Model Centric Approach of Web Services Composition

Ricardo Quintero, Victoria Torres, Vicente Pelechado

Department of Information Systems and Computation
Technical University of Valencia
Cami de Vera s/n E-46022, Spain
{iscrquinter,vtorres,pele}@dsic.upv.es

Abstract. The development of composite Web Services is being specified in a more declarative way than imperative programming. In this context, conceptual modeling has been the most accepted solution. Conceptual modeling of Web services has been done using behavioral models (like activity diagrams) considering mainly the dynamic view. We believe that, besides the dynamic aspects, the models should capture structural requirements between web service operations. In this way, behavioral models could be complemented with a structural model. In this paper we introduce a Web service composition modeling solution, following the MDA approach, considering both –structural and dynamic properties- enriched with semantic constraints in order to automatically generate composite Web services implemented in BPEL.

1 Introduction

Current e-business processes have, as an important requirement, the integration (the composition) of diverse application functionalities. The main strategy that has been followed by the industry is the use of Web Services to export the functionality and the use of programming languages to define service composition [11]. Because the majority of them were not designed with this goal in mind, they do not have abstractions for this objective, so usually the composition definitions are cumbersome.

In contrast, conceptual modeling offers abstractions and models in order to define this composition at a high level of abstraction [12,13,14]. The main focus of these approaches is on dynamic concerns (as in UML Activity diagrams) forgetting the structural concerns.

Although there are some model-driven solutions that generate in a semi-automatic way Web services and WS-BPEL [15], the problem with these modeling approaches is some lack of semantics that makes difficult to capture the composition requirements in a precise way. This drawback does make unfeasible to build modeling tools that validate models and generate complete and fully operative implementations.

We consider that structural and dynamic models are needed in order to capture these issues, especially static and dynamic binding properties between the Web services that are being composed (the main focus of this work). Moreover it could be used as a way to export the functionality of the application: by means of methodological guidelines it is possible to detect functional groups from the business layer (specified by a

structural model) and export them as a set of Web services. These Web services could be consumed by other applications to enable collaboration with other third parties.

In this work we introduce, as a main contribution, two models (the Service Model and the Dynamic Model for Service Composition) which allow us specifying the structural and dynamic requirements of Web services compositions by using aggregation/association relationships with a precise semantics, defined in the context of a multidimensional framework [3]. In order to obtain the equivalent software artifacts of these models we follow a Model driven approach where the application of a set of transformation rules generates the corresponding WS-BPEL specification. This solution extends our Web engineering method Object-Oriented Web Solutions (OOWS) [1] in order to capture the collaborative requirements that are necessary to produce (in an automatic way) complete collaborative Web applications.

The remainder of the paper is structured as follows: section 2 explains our proposal to conceptual modeling of Services; section 3 shows the introduced models from the point of view of their structural properties; section 4 explains the dynamic properties and the transformation of the models to a specific Web service composition technology (in this case we choose WS-BPEL [4], although it can be another –like BPML [5]); section 5 explains our code generation strategy and finally, section 6 presents conclusions and further work.

2 Conceptual Modeling of Services

The Model Driven Architecture (MDA) [2] is a new development strategy in which models are the first order actors within the software development process. MDA has several stages in which specific models are defined: the *platform independent models* (PIM), that describe the system with high-level constructs hiding the necessary technological details of the specific platform; and the *platform specific models* (PSM) which on the contrary, describe the system in terms of a specific technological platform. Besides these models, MDA proposes a strategy that has to be applied in order to transform these models into code.

Following the MDA strategy we define two PIM models to capture the requirements of Web services compositions: a *Service Model (hereafter SM)* and a *Dynamic Model for Service Composition (hereafter DMSC)*. As vertical arrows show in Figure 1 each of these models is mapped into a PSM model, the horizontal ones represent the existing relationships between them (both, at PIM and PSM level). The constructs of the PIMs and PSMs (its metamodels) are defined using the Meta-Object Facility (MOF) language.

The SM captures the structural requirements of both, own and external Web services of the application including their ports and operations. This PIM model can be then transformed into several PSMs such as .NET [3] or J2EE [4].

The behavior of the Web services composition is defined by the DMSC. Although traditionally the approach followed has been to compose the new Web services by specifying only the orchestration of the Web service components, we believe that the structural requirements captured in the SM are also needed to have a complete specification in order to enable the automatic code generation.

