# Using SoaML Models and Event-B Specifications for Modeling SOA Design Patterns

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

# Using SoaML Models and Event-B Specifications for Modeling SOA Design Patterns

Imen Tounsi[1], Zied Hrichi[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,
dsi.zied.hrichi@gmail.com,ahmed.hadjkacem@fsegs.rnu.tn, khalil@lass.fr

**Abstract.** Although design patterns have become increasingly popular, most of them are presented in an informal way. Patterns, proposed by the SOA design pattern community, are described with a proprietary informal notation, which can raise ambiguity and may lead to their incorrect usage. Modeling SOA design patterns with a standard formal notation avoids misunderstanding by software architects and helps endow design methods. In this paper, we present an approach that aims, first, to model message-oriented SOA design patterns with the SoaML language, and second to transform them to Event-B specifications. 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 enhanced with a tool supporting it. Specification results are imported under the Rodin platform which we use to prove model consistency.

**Keywords:** SOA Design patterns: SoaML modeling: Formal methods: Event-B method: Tool support

## 1 INTRODUCTION

The dominant architectural style for many systems is the *Service-oriented architecture* (SOA), a style that is essentially based on the message exchange. This architecture offers a model and an opportunity to solve problems related to the communication and the integration between heterogeneous and distributed applications [Erl, 2009]. However these architectures are subject to some quality attribute failures (e.g., availability, reliability, and performance problems). Design patterns, as tested solutions to common design problems within a context, have been widely used to solve these weaknesses.

Patterns, proposed by the SOA design pattern community, are described with a proprietary informal notation [Erl, 2009], which can raise ambiguity and may lead to their incorrect usage. So they require modeling with a standard notation and then formalization. The intent of our approach is to model and formalize message-oriented SOA design patterns. These 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 propose an approach for modeling and transforming message oriented SOA design patterns. The key idea is to model these patterns with the semi-formal Service oriented architecture Modeling Language (SoaML) and to transform them into Event-B specifications. We proceed by proposing the SOA design patterns metamodel conformed to the SoaML language. This modeling step is proposed in order to attribute a standard notation to SOA design patterns. Then we propose the transformation of design pattern models, according to transformation rules, into Event-B specifications. We import the generated specifications under the Rodin platform which we use to prove model consistency. We provide structural features of SOA design patterns in the modeling phase as well as in the specification phase. Structural features of a design pattern are generally specified by assertions on the existence of types of components in the pattern. The configuration of the elements is also described, in terms of the static relationships between them. We illustrate our approach through a pattern example "Event-Driven Messaging", proposed by the SOA design pattern community. We also present a tool supporting our approach.

The paper is structured as follows. Section 2 gives background information of some concepts used in this paper. Section 3 gives an overview of our approach. Section 4 describes our tool supporting our approach. Section 5 discusses related work. Section 6 concludes and gives future works.

# 2 BACKGROUND

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

## 2.1 Design Patterns

In the field of information systems, a pattern is defined as a model that provides a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection [Erl, 2009]. Patterns can be classified relatively to their level of abstraction into three categories: *architectural patterns* (or architectural styles) that provide the skeleton or template for the overall shape and the structure of software applications at a high-level design [Gomaa, 2004], *design patterns* that encode a proven solution to a recurring design common problem of automated systems [Ramirez and Cheng, 2009], and *implementation patterns* that provide a solution to a given problem in programming [Beck, 2007]. It is used to generate code.

## 2.2 SoaML

SoaML [4] (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).

## 2.3 XSLT

XSLT [5](eXtensible Styles Language Transformation) is a W3C standard that supports the XML standard. The objective of this specification is to transform XML documents into another document format. XSL is decomposed into two languages, a transformation language and a formatting language. The first one can transform an XML document into another document, while the second one can use predefined tags to represent the visual aspect of an XML document. XSLT apply the transformation written by XSL stylesheet to an XML document. In our approach, we use the XSLT language to transform SoaML diagrams into Event-B specifications.

## 2.4 Event-B

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. Each context has a name and other clauses like "Constants" to declare 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.

# 3 APPROACH OVERVIEW

The main goal of our approach is the modeling of message-oriented SOA design patterns with the semi-formal SoaML standard language, the automatic transformation of pattern diagrams to Event-B specifications and the formal verification of their correctness. Figure 3 depicts the overall approach.

