns

February 18, 2015

Imen Tounsi, Mohamed Hadj Kacem, Ahmed Hadj Kacem
Univ. Sfax, ReDCAD, Sfax, Tunisia
E-mail: imen.tounsi@redcad.org, mohamed.hadjkacem@redcad.org,
ahmed.hadjkacem@fsegs.rnu.tn
Khalil Drira
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ. Toulouse, LAAS, F-31400 Toulouse, France
E-mail: khalil@laas.fr

Abstract

Although design patterns have become increasingly popular, most of them are proposed 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 refinement-based 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.

1 Introduction

During its course of nearly five decades, software engineering has known several main evolutions regarding approaches to software development. Structured programming gave way to the concept of object-orientation. Today's current trend is clearly service orientation. *Service-oriented architecture* (SOA), as emerging architectural model, attracts attention worldwide. Recent advances in SOA, including storage and networking, are providing exciting opportunities to make significant progress in solving complex real-world challenges.

However these architectures are subject to some quality attribute failures (e.g., reliability, availability, and performance problems). *Design patterns*, as tested design solutions for common design problems within a context, have been widely used to tackle a spectrum of design problems and solve these weaknesses (Erl, 2009).

Patterns, proposed by the SOA design pattern community, are described with informal visual notations that can raise ambiguity and may lead to their incorrect usage (Erl, 2009). Modeling these 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 (Tounsi et al., 2013c,b) 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.

Our approach is based principally on three contributions. The first contribution consists in modeling SOA design patterns with a semi-formal language. We propose an SoaML-based (Service oriented architecture Modeling Language) approach for this modeling step. We introduce a metamodel, using extended UML 2.0 notations. This modeling step is proposed in order to attribute a visual standard notation to SOA design patterns. This part of our approach provides several advantages. First, the modeling process is defined in a high level of abstraction providing a generic and reusable model. Second, our approach seeks to take advantage of the expressive power of standard visual notations provided by the semi-formal SoaML 2.0 language that makes easy the understanding of design patterns. Third, a software environment supporting the different features of this approach, has been implemented and integrated as a plug-in in the open source Eclipse framework.

The second contribution consists in proposing a generic formalization of these patterns using the Event-B method. This step is enhanced with the automatic transformation of SoaML pattern diagrams to Event-B pattern specifications with respect to transformation rules (Tounsi et al., 2013a). We implement a rule-based generator which automatically translates the design pattern models that can be modeled using our tool into Event-B specifications (Tounsi et al., 2013d).

Finally, the third contribution is based on formal methods. Using the Rodin theorem prover tool supporting Event-B, we check the syntax of the generated Event-B SOA design pattern models as well as their correctness (i.e. no deadlocks...)

An other advantage of our approach is the use of refinement techniques that make the understanding of pattern models easy. At the first level of the modeling step an abstract model is specified which is further refined in the next levels to add more details. The graphical models in SoaML are automatically translated into Event-B specifications at each refinement level.

The remainder of the paper is organized as follows. In section 2, we provide some background information on the SoaML language and the Event-B notation. In section 3, we give a short overview of our approach. In section 4, we present our approach for modeling and refining SOA design patterns, then we show how we can prove their correctness. In section 5, we illustrate our approach through a case study. In section 6, we present the Eclipse plug-in that implements our approach. In section 7, we discuss the related works. We examine several research done on the modeling and the formalisation of design patterns in general. Ultimately, in section 8, we present conclusions and future work.

2 Basic concepts and notations

In this section, we provide some background information on the SoaML modeling language and the Event-B method.

2.1 SoaML

SoaML[1] (Service oriented architecture Modeling Language) (OMG, 2012) 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 metamodel that extends the UML 2.0 (Unified Modeling Language).

To model SOA design patterns, we can represent many description levels. The highest level is described as *Services Architectures* where participants are working together using services. It 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 services mentioned above and it is modeled using UML collaboration diagram stereotyped «ServiceContract». In the next level, we find the specification of *Interfaces* and *Message Types* using UML class diagrams stereotyped respectively «ServiceInterface» and «MessageType». For both the service contract and the interface levels we can specify behavioral features of patterns using any UML behavior (e.g sequence or activity diagrams).

[1] http ://www.omg.org/spec/SoaML/

2.2 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.

