Applying AOP Concepts to Increase Web Services Flexibility

Mehdi Ben Hmida¹, Ricardo Ferraz Tomaz², Valérie Monfort^{1,2}

¹ Université Paris IX Dauphine LAMSADE
Place du Maréchal de Lattre Tassigny, Paris Cedex 16
mehdi.benhmida@etud.dauphine.fr

² Université Paris 1 - Panthéon - Sorbonne
Centre de Recherche en Informatique, 90 rue de Tolbiac 75634 Paris cedex 13
{Ricardo.Ferraz-Tomaz, valerie.Monfort}@malix.univ-paris1.fr

ABSTRACT

Web Service is the fitted technical solution which provides the required loose coupling to achieve Service Oriented Architecture (SOA). In previous works, we proposed an approach, using the Aspect Oriented Programming (AOP) paradigm, to increase the adaptability of Web Services. This approach suffers from some deficiencies as dependency for both the programming language (Java) and the SOAP engine (AXIS). In this paper, we propose to increase the adaptability of Web Services by using the main AOP agreed semantics - Advices, Pointcuts and Joinpoints- to change original Web Service behavior. In the new approach, we consider that advices are themselves Web Services. Moreover, we propose to use an XML Language to describe Pointcuts, Joinpoints and for referencing advices. The invocation of advices (Web Services) is accomplished by an XQuery engine to ensure SOAP Engine independency and advices are implemented as Web Services to promote programming language independency.

1. Introduction

Web Services, as other middleware technologies, aim to provide mechanisms to bridge heterogeneous platforms, allowing data to flow across various programs. In a very short time, since Web Services were proposed, an impressive range of supporting tools has been offered that enable the development and deployment of Web Services. However, these tools suffer from three important weaknesses:

- i) The approaches typically used to integrate services in client applications are very static and do not allow any dynamic adaptations of Web Services themselves or the way they are used [16]. Once the service or the client is developed and the handlers are deployed, the Web Service cannot handle new features and it is difficult to reuse it in a different context.
- ii) XML messages [8] are at the heart of the Web Services infrastructure. Nevertheless, current approaches to Web Services development wrap the messaging layer hard coded in Java or C# source code. Consequently, whenever the structure of the messages changes all classes where these messages are codified must be rewritten and recompiled.
- iii) The service environment is very dynamic. References to Web Services are hard wired in the application and any of the service problems (method signature modification or service broken) leads to application execution problems.

Many researches [13,19,21,23] consider *Aspect Oriented Programming* (AOP) [16] as an answer for these limitations. The AOP is one of the most promising solutions to the problem of creating clean and well-encapsulated objects without extraneous functionality. It allows the separation of crosscutting concerns into single units called *aspects*, which are modular units of crosscutting implementation. Moreover, AOP allows adding new behavior to an application without touching the base source code. AOP presents three main concepts: *Joinpoints* - Points in a program execution; *Pointcuts* - Program constructs to designate a set of joinpoints and *advices* - Code that runs upon meeting certain conditions. Aspects are merged to Classes with a tool called Weaver.

In our previous researches, we provided a solution which offers a dynamic mechanism based on AOP to compute the service contract on the fly, enabling Web Services to become fully aware of the business requirements [3]. This approach increases the adaptability of the Web Services behavior depending on the service contract. However, we observed two important limitations: i) SOAP engine dependency and ii) programming language dependency.

To overcome the above mentioned limitations, we propose a new approach for changing the behavior of Web Services by using Aspect Oriented Programming concepts. We assume that *advices are themselves Web Services* and we specify an XML language for describing pointcuts, joinpoints and for referencing advices (Web Services). The problem of adding aspects to the base code turns out to be the problem of weaving aspect services' with the base service. The language independency is achieved by assuming that advices are services and by using an XML language for describing the aspects. The SOAP engine independency is reached by using an XQuery engine [6] to generate the SOAP messages to invoke the advices.