After modeling design patterns, the graphical editor generates an XML file. The plug-in transforms this XML file, according to transformation rules expressed with the XSLT language, into Event-B specifications. These specifications will then be imported under the Rodin theorem prover that supports the generation of Proof Obligations belonging to Event-B models. The Rodin Platform is also used in order to check the syntax of SOA design pattern specifications as well as their correctness.

[4] http ://www.omg.org/spec/SoaML/
[5] http ://www.w3.org/TR/xslt

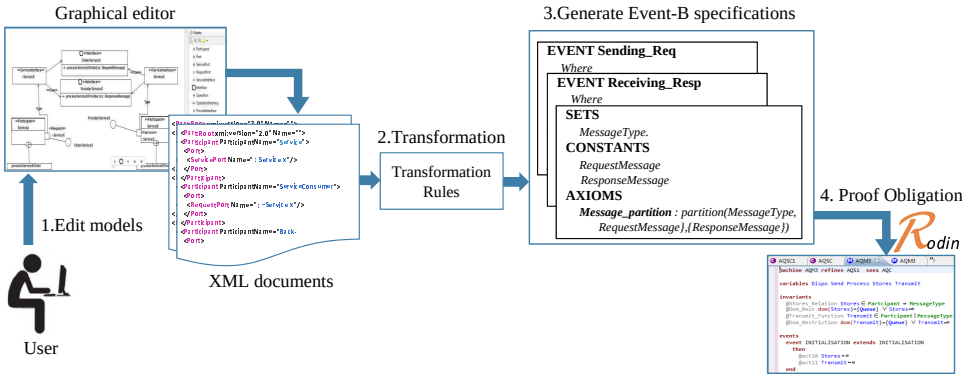


**Fig. 1.** The overall approach

### 3.1 SOA Design Patterns Modeling

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

The SoaML metamodel extends the UML 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, MessageType 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 . 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 communication path between *Services* and *Requests* within an architecture is called ServiceChannel , it extends the metaclass *Connector*.

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).

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 .

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 Belons_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 .

In this paper, we model as example the *Event_Driven Messaging* pattern [6] [Erl, 2009]. It is an SOA design pattern for inter-service message exchange. It resolves the problem of inefficient polling-based interactions for service consumer, generated in order to obtain information about events occurrence. The solution proposed by this pattern is to introduce an event manager allowing the service

[6] http ://soapatterns.org/patterns/event_driven_messaging

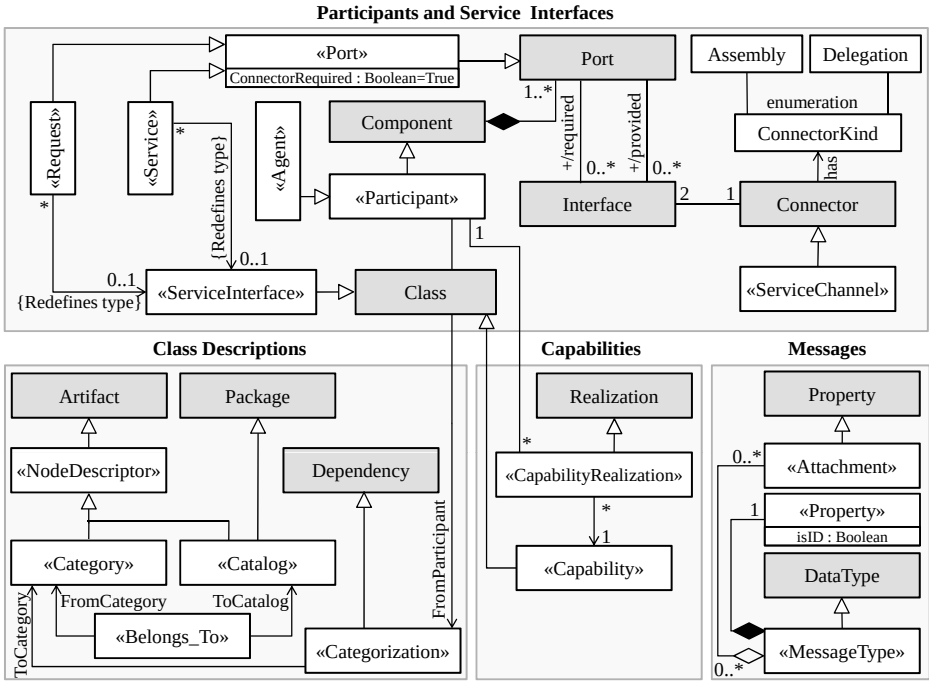


**Fig. 2.** SOA design patterns Metamodel

consumer to set itself up as a subscriber to events associated with a service that assumes the role of publisher. So that service consumers are automatically notified of runtime service events.