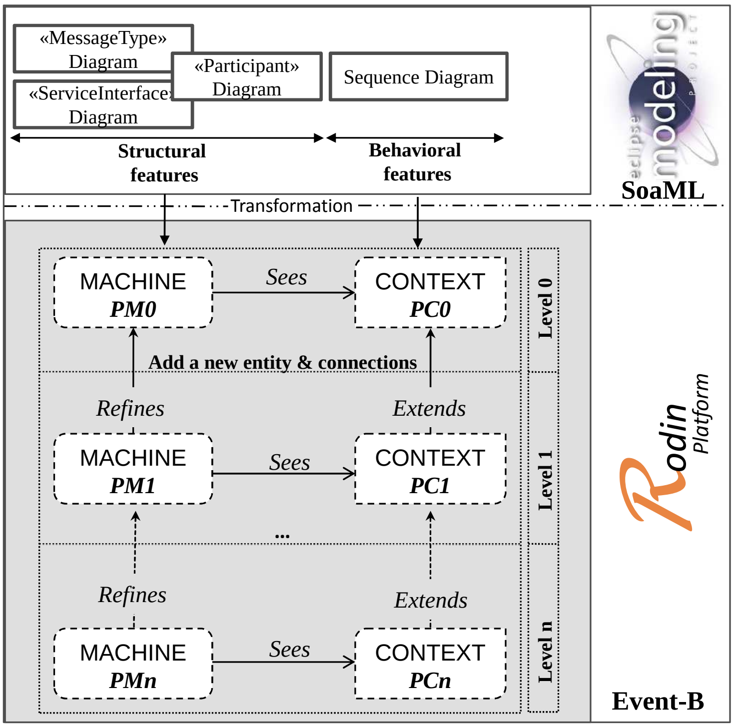
Partitions are used in two different manners. The first one is $partition(S,\ A,B)$. It means that A and B partition the set S, i.e. S=$A \cup B \wedge A \cap B = \varnothing$. The second one is $partition(S,$ $\{A\}$,$\{B\}$,$\{C\}$) which is a specialized use for enumerated sets. It means that S={A,B,C}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

3 Our approach in a nutshell

Our contribution is a formal architecture-centric design approach. It supports the graphical modeling of message-oriented SOA design patterns with the semi-formal SoaML standard language (OMG, 2012), the automatic transformation of pattern diagrams to Event-B specifications (Abrial, 2010) and the formal verification of their correctness. We provide both structural and behavioral features of SOA design patterns in the modeling step as well as in the formalization step. As presented in Figure 1, in the modeling step, structural features are described with the «Participant» diagram, the «ServiceInterface» diagram and the «MessageType» diagram. These diagrams are modeled with an Eclipse plug-in that we propose and transformed to one or several CONTEXTS in the Event-B specifications. Behavioral features, are described with the UML2.0 «Sequence» diagram that provides a graphical notation to describe dynamic aspects of design patterns. This diagram is modeled with an Eclipse plug-in that we propose and transformed to one or several MACHINES in the Event-B specifications. All the specifications are implemented under the Rodin (Abrial et al., 2010) platform in order to be checked. The specification of a pattern P will be too complicated and error prone if it is done in one shot. In order to handle this complexity, we define specification levels by using a step-wise development approach. Models are developed in a stepwise manner which are then automatically translated into Event-B specifications. Here is our strategy, it is explained in Figure 1:

- In the first level (Level0), we start with creating a very abstract model (a context $PC0$ and a machine $PM0$).
- In the next levels, we use the horizontal refinement techniques (defined in (Abrial, 2010)) to gradually introduce detail and complexity into our model until obtaining the final pattern specification. By applying a horizontal refinement, we extend the state of a pattern model by adding new variables. We can strengthen the guards of an event or add new guards. We also add new actions in an event. Finally, it is possible to add new events (Abrial, 2010). 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

Figure 1: Approach overview (see online version for colours)



refined machine sees the extended context. The Event-B specifications are proved by theorem provers at each refinement step.

4 Proposed approach

The upcoming sections provide descriptions of our proposed approach.

4.1 Modeling SOA design patterns

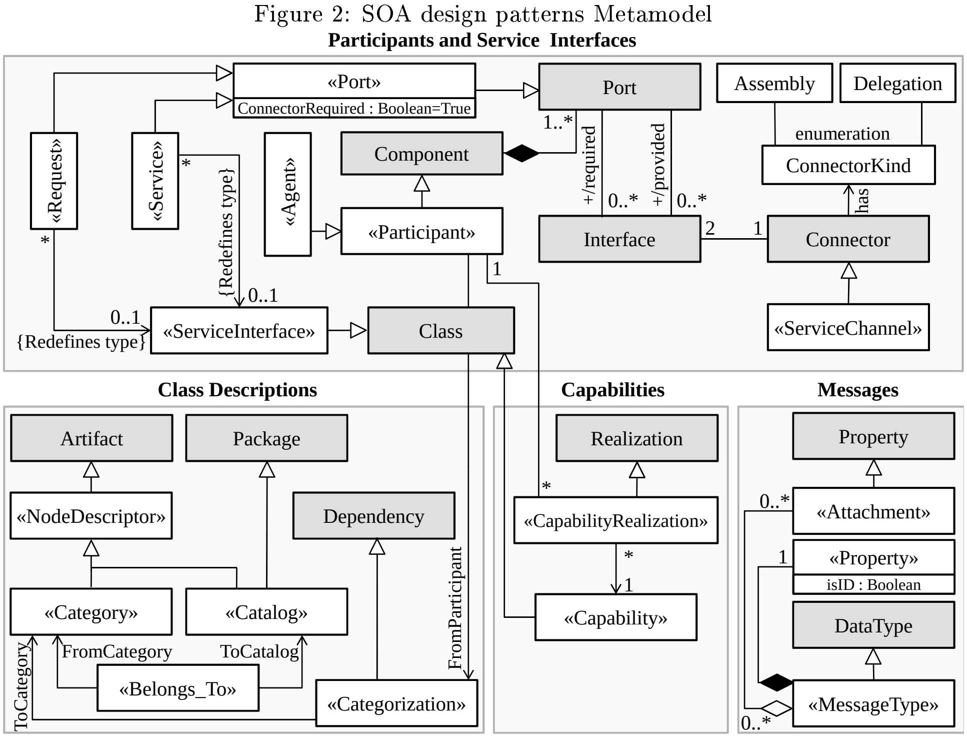
We provide a modeling solution for describing SOA design patterns using a visual notation based on the graphical SoaML language (OMG, 2012). Three main reasons lead to use SoaML. First, it is a standard modeling language defined by OMG. Second, it is used to describe SOA. Third, diagrams used in SoaML,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