This paper is organised as follows: section 2 presents Aspect Oriented Programming. Section 3 presents an overview of our previous approach and points out some limitations. Section 4 presents our new approach. Section 5 draws comparisons with other works and points out some observed limitations. Finally, we conclude and enumerate some future works in the section 6.

2. Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a programming paradigm which allows decreasing the tight coupling between different technical aspects of the software. AOP concepts were formulated by Chris Maeda and Gregor Kiczales [14].

AOP is an evolutionary way of developing software that improves Object-Oriented programming (OOP), in much the same way that OOP improved procedural programming. OOP introduced the concepts of encapsulation, inheritance, and polymorphism for creating a hierarchy of objects that models a common set of behaviors. OOP falls short, however, in providing a means of handling common behaviors that extend across unrelated objects. That is, OOP allows us to define top-down relationships but is not well suited for left-to-right relationships.

To better clarify, consider the classical example of implementing the logging functionality. Logging code is often scattered horizontally across object hierarchies and has nothing to do with the core functions of the objects it's scattered across. The same is true for other types of code, such as security, exception handling, and transparent persistency. This scattered and unrelated code is known as crosscutting code and is the reason for AOP's existence.

Instead of embedding crosscutting code in classes, AOP allows to encapsulate the crosscutting code into separate module -known as aspect- and then apply the code where it is needed. It achieves the application of the crosscutting code by defining specific places -known as Joinpoint- in the object model. The crosscutting code is injected at the specified Joinpoint.

With OOP, every time we need to introduce the logging functionality in an application part, the programmer must add manually the logging code into the appropriate objects. With AOP, we can insert the logging code into the classes that need it with a tool called Weaver. This way, objects can focus on their core responsibilities. The figure 1 shows the weaving process.

Figure 1: The weaving process

The Weaver is responsible for taking the code specified in a traditional (base) programming language and the additional code specified in an aspect language, and merging them together. The Weaver has the task to process the aspects and the component code in order to generate the specified behavior. The Weaver inserts the aspects in the specified Joinpoint transversally (left-to-right approach).

3. Our Former Approach

Our industrial experience led us to conclude that current Web Service design cannot help Service Oriented Architectures to accomplish full interoperability [3,4]. Non-functional features, such as security, routing, reliability, and transactions, cannot be defined once for all when developing or deploying an application. Otherwise, Web Services will become as monolithic as previous middleware technologies were. In other words, features that Web Services are asked to provide will become strongly coupled with the application. Designers and developers need Web Services that can automatically adapt to policies. The following subsections present our former approach and its limitations.

3.1. Overview

We noticed the existence of two major facts which are at the root of Web Service's lack of flexibility. First, there is no dynamic mechanism to bind policies and Web Service handlers. Second, there is no clean separation of concerns between the functional and the non-functional code, and also between SOAP logic within handlers and non-functional logic within handlers, as figure 2 shows.

Figure 2: Tangled logics within SOAP Services [3]

Once the client or service is developed and the handlers are deployed, the Web Service cannot handle new features and, because the different logics are tangled up, it is not easy for another developer to reuse the application in a different context. Consequently, an appropriate way to deal with these crosscutting concerns [3] would be to use different units of modularization to encapsulate these logics. Moreover, if these units of modularization could be managed by a dynamic mechanism, the whole system would be able to dynamically reconfigure itself depending on the policies.

These requirements lead us to consider AOP as an answer to Web Services reusability issues [4]. Considering Web Services, non-functional handling logic should be encapsulated within multiple aspects. It means that handler's role has to be redefined, as they will only contain SOAP logic. The idea is to replace the multiple specific handlers, which used to process SOAP messages depending on their own implementations, by a global handler whose role will be restrained to extracting non-functional data contained in incoming messages, and pushing it inside outgoing messages.

Figure 3: Aspects weaving at load time.

