A Case Study on Integrating Extra-Functional Properties in Web Service Model-Driven Development: from Model to Code

Guadalupe Ortiz
Quercus Software Engineering Group
Centro Universitario de Mérida, UEX
Mérida, Spain
gobellot@unex.es

Juan Hernández
Quercus Software Engineering Group
Escuela Politécnica, UEX
Cáceres, Spain
juanher@unex.es

Abstract— Being one of the most promising current technologies, Web Services are at the crossing of distributed computing and loosely coupled systems. Although vendors provide multiple platforms for service implementation, service integrators, developers and providers demand approaches for managing service-oriented applications at all stages of development. In this sense, approaches such as Model-Driven Development (MDD) and Service Component Architecture (SCA) can be used jointly for modeling and integrating services regardless of the underlying platform technology. Besides, WS-Policy provides a standard description for extra-functional properties, which remains independent of both the final implementation and the binding to the service in question. In this paper we show a case study in which the aforementioned MDD, SCA and WS-Policy are assembled in order to develop web services and their extra-functional properties from a platform independent model, which is later transformed into platform specific ones and then into code.

Keywords:Extra-functional property, Web service, model-driven development, aspect-oriented techniques, WS-policy.

I. INTRODUCTION

Web Services provide a successful way to communicate distributed applications, in a platform independent and loosely coupled manner, providing the systems with ample flexibility and more manageable maintenance. Although development middlewares provide a splendid environment for service implementation, methodologies for earlier stages of development, such as the modeling stage, are not provided in a cross-disciplinary scope, whereby, for instance, the automatic model-implementation transformation or the addition of extra-functional elements would be feasible.

Academy and industry are beginning to focus on the modeling stage, where it is also pursued to keep the loosely coupled notion and independence from the platform [13]. Some rising proposals focus on representing the service as a component and others base the model on WSDL elements; representative approaches are described below:

To start with, *Service Component Architecture* (SCA) and *Service Component Description Language* (SCDL) provide a way to define interfaces and references independently of the final implementation technology, which will be bound subsequently [3]. According to SCA, services are modeled as components. These components are linked to a given interface, which can be later specified in a particular one. Besides, the components will show the required references for their behavior to be completed. This proposal provides the following advantages: first of all, it defines a very high level model, allowing the developer to bind it to a specific technology at a later stage. Secondly, the model can be implemented by using different approaches such as Java, BPEL and States Machine, therefore permitting adaptability to the client's specific needs, or to the most suitable option for its integration in a specific environment. Thirdly, the model can be converted into XML, providing an intermediate language to integrate different party models into a unique system. However, this proposal does not face how to integrate this definition with other stages of development, such as implementation.

As a second trend, many proposals are emerging in the literature where a Model Driven Architecture (MDA) approach is being applied to web service development. MDA has been proposed to facilitate the programming task for developers by dividing system development into three different phases: a *Platform Independent Model* (PIM), a *Platform Specific Model* (PSM) and, finally, the code. Thus, MDA solves the integration of the different stages of development, as mechanisms are provided to model applications in a platform independent manner which may be later transformed into the specific required models and eventually into final code, but it does not provide a specific way to do so for service technology.

Let us consider now that we want to provide our modeled services with extra-functional properties, that is, with additional pieces of code which are not part of the main service functionality. It is suggested by the SCA specification that this type of property may be modeled at a different level; the way to do so and to include them in additional stages of development has not been approached as yet. Alternatively, the named MDA proposals do not consider how extra-functional properties may be included in modeled services. On the other hand, WS-Policies have emerged as a standardized way for describing extra-functional service capabilities by using the XML standard

[17]. This allows properties to remain completely decoupled when described and there is no need to establish dependences from the service description file (WSDL) to the policy ones; property description is not linked to a specific implementation, either, maintaining the platform's independent environment. However, WS-Policy does not determine how the properties are to be modeled or implemented, and an additional mechanism would be necessary so as to integrate property modeling and implementation with their description in service-based systems.

In this paper we show a case study in which a proposed model-driven methodology is applied in order to deal with extra-functional property integration in web service development, extending our previous work on the topic [9].

The rest of the paper is organized as follows: Section 2 gives an overview of the steps followed in this approach. Section 3 shows how the PIM should be implemented. Then, Section 4 explains the PSM stage, where Section 4.1 shows the specific metamodels; Section 4.2 explains the rules used for PIM to PSM transformation and, finally, Section 4.3 shows the specific models obtained from the case study PIM. Section 5 explains the rules used to obtain code from PSMs and the final generated code. Other related approaches are examined in Section 6, whereas discussion and conclusions are presented in Section 7.

II. MODEL-DRIVEN TRANSFORMATIONS

In this section we are going to provide a general overview of the presented approach, describing the order to be followed to face web services and extra-functional property development from platform independent model to code, which will be explained in detail in the next sections.

We will use one UML profile and an additional stereotype in order to define our case study platform independent model. Thus, UML will be our PIM metamodel and the developer will be able to design the system by using common standard modeling tools at this stage of development. UML is MOF compliant, and so will the PIM metamodel. The extra-functional property profile defines the abstract stereotype *extra-functional property*, which will extend *operation metaclass* or *interface metaclass*. The extra-functional property provides five attributes: the first one is *actionType*, which indicates whether the property functionality will be performed *before*, *after* or *instead of* the stereotyped operation's execution – or if no additional functionality is needed it will have the value *none,* only possible in the client side. Secondly, the attribute *optional* will allow us to indicate whether the property is performed optionally –the client may decide if it is to be applied or not– or compulsorily –it is applied whenever the operation is invoked. Then, a third attribute, *ack*, is included: when *true* it means that it is a well-known property and its functionality code can be generated at a later stage; it will have the value *false* when only the skeleton code can be generated. Finally, *PolicyId* contains the name of an existing policy or the name to be assigned to the new one in the service side and *priority* allows the developer to establish a priority in the execution of the functionality of those properties which affect the same operation. These are the necessary attributes to define the main characteristics in any property, which may be complemented with specific property attributes.Once we want to use the profile in a specific case study, we will extend it with the specific properties to be used or we can have a pool of predefined properties.