The SoaML metamodel extends the UML2.0 metamodel to support an explicit service modeling in distributed environments. This extension is perfectly applied to SOA design patterns modeling. We model structural features of design patterns with «Participant» diagram, «ServiceInterface» diagram and «MessageType» diagram. We model behavioral features with the UML2.0 sequence diagram. To model these diagrams, we use the part of the SoaML metamodel presented in Figure 2. Gray classes represent abstract metaclasses and white classes represent stereotypes. In follows, we only present the base concepts that we use in the pattern modeling.

- Entities, that make up the architecture of an SOA design pattern, can be either «Participants» or «Agents». A «Participant» represents a subclass of *Component* that provides and/or consumes services. «Agents» extend «Participants» with the ability to be active (their needs and capabilities may change over time).
- Entities can have «Ports» that constitute interaction points with their environment. These «Ports» are related to one or more *provided* or *required* $Interfaces$ and their types can be either «Service» or «Request».

- The communication path between *Services* and *Requests* within an architecture is called «ServiceChannel», it extends the metaclass *Connector*.
- A «Capability» is the ability to act and produce an outcome that achieves a result, it extends the metaclass *Class*. A «Participant» can realize zero or several capabilities with the link «CapabilityRealization».
- «ServiceInterfaces» are used to describe provided and required operations to complete services functionality, they can be used as protocols for a service port or a request port.
- The «MessageType» is used to specify information exchanged between services, it extends the metaclass *DataType*. An «Attachment» is a part of a message that is attached to it, it extends the metaclass *Property*. The stereotype «Property» extends the metaclass *Property* with the ability to be distinguished as an identifying property ("primary key" for messages).

Figure 2: SOA design patterns Metamodel



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. «Catalogs» provide a means of classifying and organizing elements by «Categories» for any purpose, they extends the metaclass *Package* and specializes the stereotype «NodeDescriptor». «Categories» are related to «Catalogs» with the relation «Belongs_to». 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».

4.2 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.

Three main reasons lead to use Event-B method. First, it allows the specification of structural and behavioral features of design patterns. Second, refinement techniques proposed by this method allow to represent patterns at different abstraction levels. Third, mathematical proofs allow to verify model consistency and consistency between refinement levels.

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

4.2.1 Structural Features

Structural features are generally specified by assertions on the existence of types of entities in the pattern. The configuration of the entities is also described in terms of the static relationships between them (Zhu and Bayley, 2010).

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 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. $Participant = \{P_1,...,P_n\} \wedge P_1 \neq P_2 \wedge ... \wedge P_{n-1} \neq P_n$.

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

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 $Agent = \{A_1,...,A_n\} \wedge A_1 \neq A_2 \wedge ... \wedge A_{n-1} \neq A_n$.

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

In the SoaML modeling a «ServiceChannel» PushE_iE_j is a connection between two entities. It can be between two participants (PushP_iP_j), two agents (PushA_iA_j) and between a participant and an agent. When the direction of the connection is from a participant to an agent, it is named PushP_iA_j and if it is from an agent to a participant, it is named PushA_iP_j. Formally, ServiceChannels are specified with an Event-B relation between two entities. ServiceChannel's name PushE_iE_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$_i$E$_j$_Domain : $dom(\{PushE_iE_j\}) = \{E_i\}$
PushE$_i$E$_j$_Range : $ran(\{PushE_iE_j\}) = \{E_j\}$

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 : $Categorization = \{Ca_1 \mapsto P_i, ..., Ca_n \mapsto A_j\}$

A «Capability» is the ability to produce an outcome that achieves a result. Each Participant is comprised of a set of capabilities. Capabilities are formally specified as follows.

