International Journal of Software Engineering
and Knowledge Engineering
Vol. 18, No. 2 (2008) 243–262

World Scientific
www.worldscientific.com

A LOOSELY COUPLED ASPECT LANGUAGE FOR SOA APPLICATIONS

NABOR C. MENDONÇA*, CLAYTON F. SILVA†,
IAN G. MAIA‡, MARIA ANDRÉIA F. RODRIGUES§

Mestrado em Informática Aplicada, Universidade de Fortaleza,
Av. Washington Soares, 1321, 60811-905 Fortaleza, CE, Brazil
*nabor@unifor.br
†clayton_fsilva@yahoo.com.br
‡iangmaia@gmail.com
§mafr@unifor.br

MARCO TÚLIO O. VALENTE

Instituto de Informática, Pontifícia Universidade Católica de Minas Gerais,
Av. Dom Jose Gaspar, 500, 30535-610 Belo Horizonte, MG, Brazil
mtov@pucminas.br

The aspect-oriented programming (AOP) paradigm offers software developers with powerful modularization abstractions to help them explicitly separate design concerns at the source code level. However, the impact of AOP in the service-oriented architecture (SOA) paradigm has been dwarfed by the fact that existing AOP solutions are tightly coupled to a particular programming language, middleware system or execution platform. Clearly, this not only restricts the implementation choices available to application developers, but it also clashes with the heterogeneous and loosely coupled nature of SOA. This paper presents the Web Service Aspect Language (WSAL) that seamlessly integrates AOP and SOA concepts, thus avoiding the drawbacks of existing solutions. In WSAL, aspects themselves are freely specified, implemented and executed as loosely coupled web services. This characteristic allows WSAL aspects to be easily woven into the message flow exchanged between service consumers and service providers, in a way that is completely independent from any particular implementation technology. This paper also reports on the implementation and preliminary evaluation of a prototype aspect weaver for WSAL, which is based on an existing web intermediary technology.

Keywords: Aspect-oriented programming; service-oriented architectures; separation of concerns.

1. Introduction

Service-oriented computing (SOC) is emerging as a powerful software development paradigm in which services constitute the fundamental elements of design [1]. SOC applications follow a standard service-oriented architecture (SOA) in which one or more service providers declaratively describe their services' operations and

invocation properties in a standard, machine-readable format, so that those services can be dynamically discovered, selected, and invoked using a suit of standard protocols [2]. The most popular manifestation of the SOC paradigm is in the form of a suite of non-proprietary XML-based technologies collectively known as *web services* [3]. These include three core technologies, namely SOAP [4], a service invocation protocol, WSDL [5], a language for describing service interfaces, and UDDI [6], a service-based repository for dynamic service discovery and location. Other related technologies are also being developed to deal with non-functional SOA requirements, such as reliability, security and performance [7].

The dynamic binding between service providers and service consumers makes SOA applications loosely-coupled in nature, allowing services to be provided and consumed independently of the client applications' programming language, execution platform, and transport protocol. This characteristic has the main benefit that it makes it easier to integrate independently developed SOA applications, even when they have not been developed with integration as a major design concern.

On the other hand, the distributed and loosely-coupled nature of SOA applications also raises a series of design concerns (typically associated with the provision of non-functional service properties) that must be properly addressed by application developers. These concerns tend to be very difficult (or even impossible) to be modularized using current SOA development technologies. The reason is that non-functional concerns usually affect both service providers and service consumers. As a consequence, their implementation becomes scattered across the implementation of several application components at both sides [8]. For example, to guarantee secure service interactions it is necessary to modify the implementation of both service providers and service consumers, so that both sides can put in place and use the appropriate security mechanisms [9]. Similarly, to allow dynamic service selection at the client side, it is necessary to change the implementation of service providers, so that they can announce their service' QoS properties using some sort of service broker [10], and also to change the implementation of service consumers, so that they can discover and select the servers that best suit their needs, before invoking their required service operations [11].

Another consequence of the lack of better modularization support for non-functional concerns in current SOA technologies is that, as the number of those concerns increases, implementing and, subsequently, maintaining them become exceedingly difficult. This problem exacerbates for applications that are implemented using different programming languages or executed over different execution platforms, a typical SOA scenario.

The fact that some design concerns, such as those discussed above, are inherently difficult to modularize using traditional (i.e., functional) decomposition techniques is the main motivation behind the aspect-oriented programming (AOP) paradigm [12]. AOP introduces a new type of abstraction — called *aspect* — which allows to explicitly separate design concerns that otherwise would have to be implemented as part of (and tangled with) the implementation of several application

components. AOP also provides novel software composition mechanisms to weave the implementation of those concerns (called *cross-cutting concerns*) back into the implementation of application components at well-defined execution points (called *join points*).

