Towards UML Modelling Extra-Functional Properties in Web Services and their Clients[1]

Guadalupe Ortiz, Juan Hernández

Quercus Software Engineering Group
University of Extremadura
Computer Science Department
Spain
{gobellot, juanher}@unex.es

Abstract. Web Services provide our systems with a platform independent and loosely coupled implementation environment, being time to face how the named systems can be modelled. *Service Component Architecture* (SCA) allows us to define services independently of the final implementation technology; however, it does not integrate the remaining development stages. Model Driven Architecture provides a method to face all stages in development from the platform independent model to final code, although it is not specific to service technologies. Regarding web service extra-functional properties, WS-Policy establishes how to describe them in a loosely coupled manner; however the loosely coupled environment is not always maintained when modelling or implementing these properties, which can be solved by using aspect-oriented techniques. In this paper, we propose to use a model driven approach for extra-functional properties in SCA service based models, where generated code will consist of the policy description and an aspect-oriented implementation.

Keywords. Extra-Functional property, web service, UML modeling, aspect-oriented techniques, WS-policy, service component architecture.

1 Introduction

Web Services provide a successful way to communicate distributed applications, in a platform independent and loosely coupled manner, providing the systems with great flexibility and easier maintenance. At present, academy and industry are focusing on the modelling stage, where it is also pursued to keep the loosely coupled platform independent notions [22]. Among the rising proposals, some focus on representing the service as a component and others on basing the model on WSDL elements; two representative approaches are described below:

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

To start with, Service Component Architecture provides a way to define interfaces and references independently of the final technology of implementation, which will be bound subsequently [6]. Based on SCA, services are modelled initially as components linked to a given interface, which can be later specified in a particular type of interface. Besides, the components will show the references they need to complete the behaviour successfully. Thus, based on this proposal, modules remain decoupled as well as avoiding being linked to a specific platform. However, this proposal does not face how to integrate this definition in all the stages of development, as no way is provided to transform the named independent model into the final selected implementation.

In addition, many proposals are emerging in the literature where Model Driven Architecture approach (MDA) is being applied to Web Service development. MDA has been proposed to facilitate the programming task for developers by allowing to generate code automatically from the application model. 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 desired models and eventually into final code, but it does not provide a specific way for service modelling.

Let us consider now that we want to provide our modelled services with extra-functional properties. It is suggested by the SCA specification that this type of property should be modelled at a different level; the way to do so has not been approached as yet. Alternatively, the named MDA proposals do not consider how extra-functional properties may be included in modelled services. Therefore, none of the previous approaches face how to integrate extra-functional properties in service models. In contrast, WS-Policy provides a way to describe them: WS-Policies have emerged as a standardized way for describing extra-functional service capabilities by using the XML standard [21]. 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 policies 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 modelled or implemented, and an additional mechanism would be necessary so as to integrate property modelling and implementation with their description. Properties are currently modelled in UML as any other element in the system, despite their being transversal elements which should be tackled at a different level. This originates dependences from the main functionality service modules to the properties to be added and therefore services' main and extra functionality are tightly coupled.

In this paper we propose to make use of all the described technologies and to join them in order to supply a model driven mechanism to integrate extra- functional properties in a loosely coupled manner at modelling and implementation stage. In this sense, the first aim of this paper is to model services in a versatile and simple manner, according to a proposal based on SCA, which provides a UML environment independent from the platform and from the implementation language in which to integrate extra-functional properties at a later stage. The second goal of this paper is to facilitate extra-functional property modelling and to include these properties in the service model and their clients, maintaining the loosely coupled and platform independent environment. Finally, code could be generated, straight from the model,

code description being the third aim of this paper. Regarding services, there are tools already based on SCA which permit code generation, hence this will not be an issue to be faced in this paper. However, as far as properties are concerned, there is no specific tool for generating their code and description. We propose *AspectJ* to be used for the implementation of the property functionality, thus maintaining properties well modularized and decoupled from the services implemented as demonstrated in [16], where Java classes are also necessary for the inclusion of optional properties. With regard to description, it is proposed to generate the WS-Policy [2] and WS-PolicyAttachment [3] documents for each property, which are now integrated with the aspect-oriented generated properties. This allows properties to remain decoupled not only in the description, but also in the implementation as explained in [17].