SETS
Capability
CONSTANTS
$Cp_1, ..., Cp_n$
Provide
AXIOMS
Capability_partition : $partition(Capability, \{Cp_1\}, ..., \{Cp_n\})$
Provide_Relation : $Provide \in Participant \leftrightarrow Capability$
Provide_init : $Provide = \{P_i \mapsto Cp_k, ..., P_j \mapsto Cp_m\}$

The «ServiceInterface» diagram models entity interfaces and their relations with messages. We don't do the formalisation of all the elements of this diagram to the event-B specifications, but we do it to know only what entity can send what message.

CONSTANTS
Can_Send
AXIOMS
Can_Send_Relation : $Can\_Send \in entity \leftrightarrow MessageType$
Can_Send_init : $Can\_Send = \{E_i \mapsto M_k, ..., E_j \mapsto M_m\}$

«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 : $partition(MessageType, \{M_1\}, ..., \{M_n\})$

4.2.2 Behavioral features

Behavioral features of a design pattern are generally defined by assertions on the temporal orders of the messages exchanged between the different pattern entities (Zhu and Bayley, 2010).

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 : $Send \in ServiceChannel \leftrightarrow MessageType$
  Process_Function : $Process \in Participant \rightarrow\!\!\!\!\!+ MessageType$

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 : $Send := Send \cup \{PushE_iE_j \mapsto M_i\}$
  end

Event Processing_M$_i$
  when
    grd : $G(v)$
  then
    act : $Process := Process \lhd\!\!\!- \{P_i \mapsto M_i\}$
  end

4.2.3 Formal verification

SoaML as a semi-formal language provides many advantages to defining SOA design patterns, such as standard visual notation. However, the fact that SoaML lacks a precise semantics is a serious drawback because it does not allow proofs and in consequence, with SoaML, we can not verify required properties like liveness (no deadlocks), and reachability property.

During our development, we use a systematic approach that consists in developing a series of more and more accurate models of the pattern we want to build. This technique is called refinement (Abrial, 2010). Each pattern model is analyzed and proved, thus enabling us to establish that it is correct relative to a number of criteria. As a result, when the last model is finished, we will be able to say that this model is *correct by construction* (Abrial, 2010).

Four formal verification techniques have been used for checking design patterns; *type checking*, *model checking*, *animation* and *theorem proving*. Type checking is a technique controlling low level properties of variables in a program. We use it to check the syntax of the generated Event-B pattern specifications and to detect modeling errors (ex. modeling incomplete ServiceChannel). It is done within the compiler. Model checking and animation are two techniques used to show the dynamic behavior of a model and they allow to systematically explore all its reachable states. We use them to check the behavior of the pattern if it is correct or not. Some temporal/behavioral properties are verified like liveness (no deadlocks present in the model) and reachability (prove that an event whose guard is not necessarily true now will nevertheless certainly occur within a certain finite time) properties. This is done by the model checker ProB (Leuschel and Butler, 2003). Theorem proving technique allows to check properties which can be experimented either as predicates (`INVARIANTS`, `AXIOMS`, `THEOREMS`) or with guards in the events. It is also ensured by proof obligations. They define what is to be proved to ensure the consistency of an Event-B pattern model. As example of consistency constraint, we check that each entity can't send a message only if it is authorised. This is controlled by the invariant Can_Send_INV. For sequence diagrams, we require that every message must start an activation.

INVARIANTS
Can_Send_INV : $\forall z, x, y \cdot z \in Entity \wedge \{x \mapsto y\} \in ServiceChannel \leftrightarrow MessageType \wedge dom(\{x\}) = \{z\} \wedge x \mapsto y \in Send \Rightarrow z \mapsto y \in Can\_Send$

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. Our approach allows developers to reuse correct SOA design patterns, hence we can save effort on proving pattern correctness.

5 Case study: Asynchronous Queuing pattern

Asynchronous Queuing pattern[2] 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 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 section 5.1.2.

5.1 Modeling step

5.1.1 Structural features

In the structural modeling step, we specify entities of the pattern and their dependencies (connections) in the «Participant» diagram (Figure 3) and we specify their interfaces and exchanged messages in the «ServiceInterface» and «MessageType» diagrams respectively (Figure 4).

ServiceA, *ServiceB* and the *Queue* are defined as participants because they provide and use services. As shown in Figure 3, *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

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

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.

Figure 3: «Participant» diagram (see online version for colours)