In view of the current need for mechanisms to separate cross-cutting concerns in SOA applications, and of the powerful modularization constructs introduced by AOP, there is a natural demand for solutions that aim at integrating these two emerging software development paradigms. Even though there has been a number of works geared toward this direction recently [13]–[17], all solutions proposed thus far suffer from the limitation of being tightly coupled to a particular programming language, middleware technology or execution platform. The high level of coupling associated with those solutions is undesirable for two main reasons: (i) it restricts the spectrum of technological choices available to application developers; and (ii) it clashes with the heterogeneous and loosely coupled nature of typical SOA applications. Therefore, any effective solution for integrating concepts from the SOA and AOP paradigms must provide technology independence as a core design principle [19].

In our previous work [18, 19], we have proposed a novel aspect model for SOA applications, in which services themselves provide the necessary abstractions to modularize cross-cutting concerns. In this paper, we present the Web Service Aspect Language (WSAL), which is a concrete realization of the concepts first introduced in that model. As such, WSAL seamlessly integrates fundamental AOP concepts, such as aspects, join points and advices [12], into the SOA context. Differently from other existing solutions that attempt to integrate these two emerging software development paradigms, in WSAL aspects can be freely specified, implemented, deployed and executed as loosely coupled web services. This characteristic allows WSAL aspects to be easily woven into the message flow exchanged between service consumers and service providers, in a way that is completely independent from any particular implementation technology.

The rest of this paper is organized as follows. Section 2 describes the syntax and semantics of WSAL. Section 3 illustrates the use of WSAL through a series of examples. Section 4 reports on the implementation of a prototype aspect weaver for WSAL. Section 5 presents the results of a preliminary evaluation of our prototype in a controlled setting. Section 6 covers related work. Finally, Sec. 7 concludes the paper and suggests some directions for future research.

2. Web Service Aspect Language

As we have mentioned, WSAL allows aspects to be both implemented and deployed as web services, which we refer to as *aspectual services* [18]. The main benefit of this approach is that the aspect weaving process can now take place at the network level externally to the target applications' execution environments, using any existing web intermediary technology [20]. This weaving approach is based on

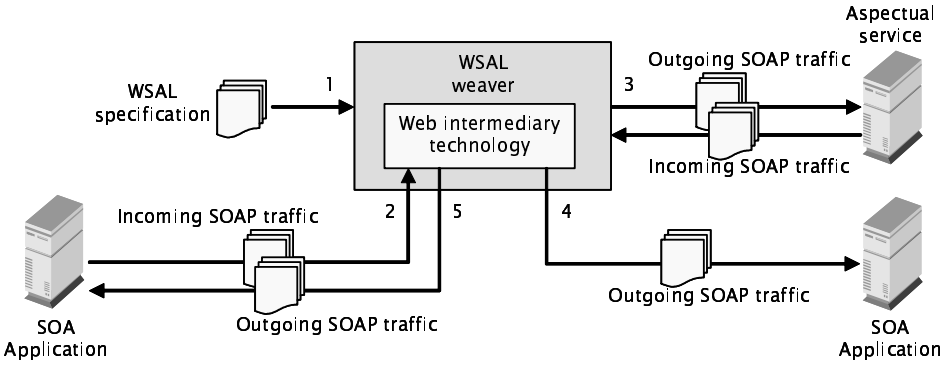

Fig. 1. WSAL aspect weaving process.

the assumption that an effective attempt to integrate the AOP and SOA paradigms must support the former without compromising the heterogeneous and loosely-coupled nature of the latter.

The next subsections describe WSAL in details, starting with its underlying aspect weaving process, and then presenting the syntax and semantics of its main language constructs.

2.1. *Weaving process*

The WSAL weaving process is illustrated in Fig. 1. The weaver is responsible for parsing and deploying a given WSAL specification (or aspect) using an appropriate intermediary technology (step 1). Once an incoming SOAP message is intercepted matching the aspect's specified join points by the intermediary technology (step 2), the weaver invokes the aspect's specified aspectual service operations (step 3) and then weaves the operations' results back into the original SOAP traffic by either forwarding them to the message's original destination (step 4) or returning them to its source (step 5). The actual behavior of the weaver will be determined by the aspect specifications it processes.

Note that the process does not impose any restriction to the roles played by the SOA applications whose messages are being intercepted. This means that the process can equally be applied to SOAP messages representing service requests as well as service responses. Moreover, since the process does not require any modification to the implementation of either service consumers or service providers, it is completely independent of any particular implementation technology. This characteristic is a key feature of WSAL, as it gives SOA developers greater flexibility in choosing an implementation technology (e.g., programming language, development platform, web service middleware) that best suits their needs and preferences.

2.2. *Syntactic elements*

Like most other languages used in SOA development, the WSAL syntax was defined as an XML extension, with its own schema. The language syntactic elements represent typical AOP constructs, such as join points, pointcuts, advices, and aspects [12]. The following subsections describe these elements.