The rest of the paper is organized as follows: in *Section 2*, firstly the necessity of an UML-oriented approach to model web services is motivated, then a profile is proposed in order to solve this need. *Section 3* outlines the need to agree on a specific way to model extra-functional properties; the profile proposed in order to do so is presented in *Section 3.2*. *Section 4* provides a case study where both proposed profiles are applied. In *Section 5*, we show how the properties' implementation and description code is generated from the model and we motivate the reason for generating aspect-oriented code and WS-Policy documents. Our proposal is discussed in *Section 6*, whereas other related approaches are examined in *Section 7* and the main conclusions are presented in *Section 8*.

2 Service Modeling

In this section, first of all *Section 2.1* will motivate the need to have a UML approach to web service modelling and *Section 2.2* will present the proposed profile to do so.

2.1 Motivation

There is no need to say how important it is to model systems before implementing them. In the case of service-oriented architecture it is even more important, as there are multiple solutions and technologies which provide us with a final service-oriented system. Therefore, we consider it essential to model this type of system in a simple way, whilst trying to maintain it as general as possible in order to specialize it at a later stage.

As said in the *Introduction*, Service Component Architecture provides a way to define interfaces and references independently of the final technology of implementation [6]. According to SCA, services are modelled as components linked to a given interface, which can be later specified in a particular one. The components will show the needed references for their functionality to be completed, which may be later linked to a required interface. This proposal allows developers to benefit from the following advantages: First of all, a very high level and independent model is defined, allowing the developer to bind it to a specific technology at a later stage. For instance, the interface may be transformed into a Java one or in a WSDL file; similarly, the references can be converted into a bind to a service interface, to an EJB

or to any other type of element, though the transformation mechanisms are not provided by the time being. Secondly, the model can be implemented by using different approaches, therefore allowing adaptability to the customer's specific needs, or to the most suitable option for its integration in a specific environment, as well as providing the possibility of transforming it into a specific model. Thirdly, the model can be converted into Service Component Definition Language code, thus providing an intermediate language among different models, which may be used to integrate different party models into a unique system.

It is due to all these factors that we decided to implement a reduced profile in UML based on SCA to avoid complex models and to maintain standardization as much as possible. Besides, this proposal lets us maintain a platform independent model, which may be later turned into any platform specific model already linked to a particular technology and language as [7], [13], [18] or [20].

2.2 The Service Profile

As shown in *Figure 1*, the service profile is very simple, since we want to raise the simplicity and versatility of the SCA proposal. First of all, we can see the *serviceComponent* stereotype which extends *component metaclass*. Secondly, we can see the *reference* stereotype which extends *port metaclass* and has the attribute *uri* to refer to the URI of the element needed to complete the service funcionality. The elements *provided interface* and *required interface*, also used in the service model, are not defined in the profile as they already belong to the UML syntax.

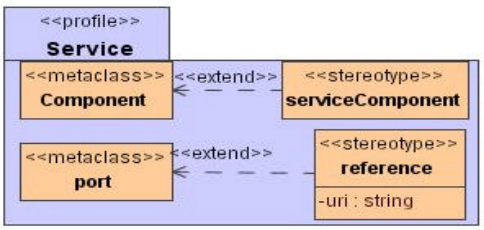

Fig. 1. Service profile.

3 Extra-Functional Property Modelling for Web service Systems

Along this section, our profile proposed to model extra-functional properties for web services and their clients is motivated in *Section 3.1*, and explained in *Section 3.2*.

3.1 Motivation

When thinking about adding further functionality to services through extra-functional properties, we have to consider how to do it whilst maintaining the loosely coupled environment of web service systems. Unfortunately, there is no standard proposal for

modelling extra-functional properties in service development: as indicated in the *Introduction*, SCA specification depicts a few suggestions in this respect, although how to manage it is not established; on the other hand, service policies provide a standardized way for XML describing extra-functional service capabilities; however, it is not an appropriate description for a model stage, nor does it provide property implementation; finally model driven approaches examined do not regard extra-functional properties in their proposals.

Therefore, when adding extra-functional properties in a service model, they are included as any other element in the model, thus causing different category elements to be mixed in it, as shown in *Figure 2*, where generic services and properties have been depicted. The figure shows the multiple dependences from the different services to property classes, which fragment the desirable loosely coupled integration.

Regarding implementation, the need for an alternative to the current implementation of extra-functional properties is motivated in [16], where it is also shown how these methodologies originate intrusive code in services when adding extra-functional properties. It is also demonstrated that aspect-oriented techniques are beneficial in order to solve this problem. Due to space restrictions, we will not go into this matter in depth here as it can be consulted in the named references.