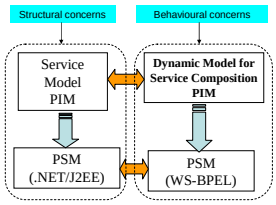

Fig. 1. MDA strategy to service composition modeling

2.1 Service Modeling

The structural requirements of produced and consumed functionality of an application are captured in the SM. Figure 2 shows the SM metamodel foundation. The included metaclasses are the basic constructs needed to model Web services, similar to other works [6,7,8].

The produced functionality is captured in the set of produced Web services from our application (called *Own-services*, see Figure 2). The consumed functionality is captured in the set of consumed Web services of our application (called *External-services*). Each service has one or more access points (Port) where each one has one or more of the following operations: (1) *one-way* (One-way-op), an asynchronous operation invoked by a client without response; (2) *notification* (Notification-op), an asynchronous operation invoked by the service without response; (3) *request-response* (Req-resp-op), a synchronous operation invoked by the client with response from the service and (4) *solicite-response* (Sol-resp-op), a synchronous operation invoked by the service with response from the client. The input and output parameters (Parameter) are one of the two specialized types from the data type (Data-type): simple (Simple-DT) or complex (Complex-DT).

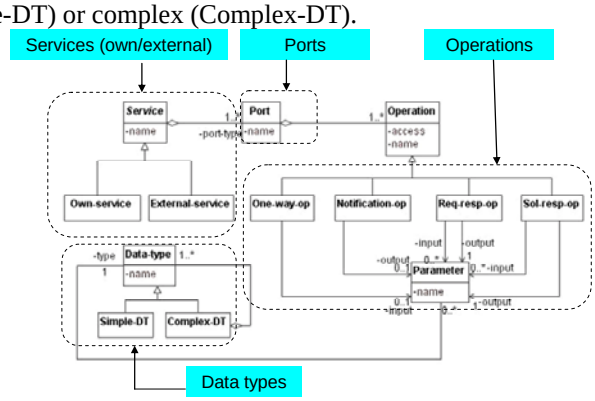

Fig. 2. The foundation SM metamodel

Figure 3 shows an instance (an extract) of the SM of Amazon.com (AWS). The operation shown (*asinSearchRequest*) allows the user to query information about a Product (*ProductInfo*) from its isbn (*AsinRequest*).

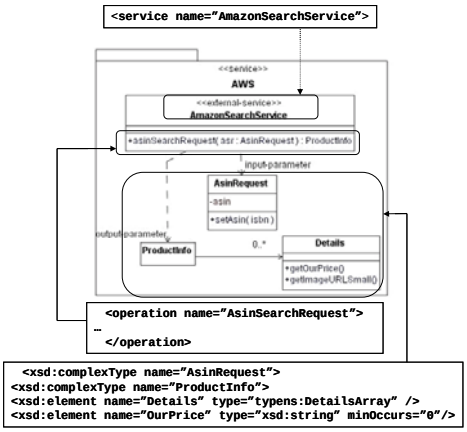

Fig. 3. SM for Amazon.com (AWS)

From the point of view of the application that produces it, services can be built in one of the following ways: (1) *Own-services*, with two possibilities: (a) those whose operations are views of the pre-existing operations in the logic layer of our application (at conceptual level they may be specified in the structural model) and (b) those whose operations are built from the composition of the operations from other own or external Services; and (2) *External-services* obtained from other applications that publish and produce them.

3 Structural Concerns

Services whose operations are built from the composition of own or external services are implemented by orchestrating their operations. This is a way of building *new functionality* (the new own-service) by reusing *functionality* through composition (from the pre-existing own or external services). From this perspective, service composition could be specified by *aggregation relationships of the services components*. Adding this new structural modelling, some tasks will be more easy to do (as we are going to show) than with the traditional dynamic approach, such as the dynamic Web service selection or the automatic and complete code generation contributing to facilitate the maintenance of the composite Service.

In order to have a precise definition of the relationships, its semantics needs to be defined. Some works have been addressed this problem in the context of object oriented conceptual modeling ([9,10]). In this work we follow the multidimensional framework proposed in [3] to characterize aggregation relationships between Web services. The use of this multidimensional framework allows us to capture the structural properties of the composition which are explained in the following subsections.