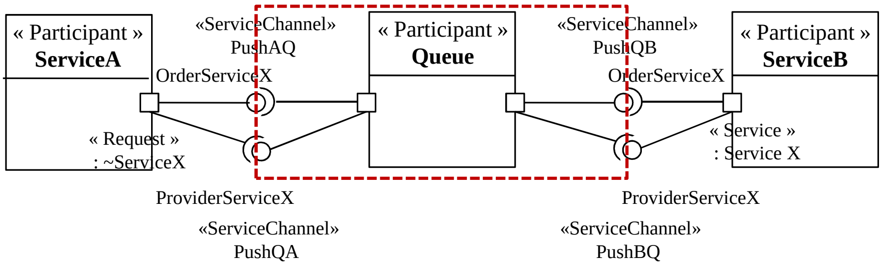
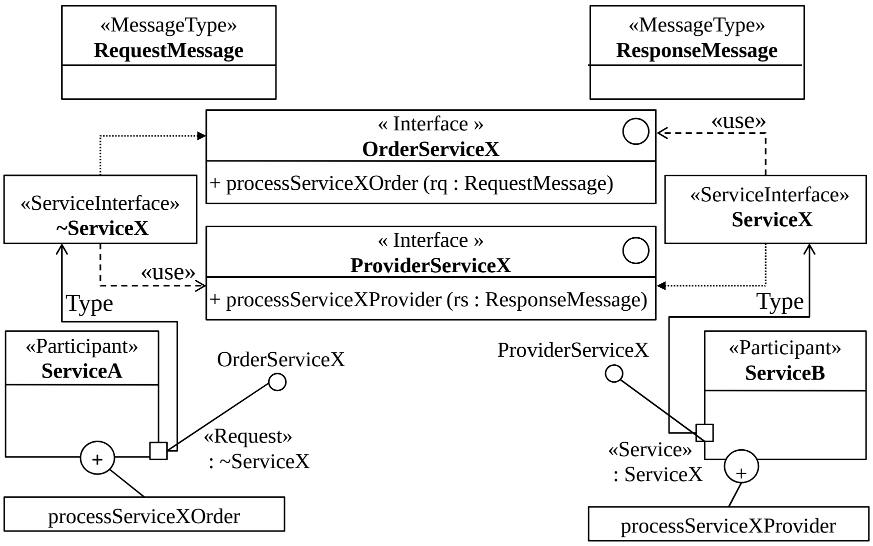


Figure 4 shows a couple of «MessageType» that are used to define the information exchanged between *ServiceA* and *ServiceB*. These messages are "`RequestMessage`" and "`ResponseMessage`", they are used as types for operation parameters of the service interfaces. The type of the *ServiceB*'s port is 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`".

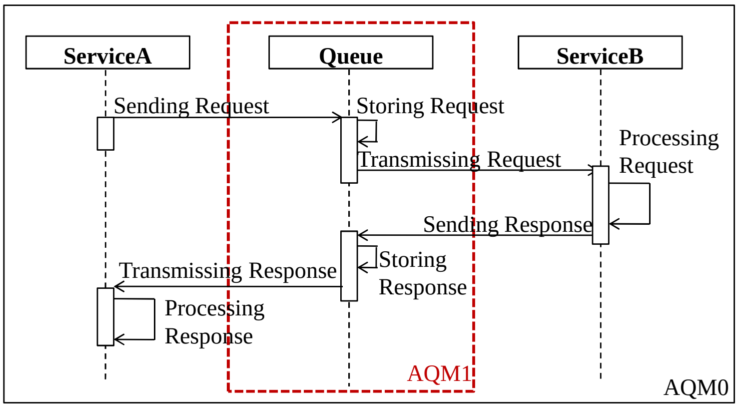
Figure 4: «ServiceInterface» and «MessageType» diagrams



5.1.2 Behavioral features

We use UML2.0 sequence diagram (Figure 5) 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.

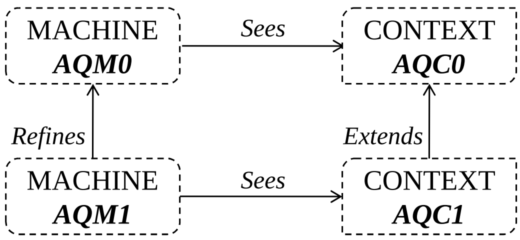
Figure 5: Sequence diagram (see online version for colours)



5.2 Formalization Step

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 6.

Figure 6: Contexts and machines relationships



5.2.1 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 3) and «MessageType» diagram (Figure 4).

5.2.2 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 in order to specify their availability. 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*.

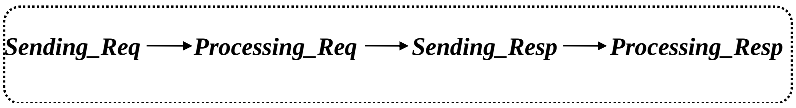
INVARIANTS
Dispo_Function : $Dispo \in Participant \pto BOOL$
Send_Relation : $Send \in ServiceChannel \leftrightarrow MessageType$
Process_Function : $Process \in Participant \pto MessageType$

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 5. 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*. This is illustrated in Figure 7.

Figure 7: *AQM*0 Events



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 : $ServiceB \in dom(Dispo) \wedge Dispo(ServiceB) = TRUE$
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 : $ServiceB \in dom(Dispo) \wedge Dispo(ServiceB) = TRUE$
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, $ServiceA$ process the received message and becomes unavailable.