- Afterwards, the specific models have to be obtained: in this case we decided our models to be EMF-complaints (http://www.eclipse.org/emf/), which facilitates a graphical edition of the element attributes within the Eclipse environment, allowing easier consultation or modification, if necessary. Service models will be based on a JAX-RPC metamodel, and three additional specific metamodels are provided for properties: an aspect-oriented one, a policy-based one and a soap tag-based one.

Subsequently, the transformation from PIM to the PSMs has to be defined. Several tools can be found for model transformations and code generation. We used *ATL* (ATLAS Transformation Language – see http://www.eclipse.org/gmt/atl/), which provides an Eclipse plugin and has its own model transformation language, also MOF-compliant. The ATL transformation file will define the correspondence between the elements in the source metamodel (PIM) and the target ones (PSMs) and will be used to generate the target model based on the defined rules and the input model. When the transformation rules are applied to the case study PIM, its platform specific models are obtained.

Finally, code can be generated from the specific models by applying additional transformation rules. In this case no target metamodel is needed since these new rules will establish correspondences from the elements in the specific metamodels to *Strings*. On the one hand, JAX-RPC web service code, to be deployed with the Java Web Service Developer Pack, will be generated from the service specific model. On the other, AspectJ will be used for the implementation of the property functionality, thus maintaining properties well modularized and decoupled from the implemented services; Java will be used to implement the code necessary for optional property inclusion. With regard to description, WS-Policy documents are obtained for each property [1], which are integrated with the aspect-oriented implementation.

III. CASE STUDY

The case study presented in this paper consists of a set of services related to a university administrative service and a web client, created for their use.

The service side consists of a set of five web services:

- PreregistrationService: using the pre-registration web service, the user will be able to create a new preregistration application for any of the courses taught in the University Centre of Mérida (CUM), to check the preregistration status and to ask for a new copy of the preregistration application to be sent to him.

Figure 1. PIM with extra-functional properties.

- RegistrationService: by using the registration service, users will be able to formalize a registration at the University Centre of Mérida, by providing their personal details, the courses to register for and payment information.
- ExamOpportunityService: through the exam opportunity service the user can obtain a list of the different subjects in a specific qualification and can bring forward or cancel any exam opportunity from the registered subjects.
- AcademicResultsService: academic results can be consulted through this service.
- TeacherService: this service can be used to obtain a list of all the CUM teaching staff in a particular area and to obtain additional information on them.

Let us imagine that we want to include some extra-functional properties to the services' model. At this stage we can discern three types of property: properties which are always applied and do not imply changes or additional information in the client code; those which are optional, so they have to be somehow chosen by the client; and those which imply changes to client code. In this sense three examples are provided, one for each option:

- First of all, a *log* property, to be applied to all operations offered by the registration service to record received invocations.
- Secondly, a property called *detailedInfo*, which will be required discretionarily by the client when invoking *bringForwardExam* in *ExamOpportunityService*: exam dates and locations can be obtained when changing the semester in which the student is going to sit the exam; the change is regularly updated and no additional information is obtained.
- Additionally, invocations to *personalData* in *RegistrationService* must be encrypted. In order to enable this functionality the *desencyption* stereotype has to be applied to the offered operation.

Regarding the client in the case study, we have created a web client for students to make use of these services by a user-friendly interface. The main web page of the web client is shown in *Figure 1*, from which the different service clients can be accessed.

IV. PLATFORM INDEPENDENT MODEL

In order to create the platform-independent model we will make use of the profile defined in the previous section and motivated in [10], which allow us to models services and their extra-functional properties in a platform-independent way.

In order to integrate the properties described in the previous section in service models we have to extend extra-functional property stereotype as shown in *Figure 2*. This figure shows us the three mentioned property stereotypes:

- *DetailedInfo* property which provides the attribute *detailedInfoFunction* in order to invoke the method which will provide us with the new functionality.
- *Log* with two attributes. *logFile* and *myLogFunction*, the first one will be the file in which we will record all the log information and the second one the will be the method which may be required for the mentioned log.
- *Desencryption* shows the attributes *keyDoc* – its value is used as the reference of the private key in the parameters decryption- and *desencryptionFunction* – it indicates the method used for the decryption.

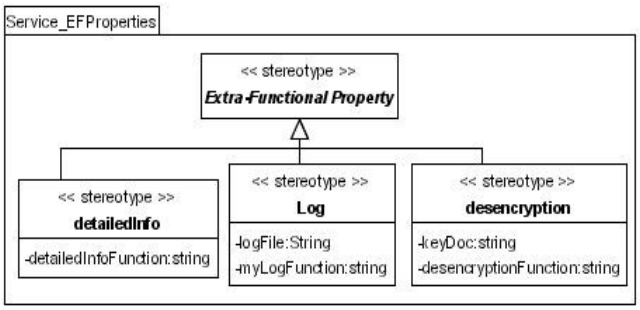

Figure 2. Extension of the extra-functional property profile with specific properties

Then we include the new property stereotypes in the service models as depicted in *Figure 3*, which are described in the following lines:

- In order to provide *bringForwardExam* in *ExamOpportunityService* with *detailedInfo* in the PIM, we have to stereotype the named operation with *<<detailedInfo>>*. Stereotype attributes are attached to models as tagged values, but they have also been included as comments in the illustration to show their values. In it the attributes for *detailedInfo* indicate that the property will be performed *optionally instead* of the execution of the named operation; it is not a *well-known* property; *policyID* is *DetailedInfo _ao4ws* and *policyDoc* is *null*.
- To provide *personalData* in *RegistrationService* with decryption in the PIM, we have to stereotype the named operation with *<<desencryption>>*. Stereotype attributes indicate that the property is not *optional* and it will be performed *instead* of the execution of the named operation – so the decryption will wrap *personalData* functionality. It is not a *well-known* property (*ack* is *true*) so that the functionality code will not be generated; *policyID* is *Desencryption _ao4ws* and *policyDoc* is http://ao4wDes.xml. Finally, for this property we can see that two specific parameters have been added: *KeyDoc* and *desencryptionFunction*, which contain the values *myPrivateKey* and *myDesencryptionFunction*, respectively and which are used as the key and function to decrypt the received message.
- Finally, log will be done for all the operations in the interface offered by *RegistrationService*. For this purpose, we have stereotyped the offered interface – *RegistrationServiceIF*– with *<<log>>* in the PIM. Stereotype attributes indicate that the application of the property will be mandatory (*optional* is *false*) and its functionality will be performed *after* the execution of the interface operations. Since *ack* has the value *true* it is a *well-known* property and therefore we will generate the complete functionality code for it. To end with, *policyID* is *log_ao4ws, policyDoc* is *null,* the method used for the logging is *myLogFunction* and the file in which the log will be recorded *myLogFile*.

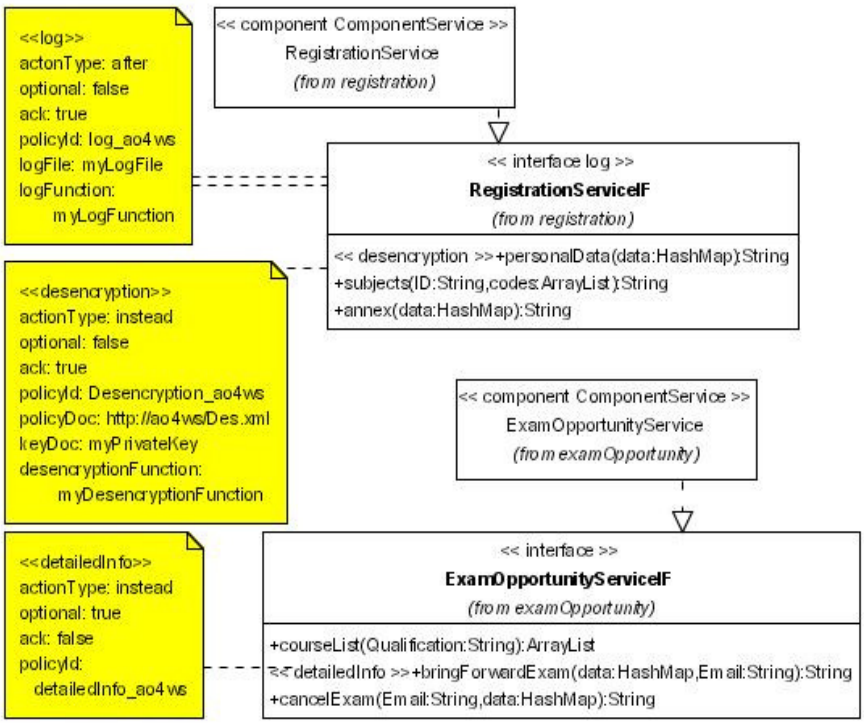

Figure 3. PIM with extra-functional properties.

It is important to remark that various stereotypes may be applied to the same operation, if necessary, thus different properties can be applied to the same element. Besides, different priorities can be assigned to those properties which are applied to the same element.

V. EXTRA-FUNCTIONAL PROPERTY PSMS

In this section we will show, first of all, the metamodels proposed for specific models, secondly the main rules used for transformations as well as the Eclipse environment configuration will be explained, and the specific models obtained for the case study will be discussed.

A. Proposed metamodels

We generate a specific model oriented to JAX-RPC services to be compiled and deployed with Java Web Service Developer Pack. In this regard the metamodel will be formed by the service Java interface and its implementation plus the necessary configuration files: *web*, *config-interface* and *jaxrpc-ri*; these elements are shown in the left-top part of *Figure 4* (properties of every element in the metamodel are not shown in the figure due to space restrictions). The metamodel, as shown in the said figure, is EMF-compliant instead of MOF-compliant, since it allows the developer to edit the generated EMF specific models to easily check and modify property attributes when necessary. The different elements shown in this part of the figure correspond to a simplified Java metamodel plus the three configuration files, which contain the main attributes necessary for their description.

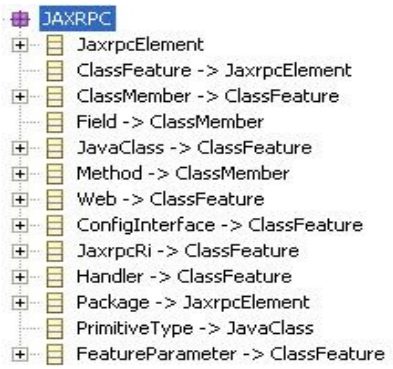

Figure 4. JAX-RPC metamodel.

As far as extra-functional properties are concerned, our specific models will be based, first of all, on an aspect-oriented approach to specify the property functionality, secondly on a *soap tags*-based approach, to lay down the necessary elements to be included or checked in the SOAP message header and, finally, a policy-based one for property description. EMF-compliant metamodels are depicted in *Figures 5, 6 and 7* and explained below:

- As shown in *Figure 5*, every *aspectClass* will have an attribute *target* which indicates the method for the property to be applied, a second attribute, *actionType,* which informs of when it has to be applied; *ack* indicates whether the property is well-known and, finally, an *action* may refer to the corresponding functionality. Besides, all additional particular property characteristics will be included as attributes. The metamodels, though represented in the EMF format, have been defined by using the KM3 syntax provided by ATL. For the better comprehension of the property-related metamodels, we have also included in this paper the KM3 definition. In this sense, in the following lines we can see the Aspect metamodel KM3 definition:

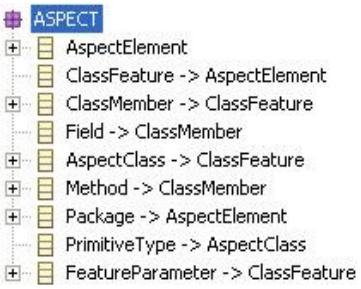

Figure 5. Aspect-based metamodel.

```
package ASPECT{

abstract class AspectElement {
     attribute name : String;       }


abstract class ClassMember extends AspectElement{
     reference type : AspectClass oppositeOf
            typedElements;
     reference owner : AspectClass oppositeOf
            members;       }

class Field extends ClassMember {
     attribute value: String;       }

class AspectClass extends AspectElement{
     reference typedElements[*] : ClassMember
            oppositeOf type;
     reference parameters[*] : FeatureParameter
            oppositeOf type;
     reference "package" : Package oppositeOf
            classes;
     reference members[*] container : ClassMember
            oppositeOf owner;
     attribute target : String;
     attribute ack: String;
     attribute actionType: String;
     attribute opt: String;
     attribute priority: String;    }

class Method extends ClassMember {
     reference parameters[*] ordered container :
            FeatureParameter oppositeOf method;}

class Package extends AspectElement {
     reference classes[*] container : AspectClass
            oppositeOf     "package";      }

class PrimitiveType extends AspectClass {}

class FeatureParameter extends AspectElement {
```

```
    reference type : AspectClass oppositeOf
            parameters;
    reference method : Method oppositeOf
            parameters;    }
}
```

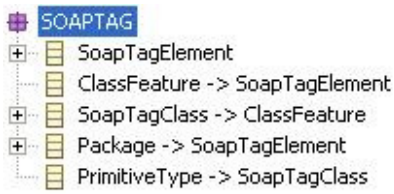

Figure 6. Soap tag-based metamodel.

- New tags are included in the SOAP Header to select –in the client side– or check –service side– relevant properties, when optional, or to deliver any other necessary information, as shown in *Figure 6*. Every *SoapTag* element will have an attribute *target* which instructs the method for the property to be applied, a second attribute, *value,* to show the tag to be included; finally, *side* indicates whether the tag is to be included by the client or checked by the service. The KM3 definition for the soap tag-based metamodel is included below:

```
package SOAPTAG {

abstract class SoapTagElement {
attribute name : String;    }


class SoapTagClass extends SoapTagElement {
    reference "package" : Package oppositeOf
            classes;
    attribute target: String;
    attribute value: String;
    attribute side: String;
    attribute providerName: String;       }

class Package extends SoapTagElement {
    reference classes[*] container : SoapTagClass
            oppositeOf "package";  }

class PrimitiveType extends SoapTagClass {}
}
```

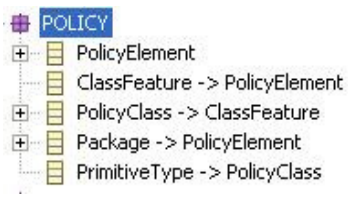

Figure 7. Policy-based metamodel.

- *Figure 7* shows that a policy will be generated for each property. The policy element will contain the policy *name*, whether the property is *optional*, well-known or domain-specific (*ack*); *targetType* indicates whether the policy is to be applied to a *portType* or an *operation* and *targetName* gives the name of the latter. For a further understanding, the KM3 description is shown in the following lines:

```
package POLICY {

abstract class PolicyElement {
attribute name : String;    }


class PolicyClass extends PolicyElement {
    reference "package" : Package oppositeOf
            classes;
    attribute opt: String;
    attribute acronym: String;
    attribute targetType: String;
    attribute targetName: String;
    attribute policyReference: String;
    attribute interface: String;
    attribute service: String;    }

class Package extends PolicyElement {
    reference classes[*] container : PolicyClass
            oppositeOf "package";  }

class PrimitiveType extends PolicyClass { }
}
```

B. *Transformation Rules*

In this section we comment briefly on one of the rules in the created transformation file in order to show how the syntax of the *ATL* declarative language is. As shown in *Figure 8* this rule applies to those properties whose *actionType* is other than *none* and which are applied to an operation. In the following lines we describe the different output results obtained in the transformation:

- The first output is an *aspectClass*; its *name* is formed by the UML package name added to the operation name and property one. Its *package* will be the one of the source element. Its *target* will be defined by the source namespace, its package name and its own name. The *actiontype* will be one in the source stereotype and its *ack* value will also be the one in the source stereotype.
- The second output –*out2*– is used for obtaining additional fields from the particular property to be included in the aspect; that is, its *name*, its *owner* and its *type* (the type will be *String* by default).
- The third output provides the aspect with the *action* and its corresponding parameters (*out3*).
- *Out4* will provide us with the *soaptag* elements to be checked to apply the property when optional. This is the reason why *type* has always the value *service* in this rule.
- Finally, policy information is found in *out5*. This information is composed of the *policy name* and *package*, the *target type* and *name*, the *ack* value and if the policy is optional or not.

The following step is to configure the Eclipse environment in order to fulfil the transformation. To start the process we will use the platform-independent model created in *Section IV* as the source for the PIM-PSM transformation. For this purpose, we have had to export the model to XMI (case tools have an