The weaving process is made as follows: firstly, aspects are weaved to the stub because it provides a natural Meta object to focus on the service itself. Secondly, the weaving is made when the service is loaded to enable one syntactical analysis of the policies document for each new instance. Thirdly, the weaver is an application able to read the policies document, interpreting these policies, select the relevant aspects and finally mix them with the plain stub (figure 3).

3.2. Limitations

As above described, to reach our purpose, we must make some changes in the SOAP engine handlers'. In fact, we must redefine the handlers' role considering only the SOAP logic and encapsulating the non-functional handling logic inside aspects. So, we must redefine the SOAP engine's architecture. This approach was tested on the AXIS SOAP engine, but each SOAP engine has its proprietary architecture. So, our previous approach is *SOAP engine dependent*.

Another drawback of our former approach is that for weaving aspects to the main stub, we must use a specific AOP Weaver. The current Weavers are language dependent. So, if a Web Service is developed in Java we must use a Java-based Weaver as AspectJ [15] or JBOSS AOP [9] to promote the changing.

Regarding these limitations, we propose a new architecture that promotes SOAP engine, programming language independency as described in the following section.

4. New Approach

In the new approach, we propose to weave the non-functional requirements to the original Web Service on a more abstract level: the SOAP messaging level. We propose to use the main concepts of AOP (joinpoint, pointcut and advice) to dynamically change the Web Service behavior. This approach offers to the Web service provider the possibility to add or change dynamically the behavior of a service without having to undeploy, redevelop and finally redeploy it. This is done in a transparent way by intercepting the SOAP messages sent by the customer to the original Web service and redirecting it to a Web Service Weaver. In this section, we present the main ideas of the new approach and the proposed architecture. To better clarify the exposed ideas, we apply our approach to a motivating scenario.

4.1. Main Ideas

We propose to use AOP concepts and not AOP platforms to realize the adaptability of Web Services. This choice is made for many reasons:

Firstly, the time of weaving in AOP paradigm (before, after and around) is convenient and sufficient to handle non-functional requirements for Web Services. The requirements which must to be satisfied occur before, after or around the invocation of one method from the Web Service. The real implementation of the Web Service does not change and it is hidden by the WSDL description. So, the WSDL description is sufficient to identify where we will add a new behavior for the service.

Secondly, the Pointcut concept in AOP can be applied to the WSDL description by using expressions path [11]. Expressions path are patterns which allows addressing parts of an XML document. They allow to navigate in the hierarchy of the XML document and to find a specific path in this hierarchy. So, these expressions allow us to recover a specific method signature or a set of methods signatures that we want to modify the behavior.

Thirdly, we consider the advice concept as a Web Service instead of applying an advice implemented in a specific AOP platform. The advice is invoked by using SOAP messaging. This allows avoiding the main problem related to AOP platforms: the lack of a standard model for specifying and declaring an aspect.

Web Services invocations are usually hard coded in the client side. To overcome this drawback, we propose to use an XQuery engine to generate the SOAP messages invocations for the advices. The XQuery engine ensures us a centralized way to generate SOAP messages instead of spreading them in the client source code. So, the advices invocations are independent of mechanism or specific software artifacts.

4.2. Motivating Scenario

To better understand the ideas described above, consider the following simplistic example:

A financial institution resolves to expose a Web Service to withdraw money from its client accounts. The original Web Service has an authentication policy. After a while, the institution resolves to change this authentication policy.

Using a classical approach, considering that is required a changing in the authentication method signature, the programmer must identify all authenticated methods and rewrite the authentication signature. In other words – the programmer must modify, recompile, retest and finally redeploy the Web Service.

Using our approach, the programmer can, for instance, develop an Advice called “authentication” and specify, in the XML file descriptor, that before the invocation of the method “withDraw” in the original Web Service, the advice “authentication” must be invoked. In the next original Web Service invocation, the engine will be aware of the rules specified in the XML file and will generate an http/SOAP invocation to the advice “authentication” in order to authenticate the user based on the new policy. In this way, no changes are required in the original Web Service. The original Web Service behavior is changed by means of its interactions with the advices.