2.2.1. *Join points*

A join point is a well-defined point in the structure or execution flow of a component where aspects can be applied to [12]. Join points related to the components' structure are called static join points, whereas join points related to the components' execution flow are called dynamic join points. Since the WSAL weaving process takes place outside the target applications' execution environment, the language offers no support for the definition of static join points. On the other hand, the language supports six types of dynamic join points. Those types reflect the general structure of SOAP messages, which in turn reflect the structures typically defined in WSDL files [5]. These structures include elements such as service operations and their input and output parameters, service endpoint, SOAP transport protocol, etc.

The six join point types supported by WSAL are described below.

***namespace*:** A *name space* join point identifies one or more XML name spaces associated with the target web service, by means of their Uniform Resource Identifiers (URIs), as defined in the service's WDSL description.

***message part*:** A *message part* join point identifies one or more XML elements that are part of the contents of the SOAP messages used as input or output parameters for any of the operations provided by the target web service.

***service operation*:** A *service operation* join point identifies the name of one or more operations among those provided by the target web service.

***service location*:** A *service location* join point identifies one or more network addresses where the target web service is physically provided, by means of its Uniform Resource Locators (URL), as defined in the service's WDSL description.

***client location*:** A *client location* join point identifies one or more network addresses where the consumer of the target web service resides, by means of its IP address or network domain.

***composite*:** The *composite* join point type is used in WSAL to define higher-level join points (or pointcuts) from the arbitrary composition of primitive join point definitions. This is done by means of three composition operators, namely *and* (conjunction), *or* (disjunction) and *not* (negation).

Although certainly not comprehensive, this set of join point types is rich enough to allow the definition of join points matching a wide range of SOAP message characteristics, either in terms of their contents (using the types *name space*, *message*

```
1 <!-- definition of a primitive join point -->
2 <pointcut name="primitive_jp" type="messagePart"
3  pattern="&lt;query*&gt;web2.0&lt;/query&gt;">
4
5 <!-- definition of a composite join point -->
6 <pointcut name="composite_jp" type="composite">
7    <and>
8        <pointcut type="serviceOperation" pattern="doGoogleSearch"/>
9        <pointcut type="clientLocation" pattern="188.188.*"/>
10   </and>
11 </pointcut>
```

Fig. 2. Example of two join points defined in WSAL.

part and *service operation*) or in terms of their underlying transport protocol (using the types *service location* and *client location*).

Syntactically, a WSAL join point is defined using the *pointcut* element. This element includes two mandatory attributes: *type*, which indicates the join point type, and *pattern*, which is used to specify the join point matching context. The *pattern* attribute can be specified using regular expressions, thus offering WSAL developers greater flexibility in describing their join points (SOA events) of interest. The *pointcut* element also includes a third (non-mandatory) attribute, *name*, which specifies a unique name to the join point. This attribute is used within the definition of *advice* elements, to associate the advice to a previously defined join point (see Sec. 2.2.2).

Figure 2 illustrates the definition of two join points in WSAL. The first is a named primitive join point of type *message part* (lines 2–3), while the second is a named composite joint point defined in terms of the composition of two (unnamed) primitive join points (of types *service operation* and *client location*, respectively) using the *and* logical operator (lines 6–11).

2.2.2. *Advices*

According to the AOP terminology, advices are programming abstractions for specifying what additional behavior an aspect should introduce at specific join points [12]. In WSAL, in contrast to most existing AOP languages, advice implementations are not bound to any particular programming language or development environment. Instead, they are implemented as loosely-coupled web service operations, so that they could be dynamically invoked by the WSAL weaver once their specified join points are met. Therefore, WSAL developers are free to implement their advices using any implementation technology, as long as they make them available via a common web service interface.

A WSAL advice is defined using the *advice* element. This element includes a *type* attribute, which specifies the advice type. In WSAL, the type of an advice denotes

the expected semantics for the advice. Included in the semantics is information related to when an advice should be invoked once a specified join point is reached; the join point context that should be passed as an invocation parameter to the advice operation; and restrictions on the actual behavior that should be implemented by the advice.

The advice types supported in WSAL are derived from the different types of interaction events that may occur between service consumer and service provider applications at runtime: service requests, service responses, and invocation failures (or exceptions). Based on these three types of events, seven advice types were considered as part of the WSAL design: *before request*, *upon request*, *after response*, *upon response*, *after exception*, *upon exception*, and *around*. These advice types are described below. To better illustrate the semantics associated with each of those types, we also compare them with some of the advice types supported by AspectJ [21], a well-known AOP Java extension.

***before request*:** The *before request* advice type adds the crosscutting behavior provided by advice's specified aspectual service operation before a SOAP request message (which has been intercepted matching the advice's specified join points) is forwarded to the target web service. The contents of the message is left unchanged. This advice type is similar to the *before* advice type of AspectJ.

