Model Driven orchestration Composition

Master thesis

HAMMOUCHE Douae

Directed by: Prs MOULINE Salma, FREDJ Mounia and BLAY-FORNARINO Mireille

Master Responsable: Pr ABOUTAJDINE Driss

Mohammed V, Agdal University Rabat Morocco

I3S Laboratory, Sophia-Antipolis University
Nice France

July 16, 2008
Acknowledgement

I would like to express my deep and sincere gratitude to my french supervisor, Madame Mireille Blay-Fornarino, Member of Rainbow team, at I3S Laboratory, University of Sophia-Antipolis de NICE in France. Her wide knowledge and her logical way of thinking have been of great value for me. Her understanding, encouraging and personal guidance have provided a good basis for the present report.

I owe my most sincere gratitude to Professor Salma Mouline, Member of GSCM research team at Sciences Faculty of Mohammed V University of Rabat, and Professor Mounia Fredj from ENSIAS, engineering school at Mohammed V-Souissi University of Rabat, who gave me the opportunity to work with them during my third master semestre in Sciences Faculty, at the Mohammed V-Agdal University of Rabat, and gave me important guidance during my first steps into computer science studies. Their ideals and concepts have had a remarkable influence on my training in the field of MDE research.

My sincere thanks are due to Professor Michel Riveill, Head of the "Computer Science" department at the Engineering School of Technology of the University of Nice - France, who welcomed me in I3S Laboratory and helped me do this work.

I warmly thank Professor Driss Aboutajdine, head GSCM research team of Sciences faculty, Mohammed V-Agdal University of Rabat, for his kind support and guidance that have been of great value in my study.

My warm thanks to Franck Chauvel, PHD student at I3S Laboratory and member of Faros project, who gave me a great support when I was working on Kermeta.

During this work I have collaborated with many colleagues for whom I have great regard, and I wish to extend my warmest thanks to all those who have helped me with my work in the Rainbow team at I3S Laboratory of Sophia-Antipolis University in France.

I owe my loving thanks to my Parents Amar et Souad, my fiance Tarik. They have been a great support to my studies abroad.

The financial support of the scholarship given by ERASMUS MUNDUS EXTERNATL WINDOW, IMAGEEN project is gratefully acknowledged.

Nice, France, Juillet 2008
Douae Hammouche


## Contents

### 1 Introduction
- 1.1 Context ............................................. 7
- 1.2 General theme .................................. 7
  - 1.2.1 FAROS project ................................. 8
- 1.3 WS-BPEL ........................................... 9
  - 1.3.1 XML and BPEL .................................. 9
  - 1.3.2 Binding ........................................ 9
- 1.4 ADORE .............................................. 9
- 1.5 Objectif of My work .............................. 10

### 2 Merging two BPEL orchestrations: The problem
- 2.1 Behavior in BPEL ................................. 11
- 2.2 The problem ...................................... 11

### 3 Meta Models to transform
- 3.1 ADORE Meta Model ............................... 14
  - 3.1.1 ADORE formalism .............................. 14
  - 3.1.2 Adore behavior Meta Model ................. 15
- 3.2 BPEL Meta Model ................................ 15

### 4 BPEL Meta Model to ADORE Meta Model transformation
- 4.1 Kermeta language ................................. 18
- 4.2 Navigating the Meta Models .................... 18
  - 4.2.1 Visitor Pattern ................................ 18
- 4.3 Building transformation ......................... 19
  - 4.3.1 Building phase ................................. 19
  - 4.3.2 Linking phase ................................ 26
  - 4.3.3 Starting the transformation ................. 27
- 4.4 Validating phase .................................. 28

### 5 Perspectives and Conclusion
- 5.1 Perspectives ....................................... 29
- 5.2 Conclusion ........................................ 30
6 Annexe

6.1 Planning ................................................................. 31
6.2 Overview of WS-BPEL .................................................. 31
   6.2.1 Structure of a process ............................................. 31
   6.2.2 BPEL activities ..................................................... 33
6.3 Transformation kermeta files ......................................... 34
   6.3.1 BPEL Visitor ......................................................... 34
   6.3.2 The Builder ......................................................... 36
   6.3.3 Aspect adding ....................................................... 39
   6.3.4 Transformation main .............................................. 40
## List of Figures

1.1 Project platform .................................................. 8
1.2 BPEL main concepts [Mig05] .................................... 9

2.1 orchestration Best Registration based on price comparison .......... 12
2.2 orchestration Best Registration based on the required date .......... 12
2.3 orchestration Best Registration based on all ........................ 13

3.1 BPEL and ADORE MM transformation ............................... 14
3.2 Adore behavior Meta Model ....................................... 16
3.3 BPEL Meta Model ................................................. 17