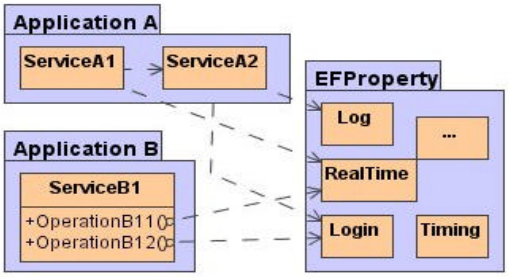

Fig. 2. Extra-functional property dependences.

3.2 The Extra-Functional Property Profile

In order to maintain our system loosely coupled when adding extra-functional properties to the model, we propose the profile in *Figure 3*, whose elements will be explained as follows:

- First of all, we define the abstract stereotype *extra-functional property*, which will extend *operation metaclass* or *interface metaclass*. This means that the stereotype may be applied to an operation – then the specified property would be applied to the stereotyped operation – or it may be applied to an interface –, in which case the property will be applied to all the operations which form the stereotyped interface. The extra-functional property provides five attributes, which will be defined as DefinitionTags of the stereotype: the first one is *actionType*, which indicates whether the property behaviour will be performed *before*, *after* or *instead* of the stereotyped operation's execution – or if no additional behaviour 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 is generated at a later stage; it will have the value *false* when it is a domain-specific property and so only the skeleton code can be generated. Finally, we have two additional attributes, namely *policyID* and *policyDoc*. *PolicyId* will contain the name of an existing policy or the name to be assigned to the new one; *policyDoc* allows the developer to reuse an existing policy document. This attribute will contain the URI where the policy document would be available; if its value is *null* then the WS-Policy document could be generated at code generation stage. The policy attachment document would be generated in every case.

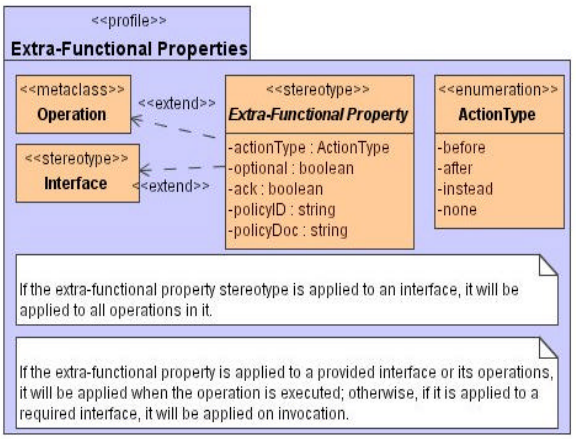

Fig. 3. The extra-functional property profile.

- In order to define *actionType*, an *enumeration* is provided with four alternative values: *before*, *after*, *instead* or *none*. These different values relate to the different options available to perform the properties at implementation time, as they may include new behaviour before, after the stereotyped operation execution or they can even replace the operation's functionality by a different one. In the case of client side properties, *actionType* will have the value *none* for any property which does not imply changes in the client code.
- It is also specified in the profile that if the property is applied in an offered interface, then it will be implemented when the stereotyped operations are executed (as the point from where it is invoked is out of scope). On the other hand, if the property is applied in a required interface, it will be performed when the operations are invoked, as the execution point is out of the service scope. Moreover, the fact that the properties are applied on a required or provided interface will have an additional implication: those which are applied to provided interfaces are the real properties applied, whereas the ones applied to required interfaces are a consequence of the former. For instance, a Login property may be applied in a service offered interface; this implies that the client who requires to use this

operation will have to add the login data in his invocation, also represented in the model by stereotyping the property in the required interface, as is explained in the next section.

- The *extra-functional property* stereotype will be specialized into different stereotypes related to well-known properties or to domain-specific ones in the particular systems modelled. Each property may have additional attributes related to their specific functionality.

4 Applying the Profile in the Case Study

A simple case study is presented in this section with a view to showing how services will be represented and how the properties will be added to it.

Consider that we have a tourist information service, which offers three different operations: the first one, *String hotelInformation (String cityName)*, provides the possibility of getting information about different hotels available in a given city; the second one, *String carRenting (String carType)*, allows us to rent a particular type of car; finally, *String weatherInformation (String cityName)* returns weather information in a destination city.

Let us consider now that we want to include some extra-functional properties, which our system needs, in the *touristInformation* service model. When describing and implementing policies, we will have three types of them: policies which could be considered as informative, as they are always applied and do not imply changes or additional information in the client code; those which could be optional, so they have to be somehow chosen by the client but do no require any extra information; and those which, optional or mandatory, if applied imply changes to client code or need additional information to be supplied. In this sense three examples are provided, one for each of the options:

First of all, as an informative property example, consider a *Log* property, which will be applied to all the operations offered by the service in order to record all the invocation-related information.