There is not a real weaving between the original Web Service and the advices. Consequently, each one can be developed in different paradigm, programming language and operating system platform.

4.3. Proposed Architecture

The new approach architecture presents three components (figure 4):

- i) *The XML schema*: it describes how an aspect must be specified and declared.
- ii) *The XQuery Module Generator (XMG)*: it takes as input the aspect XML file and generates the XQuery script to invoke the original Web Service and advices.
- iii) *The XQuery Engine*: it executes the XQuery script generated by the XMG.

Figure 4: Service weaver architecture

These components will be described in depth in the following subsections.

4.4. The XML Schema

Many AOP platforms such as AspectWerkz [7], JBoss AOP [9] and Spring AOP [20] use XML for declaring an aspect. The main difference of our approach is that the advice element references a Web Service (by means of the WSDL file URI). Figure 5 shows the XML Schema to declare an aspect.

An aspect element (lines 3 to 11) is composed of two elements: the “pointcut” and the “Advice”. It has one attribute “name” to identify the aspect. The “pointcut” element (lines 12 to 17) has two attributes: the “name” to identify the Pointcut and the “pattern” which contains an XPath expression for locating a method signature in the WSDL file. The “Advice” element (lines 18 to 27) contains two elements: the “pointcutName”, which references a “pointcut” element already defined and, the “content” element (lines 33 to 38) which references the Web service. The “content” element contains the “URN” (Universal Resource Name) of the service and the “status” attributes which receives true or false depending on the result of the advice execution. The ‘type’ attribute of the “Advice” element specify the type of the advice (before, after or replace) that is a “typeAdvice” (line 39 to line 43).

We define three kinds of advices – “before”, “after” and “replace”. We made the choice to use a “replace” type and not an “around” type because we consider that “around” type is a combination of “before” and “after” types. So:

- i) When a “before” advice is used, an http/SOAP message to the advice is generated before the original Web Service invocation message.

- ii) When an *“after”* advice is used, an http/SOAP message to the advice is generated after the original Web Service invocation message.
- iii) When a *“replace”* advice is used, an http/SOAP message to the advice is generated instead of the original Web Service invocation message. So, the original Web Service is completely replaced by the advice that accomplishes the required task.

```

1.      <?xml version="1.0" encoding="UTF-8"?>
2.      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3.
4.          <xsd:element name="Aspect">
5.              <xsd:complexType>
6.                  <xsd:sequence>
7.                      <xsd:element ref="pointcut"/>
8.                      <xsd:element ref="Advice"/>
9.                  </xsd:sequence>
10.                 <xsd:attribute name="name" type="xsd:String" use="required"/>
11.             </xsd:complexType>
12.         </xsd:element>
13.
14.         <xsd:element name="pointcut">
15.             <xsd:complexType>
16.                 <xsd:attribute name="name" type="xsd:String" use="required"/>
17.                 <xsd:attribute name="pattern" type="xsd:String" use="required"/>
18.             </xsd:complexType>
19.         </xsd:element>
20.
21.         <xsd:element name="Advice">
22.             <xsd:complexType>
23.                 <xsd:sequence>
24.                     <xsd:element ref="pointcutName"/>
25.                     <xsd:element ref="content"/>
26.                 </xsd:sequence>
27.                 <xsd:attribute name="name" type="xsd:String" use="required"/>
28.                 <xsd:attribute name="type" type="typeAdvice" use="required"/>
29.             </xsd:complexType>
30.         </xsd:element>
31.
32.         <xsd:element name="pointcutName">
33.             <xsd:complexType>
34.                 <xsd:attribute name="refName" type="xsd:String" use="required"/>
35.             </xsd:complexType>
36.         </xsd:element>
37.
38.         <xsd:element name="content">
39.             <xsd:complexType>
40.                 <xsd:attribute name="urn" type="xsd:String" use="required"/>
41.                 <xsd:attribute name="status" type="true|false"/>
42.             </xsd:complexType>
43.         </xsd:element>
44.
45.         <SimpleType name="typeAdvice">
46.             <restriction base="string">
47.                 <pattern value="before|after|replace"/>
48.             </restriction>
49.         </SimpleType>
50.     </xsd:schema>