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).

The *Subscriber*, the *Publisher* and the *Event-Manager* are defined as participants because they provide and use services. As shown in Figure 3, the *Publisher* provides an *event* used by the *Subscriber*. When the *event* occurs, the *Publisher* automatically sends the event details to the *Event-Manager*, which then broadcasts the event notification to the *Subscriber*. Both the *Publisher* and the *Subscriber* have a port typed with "Event". the *Publisher* is the provider of the service and has a Service port. The *Subscriber* is a consumer of the service and uses a Request port. In this diagram, ServiceChannels are explicitly represented, they enables communication between the different participants.

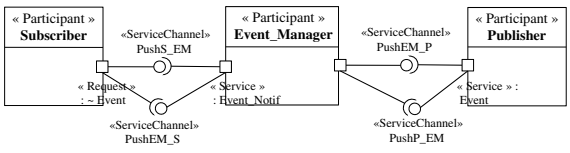


**Fig. 3.** Participant diagram

In the MessageType diagram (Figure 4) three MessageTypes are used to define information exchanged between the *Publisher*, the *Subscriber* and the *Event-Manager*. These messages are "`SubsReq`", "`SubsResp`" and "`EventInfo`", they are used as types for operation parameters of the service interfaces. As shown in Figure 4, the *Publisher*'s port type is the UML interface "`ProviderEvent`" that has the operation "`publishEvent`". This operation has a message style parameter typed "`EventInfo`". The *Subscriber* expresses its request for the "Event" using its request port. The type of this request port is the UML interface "`SubscriberEvent`". This interface has an

operation "`subscribeEvent`" with a parameter typed "`SubsReq`". The type of the *Event-Manager*'s port is the UML interface "`Notification`" that has two operations "`eventNotif`" and "`subsNotif`". These operations have two message style parameters where the type of the parameters are the MessageTypes "`SubsResp`" and "`EventInfo`".

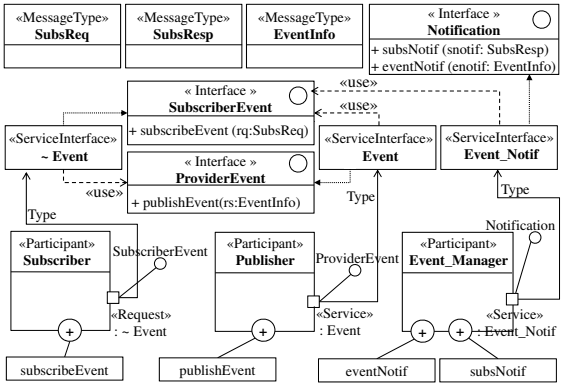


**Fig. 4.** ServiceInterface and MessageType diagrams

### 3.2 SOA Design Patterns Transformation

In the SOA design patterns transformation step, we present the transformation process of SoaML diagrams to Event-B language.

**Participant Diagram Mapping** This diagram constitute the static part of the defined pattern. It is specified in the *Context* part. The transformation of the Participant diagram is based on four major rules allowing the transformation of a graphical model into an Event-B specification.

**R1. Architecture Entities Transformation Rule**

This rule transforms entity types into new Event-B entity types. Participant names and agent names are transformed to constants. The set *Entity* is composed of the set of all *Participants* and the set of all *Agents*. This is specified by using a partition in the `AXIOMS` clause (*Entity_partition*). The following algorithm shows how to transform the architecture entities.