***upon request*:** The semantics of *upon request* is similar to the semantics of the *before request* advice type, the only difference being that an *upon request* advice is allowed to change the contents of the intercepted request message. This advice type can be seen as a particular case of the *around* advice type of AspectJ, one in which the last statement of the advice body corresponds to an invocation of *proceed*.[a]

***after response*:** The *after response* advice type adds the crosscutting behavior provided by advice's specified aspectual service operation after a SOAP response message has been intercepted matching the advice's specified join points, but before the message is forwarded to the target client application. The contents of the message is left unchanged. This advice type is similar to the *after returning* advice type of AspectJ.

***upon response*:** The semantics of *upon response* is similar to the semantics of the *after response* advice type, the only difference being that an *upon response* advice is allowed to change the contents of the intercepted response message. This advice type can also be seen as a particular case of the *around* advice type of AspectJ, but one in which the first statement of the advice body corresponds to an invocation of *proceed*.

[a]In AspectJ, calling the *proceed* construct in the body of an *around*-type advice causes the compiler to call the original (advised) method.

***after exception*:** The *after exception* advice type adds the crosscutting behavior provided by advice's specified aspectual service operation after a SOAP exception message has been intercepted matching the advice's specified join points, but before the message is forwarded to the target client application. The contents of the message is left unchanged. This advice type is similar to the *after throwing* advice type of AspectJ.

***upon exception*:** The semantics of *upon exception* is similar to the semantics of the *after exception* advice type, the only difference being that an *upon exception* advice is allowed to change the contents of the intercepted exception message. As with *upon request* and *upon response*, this advice type can also be seen as a particular case of the *around* advice type of AspectJ, but one in which the *proceed* construct is called as the first statement of the advice body and the invocation fails (i.e., the advised method throws an exception).

***around*:** The *around* advice type replaces the behavior of a requested service operation matching the advice's specified join points. This is done by having the advice's specified aspectual service operation generating a new SOAP response to be forwarded to the target client application, instead of the response message that would have been produced by invoking the target web service provider. This advice type corresponds to a more constrained version of the *around* advice type of AspectJ. The constrain is related to the fact that, because the WSAL weaving mechanism has no control over how each aspectual service is actually implemented, WSAL offers no construct for invoking the intercepted service operation from within the around advice body. This constrain is the main reason for including three *upon* advice types as part of the WSAL design. As we have mentioned above, the semantics of these three types correspond to particular uses of *proceed* within an around advice body in AspectJ.

In addition to provide the expected semantics for advice operations, in WSAL advice types also constrain the kinds of context information that will be available to a given advice operation. Specifically, for advices of types *before request*, *upon request*, and *around*, the available context information will include the contents of the intercepted request message, along with some important properties related to the message's underlying transport protocol, such as the network location of both service consumers and service providers. For the other advice types, context information will further include the contents of the intercepted response message (in the case of the advice types *after response* and *upon response*) or the contents of the intercepted exception message (in the case of advice types *after exception* and *upon exception*).

Providing context information to advice operations is particularly useful to around type advices that need to invoke the original (intercepted) web service operation. This is the case of advices that implement caching or service selection

concerns. In those cases, invoking the original service operation simply requires forwarding the intercepted request message to its original destination.

It should be stressed that we are aware that implementing and invoking advices as external web service operations is likely to have a significant impact on the performance of our aspect weaving mechanism and, consequently, on the performance of the target SOA applications. To help developers in reducing the potential performance impact imposed by our weaving mechanism, the WSAL *advice* element includes two special attributes, namely *context* and *mode*, which can be used to reduce the communication overhead between the weaver and the aspectual service operations it invokes.

The *context* attribute can be used to further restrict the kinds of context information that the weaver passes on as input to the advice's aspectual service operation. This feature is particularly useful in situations where context information may not be necessary (either entirely or in part) to the successful execution of an advice operation. For example, an advice operation only concerned with measuring the target service's reliability would not required any context information at all, but only the final status (either success or failure) of each requested service invocation.

The *mode* attribute, in turn, is only applicable to *before* or *after* type advices. It is used in WSAL to indicate to the weaver that the advice's specified operation must be invoked asynchronously, meaning that the weaver must not block waiting for the operation to complete its execution. The underlying idea is that, since *before* and *after* advice types are not required to return any kind of information to the weaver, they could – in principle, at least – be executed in parallel with the weaver without compromising their expected semantics. Nevertheless, the decision about whether or not it is safe to invoke a given *before* or *after* advice in asynchronous mode will be entirely up to the WSAL developer.