4.1 Kermeta positionning [DFF+08] .................................... 19
4.2 White box reusing in our Visitor Pattern ........................... 20
4.3 Simple mapping of final classes ................................... 21
4.4 simpleprocess.xmi .................................................. 22
4.5 simpleprocess.xmi transformed into ADORE ....................... 23
4.6 Example of sequence composed of flow ............................ 24
4.7 xmi file of process5 ............................................... 25
4.8 Generated file of process5 ........................................ 26
4.9 Declaring aspect in a class ....................................... 27
4.10 Linking activities in a flow with OrderRelation classes .......... 27

6.1 Project planification ................................................ 31
Chapter 1

INTRODUCTION

1.1 Context

The project I have been working on for my graduation in Master 2 for a period of six months is based on Faros research project with fourteen permanent researchers team of I3S Laboratory at Nice Sophia-Antipolis University. The purpose of my work is about Meta Models Transformation. More precisely, I contribute to Transformation to Platform used in Faros Project which is based on Meta Models Transformation concepts. Since platforms I am working with are based on SOA (Adore and BPEL), my work handles web services orchestration matters especially orchestrations composition. That is why we used Model driven architecture.

1.2 General theme

Services Oriented Architectures (SOA) [MLM+06] use the concept of service as an elementary brick to assemble complex systems. Services are loosely coupled by definition, and complex services are build upon basics ones using composition mechanisms. The loose coupling methodology enables the separation of concerns and helps systems evolution.

Using these elementary services for web services and orchestrations [Pel03] as a composition mechanism, web service Oriented Architectures (WSOA) provides a way to implement these loosely coupled architectures. W3C defines orchestrations as "the pattern of interactions that a web service agent must follow in order to achieve its goal" [W3C04]. Specialized code is written inside web services while business processes are described as an orchestration of those web services. Code manipulations, such as refactoring operation, help software evolution support. In [MWD+05], authors identify some challenges for future research on software evolution and focus on the abstraction need. Lehman identifies as his first "Law of Software Evolution" [Leh96] that "A program that is used must be continually adapted else it becomes progressively less satisfactory". As Faros project ensures this vision, it is expected from this work to provide a solution to the orchestrations composition in BPEL that allows evolution.

The platform to compose orchestrations is being developed by the Rainbow team (see figure 1.1)
Model driven orchestration composition

[MBFR08]. Using this platform we have to transform BPEL orchestrations into ADORE Model in order to compose them.

![Figure 1.1: Project platform.](image)

1.2.1 FAROS project

In [LMBBF+06], the objective of FAROS project is to define a composition environment for a reliable building of services oriented architecture. It aims at working on application integration using contract elements in order to provide a coherent services composition and also to define methodology that makes contracts integration process producible from business Models to their projection into execution platforms.

The idea of this project began from converging points of view among different partners that the actual services integration solutions don’t provide way of expression neither to verify specifications which are the guarantee of a sufficient level of trust in assembling different elements of an architecture. Moreover, these solutions don’t follow a guiding methods and don’t account for the strong constraints related to extra-functional environment properties.

The project proposes to express applications constraints by contracts. It defines contract as a property expression that entities have to respect when collaborating together independently of their implementation. During my training, I worked and participated in meetings concerned by this part and around the deliverable 2.3 "Meta Models of plateforms" that will be published soon.

*University Mohammed V, Agdal Sciences Faculty*

Douae HAMMOUCHE
1.3 WS-BPEL

1.3.1 XML and BPEL

web services Business Process Execution Language (WS-BPEL) is an XML based programming language used to describe high level business processes. A 'business process' is a term used to describe the interaction between two businesses or two elements in some business [Mig05]. An example of this might be company A purchasing something from company B. BPEL allows this interaction to be described easily and thoroughly such that company B can provide a web service and company A can use it with a minimum of compatibility issues. Also, a web service is typically described using web service Description Language (WSDL). This is another XML based language which allows one to describe the interface to the web service.

1.3.2 Binding

A common binding is SOAP/HTTP which uses the XML Simple Object Access Protocol (SOAP) over HTTP (standard web page fetching language) to talk to web services which are on the internet. Another common binding is the Java binding. This binding allows one to define local in-process java implementations of web services. So if wanted to write a web service that allows clients to print things (e.g. 'Hello World') one could write it in Java and then expose it as a web service [Mig05]. Next figure 1.2 illustrates BPEL main concepts:

![Figure 1.2: BPEL main concepts](Mig05).

1.4 ADORE

To perform high level composition and to allow reasoning on orchestrations and evolutions, we define a Model called Adore: "Activity Model supporting orchestration Evolution". We will develop ADORE description in farther paragraph.