3.1. Service Aggregation

The *structural* properties of the composition are captured in the SM. These properties characterize and define the semantics of the relationships between the Web services being aggregated (its *binding*).

In the aggregation relationship, the service defined is called the *composite Service* (an Own-service) and the own or external services that are being aggregated are called the *component Services*.

3.2. Properties Specification

The properties are explained with respect the MOF metamodel in Figure 4.

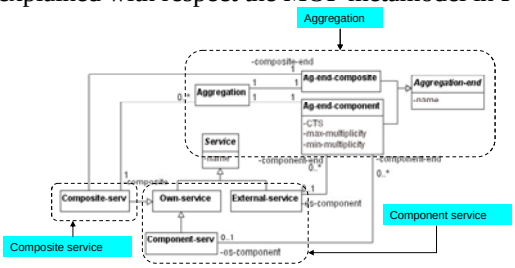

Fig. 4. Service aggregation MOF metamodel

a. **Temporal Behaviour**

– **Definition:** specifies if the *composite Service* has (or does not have) permanent *binding* with the *component Service* during its lifetime.

– **Defined over:** aggregation end (*component Service*).

– **Nomenclature:** $CTS_{aggregation\text{-}end}$

– **Values:** *Static*|*Dynamic*

- *Static*: the *component Service* is bound to the *composite service* during its life.
- *Dynamic*: the *component Service* is dynamically selected from data values (called *process variables* [11]) obtained during the execution of the composition logic, usually from a UDDI registry.

– **Semantic constraint:** expressed in OCL[9]
context Ag-end-component
inv temporal-value:
CTS=’Static’ or CTS=’Dynamic’

b. **Multiplicity**

– **Definition:** specifies the minimum and maximum number of component services (of the same type) connected with the composite service.

– **Defined over:** aggregation end (*component Service*).

– **Nomenclature:** $Min_{aggregation\text{-}end}$, $Max_{aggregation\text{-}end}$

– **Values:** nonnegative integers.

– **Semantic constraint:**
context Ag-end-component
inv multiplicity-value:

max-multiplicity >=0 and
min-multiplicity >=0 and
min-multiplicity <= max-multiplicity

3.3 Additional Semantic Constraints

One advantage of this multidimensional framework is the additional knowledge implied, which can be used to build better modeling tools with model checking features that assist the modeller in the correct construction of SMs. Some examples of the additional knowledge are as follows (expressed in OCL):

1. Every aggregation includes as a component Service an Own or External service (different from the composite Service):
   context Ag-end-component
   inv at-least-one-component-service:
   os-component->size()>0 xor es-component-size()> 0
2. The *Static* value from the *Temporal Behaviour* property implies that the *maximum* and *minimum* values from the multiplicity property are 1 (see Figure 5):
   context Ag-end-component
   inv static-multiplicity:
   CTS='Static' implies
   (multiplicity-Max=1 and multiplicity-Min=1)
3. The *Dynamic* value from the *Temporal Behaviour* property implies that the maximum multiplicity values should be greater than 1. The composite Service could be binding with 2 or more possible component Services, each one dynamically selected, as we are going to show.
   context Ag-end-component
   inv dynamic-multiplicity:
   CTS='Dynamic' implies
   multiplicity-Max>1

3.4. Service Aggregation Examples

Figure 5 shows an example of service aggregation to an e-business application that uses the Amazon Web service and B&N Web service. The composite service *BestStoreService* uses both Web services to get the best store and book price.

The aggregation relationship is defined *Static* because the composite service should have permanent binding with the component services. So they are specified using static *Temporal Behaviour* and multiplicity value equal to 1.

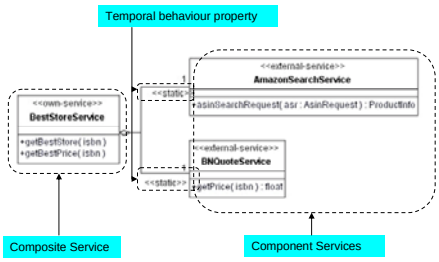

Fig. 5. SM using static aggregation

Figure 6 shows another example. In this case, the composite service (*SService*) is for a supplier application with the following process: to request the price of a product, the composite service offers the operation (*getPriceInStock*). This operation first checks the stock in the local warehouse using an own service (*LWService*). If the product is not in stock then it is checked in one of a couple of central warehouses using only one of the services *EWService1* or *EWService2* (specializations of the abstract service *EWService*, see Figure 7). So the binding between the composite service (*SService*) and the own service (*LWService*) is *Static* and to the *EWService* is *Dynamic*. In order to resolve to which concrete *EWService* is going to communicate the *SService* a condition is defined in the SM of the dynamic Web service.