option to export to XMI). Once we have our PIM in XMI format we generate the specific models. For this purpose, as previously mentioned, we have used the Eclipse environment, in which the ATL plugin is installed. In the Eclipse environment we have created a project in which the XMI file is included. In this project the UML, JAXRPPC, ASPECT, POLICY and SOAPTAG metamodels are also present, together with the predefined set of transformation rules.

```
rule TV2AO {
from e: UML!TaggedValue (
        (e.taggedValueType() = 'actionType')  and
       (e.taggedValueDataValue()<>'none')and (
        e.modelElement.oclIsTypeOf(UML!Operation)
)       )
to out: ASPECT!AspectClass(
    name<-e.modelElement.owner.namespace.name+'_'
        +e.modelElement.name+'_'+
        e.type.owner.name,
   package <-e.modelElement.owner.namespace  ,
   target                                  <-'public
   '+e.modelElement.owner.namespace.name+ ' ' +
     e.modelElement.owner.name+'.'+
   e.modelElement.name+'(..)',
   actionType <- e.taggedValueDataValue(),
   ack<-e.getAck()     ),
out2 :distinct ASPECT!Field foreach(d in
e.getFields())(
   name <- d.type.name,
   owner <- out,
   type <- String      ),
out3 : ASPECT!Action (
   name <- 'action',
   owner <- out,
   type<-e.modelElement.parameter-
   >select(x|x.kind=#pdk_return)-
   >asSequence()first().type,
       parameters <-  e.modelElement.getP()->
        collect (p |thisModule.P2F(p))        ),
out4 :distinct SOAPTAGS!SoapTag foreach(d in
e.optional='true')(
   name <- d.type.name,
   type <- String,
                     target <-
   'public'+e.modelElement.owner.namespace.name+'
                          '+
     e.modelElement.owner.name +'.'+
   e.modelElement.name+ '(..)',
   value:<- true
   side <-service,
   package <-e.modelElement.owner.namespace  ),
out5: POLICIES!Policy(
    name<-e.modelElement.owner.namespace.name+'_'
        +e.modelElement.name+'_'+
        e.type.owner.name,
   package <-e.modelElement.owner.namespace,
   targetType<-'Operation',
   targetName <- 'public '+
   e.modelElement.owner.namespace.name       + '
   '+   e.modelElement.owner.name +'.'+
   e.modelElement.name+'(..)',
   ack<-e.getAck(),
   optional<-e.getOptional()  )             }
```

Figure 8. Transformation rules.

In order to execute the transformation we had to configure the Eclipse running environment: first of all the ATL file containing transformation rules is selected (see top part of *Figure 9*), then the source and target metamodels and source model has to be indicated, as well as the location where we desire the target generated model to be stored. An example is shown in the lower part of *Figure 9*; in it we can see the configuration for the UML2JAXRPC transformation, therefore we only have one target metamodel. When the transformations from UML to ASPECT, POLICY and SOAPTAG are configured, the three specific metamodels and output models have to be specified.

Thus, using Eclipse and the ATL plugin, we can perform PIM to PSMs transformation from the case study, whose result is shown in the next subsection.

C. Specific Models in our Case Study

Some of the specific models obtained from the case study PIM transformation are shown in *Figure 10, 11, 12 and 13*, where service and property models can be examined. Only some branches of structure have been deployed to make the illustration easier to understand. Specifically, we have chosen, for instance, the *detailedInfo* property as a characteristic example for the remainder of this paper.

Figure 10 shows the created web services with their corresponding generated elements, namely service Java interfaces and implementations and configuration files. For instance, *examOpportunity* service package is deployed in the figure, where we can see the Java interface *examOpportunityServiceIF* with its corresponding methods and parameters and its implementation. We can also see the properties corresponding to the three configuration files.

Figures 11, 12 and 13 show the property models obtained, *detailedInfo* property is explained as follows:

- An aspect, *examOpportunity_bringForwardExam_detailedInfo*, will be generated for *detailedInfo* in the service side. As we can see in *Figure 11* its attributes *target* will be the method *bringForwardExam*, for which we are aware they have two parameters –*data* and *Email*. For inspecting the remaining attribute values in the EMF environment, we would have to click on the aspect element so that the remaining values would be shown in the Eclipse property window. If we did so we would see that *actionType* has the value *instead*, and *ack false*.
- Regarding the policy element, as represented in *Figure 12*, its name will *be detailedInfo_ao4ws*. If we inspected the property window we would see that its *optional* value is *true,* for *policyAttachmen*t, *targetType* is *operation* and *targetName bringForwardExam*.
- Due to its optional nature, we ought to include code whose function is to check whether *detailedInfo* has been selected: the corresponding *SOAPTag target* will be *bringForwardExam*, its value *detailedInfo* and it will operate as a *side service* – depicted in *Figure 13*.