University Mohammed V, Agdal Sciences Faculty
Douae HAMMOUCHE
1.5 Objectif of My work

In the context of Faros project, the expression of process orchestration is carried out using BPEL [AAA+07] since it is a standard of process oriented implementation of web services. Here we focus on BPEL activities processing order as defined in OASIS Standard.

The aim of this work is threefold. First, we identify the problem of merging two orchestrations in BPEL. Second, we obtain our own BPEL Meta Model to simplify the transformation process. The target of this Meta Model is an Adore Meta Model which already exists. Third, having both source and target Meta Models we proceed with the transformation using the Visitor Pattern [GJHM94] and Kermeta language [BCF+05] within ECLIPSE.

we will focus on a transformation between Meta Models and not Models in order to make this transformation general to all instanciations of these Meta Models. We would indeed use the Visitor Patten to cover the Meta Models as trees and entities as leaves.

Having described the context, reviewed concepts that will be involved and set the objectives to attain for this work the main methodology steps that will be followed for this work are as follow:

1. Obtain both source and target Meta Models.
2. Write the transformation using the Visitor Pattern.
3. Implement the transformation with kermeta.
Chapter 2

Merging two BPEL orchestrations: The problem

2.1 Behavior in BPEL

WS-BPEL, which stands for web services Business Process Execution Language, is a language for specifying business process behavior based on web services. Processes in WS-BPEL export and import functionality by using web service interfaces exclusively [AAA+07]. It is integrated within Eclipse and provides a graphical interface to handle services orchestration. In this work, we will focus on this part of BPEL.

2.2 The problem

As figures 2.1, 2.2, and 2.3 show, here we have the problem of merging two different BPEL orchestrations. First, 2.1, we have to orchestrate between two services: booking a hotel and booking a flight in order to build the cheapest trip.

Second, we carry out an orchestration (2.2) that allows to reuse an existing web service (bookFlightHotel) that provides different trips in different dates. The result of this orchestration is the selection of the trip with the best date.

As we see, BPEL provides complete tools to describe a business process and especially ordering services. But having these two orchestrations done, can we know which booking gives the client the best price and date? (Booking hotel and flight separately or booking their combination offer). The answer will be an orchestration (see figure 2.3) which is the composition of both the orchestrations of 2.1 and 2.2.

Therefore, composing these different orchestrations in BPEL would not be automatic. Indeed, if it is a big orchestrations composition it may be hardly done. Thus, we chose to use the ADORE platform that allows us to addresses the problem of the merging of different orchestrations [MBFR08].
Figure 2.1: orchestration Best Registration based on price comparison.

Figure 2.2: orchestration Best Registration based on the required date.
Figure 2.3: orchestration Best Registration based on all.
Chapter 3

Meta Models to transform

Taking advantage of ADORE to compose Activities, we aim to generate automatically BPEL orchestrations. In the opposite, to reuse existing orchestrations we need to transform BPEL orchestration in ADORE. The first step of this work is to set up the Meta Models needed in the transformation, see figure 3.1. At the end, we want to reach a transformation Model that maps the BPEL Meta Model to the ADORE Meta Model so that we can propose automatic orchestrations composition to the client. First let us introduce both of these Meta Models.

Figure 3.1: BPEL and ADORE MM transformation.

3.1 ADORE Meta Model

3.1.1 ADORE formalism

This section describes formally this Model based on [Mos07].

orchestration: An orchestration is a tuple $(A^*, <^*)$ that defines a behavior. $A^*$ is a set of Activities $a_1, ..., a_n$ and $<^*$ a partial ordering among these activities.
Activity: An activity is a tuple \((uid,K,V^*,V_{\text{out}},G^*)\). Each activity is unique inside an orchestration and identified by \(uid\). \(K\) refers to the Kind of this activities. \(V^*_\text{in}\) (resp. \(V_{\text{out}}\)) represent inputs (resp. output) Variables. \(G^*\) represents conditional guards and allows conditional expressions (if/then/else). In order to respect the schedule of this work we didn’t include the guards in the analysis and we worked only with behavior Meta Model of Adore.

Partial ordering \((<, \text{precedesence rule})\): Activities are ordered using an operator \(<\). The expression \(a_1 < a_2\) is called a precedence rule and means that \(a_2\) must wait the end of \(a_1\) before starting its own executing.

Kind: We use in ADORE behavior Meta Model a subset of ADORE specifications. We consider the following kind of allowed activities:

* variable assignment (Assign)
* service invocation (Invocation)
* message reception (Receive)
* response sending (Reply)
* fault report (Throw)

Evolution: An Evolution can be considered as a piece of orchestration which can be plugged into existing orchestrations. Evolution is therefore as a superset of orchestrations.