---

**Algorithm** Architecture entities transformation rule

```
1: begin
2: Write (" SETS ")
3: Write ('Entity')
4: Write (" CONSTANTS ")
5: if exist Participant then
6:    Write ('Participant')
7:    for each Participant do
8:       Write (Participant.Name)
9:    end for
10: end if
11: if exist Agent then
12:    Write ('Agent')
13:    for each Agent do
14:       Write (Agent.Name)
15:    end for
16: end if
17: Write (" AXIOMS ")
18: Write ('Entity_partition:partition(Entity')
19: if exist Participant then
20:    Write(',Participant')
21: end if
22: if exist Agent then
23:    Write(',Agent')
24: end if
25: if exist Participant then
26:    Write('Participant_partition (Participant,')
```

```
27:    for each Participant do
28:      Write (Participant.Name)
29:    end for
30:  end if
31:  if exist Agent then
32:    Write('Agent_partition (Agent,')
33:    for each Agent do
34:      Write (Agent.Name)
35:    end for
36:  end if
37:  end
```

**R2. Connections Transformation Rule**

In the SoaML modeling, a ServiceChannel is a connection between two architecture entities. This rule define the graphical connection with an Event-B relation between two entities (ServiceChannel) and transforms ServiceChannels name into constants in the `CONSTANTS` clause. The set of ServiceChannels is composed of all ServiceChannel's name. This is transformed formally to a partition (*ServiceChannel_partition*). This rule also generates *Domain* and *Range* axioms for each service channel to define its source and its target. The following algorithm shows how to transform a service channel.

**Algorithm** Connections transformation rule

```
1:  begin
2:  Write (" CONSTANTS ")
3:  if exist ServiceChannel then
4:    Write ('ServiceChannel')
5:    for each ServiceChannel do
6:      Write (ServiceChannel.Name)
7:    end for
8:  end if
9:  Write (" AXIOMS ")
10: if exist ServiceChannel then
11:   Write('ServiceChannel_partition:partition(ServiceChannel')
12:   for each ServiceChannel do
13:     Write (ServiceChannel.Name)
14:   end for
15:   Write('ServiceChannel_Relation : ServiceChannel ∈ Entity ↔ Entity')
16:   for each ProviderInterface do
17:     Write (ProviderInterface.Origine)
18:     Write('_Domain:dom')
19:     Write (({ProviderInterface.Origine}))
20:     Write('=')
21:     Write ({ProviderInterface.Destinataire})
22:   end for
23:   for each RequireInterface do
24:     Write (RequireInterface.Destinataire)
25:     Write('_Range:ran')
26:     Write (({RequireInterface.Destinataire}))
27:     Write('=')
28:     Write ({RequireInterface.Origine})
29:   end for
30: end if
31: end
```

**R3. Class Descriptions Transformation Rule**

This rule transforms catalog type to a new Event-B catalog type and catalogs name into constants in the `CONSTANTS` clause. The set of Catalogs is composed of all catalogs name. This is transformed formally to a partition (*Catalog_partition*). This rule also transforms category type to a new Event-B category type and categories name into constants in the `CONSTANTS` clause. The set of Categories is composed of all Categories name. This is transformed formally to a partition (*Category_partition*). The relation of containment of a Catalog with Categories is transformed to the relation *Belongs_to*. The link of *Categorization* is transformed to a relation between a Category and an Entity. The following algorithm shows how to transform class descriptions.

**Algorithm** Class descriptions transformation rule

```
1:  begin
2:  Write (" SETS ")
3:  if exist Catalog then
4:    Write ('Catalog')
5:  end if
6:  if exist Category then
7:    Write ('Category')
```

```
 8: end if
 9: Write (" CONSTANTS ")
10: if exist Catalog then
11:    for each Catalog do
12:       Write (Catalog.Name)
13:    end for
14: end if
15: if exist Category then
16:    for each Category do
17:       Write (Category.Name)
18:    end for
19: end if
20: if exist Category and exist Catalog then
21:    Write ('Belongs_To')
22:    Write ('Categorization')
23: end if
24: Write (" AXIOMS ")
25: if exist Catalog then
26:    Write('Catalog_partition:partition(Catalog,')
27:    for each Catalog do
28:       Write (Catalog.Name)
29:    end for
30: end if
31: if exist Category then
32:    Write('Category_partition:partition(Category,')
33:    for each Category do
34:       Write (Category.Name)
35:    end for
36:    Write('Belongs_to_Relation : Belongs_to ∈ Catalog ↔ Category')
37:    Write('Categorization : Categorization ∈ Category ↔ Entity')
38:    Write ('Belongs_to_init:Belongs_to = {')
39:    for each Category do
40:       for each Catalog do
41:          Write (Catalog.Name)
42:          Write('↦')
43:          Write (Category.Name)
44:       end for
45:    end for
46:    Write ('}')
47:    Write ('Categorization_init:Categorization = {')
48:    for each Categorization do
49:       Write (Categorization.TransitionToNoeud)
50:       Write('↦')
51:       Write (Categorization.TransitionFromNoeud)
52:    end for
53:    Write ('}')
54: end if
55: end
```