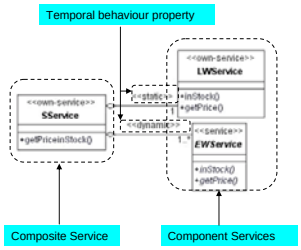

Fig. 6. SM using static and dynamic aggregation

Figure 7 shows the SM for the dynamic EWService. When a set of Web services (in this case EWService1 and EWService2) are going to be managed dynamically, first they are imported to the SM from a UDDI registry. Then, a façade class (*FEWService*), with an operation *getXService*, is generated into the model in order to delegate it the responsibility of dynamic Web service selection. This operation includes the condition necessary to select the concrete Web service in the dynamic model. This condition can be established in the SM or the DMSC. In the first case, the condition is based on the different model elements of the application. Because the condition is not defined as part of the DMSC definition, the reuse and flexibility in this model is improved (by example, if a new central Warehouse is added then only a *EWService3* is added to the SM, the DMSC is not changed and the dynamic selection responsibility continues in the SM). In the second case, the modeler defines the condition based on process variables from the DMSC and this condition is passed as parameter to the *getXService* in the SM in order to select de Web service. An example of this mechanism is presented in subsection 4.3.

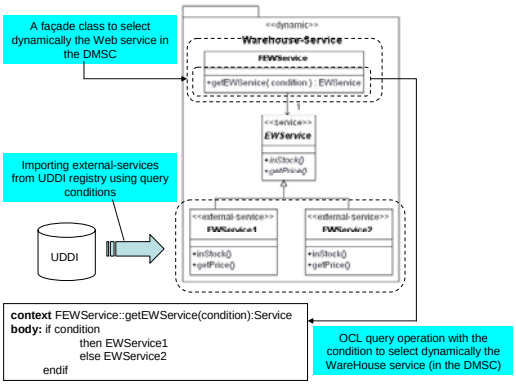

Fig. 7. Importing Web services to the SM to be used in a dynamic way

4 Behavioral Concerns

The logic of the composed Web services is captured in the DMSC. This model is defined as an UML activity diagram whose actions define the invocation of some of the component services operations. Each of those operations is defined as one of the types listed in Table 1 which correspond to the possible Web services operation types. (see Figure 2).

Table 1. DMSC operations types and stereotypes

Operation type	Stereotype
One-way	<<one-way>>
Request-response	<<request-response>>
Notify	<<notify>>
Solicite-response	<<solicite-response>>

To manage the data of the process two more actions are defined: variable declaration (stereotyped with the keyword <<variable>>) and variable assignment (stereotyped with the keyword <<assign>>).

In the case of dynamic selection of Web services, we define a special data type Service. An action to select dynamically the service (stereotyped with the keyword <<select-service>>) is also defined.

DMSC could be mapped to WS-BPEL (or another language such as BPML). In the following subsections the WS-BPEL mappings are introduced. An example is presented next to show the use of the DMSC and its mapping to WS-BPEL for an e-commerce service (*BestStoreService*).

4.1 The BestStoreService Case Study

Once the structural concerns of the *BestStoreService* (see Figure 5) have been defined, the composition logic of each operation needs to be specified by a DMSC. Figure 8 shows the composition logic of the *getBestStore* operation. Its DMSC is included in an action *getBestStore* (stereotyped with the keyword <<operation>>). The action has an input-pin, with the input parameter of the operation (*isbn*), and an output-pin (*BestStore*) with the return value.

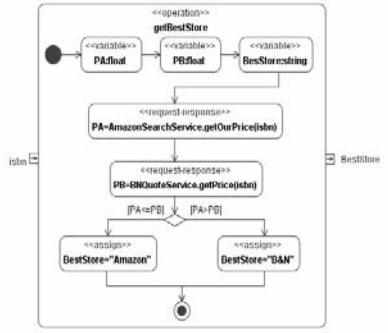

Fig. 8. DMSC of the getBestStore operation

The translation of the operation to WS-BPEL needs the structural knowledge captured in the SM (Figure 5) and the composition logic captured in the DMSC (see Figure 8):

- **Process definition:** corresponding to the main XML element of the WS-BPEL process. This is obtained from the action name of the operation (see Figure 9).