We use then ADORE formalism which enables the automatic integration of n evolutions into an orchestration automatically.

3.1.2 Adore behavior Meta Model

Adore behavior Meta Model is derived from concepts defined in [Mos07], see figure 3.2. Because it is still in development, we will use for this work the version of ADORE Meta Model made available for us at the beginning of this project. In this Meta Model we find the needed container entity named behavior that possesses activities and order entities.

3.2 BPEL Meta Model

Since BPEL is a large and complex language and in order to define a simple Bpel Meta Model, we used the concepts we need for the targeted transformation of this work, such as the behavioral concepts from OASIS STANDARD of WS-BPEL. OASIS STANDARD introduce the behavioral notion as follow: "Executable business processes Model actual behavior of a participant in a business interaction" [AAA+07]. A basic structure of a process is given in the Annexe. This structure defines the components of a process.

Since BPEL description has many concepts and in order to simplify carrying out a BPEL Meta Model we used only some of the principal concepts which are involved in an orchestration behavior. These concepts are:

* Process
* Partner Link
Figure 3.2: Adore behavior Meta Model.
Based on these concepts we obtained an example of BPEL Process Meta Model as follows in figure 3.3:

Figure 3.3: BPEL Meta Model.

This Meta Model will be used as the source of the transformation from BPEL Meta Model to ADORE Meta Model.
Chapter 4

BPEL Meta Model to ADORE Meta Model transformation

Now that we have both source and target Meta Models we can map them. To implement the transformation between these two Meta Models we need a language which permit their loading in order to scan each source entity and bind it with the associated target. For the Faros project the Kermeta language [BCF+05] has been chosen, so we will use this language for the transformation.

4.1 Kermeta language

Kermeta is a Triskell project of INRIA. It is a Meta Modeling language that defines both the structure and the behavior of Meta Models. Kermeta has been designed to be fully compliant with the OMG Meta Modeling language EMOF (part of the MOF 2.0 specification) and provides an action language for specifying the behavior of Models. It is developed with an open source license EPL (Eclipse Public License). Figure 4.1 shows kermeta positioning.

Kermeta is intended to be used as the core language of a Model oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language. It intends so to provide useful Models manipulation tools.

4.2 Navigating the Meta Models

Before mapping both Meta Models, we need to know each element and its associates (each element can have many targets). That is why we navigate in each of Meta Model to determine the different transformations needed. For example, a process, which is the container in BPEL Meta Model, will be mapped to ADORE Meta Model container Behavior. And so on we associate to each entity source its entity (or entities) target, and make the transformation.

4.2.1 Visitor Pattern

To scan a Meta Model we programmed a Visitor Pattern, one of the Design Patterns defined in [GJHM94]. As Christopher Alexandre said: "Each pattern describes a problem which occurs over
and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice, we can clearly find out that using a Visitor Pattern we will be able to provide the genericity we want for our transformation. There are two ways of reuse: white box reuse and black box reuse. In our case we used white box reuse since we reuse by subclassing, which means using inheritance instead of interface. We can see this in figure 4.2.

4.3 Building transformation

Building a transformation consists of determining generated elements for each source elements. For that, Kermeta permits loading the Meta Model and adding new properties that didn’t exist first in this Meta Model, like the property generated that we need only at the transformation process. Also, we need the Visitor Pattern to cover all elements in our Meta Model. After that we can map each found element to its generated element. We have then to link the generated elements among them to obtain the targeted Model. Finally, having a Visitor, a builder and a linker we can launch the transformation process.

4.3.1 Building phase

In the beginning, we start working on final entities in our BPEL Meta Model, like all Activity subclasses. We can map these entities simply since their targets are final entities in ADORE behavior Meta Model too. As for the invoke class, in BPEL, its generated element is Invocation class in ADORE. Same thing to all other subclasses of Activity except sequence and flow classes.
Figure 4.2: White box reusing in our Visitor Pattern.
which are complex types. The segment code below in figure 4.3 shows how we generate one class in ADORE:

```java
method visitInvoke(target : invoke) : Void is
  do
    var resultElement : behaviour:Invocation init behavioural:Invocation:new
    resultElement.uid := String.clone(target.name)
    target.abstractGenerated := resultElement
    resultBehaviour.activity.add(resultElement)
  end

method visitReceive(target : receive) : Void is
  do
    var resultElement : behaviour:Receive init behavioural:Receive:new
    resultElement.uid := String.clone(target.name)
    target.abstractGenerated := resultElement
    resultBehaviour.activity.add(resultElement)
  end

method visitReply(target : reply) : Void is
  do
    var resultElement : behaviour:Reply init behavioural:Reply:new
```

Figure 4.3: Simple mapping of final classes.