**R4. Capabilities Transformation Rule**

This rule transforms capability type to a new Event-B capability type and capability name into constants in the `CONSTANTS` clause. The set of Capabilities is composed of all capabilities name. This is transformed formally into a partition (*Capability_partition*). The link between a Participant and a capability is transformed to a relation *Provide*. The following algorithm shows how to transform capabilities.

**Algorithm** Capabilities transformation rule

```
 1: begin
 2: Write (" SETS ")
 3: if exist Capability then
 4:    Write ('Capability')
 5: end if
 6: Write (" CONSTANTS ")
 7: if exist Capability then
 8:    for each Capability do
 9:       Write (Capability.Name)
10:       Write ('Provide')
11:    end for
12: end if
13: Write (" AXIOMS ")
14: if exist Capability then
15:    Write('Capability_partition:partition(Capability,')
16:    for each Capability do
17:       Write (Capability.Name)
18:    end for
19:    Write('Provide_Relation : Provide ∈ Participant ↔ Capability')
20:    Write('Capability_init:Capability={')
21:    for each Realization do
22:       Write (Realization.TransitionFromProperty)
23:       Write('↦')
```

```
24:       Write (Realization.TransitionToCapability)
25:     end for
26:     Write ('}')
27:  end if
28:  end
```

**MessageType Diagram Mapping** This diagram is also specified in the *Context* part. The transformation of this diagram is based on a single rule that allows to transform the graphical model into an Event-B specification. This rule transforms MessageType to a new Event-B message type and messages name into constants in the `CONSTANTS` clause. The set of MessageType is composed of all messages name. This is transformed formally to a partition ($Message\_partition$). The following algorithm shows how to transform MessageTypes.

**Algorithm** MessageType transformation rule

```
1:  begin
2:  Write (" SETS ")
3:  if exist Message then
4:     Write ('MessageType')
5:  end if
6:  Write (" CONSTANTS ")
7:  if exist Message then
8:     for each MessageType do
9:        Write(MessageType.Name)
10:    end for
11: end if
12: Write (" AXIOMS ")
13: if exist Message then
14:    Write('Message_partition:partition(MessageType')
15:    for each MessageType do
16:       Write (,{ MessageType.Name})
17:    end for
18: end if
19: end
```

**Service Interface Diagram Mapping** This diagram is specified in the same *Context*. The transformation rule of this diagram define the relation $Can\_Send$, which is the link between an *Entity* and a $MessageType$. The following algorithm shows how to transform Service Interfaces.

**Algorithm** Service Interface transformation rule

```
1:  begin
2:  Write (" CONSTANTS ")
3:  if exist Participant then
4:     Write ('Can_Send')
5:  end if
6:  Write (" AXIOMS ")
7:  Write('Can_Send_Relation : Can_Send ∈ Entity ↔ MessageType')
8:  Write('Can_Send_init:Can_send = {')
9:  Var1 ← Participant.RequestPort.Name
10: Var2 ← ServiceInterface.Name
11: Var3 ← AssociationUse.Origine
12: for each Participant do
13:    Write(Participant.Name)
14:    Write('↦')
15:    if Var1 = Var2 and Var1 = Var3 then
16:       Select(AssociationUse.Destinataire)
17:       Write(Interface.OperationInterface.Name)
18:    end if
19: end for
20: Write ('}')
21: end
```

## 4 TOOL SUPPORT

Our approach is enhanced by an Eclipse plug-in based on its development on the Frameworks; GMF (*Graphical Modeling Framework*) [Eclipse, 2010b], EMF (*Eclipse Modeling Framework*) [Steinberg et al., 2009] and GEF (*Graphical Editing Framework*) [Eclipse, 2010a]. It is a graphical modeling tool that ensures an easy and efficient modeling way of SOA design patterns. Several diagrams are available in the plug-in; we can model *Participant* diagram, *Service Interface* diagram, and *Message Type* diagram.