Secondly, as an optional property instance, let us consider a *RealTime* property, which will be required discretionarily by the client when invoking *weatherInformation*: subject to a different pricing, the real time weather in a city may be obtained; under the regular price the average weather for the selected date will be obtained.

Finally, one capability which implies additional information to be supplied could be a *Login* property, to be used in order to control access to the *RentingCar* operation, since only those who have a username and password will be able to rent a car. If they are not used during the invocation, the operation will return an error message.

In order to model the describes properties, first of all, we will have to extend the extra-functional property stereotype with the specific properties to be applied, as shown in package *EFProperties* in *Figure* , where we can see *Log*, *Login* and *RealTime* properties, each of them with the additional attributes necessary for their functionality.

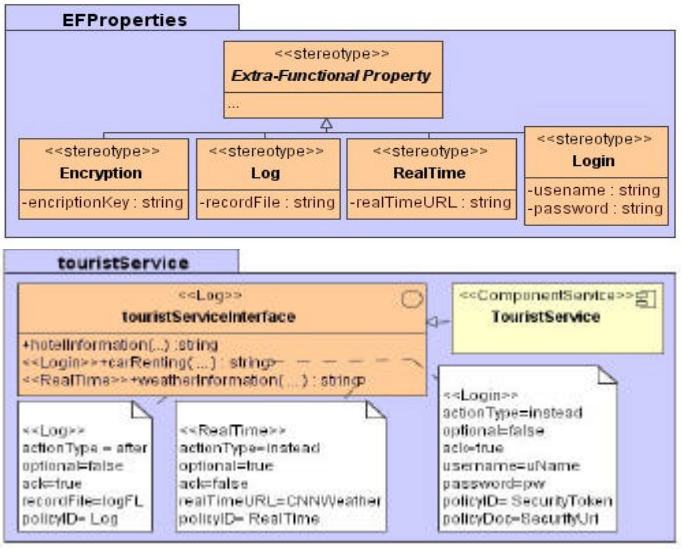

Fig. 4. Model with extra-functional properties.

Regarding the service side, it can be seen in *Figure 4* that properties are added to *touristServiceInterface*. In order to provide all the operations with *Log* in the model, we only have to stereotype the provided interface with the *<<log>>* stereotype. Generally, stereotype attributes are included as tagged values in the model, but in order to visualize them in the printed figures we have also included them as comments. The attributes for *log* in the figure indicate that the property will be performed when any operation in the interface is executed, as it is a non-optional property; l*og* will be performed after the execution of the named operations, since *actionType* is *after*; the information will be recorded in *logFile*; it is a well-known property since *ack* is *true*; *policyID* is *Log* and *policyDoc* is null[2], as it is not specified in the model.

W*eatherInformation*, as previously explained, offers clients the possibility of receiving weather information at real or delayed time, thus it will be stereotyped with *<<Real Time>>*, also depicted in *Figure 4*. The attribute *optional* will have the *true* value, as this property is not always applied; *actionType* will be *instead*, since the property functionality will replace the original operation behaviour if it is selected; *ack* will be *false* as it is not a well-known property, and finally, *realTimeURL* has the *CNNWeather* value, which will be used to obtain real time weather; *policyId* is RealTime and *policyDoc* is null.

Finally, *carRenting* will be performed when a username and a password are provided by the client. This condition will be controlled by the *login* property, also included in *Figure 4* which therefore will be non-optional as the client needs to login for access to this service, and whose *actionType* will be *instead* as it will not allow the operation to be executed unless login is successful and *ack* will be *true*. Lastly,

[2] Null values have not been depicted for simplicity.

username and *password* are *uName* and *pw*, respectively. In this case *policyID* is *securityToken* and *policyDoc* contains an URI from an already existing policy document.

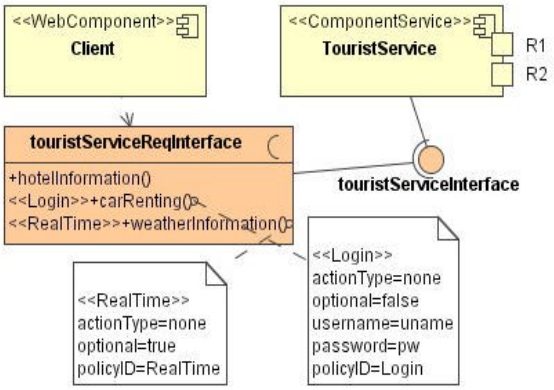

Fig. 5. Property addition in the client side

Regarding the client side, *Figure 5* shows the case study system model, where a client of the *TouristService* required interface has been included. The extra-functional properties *RealTime* and *Login* have been added, this time in the client side. In the next paragraphs, properties included in the client side are going to be discussed. It is important to note that these properties do not have to always appear both in service and client side within the same model, as it may not include both sides; therefore, it is not redundant information, but necessary.