2.2.3. *Aspects*

In WSAL, an aspect is defined using the *aspect* element. This element relates a set of *pointcut* elements to an *aservice* element. The *aservice* element is used in WSAL to specify an externally provided aspectual service. It includes the following attributes: a *name*, which assigns a unique name to the aspectual service; a *location*, which specifies the network location (URL) where the aspectual service is physically provided; and one or more *advice* elements. In this way, an *aspect* element comprises all the information necessary for the weaver to manipulate the intercepted SOAP traffic and invoke the provided aspectual service operations, as specified by the WSAL developer. In addition, by requiring that *aspect* and *aservice* elements be specified separately, WSAL makes it easier to reuse the same aspectual service operations across different aspect specifications.

The next section illustrates the use of WSAL by presenting the specification of three aspects: *client authentication*, *cache*, and *billing*.

3. WSAL Examples

3.1. *Client authentication aspect*

Figure 3 shows an example of a WSAL specification for a client authentication aspect. This aspect encapsulates all the authentication details necessary for a client application to access the Google search service [22]. In this example, the aspect defines a *pointcut* element of type *service location*, whose *pattern* attribute contains the Google service's access URL (lines 2–4). The aspect also defines an *aservice* element with a single advice, of type *upon request* (line 7). This advice is associated with an aspectual service operation named *AddCredential* (line 9), provided by the *AuthenticationService* service (line 5), whose main purpose is to insert the necessary credentials for accessing the Google service into the intercepted request messages. In this way, all SOAP request messages intercepted by the weaver, which have the Google service URL as their destination, will be preprocessed by the authentication aspectual service to have the required credential properly included, before the message is forwarded to its original destination.

```
1 <aspect id="ClientAuthenticationAspect">
2     <pointcut name="googleService"
3      type="serviceLocation"
4      pattern="http://api.google.com/search/beta2"/>
5     <aservice name="AuthenticationService"
6      location="http://server1/axis/AuthService">
7         <advice type="uponRequest"
8          pointcut-ref="googleService"
9          operation="AddCredential"
10         context="infoRequest" />
11    </aservice>
12 </aspect>
```

Fig. 3. Example of a client authentication aspect specified in WSAL.

With this aspect, all information related to the client authentication concern would be completely encapsulated in the aspect's WSAL specification and the implementation of its corresponding aspectual service. This level of modularization is beneficial, in that any modification required to the authentication concern (e.g., changing or renewing the client credentials) would be implemented exclusively in those two components, without affecting the implementation of any client application directly. In fact, with the authentication aspect in place, the authentication process would be totally transparent to client applications developers, who could then focus their development effort in the applications' main business concerns.

3.2. *Cache aspect*

Cache mechanisms are largely used in distributed system to improve the performance and availability of remote services [23]. In the case of web services, caching

is only valid for operations that produce the same results when invoked with the same set of parameters [24]. Figure 4 shows an example of a WSAL specification for a web service cache aspect. This aspect defines a composite pointcut from the combination of three primitive pointcuts, of types *service location*, *service operation* and *message part*, respectively, whose pattern attributes will match every invocation issued to the *doGoogleSearch* operation, provided by the Google search service, and whose query parameter contains the string "Internet Standards" (lines 2–11). The aspect's additional behavior is specified by means of an advice element of type *around* (line 14), whose implementation is given by the *GenerateResponse* aspectual service operation (line 15), provided by the *CacheService* service (line 12).

```
1 <aspect id="CacheAspect">
2     <pointcut name="googleSearch" type="composite">
3         <and>
4             <pointcut type="serviceLocation"
5              pattern="http://api.google.com/search/beta2"/>
6             <pointcut type="serviceOperation"
7              pattern="doGoogleSearch"/>
8             <pointcut type="messagePart"
9              pattern="<query*>Internet␣Standards</query>"/>
10        </and>
11    </pointcut>
12    <aservice name="CacheService"
13     location="http://localhost:8080/axis/services/ASCaching1">
14        <advice type="around" pointcut-ref="googleSearch"
15         operation="GenerateResponse" context="all"/>
16  </aservice>
17 </aspect>
```

Fig. 4. Example of a web service cache aspect specified in WSAL.

The cache aspect works as follows. Every time the WSAL weaver intercepts a SOAP request message matching the cache aspect's specified join points, it invokes the *GenerateResponse* aspectual service operation, passing the intercepted message and other context information as parameters. The *GenerateResponse* operation then checks whether a similar request message (sent to the same service operation and involving the same set of parameters) has been registered in the cache before. If it finds a similar message in the cache, it returns the cached response to the weaver. Otherwise, it forwards the intercepted request message to its original target service (in this case, the Google search service), waits for a response, stores both the request message and the response message in the cache, and, finally, returns the response to the weaver. Once the weaver receives a response back from the aspectual service, it simply returns this message back to the client application. If the aspectual service fails to contact the target service, it returns an exception to the weaver, which then propagates the exception to the client application.