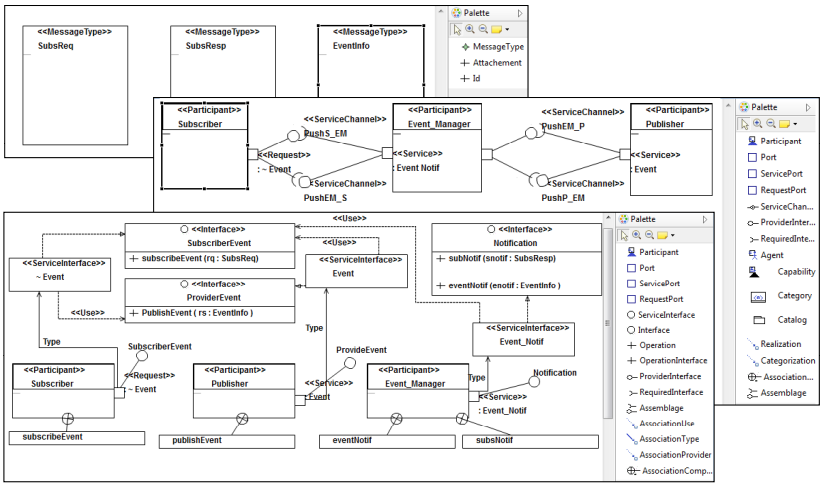


**Fig. 5.** SOA design patterns plug-in

The SOA design patterns diagram editor is a tool where diagrams can be created to model patterns. Figure 5 shows the diagram editor of the SOA design patterns with an illustration of the pattern example "Event-Driven Messaging". After modeling a design pattern, the plug-in generates an XML specification describing it. The generated XML specification corresponding to the participant diagram presented in Figure 3, is depicted in follows.

```
          <!-- =======Entities======= -->
<Participant ParticipantName="Subscriber">
    <Port>
      <RequestPort Name=": ~ Event"/>
    </Port>
</Participant>
...
          <!-- =======Connexions======= -->
<RequireInterface Origine="..." Destinataire="//@Assemblage.0"/>
<RequireInterface Origine=".../@Port.0" Destinataire="..."/>
...
```

The plug-in transforms the generated XML file, according to transformation rules expressed with the XSLT language, into Event-B specifications. These specifications can be imported under the Rodin platform to verify their correctness. Transformation rules described in section 3.2 are expressed with the XSLT language. For reasons of space, we only present the following XSLT fragment for transforming the architecture entities.

```
          <!-- =======CONTEXT======= -->
&lt;org.eventb.core.contextFile
org.eventb.core.configuration="org.eventb.core.fwd"version="3"&gt;
          <!-- ========SETS========= -->
&lt;org.eventb.core.carrierSet name=")"
org.eventb.core.identifier="Entity"/&gt;
<xsl:if test="document('MessageDiagram.xml')">
&lt;org.eventb.core.carrierSet
name="("org.eventb.core.identifier="MessageType"/&gt;
</xsl:if>
...
          <!-- ======CONSTANTS======-->
<xsl:if test="//Participant"> &lt;org.eventb.core.constant
name="00"org.eventb.core.identifier="Participant"/&gt;
</xsl:if>
...
          <!-- =======AXIOMS======= -->
<xsl:if test="//Participant"> &lt;org.eventb.core.axiom
name="002"org.eventb.core.label="Participant_partition"
org.eventb.core.predicate="partition(Participant,
<xsl:for-each select="Part:Root/Participant">
{<xsl:value-of select="@ParticipantName"/>}
</xsl:if>
</xsl:for-each>)
"/&gt;
</xsl:if>
 ...
```

By applying transformations rules on the generated XML specifications, we obtain Event-B specifications presented in Figure 6.