```

Figure 5: The XML schema

Considering the motivating scenario, the XML file descriptor that contains aspects for the original Web Service would be like this (Figure 6):

```

1.    <Aspect name="authentication">
2.        <pointcut name="Withdraw" pattern="*"WithD*" urn="urn_wsdl_service2"/>
3.        <Advice name="Authentication" type="before">
4.            <pointcutName name="Withdraw" />
5.            <content urn="urn_wsdl_service1" status="true" >
6.        </Advice>
7.    </Aspect>

```

Figure 6: Part of the XML file descriptor

This XML file descriptor asks the engine to invoke the advice “authentication” before invocation of any method whose signature matches the regular expression “*WithD**”. The original method invocation is conditioned by the answer obtained from the authentication method. The method must be invoked, only if the status is true.

4.5. XQuery Module Generator

The XQuery Module Generator receives as input the XML file descriptor depicted above and generates the XQuery script which makes the invocation of the original Web Service and the advices.

Figure 7: XQuery Module generator

The XQuery Module Generator (Figure 7) is composed of two components:

- i) The “WSDL Parser” parses the WSDL files to obtain information as: ports, method signatures, etc.
- ii) The “XQuery Generator” takes as input the aspect XML description and interacts with the “WSDL parser” to generate the XQuery script.

Based on the XML file descriptor, depicted in the figure 6, the engine must generate the following XQuery script (Figure 8):

```

1. import service namespace Service1 = "urn:urn_wsdl_service1" name "Authentication";
2. import service namespace Service2 = "urn:urn_wsdl_service2" name "Withdraw";
3. let $i Service1:Authenticate (UserId, Passwd);
4. if $i/result=true then
5. let $j:= Service2:Withdraw(AccountNumber, ValueWithdraw);

```

Figure 8: XQuery script

The line 1 imports the Web Service named “*Authentication*”. This Web Service has the method signature “*authenticate (userId, passwd)*” that returns a *Boolean* depending on the result of the authentication. The line 2 imports the Web Service named “*Withdraw*”. This Web Service has the method signature “*Withdraw (AccountNumber, ValueWithdraw)*”. In the line 3, the method “*authenticate*” is invoked. The XQuery engine automatically generates the SOAP messages based on the WSDL description localized in the URN (Uniform Resource Name) specified in the import statement (line 1). The line 4 tests if the result returned by the first Web service is true. The line 5 invokes the “*Withdraw*” if the condition at line 4 is satisfied.

Although the engine can generate complex interactions, based on a well defined set of rules, among original Web Services methods and advices, the mechanism to accomplish the interactions is simple.

For example, when a “before” advice is detected, the engine intercepts the http/SOAP request to the original Method (1) and looks for aspects defined on this method in the XML file descriptor (2). If such an aspect exists, then the weaver redirects the request to the proper advice (3). Then, it captures the advice http/SOAP answer (4) and sends it to the original invoked Web Service method (5) that performs the task and sends the answer to the engine (6). Finally, the engine dispatches the final answer to the client (7). This process is showed in figure 9.

Figure 9: Schema interaction for a before advice

Whenever an “after” advice is detected, the engine intercepts the original method http/SOAP answer (4) and redirects it to the advice (5). The advice performs some tasks and sends an answer to the engine (6). Finally, the engine dispatches the final answer to the client (7). This process is showed in figure 10.

Figure 10. Schema interaction for an after advice

Finally, whenever a “replace” advice is detected, the engine intercepts the http/SOAP request to the original Web Service (1) and redirects it to the advice (3); the advice performs the required task and sends an answer to the engine (4). Finally, the engine dispatches the final answer to the client (5). The original Web Service behavior is completely replaced by the advice behavior. As we can observe in the figure 11, there are not arrows to the Web Services Provider (where the original Web Service is stored).