Fig. 9. Mapping the WS-BPEL process name

- **Partner links identification:** the partner links corresponds to: (1) the client, who invokes the WS-BPEL process and (2) the Web services invoked by the BPEL process. In the first case, the client name of the process is specified with the following syntax: ***<service-name>*-clientLT.** The *service-name* is taken from the name of the composite service in the SM. The role corresponds to the service-name too and the *portType* is corresponding from the port name in the SM. For simplicity reasons the Port class is not shown in the SM (see Figure 10). Each one of the component services are the Web Services collaborators. Its name has the following syntax: ***<service-name>-clientLT***

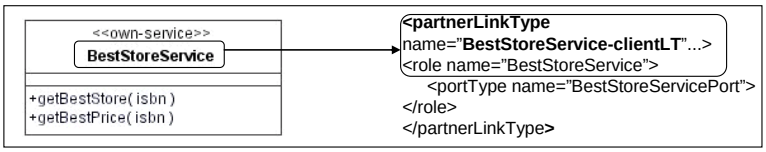

Fig. 10. The client as a partner link

Figure 11 shows and example for Amazon as collaborator

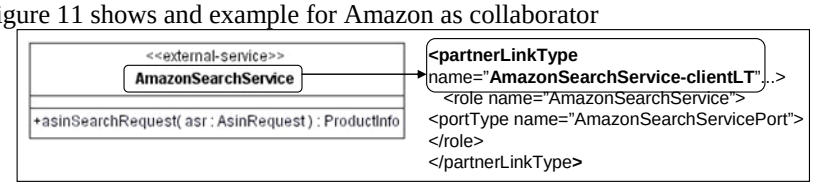

Fig. 11. Amazon Web Service as collaborator

- **Operation logic:** is defined in terms of WS-BPEL:

```
<process name=...>
<partnerLinks></partnerLinks>
<variables></variables>
<sequence></sequence>
</process>
```

Fig. 12. Basic operation logic for the composite operation in WS-BPEL

- **Client partnerLink definition:** from the client definition as collaborator (see Figure 10) is possible to obtain its *partnerLink* in the SM:

```
<partnerLinks>
<partnerLink name="client" partnerLinkType name="BestStoreService-clientLT"
myRole="BestStoreService"
partnerRole="BestStoreServiceClient" />
</partnerLinks>
```

Fig. 13. Client partnerLink definition

- **Definition of the other partnerLinks:** from the component services and its *partnerLinks* is possible to obtain the *partnerLink* of the other collaborators:

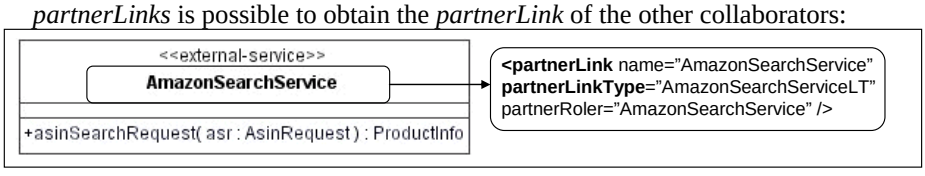

Fig. 14. Amazon collaborator partnerLink definition

- **Variable definition:** for each message sent to the collaborators corresponding to an operation invocation is necessary to define at least one variable (*request* for the invocation) if none value is returned; in other case, it is necessary to define two

variables (additionally a *response* for the return value). This can be obtained from the operation definition in the SM and its use in the DMSC (see Figure 15):

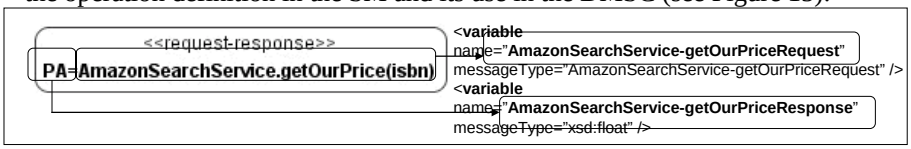

Fig. 15. Variable definitions for operation invocation

Declared variables in DMSC actions can be mapped to variable elements in the process (see Figure 16).