To illustrate transforming this simple classes, we give in figure 4.4 a very simple example of orchestration simpleprocess:
The transformed simpleprocess is shown in figure 4.5.

After determining generated classes for entities above, we go on to other more complex entities. For example, the class sequence. This class is defined in BPEL such as a collection of activities [AAA+07]. Here we show how Kermeta is useful since it provides all tools we need such as Collectionspackages. (See Annexe for more details).

The difficulty with sequence activity was when building transformation we needed to conserve order between all its activities while building the generated class for each of these sequence activities. More importantly, since a sequence can also include a flow we have to conserve in the same time the order sequence of activities and the parallelism of the flow activities. This was the major difficulty in the transformation because there isn’t a generated class for sequence neither for flow.

In the following example we have an orchestration named process5 composed of a sequence that has simple activities and flow, see figure 4.6:
Figure 4.5: simpleprocess.xmi transformed into ADORE.
We instantiate our BPEL Meta Model to have this orchestration as .xmi file, see figure 4.7. After applying the transformation on this orchestration Model we finally have the ADORE generated orchestration. As we see in figure 4.8 it has more elements than the source orchestration:
Figure 4.7: xmi file of process5.
We had more elements in the generated file than the source file because, for every order in the sequence and every order in the flow, expressed by a link, we generate an orderRelation class in ADORE. This difference is explained by the fact the order in a sequence in BPEL is defined as an ordered list and as a relationship between two activities.

4.3.2 Linking phase

Now that we have all our element source and their generated elements we have to link elements in the generated Model, in order to be conform to ADORE Meta Model. Here we need the ability of kermeta to add new property, defined by the word aspect that we add when declaring a class, see figure 4.9:
Having the generated elements, we have to link them to have a Model conform to ADORE Meta Model. Such as \textit{behavior} in Adore Meta Model is associated to \textit{OrderRelation} by the reference \textit{order}, that is linked to \textit{Activity} with the reference \textit{before} and \textit{after}. For an illustration see figure 4.10:

4.3.3 Starting the transformation

After having visited all source elements in BPEL Model, built their generated elements and linked them we can now begin the transformation. For this, we create a transformation file that call all the files before by specifying the source and target files. Finally, by executing our kermeta file we reach the Adore Model, wich is the target. See the Annexe for the transformation files.
4.4 Validating phase

Although we can build our transformation, to validate it is an other matter. At first we are sure that the resulting Model conforms to ADORE Meta Model. And this since Eclipse environment supports such validation. However to check the equivalence of two Models, the generated and the real ADORE Models, at present time we need to do the validation manually. But as we worked on examples, it isn’t easy in a general case. And since ADORE is still in development we can not perform yet a formal validation of the transformation. And as the ADORE Meta Model has changed from the beginning of my work, we can’t for the moment use tools such as visualisation.
Chapter 5

Perspectives and Conclusion

5.1 Perspectives

BPEL Meta Model transformation into ADORE Meta Model is an important part of the Model driven composition orchestration. Having it done, we have completed the main part of making orchestrations composition automatic. And while working on our transformation, between different technologies Meta Model, we thought about what would be next. Consequently, more work has to be done to make the composition process automatic now that we have the main transformation achieved.

Possible transformations may be as follows:

* From Adore to BPEL ($T^{-1}$ of our work).
* From .bpel to .xmi files.
* From .xmi to .bpel files.

The first transformation will be clearly done after having our generated orchestrations composed in one orchestration in ADORE. This step would also be an elegant solution to validate the transformation we proposed, as soon as we will be able to check if two orchestrations are equivalent. In case we would like to transform a sequence in ADORE, we will need to know if it will be generated in BPEL as a sequence or as a flow that has activities linked as a sequence. However, it will permit coming back to our source language BPEL, but at this stage, it will provide an .xmi file that is conforme to our BPEL Meta Model.

That is why, the second transformation will then be needed to provide our final .bpel file and make all of the orchestration composition process an automatic one. Finally, the third transformation is clearly the first one that gives the orchestrations we want to combine in one, they are the source for our transformation.

Some ideas have been put forward to handle these next transformations will be handled. Of course, there are many ways to answer these questions. Some of these answers are:

First, the reverse transformation, from ADORE to BPEL, will be inspired from the transformation
discussed in this work, since they both use the same Meta Models. Nevertheless, it doesn’t mean that it will be the same or easier, because we don’t know if it is an bijective transformation. Second, for second and third transformation (BPEL to XMI, XMI to BPEL), we thought of using an XML parser since both of these languages are based on XML files. For that, there are many tools that permit a XML to XML transformation.