Figure 11. Schema interaction for a replace advice

4.6. XQuery Engine

XQuery is a concise and flexible query language for XML. XQuery combines important aspects of the last twenty-five years of research in languages development and data management, such as functional semantics, formal type systems and composability. XQuery is a powerful language that represents to XML world what SQL represents to relational database [6].

XQuery makes a similar promise as Web Service: to present a single front-end to all of the databases and data sources available on the network. In both cases, the goal is to bring together the processes and data, we interact with individually, in a manner that reduces the amount of custom mapping required for each source. In [17], the author demonstrates how Web Services and XQuery combined together can facilitate the task of developing applications on top of heterogeneous distributed systems.

Since XQuery work draft was published in May 2003 [6], many XQuery implementations have incorporated the notion of modules. One of the vanguard works was developed at IBM labs [17]. Nevertheless, nowadays there are a lot of commercial tools such as XQuery Stylus Studio[12], TigerLogic [18] and Liquid Data [5] that implement this feature. A Web Service can be imported in a module. Once the Web Service is imported, its methods are available and, when it is invoked, the SOAP messages are generated automatically by the XQuery engine. The tools above mentioned occult SOAP messages generation details for the programmer. Using a XQuery engine, it is possible to combine Web Services as pipes (Software development philosophy introduced for UNIX operating system where complex tasks are performed by bringing together a collection of simpler tools), routing the output of a Web Service as input for another Web Service. This characteristic suggests XQuery as an efficient mechanism to weave the aspects to the original Web Services.

5. Discussion and Related Works

One important characteristic of this work is that our proposal requires minimal changes on the existing infrastructure.

There are many XQuery engines that implement interoperability with Web Services. In addition, many tools vendors want to become full compliant with the W3C specification.

AOP is widely used to improve component based software development. There are many available tools to implement AOP techniques in several programming languages. Nevertheless, these tools are language or platform dependent. For example, AspectJ [15] extends Java and Jasco.Net [24] extends C#. Our approach is language independent because we consider that advices are Web Services.

The Web Service Management Layer (WSML) is a platform for Web Services allowing a more loosely coupling between the client and server sides [23]. This platform is based on JasCo (an AOP platform) [22]. The idea of this technology is to transfer the Web Service related code from the client code to this new management layer. The advantages are the dynamic adaptation of the client to find the most fitted Web Service, and it also deals with the non-functional properties like Traffic Optimization, Billing Management, Accounting, Security, Transaction. Regarding our approach, the difference is that we use the main concepts of AOP (advice, pointcut and joinpoints) and not a ready to use AOP platforms and we developed our own weaver based on XQuery.

The Aspect-Oriented Component Engineering (AOCE) [19] has been developed to capture the crosscutting concerns, such as transaction, co-ordination and security, etc. To achieve this solution, the WSDL grammar was extended by enriching it with aspect-oriented features so that it becomes better characterized and categorized. However, there are no universally accepted standards in the terminology and notations used in AOCE by the various interested parties trying to use it. Regarding our proposal, it does not require changes in the existing Web Service Standards.

There are non AOP-based approaches to improve the Web Service flexibility.

The Web Services Mediator (WSM) is a middleware layer that sits above standard Web Services technologies such as Simple Object Access Protocol (SOAP) Servers [10]. It aims to decouple an application from its consumed Web Services, and to isolate the application's characteristics (e.g., reliability, scalability, latency).

The Web Service Description Framework (WSDL) consists in a suite of tools for the semantic annotation and invocation of Web Services, by mixing both Web Service and Semantic Web communities [25]. Instead of establishing a hard wired connection between the client and the service, by specifying the Web Services through addresses, WSDL enables the developer to formally specify a service using rules and ontological terms.