First of all, *WeatherInformation*, as previously explained, offers the client the possibility of receiving weather information at real or delayed time, thus this operation will be stereotyped with <<*RealTime*>>, as depicted in *Figure 6*. As it is an optional property in our case study service, the client has to indicate his interest in the property for it to be applied. This is the reason why the operation *weatherInformation* in the client required interface is also stereotyped with <<RealTime>>, as can be seen in the named figure. In our case study, the attribute *optional* will have the *true* value, as this property is not always applied; *actionType* will be *none* and *ack false*, since no additional behaviour will be necessary in the client, only the *RealTime* selection has to be executed; *policyId* is *RealTime* and *policyDoc* is *null*. *policyId* is necessary to reference the property to be included, whereas *policyDoc* may even be useful to show how to add the information in the message header.

Secondly, the *carRenting* operation will be performed when a username and a password are provided. This condition will be controlled by the *login* property in the service side. In *Figure 6* we can see how the client side operation *carRenting* has been stereotyped in the required interface, thus indicating the need to include the username and password on invocation. This property will be non-optional as the client needs to login for access to this service, and its *actionType* will be *none* and *ack false* as it will not include a new behaviour in the client side, but only new information – username

and password – will be added to the invocation data; finally *policyID* is *Login* and *policyDoc* is null. *policyDoc*, for instance, may be known at later stages and then included in the model, as an informative value.

We could have additional policies, as *Encryption*, for instance, where aspects would also be necessary in the client side. Although we have not included any properties of this type in our case study, its inclusion would be analogous to those previously examined.

5 Implementing the Extra-Functional Property Model

Once our system is modelled, we may desire to generate code from it. As was mentioned earlier, different types of implementation could be generated from the service profile. It is not the aim of this paper to provide a methodology to convert the service model into a specific one or particular code, as justified in *Section 6*.

Regarding extra-functional properties we can choose among various alternatives; if we are going to use a platform which deals with this type of property we only have to generate the policy documents; on the contrary, if the platform does not deal with these properties or we want to tackle them ourselves, we could generate some code which performs the property behaviour plus policy documents. In this paper, we choose the second approach, where we have opted for an aspect-oriented implementation of the properties.

AspectJ has been chosen among the different aspect-oriented languages to illustrate our examples, but any other aspect-oriented language could have been used. To generate an aspect code we have to determine the point in the original system where we want to introduce the new behaviour and the named new behaviour to be included, which are called *pointcut* and *advice* respectively in AspectJ (further information on *aspect-oriented programming* (AOP) can be found at [10] [14]). In the case of our properties, the *pointcut* is the execution of the stereotyped operation in offered interfaces and the invocation of the stereotyped operations in required interfaces. Regarding the advice, depending on the *actionType* attribute value, *before*, *after*, *instead* or *none*, the advice type will be *before*, *after* or *around*, respectively, or they will be no aspect at all. The functionality is determined by the well-known property if *ack* is true, otherwise only the advice skeleton will be generated. Regarding property description, we have opted for implementing it by using the WS-Policy standard and attaching it by using the WS-PolicyAttachment standard. To do so we generate the policy model using the specification data related to the known property if *ack* is true; if false then only the skeleton is generated. The policy attachment is done by using the stereotyped element.

Therefore, for each property stereotype in the service side an aspect, a policy, and SOAP header-related information may be generated. *Figure 6* depicts the skeleton of a general property in order to show the obtained code for the referred elements. As shown in the figure the aspect name is obtained by linking property name and the stereotyped operation name. Pointcut name is obtained in the same way, but adding a 'P' at the end; parameters from the operation will be generated when the *action* is *around;* as explained in *Section 3.2* all pointcuts in service side are *execution* ones,

followed by the name of the stereotyped operation. Regarding the advice, the type is obtained from *actionType*; then the pointcut name and necessary parameters are used. Concerning the policy, it can also be seen how the name of the property and stereotyped element are used for its construction, also including whether or not it is optional. Finally, the tag to be checked in the SOAP Header is the property name.