5.2 Conclusion

My work in this training can be summerized into two main parts. The first one consisted of discovering BPEL language and working with its Meta Model for which an example of BPEL Meta Model has been developed. While the second part was about building the transformation in the context of FAROS project using MDA concepts and Meta Model transformations.

By finding out how to realise a Meta Model of BPEL, I had the opportunity to discover and work with this language. With this part of my work, I had the opportunity to develop the required knowledge of all BPEL concepts and web services technologies related to it. And these are from programming services with JAVA to deploying them on a web page, by using all sorts of technologies like: JETTY, AXIS 2 and ODE.

Moreover, while realising the Meta Model I worked with the MDE side. And this is when I used EMF to write my Meta Model since it is a graphic tool like Modelising with UML. Finally, by making a small BPEL Meta Model, I provide to every one interested in transformation from BPEL language to any other language a Meta Model source. Also, will be an sample for Faros as a Meta Model for future platform transformations from BPEL.

The Meta Model transformation is a major concern for the FAROS project. Since this transformation provides a bridge between different platforms and technologies, the rainbow that I worked during this project is undoubtedly enlarging possibilities of using multiple platforms and reusing them. Also, providing transformation from BPEL to ADORE is to give the first and important step for orchestrations composition to become automatic. And it can be an opening to ADORE developers or users to use anything from BPEL in order to reuse its concepts or ideas.

To conclude this work, I must say that this training allowed me to improve my knowledge with different new concepts and technologies. It also allowed me to enhanced my computer science competencies by working with a highly skilled team of computer science professionnals and by participating in many discussions and meetings of the Faros project.
Chapter 6

Annexe

6.1 Planning

Before concluding, next figure 6.1 represents a Gantt diagram of the project planning:

![Gantt Diagram]

Figure 6.1: Project planification

6.2 Overview of WS-BPEL

6.2.1 Structure of a process

OASIS STANDARD provides this basic structure of a process [AAA+07]:

---

31
<process name="NCName" targetNamespace="anyURI"
queryLanguage="anyURI"?
expressionLanguage="anyURI"?
suppressJoinFailure="yes|no"?
exitOnStandardFault="yes|no"?
xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
<extensions>?
  <extension namespace="anyURI" mustUnderstand="yes|no" />+
</extensions>
<import namespace="anyURI"?
location="anyURI"?
importType="anyURI" />*
<partnerLinks>?
  <!-- Note: At least one role must be specified. -->
  <partnerLink name="NCName"
    partnerLinkType="QName"
    myRole="NCName"?
    partnerRole="NCName"?
    initializePartnerRole="yes|no"?>+
  </partnerLink>
</partnerLinks>
<messageExchanges>?
  <messageExchange name="NCName" />+
</messageExchanges>
<variables>?
  <variable name="BPELVariableName"
    messageType="QName"?
    type="QName"?
    element="QName"?><
    from-spec?
  </variable>
</variables>
<correlationSets>?
  <correlationSet name="NCName" properties="QName-list" />+
</correlationSets>
<faultHandlers>?
  <!-- Note: There must be at least one faultHandler -->
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    ( faultMessageType="QName" | faultElement="QName" )? >*
  activity
  </catch>
  <catchAll>? 
  activity
</catchAll>
</faultHandlers>
<eventHandlers>?
   <!-- Note: There must be at least one onEvent or onAlarm. -->
   <onEvent partnerLink="NCName"
      portType="QName"?
      operation="NCName"
      ( messageType="QName" | element="QName" )?
      variable="BPELVariableName"?
      messageExchange="NCName"?>*
   <correlations>?
       <correlation set="NCName" initiate="yes|join|no"?>++
   </correlations>
   <fromParts>?
       <fromPart part="NCName" toVariable="BPELVariableName"?>++
   </fromParts>
   <scope ...>...</scope>
</onEvent>
<onAlarm>*
   <!-- Note: There must be at least one expression. -->
   ( 
      <for expressionLanguage="anyURI"?>duration-expr</for> |
      <until expressionLanguage="anyURI"?>deadline-expr</until> )?
   <repeatEvery expressionLanguage="anyURI"?>
       duration-expr
   </repeatEvery>?
   <scope ...>...</scope>
</onAlarm>
</eventHandlers>
activity
</process>

6.2.2 BPEL activities

A WS-BPEL activity can be any of the following:

. <receive>
. <reply>
. <invoke>
. <assign>
. <throw>
. <exit>
. <wait>
6.3 Transformation kermeta files