Our approach is coarse-grained and provides a non-intrusive way to hot fix Web Services. However, the non-invasive approach produces an important limitation to change existing behavior developed as private methods, that are executed internally for the public methods. For instance, if in the above sample the “authentication” method was codified as a private method and it was hard coded inside the “withDraw” method, we can not change this behavior using our approach.

So, to better use our approach, the developer must identify, a priori, the change-prone behaviors and codify them as public methods.

Using our approach, it is possible to add new functionalities seamlessly. But, to remove or change existing ones, they must be implemented as public methods and have public signatures described in WSDL.

We advocate the development of patterns in accordance with the early Aspects initiative [2] to modularize efficiently the change-prone behaviors and codify them as public methods.

6. Conclusion and Future Works

This paper presents how to apply AOP techniques, to increase hot fix capacities of SOA, using Web Services as assembly blocks to build advices. We propose an aspect-oriented approach to support the adaptation of existing services to different needs. For example, authentication can be added seamlessly to a non-authenticated service using aspects. We assume that the aspects (such as authentication) are themselves Web Services. Thus, the problem of weaving aspects to base code turns out to be the problem of composing aspect services with base Service. We propose an XML specification for weaving these aspects. This specification is automatically translated into an XQuery script. This XQuery script can, henceforth, be executed on any standard XQuery engine.

Our approach is very useful to add new functionalities and presents some limitations to remove or change behavior implemented as private methods. We propose to modify the Web Services behavior by means of its interactions. As private methods do not interact with the external world, we cannot change the behaviors implemented internally as private methods.

These limitations are resulted from our choice for a coarse-grained model. Our option is justified for the main idea behind Web Services paradigm: to minimize what the applications need to know to interact meaningfully ones each other.

Currently, we are finishing the development of our first prototype as proof-of-concepts. We are developing another prototype using a BPEL4WS [1] engine (a language for Web Service Composition) instead of using an XQuery engine.

In fact, we can see AOP weaving as a simplistic manner to compose Web Services. The weaving between these advices and the original Web Service constitutes a composite Web Service. Another motivation to use BPEL4WS is that BPEL4WS is a general agreed standard for service composition. The XQuery specification is not yet a W3C standard and in spite of that XQuery has already demonstrates to be a good alternative to combine Web Services, this task is not its main purpose. We will continue using XQuery to generate the BPEL4WS script.

Another direction we are investigating is to develop design patterns regarding early aspects initiative [2] know-how to better identify the change-prone behaviors and modularizes their implementations as public methods.

7. References

- [1] Andrews, T., Curbera, F., Dholakia, H., Golland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., & Weerawarana, S. (2003). BPEL4WS Version 1.1 Specification. FTP available at <ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf>.
- [2] Araujo, J., Baniassad, E., Clements, P., Moreira, A., Rashid, A., & Tekinerdogan, B. (2005). Early Aspects: The current Landscape. Web site available at <http://www.comp.lancs.ac.uk/computing/users/marash/early-aspects/landscapereportv1.pdf>.
- [3] Baligand, F., & Monfort, V. (2004). A concrete solution for Web Services adaptability using policies and Aspects. Proceedings of the 2nd international conference on Service oriented computing (ICSOC) .ACM press. 134-142.
- [4] Baligand, F., & Monfort, V. (2004). Standards and Web Services: some concrete limitations. Proceedings of the International IEEE Symposium on Web Services and Applications (ISWS'04).