| CONTEXT | AXIOMS |
|---|---|
| EventDrivenM | Entity_partition: partition(Entity, Participant) |
| SETS | Participant_partition: partition(Participant, {Subscriber}, {Event_Manager}, {Publisher}) |
| Entity | Message_partition: partition(MessageType, {SubsReq}, {SubsResp}, {EventInfo}) |
| MessageType | ServiceChannel_Relation: ServiceChannel ∈ Entity ↔ Entity |
| CONSTANTS | ServiceChannel_partition: partition(ServiceChannel, {PushS_EM}, {PushEM_S}, {PushEM_P}, {PushP_EM}) |
| Participant | PushS_EM_Domain: dom({PushS_EM}) = {Subscriber} |
| ServiceChannel | ... |
| Subscriber | PushEM_S_Range: ran({PushEM_S}) = {Subscriber} |
| Event_Manager | ... |
| Publisher | Can_Send_Relation: Can_Send ∈ Entity ↔ MessageType |
| SubsReq | Can_Send_init: Can_Send = {Subscriber ↦ SubsReq, Publisher ↦ EventInfo, Event_Manager↦SubsResp, Event_Manager↦EventInfo} |
| EventInfo | END |
| PushS_EM | |
| Can_Send | |
| ... | |

**Fig. 6.** Excerpt of Event-B specification results

## 5 RELATED WORK

In the literature 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 Hohpe, 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].

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 specify design patterns using First Order Logic, but we use a different formal method which is Event-B.

After the OMG publication of the SoaML language, some works that provide SoaML support appeared. Delegado et al. [Delgado et al., 2011] developed an Eclipse plug-in based on EMF and GMF that implements the SoaML standard. Modeling with this plug-in is quite heavy and we can not model a provided/required connection with SoaML. Other tools that allow modeling service oriented architectures according to the OMG standard exists like Modeliosoft [Modeliosoft, 2011] and modelDriven [ModelDriven, 2009] however, these tools do not use transformation techniques for generating formal specifications.

In this context, we proposed a tool for modeling SOA design patterns, that is not only easy to use, specific for our diagrams, and adaptable with Rodin environment but also it allows importing and exporting XML files of model that will be subsequently converted to Event-B specifications. Moreover, we use the XSLT language for the automatic transformation of our model to Event-B language.

## 6 CONCLUSIONS

In this paper, we presented an architecture-centric approach supporting the modeling and the transformation of message-oriented SOA design patterns to formal specifications. The modeling phase allows to describe SOA design patterns with a graphical standard notation using the SoaML language. The transformation phase allows to formally specify structural features of these patterns at a high level of abstraction. We proposed an Eclipse plug-in that supports our approach. More precisely, it allows the modeling of SOA design patterns and then generating the corresponding XML file. Each XML file is transformed according to transformation rules expressed with the XSLT language into Event-B specifications. These specifications are then imported under the Rodin platform. We illustrated

our approach through a pattern example ("*Event_Driven Messaging*"). In this paper, we presented structural features of design patterns, behavioral features are presented in the modeling phase with sequence diagram which are then transformed to machines in the Event-B method. Currently, we are working on defining transformation rules in order to automate this phase.

## 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.

[Beck, 2007] Beck, K. (2007). *Implementation Patterns*. Addison Wesley; 1 edition (23 Oct 2007).

[Blazy et al., 2003] Blazy, S., Gervais, F., and Laleau, R. (2003). Reuse of specification patterns with the b method. In Bert, D., Bowen, J., King, S., and Waldn, M., editors, *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 626–626. Springer Berlin / Heidelberg.

[Delgado et al., 2011] Delgado, A., Laura, G., Sofia, L., Andrs, P., FranciscoRuiz, I., and Garcia, R. (2011). SoaML Eclipse plug-in para modelado de servicios. Technical report, Technical report.

[Eclipse, 2010a] Eclipse (2010a). Graphical Editing Framework. http://www.eclipse.org/gef/.

[Eclipse, 2010b] Eclipse (2010b). Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/.

[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.

[Gomaa, 2004] Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

[Gregor Hohpe, 2003] Gregor Hohpe, B. 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.

[ModelDriven, 2009] ModelDriven, C. (2009). ModelDriven. http://portal.modeldriven.org/.

[Modeliosoft, 2011] Modeliosoft (2011). Modelio: The open source modeling environement. http://modeliosoft.org/.

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

[Ramirez and Cheng, 2009] Ramirez, A. J. and Cheng, B. H. (2009). Developing and applying design patterns for dynamically adaptive systems. Technical Report MSU-CSE-09-8, Department of Computer Science, Michigan State University, East Lansing, Michigan.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[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.