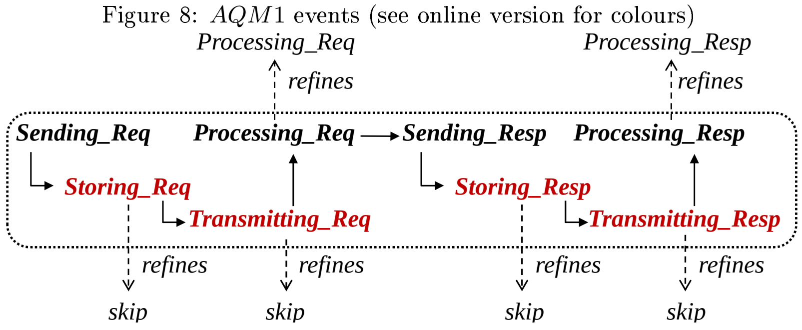
Event **Processing_Resp**
when
grd1 : $RequestMessage \in ran(Send)$
grd2 : $ServiceA \in dom(Dispo) \wedge Dispo(ServiceA) = TRUE$
then
act1 : $Process := Process \lhd\!\!\!- \{ServiceA \mapsto ResponseMessage\}$
act2 : $Dispo(ServiceA) := FALSE$
end

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 restricts the domain of this relation to only the $Queue$. Consequently, $Store$ reveals what message the queue is storing. $Transmit$ is specified with a partial function that relates a $Participant$ to a $MessageType$. We add an invariant that restricts the domain of this function to only the $Queue$. Consequently, Transmit reveals what message the $Queue$ is transmitting. Initially $Store$ relation and $Transmit$ function are both initialized to the empty set.

INVARIANTS
Store_Relation : $Store \in Participant \leftrightarrow MessageType$
Store_Dom_Rest : $dom(Store) = \{Queue\} \vee Store = \varnothing$
Transmit_Function : $Transmit \in Participant \rightarrow\!\!\!\!\!+ MessageType$
Transmit_Dom_Rest : $dom(Transmit) = \{Queue\} \vee Transmit = \varnothing$

The $AQM1$ machine events are defined in Figure 8. We keep the **Sending_Req** and the **Sending_Resp** events. We add four new events namely **Storing_Req**, **Transmitting_Req**, **Storing_Resp** and **Transmitting_Resp**. These events are related to the $Queue$ behavior. We add more details to the abstract events **Processing_Req** and **Processing_Resp**.

Figure 8: $AQM1$ events (see online version for colours)



Due to space restrictions, we did not present the four new events. We present only **Storing_Req** and **Transmitting_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 **Transmitting_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 **Transmitting_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*.

6 Tool support

Our approach is enhanced by an Eclipse plug-in[3] . It is a graphical modeling tool that makes the modeling of SOA design patterns easier. It ensures an easy and efficient modeling way of SOA design patterns. For the development of the plug-in, we have used several Eclipse frameworks, i.e., GMF (*Graphical Modeling Framework*) (Eclipse, 2010a), EMF (*Eclipse Modeling Framework*) (Steinberg et al., 2009) and GEF (*Graphical Editing Framework*) (Eclipse, 2010b). Several diagrams are available in the plug-in; we can model «Participant» diagram, «Service Interface» diagram, «Message Type» diagram and UML2.0 Sequence diagram.

The SOA design patterns diagram editor is a tool where diagrams can be created to model patterns. Graphical elements can be picked up from a tool palette and created in the Diagram editor pane in a "drag-and-drop" way. Elements of the palette are listed under *Nodes* and *Links* elements. The "Property Editor" can be used for changing properties of the object selected in the diagram editor pane. Property elements vary depending on the type of the chosen object. Figure 9 shows the diagram editor of the SOA design patterns with an illustration of the pattern example "Asynchronous Queueing". After modeling a design pattern, the plug-in generates an XML file describing it.

The plug-in transforms the generated XML file, according to transformation rules (described in (Tounsi et al., 2013a)) expressed with the XSLT language (Tounsi et al., 2013d), into Event-B specifications. These specifications can be imported under the Rodin platform to verify their correctness.