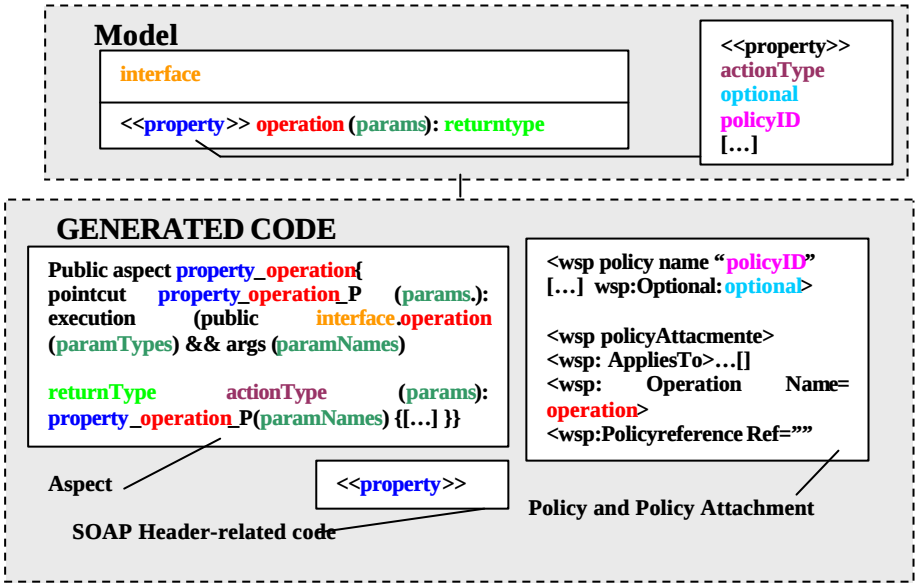

Fig. 6. Code generation from model

Let us now examine the code generated for the case study properties. The aspect for the *Log* property, shown in *Figure 7.1*, contains a *pointcut* corresponding to any execution of the interface operations, the *actionType* is *after* and the method is determined by the well-known property functionality, using the log file provided. since it is a mandatory property, there is no need to check whether the client has chosen it. The case study policy document for the *Log p*roperty would be the one in *Figure 7.2*, where we can see the policy description (in a fictitious specification). Finally, the corresponding policy attachment would be the one shown in the same figure. The same would apply to the remaining properties. We remark, for instance, the optional attribute included in the *RealTime* policy (*Figure 7.4*) and how the new behaviour may be included in the *RealTime* aspect code depending on the client's choice (*if-else* structure in *Figure 7.3*), although only the skeleton is generated for *RealTime* as *ack* was *false*. Regarding *Login* aspect (*Figure 7.5*) and policy (*Figure 7.6*) we can mark that the latter may be described by using a standard specification (*WS-Security*). In this case, the policy code would have been recovered from the URI provided in *policyDoc* in the corresponding stereotype, rather than generated.