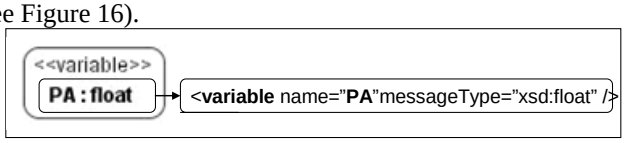

Fig. 16. Variable definition

In operations with response, it is necessary to define two variables: one for the request and one for the response. The SM is used for this mapping.

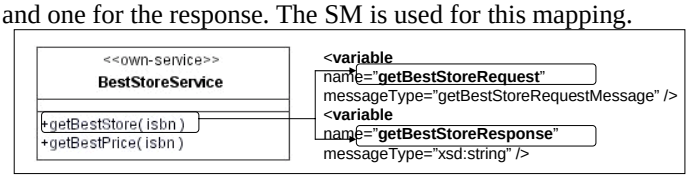

Fig. 17. Variable definition for a request and response operation

- **Main body process:** which starts with the message reception from the client:

```
<receive partnerLink="client"
operation="getBestStore" variable="getBestStoreRequest"createInstance="yes"/>
```

Fig. 18. Initial message from the client

Each action of the DMSC is mapped to a WS-BPEL activity. Figure 19 shows a DMSC comparison and its mapping to a WS-BPEL switch. As another example, Figure 20 shows a DMSC assignment and its mapping to a WS-BPEL assign.

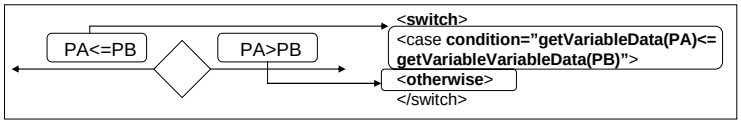

Fig. 19. From DMSC comparison to switch WS-BPEL activity

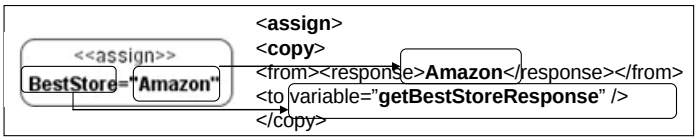

Fig. 20. DMSC assign and assign WS-BPEL

4.2 Web Service Dynamic Selection

The dynamic selection of Web services is specified in the DMSC with two special actions: (1) a *variable declaration* action to define a variable of the data type *Service* (see Figure 2). This variable will be set using the *select-service* action. And a (2) *select-service* action: in which the *getXService* operation, defined in the façade of the dynamic service in the SM, is invoked in order to select dynamically the Web based on a condition defined by the modeler using an OCL expression.

With respect to the example of dynamic Web service selection of central warehouses from Figure 6, in order to implement the *SService.getPriceinStock* operation (see Figure 6) and taken into account the SM for the composite Web service (see Figure 7), the actions needed to specify the dynamic selection of web services are shown in Figure 21. In this case the condition has been defined on a DMSC process variable (country) which is passed as parameter to the condition in the SM.

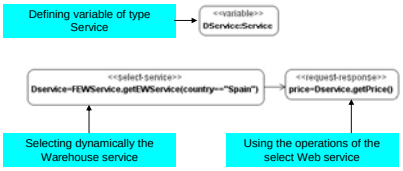

Fig. 21. Dynamic Web service selection in the DMSC

5. Code Generation

Figure 22 shows the three steps of the code generation to implement our proposal. In the first step (see box 1) we define a set of model to model transformations: from PIM models -SM and DMSC- to PSM models -Java and BPEL-. Both kinds of models are defined using the Eclipse Modeling Framework (EMF) [16] and the transformations are defined using the ATLAS Transformation Language (ATL) [17].

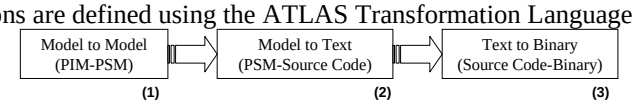

Fig. 22. Code generation strategy

In the second step (see box 2), another set of model to text transformations are also defined, from PSM models to source code. For each one of the models, a group of templates are defined to enable the complete source code generation (Java and WS-

BPEL). We implement this step by using the ERb tool for templates and the Ruby language [18].