In [24], the authors propose annotating the WSDL specification with information indicating whether a given service operation is considered to be cache-safe by the web service developers. In this way, it would be up to client application developers to decide whether it would be worth caching the results of those operations marked as "cacheable" in the WSDL document. With the WSAL cache aspect in place, any cache-related concern (such as the decision about whether or not to cache the results of a given service operation) would be completely modularized in the aspect specification and in the implementation of its associated aspectual service, thus freeing software developers from having to deal with those issues in every new application they implement.

3.3. *Billing aspect*

Figure 5 shows an example of a WSAL specification for a billing aspect. This aspect is used to register each successful invocation to a given service, so that client applications can later be charged for their accesses. The aspect defines a *pointcut* element of type *client location*, whose *pattern* attribute contains a single network domain name (lines 2–4). This means that only invocations generated from that domain will be charged by the aspect's associated aspectual service operation. The aspect also defines an *aservice* element with a single advice, of type *after response* (line 7). This type was chosen as only successful invocations must be charged. Note that the *after response* advice is specified to be invoked in asynchronous mode (line 11). This is possible because its aspectual service operation does not return any relevant information back to the weaver.

```
1 <aspect id="ServiceBillingAspect">
2     <pointcut name="UseService"
3         type="clientLocation"
4         pattern="http://client.domain/"/>
5     <aservice name="BillingService"
6         location="http://server2/axis/BillingService">
7         <advice type="afterResponse"
8             pointcut-ref="UseService"
9             operation="BillingPerUse"
10            context="infoHTTP"
11            mode="asynchronous"/>
12    </aservice>
13 </aspect>
```

Fig. 5. Example of a service billing aspect specified in WSAL.

3.4. *Discussion*

It is worth noting that, because WSAL is a specification language, rather than an implementation one, all details pertaining to the implementation of aspectual service operations are under the responsibility of their respective developers. Therefore,

it is the developers' responsibility to guarantee that the implemented operations comply with their execution semantics, as required by the WSAL weaver.

On the other hand, by decoupling aspect specifications from their implementations, WSAL allows aspect implementations to be reused across applications developed using different implementation technologies. For example, the client authentication aspect (shown in Fig. 3) could be reused to add credentials to SOAP request messages generated from client applications running in different network domains, independently of the their programming language, midlleware system or execution environment. Similar arguments also apply to the cache and billing aspects. Such level of reuse is simply not possible with existing solutions that also attempt to integrate the AOP and SOA paradigms.

A more detailed comparison between WSAL and a number of related approaches can be found in Sec. 6.

4. Development of a Prototype Aspect Weaver for WSAL

As mentioned previously, the aspect weaving process in WSAL is based on the interception of SOAP messages exchanged between SOA applications at the network level, which means that messages are intercepted externally to the applications' execution environments. In this way, the WSAL weaver plays the role of a web intermediary (or proxy) [20] between service consumers and service providers.

In our prototype, we have implemented the WSAL weaver on top of IBM's WBI [25]. WBI was chosen because it already offers an extensible web intermediary platform for intercepting and handling HTTP messages. Since HTTP is still the most commonly used transport protocol for delivering SOAP messages, WBI has proved a handy solution to implement our weaver prototype.

More specifically, WBI is a programmable HTTP proxy written in Java. Its extension mechanism is based on the concepts of plugins and MEGs. Plugins are Java classes that can be dynamically loaded into the WBI runtime environment (JVM). They are used to configure and control the execution of one or more MEGs. MEGs are special objects used to handle HTTP request and response messages intercepted under the conditions configured in the MEG by the plugin. The term MEG is an acronym for Monitor, Editor and Generator, which reflect the three different types of MEG supported by WBI. A monitor MEG is used to monitor HTTP messages that have been intercepted by WBI without applying any modification to their contents or destination. An editor MEG in turn is used to modify the contents of intercepted HTTP messages without changing their destination. Finally, a generator MEG is used to generate new HTTP messages that can be then sent to a destination in place of any intercepted HTTP message. WBI also supports the definition of new types of MEGs, by extending these three basic types.

To avoid coupling our implementation to a specific intermediary technology, we have taken some design decisions aimed at improving the reusability of the weaver components. These design decisions are discussed in Secs. 4.1 and 4.2.

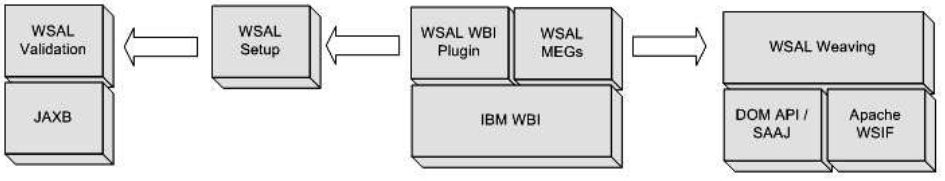

Fig. 6. WSAL weaver architecture.