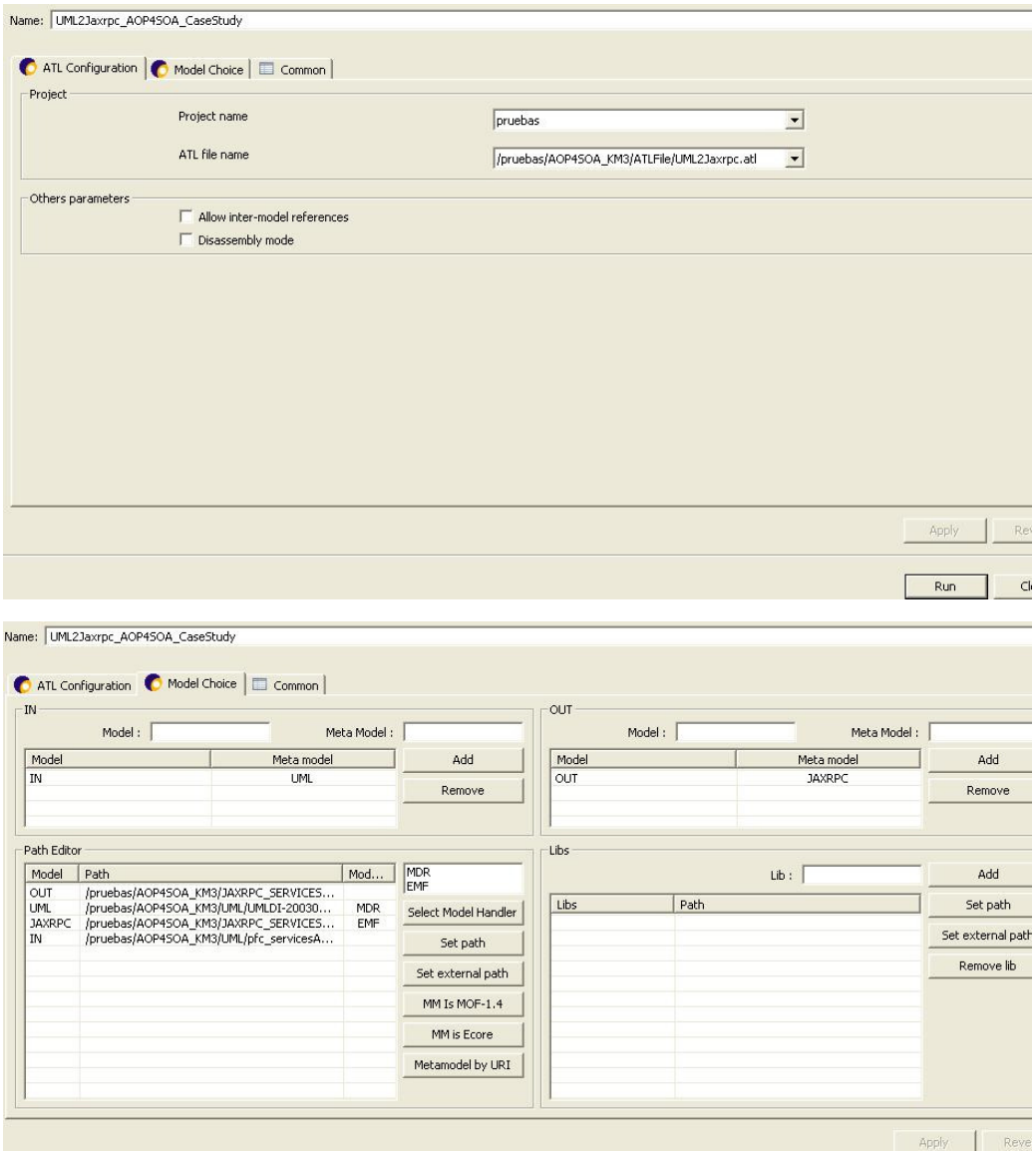

Figure 9. Eclipse configuration for model transformations.

VI. CODE LAYER: EXTRA-FUNCTIONAL PROPERTY GENERATED CODE

Once we have our models for the case study we may also apply additional rules to generate code from them. For this purpose we will make use of the platform-specific models obtained in previous section as the source for the PSM-code transformation. The developer may have modified any attribute value should they be necessary in the eclipse environment.

In order to generate the code, we again use the Eclipse environment. We have created a project in which the Ecore files obtained in previous subsections are included. In this project the UML, JAXRPPC, ASPECT, POLICY and SOAPTAG metamodels are also present, together with the set of transformation rules corresponding to this stage of development.

In order to execute the transformation we have to configure the running environment. First of all the ATL file invoking the transformation rules is selected, then the source metamodels and models are indicated, as well as the location of the libraries with the complete set of transformation rules, as shown in *Figure 14*. In this figure the ALL2STRING transformation is chosen, in this sense we have four source metamodels, JAXRPC, ASPECT, POLICY and SOAPTAG, and their respective models. We have also included six separate libraries with transformation rules: one library is included for each type of metamodel, one more in order to maintain the code generation for the compilation and deployment files separate from the implementation and description code, and one more in order to generate the full code of well-known properties.

From the service specific model, where additional attribute values can be added or modified (e.g. *deployment endpoint*),

Figure 10. Jax-rpc PSM model.

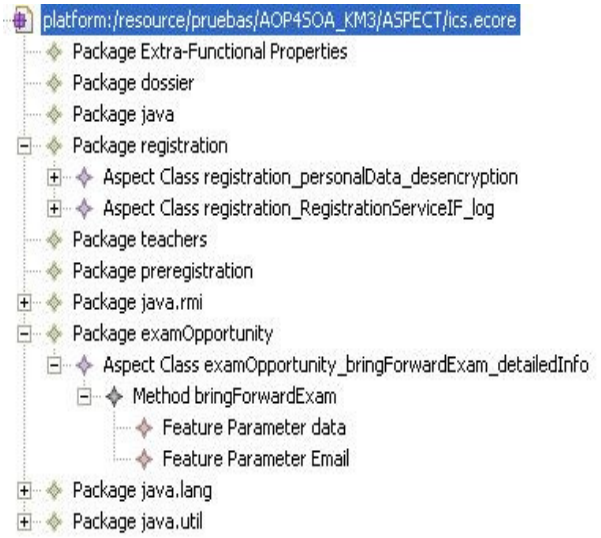

Figure 11. Aspect-based PSM model.

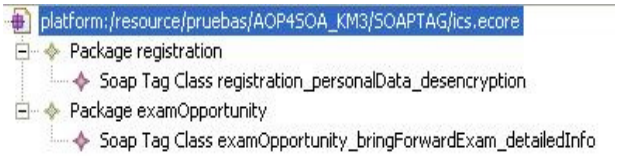

Figure 12. Soap tag-based PSM model.

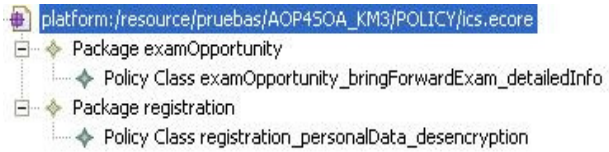

Figure 13. Policy-based PSM model.

the JAX-RPC service skeleton code for JWSDP compilation and deployment is generated. In this sense, the Java interface and implementation skeleton will be generated and complete configuration files created. *Figure 15* shows the Java interface generated for *examOpportunityService*.

From the property models, transformation rules will generate skeleton code for the three extra-functional property model elements: *Figure 15* shows code generated for *detailedInfo*. However, in the case of well-known or user-defined properties, a repository with specific code may be maintained to generate additional code for the three of them. In these cases, in which *ack* is *true*, it is possible to generate the advice functionality and further policy content.