In the third step (see box 3), the Java source code generated from the step 2, it is compiled with a common Java compiler (as javac) and it is installed in an application server. In the case of WS-BPEL, there is no necessity to compile; only the code is deployed in a BPEL execution environment. The details about this code generation strategy are given in the following subsections.

5.1. Metamodels Definition

The PIM and PSM models are defined using KM3 [19]. Once the metamodels are specified in KM3, then they are transformed to EMF Ecore.

As an example, Figure 23 shows an excerpt of the SM. By using the *transformation* from *KM3 to Ecore* existing in the ATL tool, they are transformed to EMF Ecore.

```
package Service {
        abstract class Service {
                attribute name : String;
                reference port[1-*] : Port;
        }
        class EService extends Service {
        }
        class OService extends Service {
        }
        class Port {
                attribute name : String;
        }
}

package PrimitiveTypes {
        datatype String;
}
```

Fig. 23. An excerpt of the PIM-Service metamodel in KM3

We also define metamodels for the DMSC, PSM-Java (based on platform Axis [20]) and PSM-BPEL. The models are specified in XMI 2.0 format [21].

5.2. Model to Model Transformations

The model to model transformations are defined using ATL rules. The Model to Model strategy is shown in Figure 24. As an example, Figure 26 shows the transformation rule from the SM (see Figure 23) to PSM-BPEL (see Figure 25) which implements the transformation defined in Figure 10. This rule matches the Service name in the SM and generates the *Partnerlinks*, *role* and *PortType* of the PSM-BPEL.

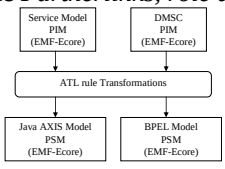

Fig. 24. Model to model strategy

The input to the rule is a SM and the output is a PSM-BPEL model. From this last model, a model to code transformation algorithm, based on templates, are applied to obtain the final WS-BPEL (or Java) code.

```
package PSM_BPEL {
        class Process {
                attribute name : String;
                reference pls : Partnerlinks;
        }
        class Partnerlinks {
                reference pl[1-*] : Partnerlink;
        }
        class Partnerlink {
                attribute name : String;
                reference role : Role;
        }
        class Role {
                attribute name : String;
                reference pt : PortType;
        }
        class PortType {
                attribute name : String;
        }
}
package PrimitiveTypes {
        datatype String;
}
```

Fig. 25. An excerpt of the PSM-BPEL metamodel in KM3

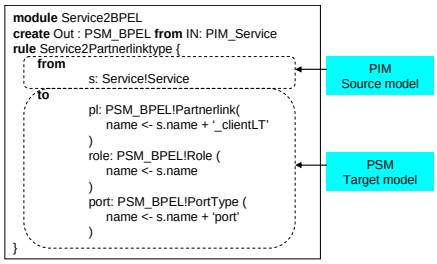

Fig. 26. An ATL transformation rule from PIM-Service to PSM-BPEL

5.3. Model to Code Transformations

Figure 27 shows the general Model to Code strategy. From the PSM models a final transformation is needed. For each one of the PSM models a set of templates are designed, depending on the source files needed to the final platforms. So we design templates to the AXIS platform and to the WS-BPEL platform.

As an example, Figure 28 shows the template (with a format defined by the ERb tool) for the example of the PSM-BPEL in the previous section. This template and the XMI model generated in the previous step are inputs to the code generator and the final WS-BPEL is obtained.

In the case of the PSM-Java AXIS Model, the code generator produce Java code that will need a final compilation using the java compiler (javac).

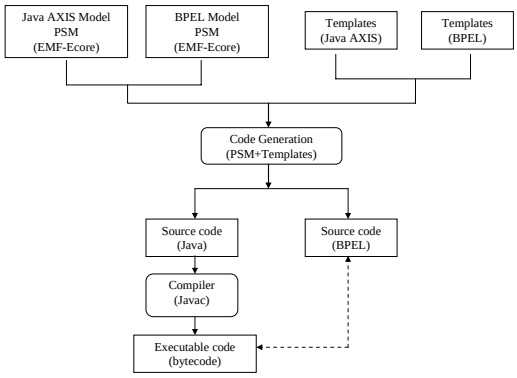

Fig. 27. From PSM models to target source code

```
<PartnerLinkType name="<%= name_pl %>" >
  <role name="<%= name_service %>">
    <portType name="<%=name_port %>">
  </role>
</PartnerLinkType>
```

Fig. 28. WS-BPEL template in ERb tool

6. Conclusions and Further work

In this work we have presented a solution for the conceptual modeling of Web service compositions. The proposed models capture structural (SM) and dynamic (DMSC) requirements of the composition. As we have shown, the captured aspects in both models are complementary and needed to enable the complete code generation of the composite Web service.