6.3.1 BPEL Visitor

```java
package ProcessModel2;

require kermeta
require "platform:/resource/MBPELBis/ProcessModel2.ecore"
using kermeta::standard

abstract class BPELVisitor
{
  operation visitProcess (p: Process) : Void is abstract
  operation visitInvoke (i: invoke) : Void is abstract
  operation visitReceive (r: receive) : Void is abstract
  operation visitFlow (f: flow) : Void is abstract
  operation visitSequence (s: sequence) : Void is abstract
  operation visitReply (r: reply) : Void is abstract
  operation visitLink (l: link) : Void is abstract
}

abstract class VisitableElement
{
```

University Mohammed V, Agdal Sciences Faculty
Douae HAMMOUCHE
 operation accept(visitor : BPELVisitor) : Void is abstract

aspect abstract class Activity inherits VisitableElement{}

aspect class Process inherits VisitableElement
{
    method accept(visitor : BPELVisitor) : Void is
    do
        visitor.visitProcess(self)
    end
}

aspect class invoke inherits VisitableElement
{
    method accept(visitor : BPELVisitor) : Void is
    do
        visitor.visitInvoke(self)
    end
}

aspect class receive inherits VisitableElement
{
    method accept(visitor : BPELVisitor) : Void is
    do
        visitor.visitReceive(self)
    end
}

aspect class flow inherits VisitableElement
{
    method accept(visitor : BPELVisitor) : Void is
    do
        visitor.visitFlow(self)
    end
}

aspect class sequence inherits VisitableElement
{
    method accept(visitor : BPELVisitor) : Void is
    do
        visitor.visitSequence(self)
    end
}

aspect class receive inherits VisitableElement
{
6.3.2 The Builder

```kotlin
package ProcessModel2;  
require "platform:/resource/MBPELBis/Simulation/Adorelinkage.kmt"  
require "platform:/resource/MBPELBis/Simulation/BPELVisitor.kmt"  
require kermeta  
using kermeta::standard  

class ADOREBuilder inherits BPELVisitor  
{  
  reference resultBehaviour: behaviour::Behaviour  
  operation getResultBehaviour() : behaviour::Behaviour is  
    do  
    result := resultBehaviour  
    end  

  method visitProcess(target : Process) : Void is  
    do  
    var resultElement : behaviour::Behaviour init behaviour::Behaviour.new  
    resultElement.name := String.clone(target.name)  
    target.generated := resultElement  
}  
```
resultBehaviour := resultElement

target.activity.accept(self)
end

method visitSequence(target : sequence) : Void is
do
  var s : Integer init 0
  s:=target.inorder.size()
  target.inorder.each{s | s.accept(self)}
//stdio.writeln(target.inorder.at(0).initialElements().at(0).name)
  from var i : Integer init 0
  until i == s-1
  loop
    if target.inorder.at(i).finalElements().size == 1
    then
      var t: Integer init 0
      t:=target.inorder.at(i+1).initialElements.size()
      from var j: Integer init 0
      until j == t
      loop
        var resultOrder : behaviour::OrderRelation init behaviour::OrderRelation .new
        resultBehaviour.order.add(resultOrder)
        resultOrder.before:=target.inorder.at(i).finalElements().at(0).abstractGenerated
        resultOrder.after:=target.inorder.at(i+1).initialElements().at(j).abstractGenerated
        j:=j+1
      end
    else //target.inorder.at(i+1).initialElements().size == 1
    var t: Integer init 0
    t:=target.inorder.at(i).finalElements.size()
    from var j: Integer init 0
    until j == t
    loop
      var resultOrder : behaviour::OrderRelation init behaviour::OrderRelation.new
      resultBehaviour.order.add(resultOrder)
      resultOrder.before:=target.inorder.at(i).finalElements().at(j).abstractGenerated
resultOrder.after:=target.inorder.at(i+1).initialElements().at(0).
abstractGenerated
j:=j+1
end
i:=i+1
end

method visitFlow(target : flow) : Void is
do
target.inparallel.each{s|s.accept(self)}
target.links.each{s|s.accept(self)}
end

method visitInvoke(target : invoke) : Void is
do
var resultElement : behaviour::Invocation init behaviour::Invocation.new
resultElement.uid := String.clone(target.name)
target.abstractGenerated := resultElement
resultBehaviour.activity.add(resultElement)
end

method visitReceive(target : receive) : Void is
do
var resultElement : behaviour::Receive init behaviour::Receive.new
resultElement.uid := String.clone(target.name)
target.abstractGenerated := resultElement
resultBehaviour.activity.add(resultElement)
end

method visitReply(target : reply) : Void is
do
var resultElement : behaviour::Reply init behaviour::Reply.new
resultElement.uid := String.clone(target.name)
target.abstractGenerated := resultElement
resultBehaviour.activity.add(resultElement)
end