4.1. *Deployment strategies*

Firstly, we had to decide how the WSAL specifications would be deployed by the weaver, and how the weaver would use that information to handle the HTTP message traffic according to what was specified in the WSAL aspects. Two alternatives were considered:

(1) *Static*: An external tool could be created to read the WSAL specifications and then, using predefined code templates, to generate the WBI code necessary to handle the intercepted HTTP messages according to the behavior specified for each aspect. A possible implementation strategy would be to generate a new WBI plugin for each WSAL aspect.
(2) *Dynamic*: A set of generic classes could be created, which would be responsible for loading the WSAL specifications into memory. At execution time, those classes would check the execution conditions for each deployed aspect and, if the conditions are met, handle the HTTP traffic as necessary.

Both strategies have advantages and disadvantages. For example, while the former requires generating new code for each new WSAL specification, the generated code could be made more efficient by exploring the characteristics of a specific intermediary technology. The latter, on the other hand, would trade performance for reusability, as it could be provided in the form of a generic weaving API, which then could be implemented using different web intermediary technologies. Since we favor platform-independence as a major design concern, we adopted the latter solution in the implementation of our weaver prototype.

4.2. *Weaver architecture*

Figure 6 shows the main architectural elements of our weaver prototype. These include components for parsing and validating the WSAL specifications, and deploying the specified aspects using the underlying intermediary technology. A more detailed description of these components in terms of their functionalities and implementation strategies are provided below.

WSAL Setup: This is a simple mechanism used to load the WSAL specification files into memory and pass them to the validation component (WSAL Validation). It works by obtaining a list of files which end with the extension

".wsal.xml", and then invoking the validation component to validate the loaded aspect specifications.

WSAL Validation: This component converts the XML files loaded by *WSAL Setup* into Java objects, using an XML-Java binding technology (currently we use JAXB [26] as the binding technology). The component also provides an XML-independent WSAL validation API, which consists of a set of interfaces for traversing and validating the WSAL elements. This design decision isolates the validation process from any specific binding technology, thus making it easier to replace the latter in future versions of the weaver.

WSAL WBI Plugin: This component is a single class that is responsible for coordinating the other components at deployment time. It invokes the WSAL Setup and Validation components, obtains the aspect definitions, and instantiates and configures the WBI MEGs necessary to intercept and handle the HTTP traffic.

WSAL MEGs: These are three WBI MEGs (namely *WsalWeavingRequestEditor*, *WsalWeavingDocumentEditor* and *WsalWeavingGenerator*) created specifically for the WSAL weaver. Each MEG handles different types of advices, as described below:

- *WsalWeavingRequestEditor*: this MEG is a subtype of *RequestEditor*, which is the MEG type defined for WBI to handle and possibly modify HTTP requests. In our weaver prototype, this MEG is used to handle advices of types *beforeRequest* and *uponRequest*;
- *WsalWeavingDocumentEditor*: this MEG is a subtype of *DocumentEditor*, which is the MEG type defined for WBI to handle and possibly modify HTTP responses. In our weaver prototype, this MEG is used to handle advices of types *afterException*, *afterResponse*, *uponException* and *uponResponse*;
- *WsalWeavingGenerator*: this MEG is a subtype of *Generator*, which is the MEG type defined for WBI to generate new HTTP responses. In our weaver prototype, this MEG is used to handle advices of type *around*.

WSAL Weaving: This is the core component of our weaver prototype. It is responsible for realizing the full semantics of WSAL, i.e., evaluating join points, compiling context information, invoking aspectual services, and resuming the intercepted HTTP flow. It was implemented using existing technologies for manipulating the XML structure of SOAP messages [27] and for dynamic service invocation [28]. The relationships between this component and these two technologies is illustrated on the right of Fig. 6.

5. Preliminary Evaluation

This section presents the results of a preliminary evaluation of our WSAL weaver prototype. Our motivation at this stage is twofold: (i) to provide some early evidence of the performance impact of WSAL on existing SOA applications; and (ii) to

establish a methodological basis upon which to conduct further experiments with larger systems, possibly using a more robust implementation of the weaver.

The evaluation consisted of several experiments carried out to measure the performance impact of the billing aspect (presented in Fig. 5) on a toy web service. This aspect affects only the implementation of a given service. However, other non-functional requirements may demand aspects that crosscut both client and service providers, such as authentication, authorization, caching, etc [8]. The experiments measured the average service response time, as perceived at the client side, in three different scenarios: with the billing concern implemented as part of the toy service implementation (Scenario 1); with the billing concern implemented by an external aspectual service operation invoked by the weaver in asynchronous mode (Scenario 2); and with the billing concern implemented by an external aspectual service operation invoked by the weaver in synchronous mode (Scenario 3).

The evaluation involved three machines connected through a Fast Ethernet (10/100 Mbps) local area network. The three machines had the same configuration: Windows XP Professional (SP2); Pentium IV processor (3.2 GHz) and 512 MB RAM. One machine was used to run the client application; a second machine was used to run both the WBI components required by our prototype weaver implementation and the billing aspectual service; finally, the third machine was used to deploy the target toy service whose successful invocations would be registered by the billing aspectual service.