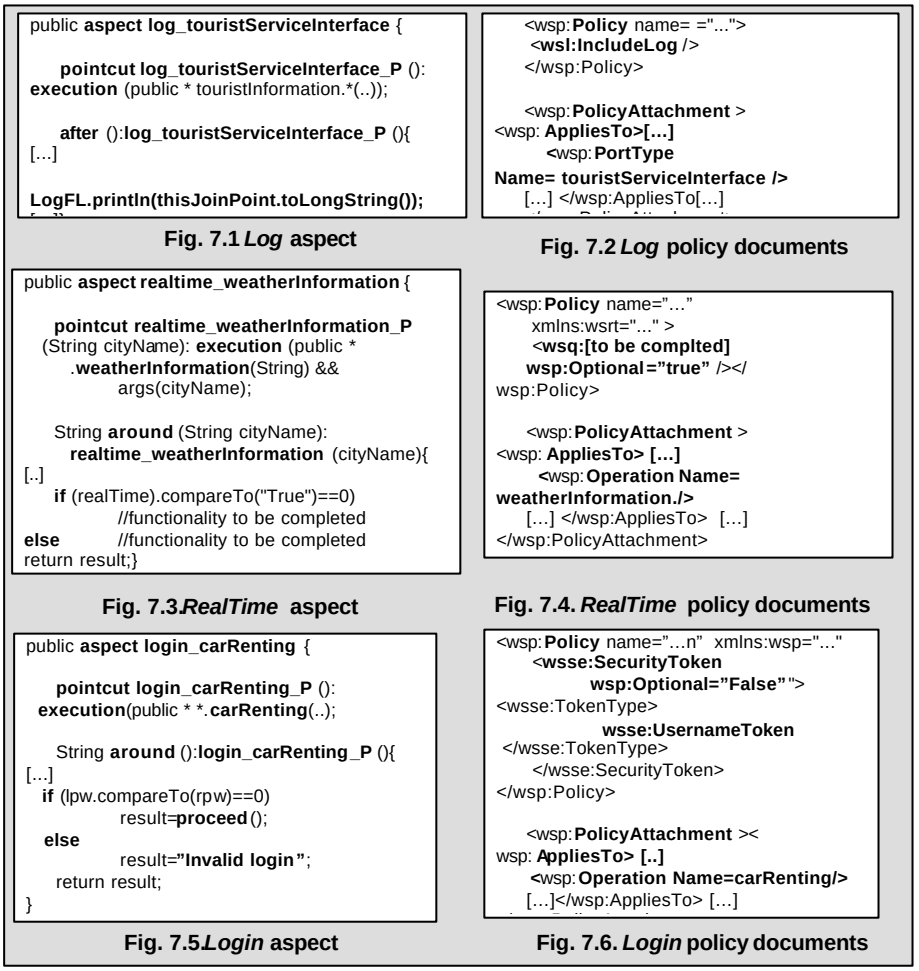

Fig. 7. Properties implementation and description.

As far as property selection is concerned, we have mentioned we are going to include new information in the SOAP message header. In this sense *Figure 8* shows the information added to the headers for the three properties in our case study. No information has to be added to the SOAP Header for *Log*, as it is not an optional property and it does not require any additional information from the client. For *RealTime*, we only need to indicate that we want the operation to be applied, which will be done by including the tags in *Figure 8.1*. This has to be done because the stereotype attribute *optional* had the *true* value. For *Login*, we need to include the username and password, shown in *Figure 8.2*. We have to bear in mind that when talking about well-known properties, associated tags are known beforehand, thus facilitating reusability; if we had properties without known associated functionalities, only skeletons would be generated.

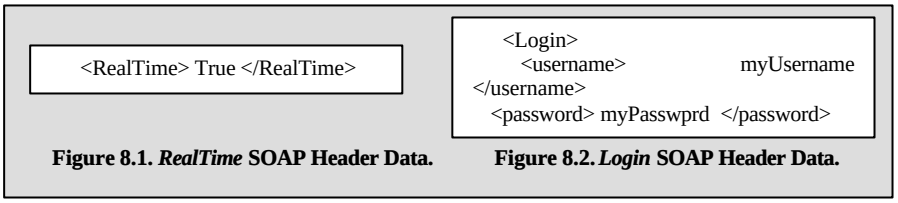

Figure 8.1. *RealTime* SOAP Header Data. **Figure 8.2. *Login* SOAP Header Data.**

Fig. 8. Generated code in the client side.

6 Discussion

Concerning the extra-functional property profile, our starting point is the idea that these types of property are getting more and more specific and defined every day, an unequivocal sign being how some frameworks are already including these properties in their management layer. The profile defines the different properties by using individual stereotypes, which is also comparable to the emerging OMG Quality of Service profile. Finally, it is also clear that many implementation and modelling approaches are using a large number of annotations as a very good way to include or mark new behaviours in a system; we use stereotypes to do so.

With respect to code generation, as said in the previous section, it is not the aim of this paper to provide a methodology for converting the service model into a specific model or particular code. This is because this generation is already supposed to have been achieved by some other approaches or tools available in the market. Besides, despite not generating service code, the code obtained for extra-functional property can be used together to the service one as we are going to explain: services are black boxes which show an interface with a set of operations, therefore, regardless of the final implementation, interface and operation names will remain. Aspects and policies generated only make references to these names, as shown in *Figure 7*, thus providing compatibility between property and service code. Besides, aspect-oriented techniques inherently avoid the use of intrusive code in the main functionality code, thus preserving service code.

Finally, it has been shown that the aspect code or skeleton for property implementation can be generated by using the stereotype information. Thanks to AOP, traceability remains for all the development stages: if we want to delete an aspect in the implementation we only have to delete the stereotype mark for one operation in the model and vice versa. In regard to the policy description, established standards, as WS-Security, may be used for the policy document generation; it may also be necessary to use custom policies, which may be included in the code generator.

7 Related Work

As regards Web Service modelling proposals, such as [13] [20], it can be noted that most of the literature in this area tries to find an appropriate way to model service

compositions with UML and most of them use the WSDL structure for service modelling. The research presented by J. Bezivin et al [7] is worth a special mention; in it web service modelling is covered in different ways, using *Java* and *JWSDP* implementations in the end. It is also worth mentioning the paper from M. Smith et al [18], where a model driven development is proposed for Grid Applications based on the use of web services. Our work differs from these in two respects: first of all, our service modelling proposal is platform independent, while theirs is oriented to a specific implementation; secondly, ours provides the possibility of adding extra-functional properties to the services. In this sense both proposals could be considered complementary approaches, since platform specific proposed models could be generated from our general one, where properties could also be included.