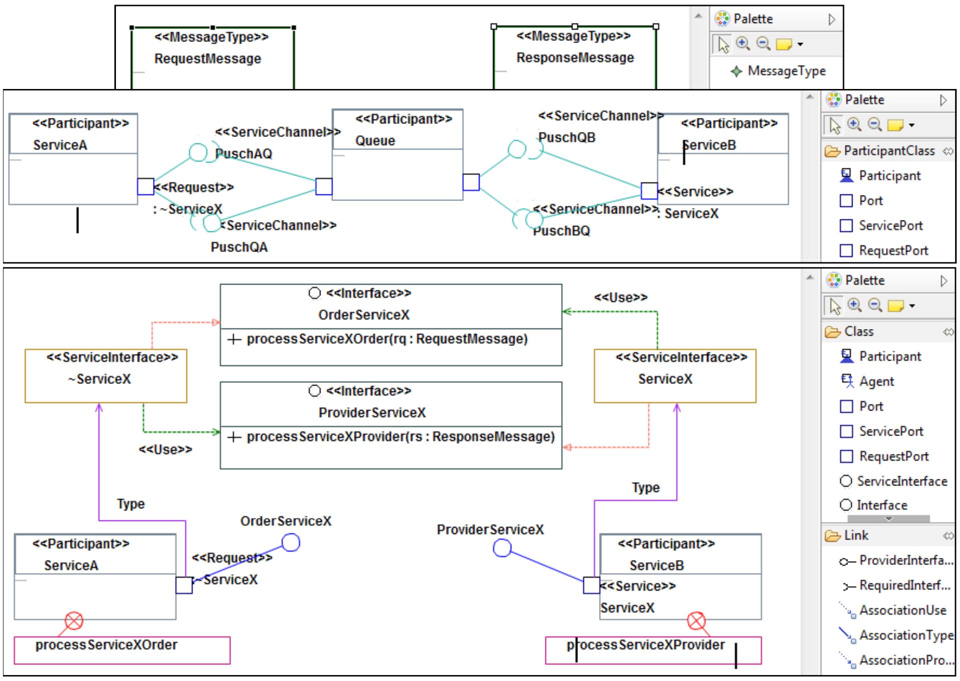
By applying transformations rules on the generated XML specifications, we obtain Event-B specifications presented in Figure 10.

7 Related work

This section surveys related research to design patterns in the field of software architecture. These research are mainly classified into three branches of work according to

[3]The plug-in is available for download in:
http://www.redcad.org/members/imen.tounsi/

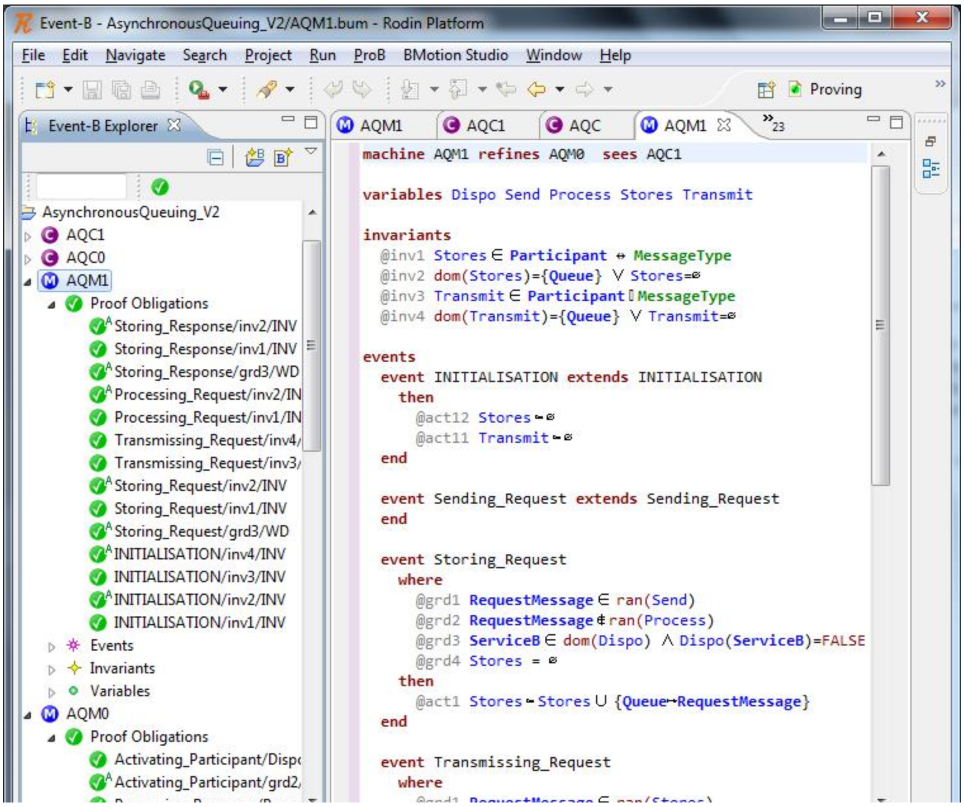
Figure 9: SOA design patterns plug-in (see online version for colours)



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. Taibi et al. (Taibi, 2006; Taibi and Ngo, 2003) develop the 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 FOL and behavioral features in the Temporal Logic of Action (TLA). Taibi et al. use as a case study a pattern composition proposed by GoF. Dong et al. (Dong et al., 2007) focus on the specification of design pattern component. They use the FOL to specify structural features of patterns with Object-Z and TLA to specify their behavioral features. As examples, they use GoF 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

Figure 10: Excerpt of Event-B specification results (see online version for colours)



features.

Among research related to design patterns for EAI, we present the work of Gregor et al. (Hohpe and Woolf, 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. These patterns are modeled without any formal specification. 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.