Moreover, with the structural model the modeler gain flexibility and reuse as we have shown in the case of dynamic Web service selection in the DMSC. If a new Web service is introduced, in the traditional approach changes in the dynamic model would be needed. Because our approach follows a polymorphic strategy we only need to add a new Service in the SM and, maybe, a change in the select condition. The DMSC will not change. Finally, from these models a transformation to WS-BPEL specification has been presented too.

This proposal has been proven successfully in the construction of the collaborative aspects of the *Technical University of Valencia General Library* Web application (http://www.upv.es/bib/index_i.html).

We are currently including this proposal in the CASE tool of our Web Engineering method OOWS. With this proposal, the designer specifies the *Service view* of the Web application that is being modeled.

As further work we consider exploring the reuse of Web services using specialization relationships in the structural and dynamic models. Presentation concerns should be another topic that we need to explore.

7. References

1. Fons J., Pelechano V., Albert M. And Pastor O. Development of Web Applications from Web Enhanced Conceptual Schemas. *Proc. of the International Conference on Conceptual Modeling. 22nd Ed. ER'03*, pp. 22-45, EEUU, 13-16 october 2003.
2. OMG. MDA. http://www.omg.org/mda
3. Albert M., Pelechano V., Fons J. Ruiz M. Pastor O. Implementing UML association, Aggregation and Composition. A particular Interpretation based on a Multidimensional Framework. *CaiSE 2003*: 143-148.
4. Andrews T. Et al. Business Process Execution Language for Web Services. Version 1.1. http://www128.ibm.com/developerworks/library/specification/ws-bpel/
5. BPMI. Business Process Management Language. http://www.bpmi.org
6. Colombo Massimiliano, Di Nitto Elisabetta, Di Penta Maximiliano, Distante Damiano, Zuccalà Maurilio. Speaking a Common Language: A conceptual Model for Describing Service-Oriented Systems. *ICSOC 2005*: 48-60
7. Gronmo R., Slogan D., Solheim, Oldevik J. Model-driven Web Services Development. SINTEF Telecom and Informatics. *EEE'04*
8. Bézivin J., Hammoudi S., Lopes D., Jouault F. Applying MDA Approach for Web Service Platform. Atlas Group, INRIA and LINA. ESEO. TNI-Valiosys. *EDOC 2004*.
9. Warmer J. Kleepe A. The object constraint language second edition. Addison Wesley. 2003.
10. Kristensen B. B. Osterbye K. *Roles:* Conceptual abstraction theory and practical languages issues. Theory and practice of Object Systems, 2(3): 143-160, 1996.
11. Alonso G., Casati F., Kuno H., Machiraju V. Web Services. Concepts, Architectures and Applications. Springer 2004.
12. Object Management Group. Unified Modeling Language Specification. http://www.uml.org
13. Reisig W. And G.R. (editors). Lectures on Petri Nets I: Basic Models. *Lecture Notes in Computer Science.* Springer-Verlag, 1998.
14. Milner r. Parrow J., Walker D. A calculus of mobile processes. *Information and Computation*. 100(1):1-40, Sept. 1992.
15. Anzböck R., Dustdar, S. Semi-automatic generation of Web services and BPEL processes – A Model-driven approach (Appendix), BPM 2005, 5-7 September, Nancy France. *Springer LNCS*
16. Eclipse project. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf
17. Eclipse project. ATL Home page. http://www.eclipse.org/gmt/atl
18. Herrington Jack. *Code generation in action*. Manning Ed. 2003.
19. Joualt, F., Bézivin J.: KM3: a DSL for metamodel specification. In: *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy (2006).
20. Apache project. Web Services – Axis. http://ws.apache.org/axis
21. OMG/XMI XML Model Interchange (XMI) 2.0. Adopted Specification. Formal/03-05-02, 2003.