method visitLink(target : link) : Void is
do
var resultElement : behaviour::OrderRelation init behaviour::OrderRelation.new
target.generated := resultElement
resultBehaviour.order.add(resultElement)
target.generated.before := target.source.abstractGenerated
target.generated.after := target.istarget.abstractGenerated
end
6.3.3 Aspect adding

```java
package ProcessModel2;

require "platform:/resource/MBPELBis/Simulation/BPELVisitor.kmt"
require "platform:/resource/MBPELBis/MADORE/behaviour.ecore"
require kermeta

using kermeta::standard

aspect class Process
{
    reference generated : behaviour::Behaviour
}

aspect class Activity
{
    reference abstractGenerated : behaviour::Activity

    operation initialElements() : Sequence<Activity> is
    do
        result := Sequence<Activity>.new
        result.add(self)
    end

    operation finalElements() : Sequence<Activity> is
    do
        result := Sequence<Activity>.new
        result.add(self)
    end
}

aspect class invoke
{
}

aspect class receive
{
}

aspect class flow
{
    method initialElements() : Sequence<Activity> is
    do
        result := self.inparallel.select{ a | not self.links.exists( l | l.istarget.equals( a))}
    end
```
method finalElements() : Sequence<Activity> is
do
  result := self.inparallel.select{ a | not self.links.exists{ l | l.source.equals(a)}}
end
}
aspect class sequence
{
  method initialElements() : Sequence<Activity> is
do
  result := Sequence<Activity>.new
  result.add(self.inorder.first)
end

  method finalElements() : Sequence<Activity> is
do
  result := Sequence<Activity>.new
  result.add(self.inorder.last)
end
}
aspect class reply
{

}
aspect class link
{
  reference generated : behaviour::OrderRelation
}

6.3.4 Transformation main

@mainClass "ProcessModel2::Bpel2Adore"
@mainOperation "main"
package ProcessModel2;

require kermeta

require "platform:/resource/MBPELBis/Simulation/AdoreBuilder.kmt"
require "platform:/resource/MBPELBis/Simulation/Adorelinker.kmt"

using kermeta::standard
using kermeta::persistence
class Bpel2Adore
{
    reference process: Process
    reference behaviour1: behaviour::Behaviour

    operation transform(processModel : String, behaviour1Model : String) : Void is
        do
            loadProcessModel(processModel)
            stdio.writeln(">>> Building the behaviour Model ... ")
            var builder : ADOREBuilder init ADOREBuilder.new
            process.accept(builder)
            behaviour1 := builder.getResultBehaviour()
            stdio.writeln("Done.")
        end

        stdio.writeln(">>> Linking the behaviour Model ... ")
        var linker : Adorelinker init Adorelinker.new
        process.accept(linker)
        stdio.writeln("Done.")

        savebehaviourModel(behaviour1Model)
    end

    // Helpers
    operation loadProcessModel(processModel : String) : Void is
        do
            stdio.writeln(">>> Loading Process Model " + processModel)
            var repository : EMFRepository init EMFRepository.new
            var resource : EMFResource
            resource ?= repository.getResource(processModel)
            resource.load
            process ?= resource.instances.one
        end

    operation savebehaviourModel(behaviour1Model : String) : Void is
        do
            stdio.writeln(">>> Saving behaviour model ... ")
            var resourceToSave : Resource
            var repository : EMFRepository init EMFRepository.new
            resourceToSave := repository.createResource(behaviour1Model, "platform:/resource/MBPELBin/MADORE/behaviour.ecore")
            resourceToSave.add(behaviour1)
            resourceToSave.save
            stdio.writeln("Done.")
        end

    // Launcher
operation main() : Void is

do

  //var process : String init "platform:/resource/MBPELBis/Test2Process.xmi"
  //var behaviour1 : String init "platform:/resource/MBPELBis/MADORE/TestProcess_Adore.xmi"

  //var process : String init "platform:/resource/MBPELBis/Test1Process.xmi"
  //var behaviour1 : String init "platform:/resource/MBPELBis/MADORE/Test1Process_Adore.xmi"

  //var process : String init "platform:/resource/MBPELBis/Process3.xmi"
  //var behaviour1 : String init "platform:/resource/MBPELBis/MADORE/Process3_Adore.xmi"

  //var process : String init "platform:/resource/MBPELBis/Process4.xmi"
  //var behaviour1 : String init "platform:/resource/MBPELBis/MADORE/Process4_Adore.xmi"

  //var process : String init "platform:/resource/MBPELBis/Process5.xmi"
  //var behaviour1 : String init "platform:/resource/MBPELBis/MADORE/Process5_Adore.xmi"

  transform(process, behaviour1)

end
Bibliography