Concerning extra-functional properties, we can especially mention two more proposals. To begin with, WSMF from D. Fensel et al. [11], where extra-functional properties could be considered goals, which implies pre and post conditions in an ontology description. Finally, L. Baresi et al. extend WS-Policy by using a domain-independent assertion language, WSCoL, in order to embed monitoring directives into policies [5]. Both are interesting and thorough proposals, however they do not follow the UML standard, which we consider essential for integrating properties in the different service models.

As far as aspect-oriented modelling is concerned, we can find undoubtedly reference proposals for general aspect modelling, such as [1], [19] and [4]. However we aim to focus modelling within the specific scope of web services, removing unnecessary elements and considering additional requirements, such as policy descriptions.

Among policy and quality-related contributions we can specially remark the contribution from T. Gleason et al., which provides very interesting discussion on policy management [21]. H. Ludwig's work in [15] is also worth a special mention, where he comments on Quality of Service representation status. Representations are mainly by semantic expressions or standard ones (such as WS-Policy or WSLA), which was argued at the beginning of this paper to be complementary with a necessary UML design to consider these properties at the modelling stage. We can also mention a paper that concentrates on a new language, AO4BPPEL, an aspect-oriented extension for BPEL [8]. A. Charfi *et al.* use it for implementing policies in BPEL compositions. Along the same line, we can mention the paper from C. Courbis et al. [24], where a way to weave new capabilities in BPEL compositions is explained. These proposals only centre on service compositions based on BPEL, and are therefore platform specific proposals, whereas we consider that properties should be added to single or composed services, regardless of the final execution or implementation platform.

8 Conclusions

This paper has shown how services and extra-functional behaviours can be modelled in a loosely coupled and platform independent manner by extending UML with profiles. Additionally, the code which may be generated from the model for property

implementation and description has been designed and explained. For this purpose, an aspect-oriented approach has been selected for the implementation and a WS-Policy based one for its description.

9 References

[1] Aldawud, O., Elrad, T., Bader, A. A UML Profile for Aspect Oriented Modeling. OOPSLA 2001 Workshop on Aspect Oriented Programming.

[2] 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.

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

[4] Baniassad, E. Clarke, S. Theme: An Approach for Aspect-Oriented Analysis and Design. 26th Int. Conference on Software Engineer, Edinburgh, Scotland, UK, 2004

[5] Baresi, L., Guinea, S., Plebani, P. WS-Policy for Servi7e Monitoring. VLDB Workshop on Technologies for E-Services, Trondheim, Norway, September 2005.

[6] 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_0 9.pdf, November 2005

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

[8] Charfi, A., Mezini, M., Using Aspects for Security Engin4ering of Web Service Compositions, Proc. Int. Conf. on Web Services, Orlando, Florida, USA, July 2005.

[9] Courbis, C., Finkelstein, A. Towards Aspect Weaving Applications, Proc. at Int. Conf. on Software engineering, St. Louis, Missouri (USA), May 2005.

[10] Elrad, T., Aksit, M., Kitzales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM, Vol.44, No. 10, October 2001.

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

[12] 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

[13] Grønmo, R, Solheim, I Towards Modeling Web Service Composition in UML. Int. Workshop Web Services: Modeling, Architecture and Infrastructure, Porto, Portugal, 2004.

[14] Kiczales, G. Aspect-Oriented Programming, ECOOP'97 Conference proceedings, Jyväskylä, Finland, June 1997.

[15] Ludwig, H., Web Service QoS: External SLAs and Internal Policies Or: How do we deliver what we promise?, proc. at IEEE International Conference on Web Information Systems Engineering, Brisbane, Australia, November 2004

[16] 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.

[17] 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

[18] Smith, M., Friese, T. Freisbelen, B. Model Driven Development of Service-Oriented Grid Applications. Proceedings Advanced Int. Conference on Telecommunications and International Conference on Internet and Web Applications and Services, Guadeloupe, French Caribbean, February 2006.

[19] Stein, D., Hanenberg, S. and Rainer, U.: A UML-based Aspect-Oriented Design Notation for AspectJ. Proc. 1st Int. Conf. on AOSD, Enschede, The Netherlands, 2002

[20] 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

[21] 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.

[22] Zimmermann, O., Krogdahl, P, Gee, C. Elements of Service-Oriented Analysis and Design, http://www-128.ibm.com/developerworks/webservices/library/ws-soad1/, May 2004