All cited research work are dealing with object oriented design patterns, 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 (Tounsi et al., 2013c,b), 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 (Hohpe and Woolf, 2003), (Erl, 2009), in order to make them easy

Approach	Object Oriented Design Patterns						EAI Design Patterns	SOA Design Patterns	
	Gamma et al. (Gamma et al., 1995)	**Zhu et al. (Zhu and Bayley, 2010)**	**Taibi et al. (Taibi, 2006)**	**Dong et al. (Dong et al., 2007)**	**Kim et al. (Kim and Carrington, 2009)**	**Blazy et al. (Blazy et al., 2003)**	**Hope et al. (Hohpe and Woolf, 2003)**	**Erl (Erl, 2009)**	**Our appr** (Tounsi et al., 2013b)
Pattern modeling	OMT	GoF (OMT)	GoF (OMT)	GoF (OMT)	GoF (OMT)	GoF (OMT)	Prop Nota	Prop Nota	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

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. Table 1 is a recapitulation of related works that contains a comparison between the above-mentioned approaches and our approach.

8 Conclusions

In this paper, we presented a formal refinement-based 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 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. In order to reach the generality and the validity of our approach, we have applied it to more pattern examples within the "Service messaging patterns" category and "Transformation patterns" category.

In real applications, problems are complex and their solutions can be represented by compound patterns that require the combination and reuse of other design patterns. So, as future work, we are working on formally specifying pattern composition and verifying some related properties.

9 Acknowledgment

This paper is done with the support of the Ministry of Higher Education and Scientific Research of Tunisia within the Tunisian-French scientific cooperation (DGRS/CNRS).

References

J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521895561, 9780521895569.

J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, November 2010. ISSN 1433-2779. doi: 10.1007/s10009-010-0145-y. URL http://dx.doi.org/10.1007/s10009-010-0145-y.

S. Blazy, F. Gervais, and R. Laleau. 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, 2003. Springer-Verlag. ISBN 3-540-40253-5. URL http://dl.acm.org/citation.cfm?id=1761968.1761972.

J. Dong, P. S. C. Alencar, D. D. Cowan, and S. Yang. Composing pattern-based components and verifying correctness. *J. Syst. Softw.*, 80:1755–1769, November 2007. ISSN 0164-1212. doi: 10.1016/j.jss.2007.03.005. URL http://portal.acm.org/citation.cfm?id=1290192.1290213.

Eclipse. Graphical modeling framework. https://wiki.eclipse.org/Graphical_Modeling_Framework, February 2010a.

Eclipse. Graphical editing framework. http://www.eclipse.org/gef/, February 2010b.

T. w. a. c. Erl. *SOA Design Patterns (The Prentice Hall Service-Oriented Computing Series from Thomas Erl).* Prentice Hall PTR, 1 edition, January 2009. ISBN 0136135161. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0136135161.

E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.

G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321200683.

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

M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: FORMAL METHODS, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

D. Mapelsden, J. Hosking, and J. Grundy. 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., 2002. ISBN 0-909925-88-7. URL http://dl.acm.org/citation.cfm?id=564092.564094.

OMG. Service oriented architecture Modeling Language (SoaML) Specification. Technical report, Fundacion European Software Institute, May 2012.

D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

T. Taibi. Formalising design patterns composition. *Software IEE Proceedings*, 153(3):127–136, 2006.

T. Taibi and D. C. L. Ngo. Formal specification of design pattern combination using BPSL. *Information and Software Technology*, 45(3):157 – 170, 2003. ISSN 0950-5849. doi: DOI:10.1016/S0950-5849(02)000195-7. URL http://www.sciencedirect.com/science/article/pii/S0950584902001957.

I. Tounsi, M. Hadj Kacem, and A. Hadj Kacem. Building Correct by Construction SOA Design Patterns: Modeling and Refinement. In *Software Architecture: Proceedings of the 7th European Conference, ECSA*, volume 7957 of *Lecture Notes in Computer Science*, pages 33–44, Monpellier, France, July 2013a. Springer Berlin Heidelberg.

I. Tounsi, M. Hadj Kacem, and A. Hadj Kacem. An Approach for Modeling and Formalizing SOA Design Patterns. In *Proceedings of the IEEE 22nd International WETICE Conference (WETICE 2013)*, pages 330–335, Hammamet, Tunisia, June 2013b. IEEE Computer Society.

I. Tounsi, M. Hadj Kacem, A. Hadj Kacem, K. Drira, and E. Mezghani. Towards an Approach for Modeling and Formalizing SOA Design Patterns with Event-B. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC 2013, pages 1937–1938, Coimbra, Portugal, March 2013c. ACM.

I. Tounsi, Z. Hrichi, M. Hadj Kacem, A. Hadj Kacem, and K. Drira. Using SoaML Models and Event-B Specifications for Modeling SOA Design Patterns. In *Proceedings of the 15th International Conference on Enterprise Information Systems (ICEIS 2013)*, pages 294–301, Angers, France, July 2013d. doi: 10.5220/0004453302940301.

H. Zhu and I. 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. ISBN 3-642-16900-7, 978-3-642-16900-7. URL http://portal.acm.org/citation.cfm?id=1939864.1939915.