- [5] BEA. Web site available at http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/liquid_data.
- [6] Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., & Siméon, J. (2005). XQUERY 1.0: An XML query language. Web site available at : <http://www.w3.org/TR/xquery/>.
- [7] Boner, J. (2004). AspectWerkz- Dynamic AOP for Java. Proceedings of the third International Conference of Aspect-Oriented Software Development (AOSD'04).
- [8] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2004). Extensible Markup Language (XML) 1.0 (Third Edition). Web site available at <http://www.w3.org/TR/REC-xml>.
- [9] Burke, B., Fleury, M., Brock, A., Hussenet, C., Khan, K., Culpepper, M. (2004). JBOSS Aspect Oriented Programming. Web site available at <http://www.jboss.org/developers/projects/jboss/aop/>.
- [10] Chatterjee, S. (2003). Developing Real World Web Services-based applications. Web site available at: <http://javaboutique.internet.com/articles/WSApplications/>.
- [11] Clark, J., & DeRose, S. (1999). XML Path Language (XPath) version 1.0. Web site available at <http://www.w3.org/TR/xpath>.
- [12] DATA DIRECT TECHNOLOGY. Web site available at <http://www.stylusstudio.com/>.
- [13] Jeckle, M., & Zhang, L.J. (2003). AOP for Dynamic Configuration and Management of Web Services. ICWS-Europe, LNCS 2853, pp. 137–151.
- [14] Kiczales, G., & al. (1997). Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming, (ECOOP'97). LNCS 1241, Springer-Verlag.
- [15] Kiczales, G., Hilsdale, E., Hugunin, E., Kersten, M., Palm, J., & Griswold, W. G. (2001). An Overview of AspectJ. Proceedings of the 15th European Conference on Object Oriented Programming. (ECOOP). 327-353.
- [16] Laddad, R. (2004). ASPECTJ in Action: Practical Aspect-Oriented Programming. Portland : Book News, Inc.
- [17] Onose, N., & Simeone, J. (2004). XQuery at Your Web Service. Proceedings International WWW Conference, New York, USA.
- [18] RAININGDATA. Web site available at <http://www.rainingdata.com/>.
- [19] Singh, S., Grundy, J.C., & Hosking, J.G. (2004). Developing .NET Web Service-based Applications with Aspect-Oriented Component Engineering. Proceedings of the Fifth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia.
- [20] Spring AOP platform. Web site available at www.springframework.org/docs/reference/aop.html.
- [21] Suvee, D., Vanderperren, W., & Jonckers, V. (2003). JasCo: an Aspect-Oriented approach tailored for Component Based Software Development. Proceedings of the second International Conference of Aspect-Oriented Software Development (AOSD). Boston, USA.
- [22] Vanderperren, W., Suvee, D., Verheecke, B., Cibrán, M.A., & Jonckers V. (2005). Adaptive Programming in JAsCo. Proceedings of the 4th international conference on Aspect-oriented software development (AOSD) Chicago, Illinois. 75 – 86.
- [23] Verheecke, B., Cibrán, M.A., & Jonckers, V. (2004). Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection. Proceedings of the European Conference on Web Services (ECOWS'04), Erfurt, Germany.
- [24] Verpecht, D., Vanderperren, W., Suvee, D., & Jonckers, V. (2003). JasCo.NET: Unraveling Crosscutting Concerns in .NET Web Services. Vrije Universiteit Brussel, Pleinlaan 2, Bruxelles, Belgique.
- [25] Wilson, M., Eberhart, A., Hopgood, B., & Matthews, B. (2002). Towards universal Web Service clients. In Proceedings of the Euroweb 2002.

8. Biography

Mohamed Mehdi Ben Hmida:

Mohamed Mehdi Ben Hmida is a PhD student within the Paris 9 Dauphine University, LAMSADE CNRS. His thesis is concerned with the auto-adaptability of complex Web Services by the use of the Aspect Oriented Programming paradigm.

Ricardo Tomaz Ferraz:

Ricardo Tomaz Ferraz is a PhD student within the Paris 1 Sorbonne University, CRI. His thesis is concerned with the conception of an auto-adaptable middleware by the use of the Aspect Oriented Programming paradigm.

Valérie Monfort:

Valérie Monfort is an assistant Professor in Paris 1 Sorbonne University and associate researcher in Dauphine University, LAMSADE CNRS. She comes from industry where she was an IBM senior consultant in Integration Solutions. She wrote several books and papers about Web Services and AOP in Web Services.