Regarding property implementation, Java code will be generated to check if soap tags are included in the SOAP message and AspectJ has been chosen for the implementation of the property's functionality, consequently properties remaining well modularized and decoupled from implemented applications, as demonstrated in [11]. An AspectJ aspect will be generated for each aspect class in our model. AspectJ pointcuts will be determined by target element's execution. Concerning the advice, depending on the *actionType* attribute value, *before*, *after* or *instead*, the advice type will be *before*, *after* or *around*, respectively; its name will be the one in the *action* attribute. With regard to property description, it is proposed to generate the WS-Policy documents for each property, integrated with the aspect-oriented generated properties as explained in [12]. In this sense, an xml file based on the WS-Policy and WS-PolicyAttachment standards is generated. The policy is attached to the stereotyped element in the PIM, represented by the attribute *targetName* in the policy specific model.

VII. Related Work

As regards Web Service modeling proposals, such as [16] and [4], it can be noted that most of the literature in this area tries to find an appropriate way to model service compositions with UML. The research presented by J. Bezivin *et al.* [4] is worth a special mention; in it Web Service modeling is covered in different levels, using *Java* and *JWSDP* implementations in the end. It is also worth mentioning the paper from M. Smith *et al.* [15], where a model-driven development is proposed for grid applications based on the use of Web Services. Our work differs from these in the sense that ours provides the possibility of adding extra-functional properties to the services and is not oriented to the service modeling itself; therefore it could be considered as complementary to them. We can mention the ASG (*Adaptive Services Grid*) project, which takes into consideration some specific extra-functional properties in their WSDL-centric model-driven development [14]; however, services and properties have to be initially described by a semantic language, and, being a WSDL-centric approach from the very beginning, the possibilities of implementation for the services described are limited.

Concerning proposals which focus on extra-functional properties, we can especially mention two. To begin with, WSMF from D. Fensel *et al.* [6], where extra-functional properties are modeled as pre and post conditions in an ontology description. Secondly, L. Baresi *et al.* extend WS-

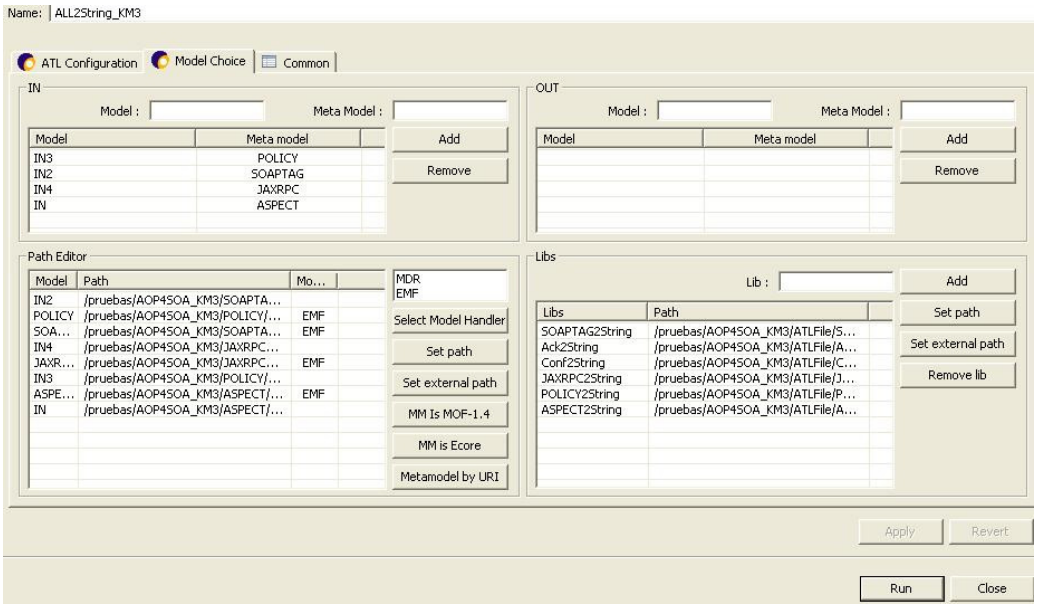

Figure 14. Eclipse configuration for code generation.

Policy by using a domain-independent assertion language in order to embed monitoring directives into policies [2]. Both are interesting proposals, however they do not follow the UML standard, which we consider essential for integrating properties in future service models.

Many policy-related contributions are emerging as policies are a very popular issue nowadays. Among them we can especially remark the contribution from T. Gleason *et al.*, which provides very interesting discussion on policy management [7]. On the other hand, plenty of literature can be found on model-driven development for web service compositions (for instance [5]); our proposal aims at providing support for extra-functional properties in isolated or composed services, and therefore could be complementary to the named proposals.

VIII. Discussion and Conclusion

This paper has shown a case study in which a model-driven approach to web service and their extra-functional properties development has been followed. Several issues arise for discussion:

- First of all, it is important to remind that the profiles provided for PIM level are motivated and further discussed in [10], as previously said. We wish to mention that both profiles attempt to keep the modularization and decouplessness of the different level elements in our model from this initial stage of development.
- Secondly, four different specific metamodels are used at PSM stage in order to maintain the service development independent from the property one on the one hand (Jax-rpc metamodel), and, on the other, in order to keep the properties decoupled from the implementation (aspect metamodel) and description (policy metamodel) perspectives. Besides, more versatile services are provided when the extra functionality is optional for the client, thus it ought to be possible for properties to be selected somehow in a transparent way for the service (soap tag metamodel). This last case is especially suitable for domain specific properties.
- Concerning the generated code, AspectJ has been used for the implementation of the property functionality, thus maintaining properties well modularized and decoupled from the services implemented as demonstrated in [14], where additional elements are also necessary for optional property inclusion. With regard to description, it is proposed to generate the WS-Policy [1] documents for each property, which are integrated with the aspect-oriented generated properties as explained in [15]. This allows properties to remain decoupled not only at modeling stage, but also during implementation. Besides, SOAP Tags will be used to select optional properties and transfer the additional data necessary due to the property inclusions in a transparent way.
- Moreover, having service and property metamodels separated, model-driven transformations remain simpler, but still complementary.
- Besides, traceability is maintained from the very independent model to code, so properties are easily eliminated or added from any stage of development at any level of abstraction, without damaging the remainder of the system.
- Finally, regarding performance, it is important to mention that no payload is included due to the aspect-oriented implementation as it is a static approach.

```
**********EXAMOPPORTUNITY SERVICE INTERFACE*******
package examOpportunity;
public interface ExamOpportunityServiceIF extends
Remote {
    public ArrayList courseList (String
        Qualification) throws RemoteException;
    public String bringForwardExam(HashMap data,
        String EMail) throws RemoteException;
    public String cancelExam(HashMap data, String
        Email) throws RemoteException;}
```

```
***************DETAILED INFO  ASPECT**************
public aspect
pportunityExam_bringForwardExam_detailedInfo {
pointcut bringForwardExam_detailedInfoP
   (data: hashMap, Email:String): execution
   (public     *.opportunityExam_bringForwardExam
   (HashMap, String)) && args(data, Email);

String around ((data: hashMap, Email:String):
bringForwardExam_detailedInfoP (data, Email){
  if ((((String)opportunityHandlerHandler.
  operDetailedInfo.get("operationName")).compareTo
  ("bringForwardExam") ==0) &&(((String)
  opportunityHandler. operdetailedInfo.get
 ("propertyName")).compareTo("detailedInfo")==0))
   { [...]
       [functionality to be completed]    [...]}
    else result=proceed(data, Email)        […]}
```

```
***************** DETAILED INFO POLICY***********
<wsp:PolicyAttachment >
<wsp:AppliesTo>[…]
<wsp:Operation  Name= bringForwardExam/>[…]
</wsp:AppliesTo>
<wsp:Policy name=detailedInfo_ao4ws [...] ">
        <[to be completed]/>
</wsp:Policy></wsp:PolicyAttachment>
```

```
***************SERVICE SIDE SOAP CODE************
if element.getElementName().getLocalName().
 equals ("operationName")){
  String operationName = element.getValue();
  operdDetailedInfo.put("opName",operationName);
  Iterator iter2= element.getAllAttributes() ;[…]
  if (name.getLocalName().equals("propertyName")){
    String propertyName=
        Element.getAttributeValue(name);
    operDetailedInfo.put("propertyName",
        propertyName); }  }
```

Figure 15. Generated Code.

In regard with our present and future work, we are extending our approach in order to cover Quality of Service monitoring. Further information on this topic can be found in [8].

REFERENCES

[1] Bajaj, S., Box, D., Chappeli, D., et al.. Web Services Policy Framework (WS-Policy), ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf, September 2004

[2] Baresi, L. Guinea, S. Plebani, P. WS-Policy for Service Monitoring. VLDB Workshop on Technologies for E-Services, Trondheim, Norway, September 2005

[3] Beisiegel, M., Blohm, H., Booz, D.,et al. Service Component Architecture. Building Systems using a Service Oriented Architecture. http://download.boulder.ibm.com/ibmdl/ pub/software/dw/specs/ws-sca/SCA_White_Paper1_09.pdf, November 2005

[4] Bézivin, J., Hammoudi, S., Lopes, D. et al. An Experiment in Mapping Web Services to Implementation Platforms. N. R. I. o. Computers**:** 26, 2004

[5] Castro, V. Marcos, E. Lopez, M. A model driven method for service composition modelling: a case study, Int. Journal in Web Engineering and Technology, V. 2, I. 4, 2006.

[6] Fensel, D., Bussler, C. The Web Service Modeling Framework WSMF. http://informatik.uibk.ac.at/users/c70385/ wese/wsmf.bis2002.pdf

[7] Gleason, T., Minder, K., Pavlik, G. Policy Management and Web Services, Proc. Policy Management for the Web Workshop at IWWW Conf., Chiba, Japan, May 2005.

[8] Ortiz G., Bordbar B. Model-driven Quality of Service for Web Services: an Aspect-Oriented Approach. Proc. Int. Conf. on Web Services, Beijing, China, September 2008.

[9] Ortiz G., Hernandez J. A Case Study on Integrating Extra-Functional Properties in Web Service Model-Driven Development. Proceedings International Conference on Internet and Web Applications and Services, 2007. Digital Identifier: 10.1109/ICIW.2007.2

[10] Ortiz G., Hernández J., Toward UML Profiles for Web Services and their Extra-Functional Properties, Proc. Int. Conf. on Web Services, Chicago, EEUU, September 2006.

[11] Ortiz G., Hernández J., Clemente, P.J. How to Deal with Non-functional Properties in Web Service Development, Proc. Int. Conf. on Web Engineering, Sydney, Australia, July 2005

[12] Ortiz, G., Leymann, F. Combining WS-Policy and Aspect-Oriented Programming. Proc. of the Int. Conference on Internet and Web Applications and Services, Guadeloupe, French Caribbean, February 2006

[13] Papazoglou, M. Van Den Heuvel, W. Service-oriented design and development methodology, International Journal in Web Engineering and Technology, V.2, Issue 4, 2006.

[14] Roman, D et al. Requirements Analysis on the ASG Service Specification Language. Deliverable D1.1-1, DERI Innsbruck, 2005.

[15] Smith, M., Friese, T. Freisbelen, B. Model Driven Development of Service-Oriented Grid Applications. Proc. of the Int. Conference on Internet and Web Applications and Services, Guadeloupe, French Caribbean, February 2006

[16] Thöne, S. Depke, R, Engels, G.. Process-Oriented, Flexible Composition of Web Services with UML. Int. Workshop on Conceptual Modeling Approaches for e-business: A Web Service Perspective, Tampere, Finland, 2002

[17] Weerawarana, S. Curbera, F. Leymann, F., et al. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More, Ed. Prentice Hall, ISBN 0-13-148874-0, March 2005

This work has been developed thanks to the support of MEC under contract TIN2005-09405-C02-02.