The target service provides a single operation which receives a text string as input and returns the same text string as output, but with all characters non-capitalized. Four sequences of invocations were performed as part of the experiments. In each sequence, the service was invoked 15 times, using the same string as input. At each new sequence, the string size (in bytes) was increased. The string sizes used in each sequence were 1, 20, 40, and 60 KBytes, in that order. For each service invocation the client application computed the service response time by measuring the elapsed time between invoking the service and receiving a complete response. To avoid service initialization time interfering in our results, we discarded the first five invocations in each sequence when computing the mean service response time.

The evaluation results are depicted in Fig. 7. As expected, the results show that the weaver may impose different levels of performance overhead, depending on the synchronization mode used to invoke the aspectual service.

As described before, in the first scenario the billing concern is part of the toy service implementation, which means that its execution was considered when measuring the service response times. In Scenario 2, an aspectual service modularizes the billing concern. Because this service was invoked asynchronously, its execution was not considered in the service response times. For this reason, the average response time observed in Scenario 2 is slightly inferior to the results of Scenario 1. Scenario 3 is similar to Scenario 2, but with the aspectual service being invoked synchronously. Thus, in Scenario 3, the measured service response times include the overhead of accessing the billing service and waiting for its execution.

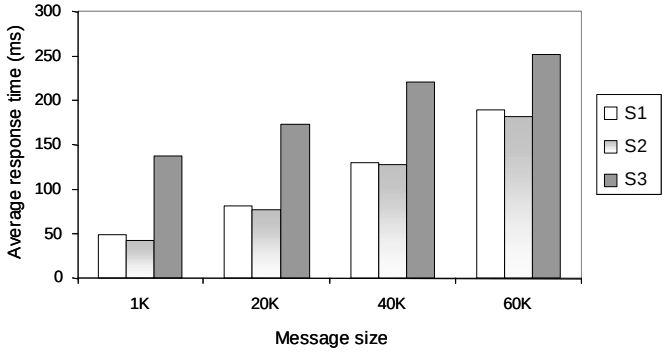

Fig. 7. Billing service's average response time in the three scenarios investigated.

Note that the relative performance overhead observed in Scenario 3 with respect to the other two scenarios gradually decreases as the message size increases. This is explained by the fact that the toy service response time increases proportionally to the size of the input parameter, while the overhead imposed by the weaver remains under 100 ms for all message sizes.

Even though the results show that the weaver may impose a noticeable performance overhead when an aspectual service is invoked in synchronous mode, we believe that, in many situations, this may be compensated by the reusability and platform-independence benefits that are brought about when using a loosely coupled aspect language such as WSAL. For example, in the particular case of the billing aspectual service used in our preliminary evaluation, the overhead was under 100 ms for all message sizes. This level of performance degradation could be perfectly acceptable, in so far as it does not compromise the QoS requirements of the client application.

6. Related Work

Using SOAP intermediaries as a way to implement crosscutting SOA concerns is not a novel ideal. In fact, it has been part of the SOAP specification since its early versions [4]. In most existing web service frameworks, such as .NET [29] and JAX-WS [30], such intermediaries are implemented as message handlers, which can be deployed at both the client and service sides. The weaving model supported by WSAL takes this idea one step ahead, by using SOAP intermediaries not as the technology to implement SOA concerns directly (which would bind concern implementations to a particular intermediary technology), but as the means to decouple, in a SOA context, concern composition (weaving) from concern implementation.

A number of other approaches exist that aim at integrating concepts from the AOP and SOC paradigms [13, 17, 31]. However, those solutions all rely on an aspect weaving mechanism that is tightly bound to a specific programming language or development environment. For instance, the approaches by Verheecke and Cibrán [13]

and Baligand and Monfort [31] rely on different weaving mechanisms specific to the Java platform, and thus require that both the aspects and their target applications be implemented in Java. Similarly, the approach by Verheecke *et al.* [17] relies on a weaving mechanism specific to .NET, and thus requires aspects and applications only to be implemented using that framework. In contrast to the above approaches, in WSAl aspects are implemented as loosely-coupled web services that can be weaved dynamically into SOAP-based interaction events captured over the network.

Two other related works follow a slightly different approach [14, 15]. They both integrate AOP and SOC concepts at the service composition level, and both extend BPEL4WS [32], a web service-based process composition language. The extension includes creating new constructs for specifying process composition aspects, and a new composition mechanism for weaving the aspects back to the original BPEL4WS processes. By relying on a weaving mechanism that works at the service composition level, those two works, like our work on WSAL, also support aspect implementation and weaving in the form of loosely coupled web services. However, the weaving mechanism used in those approaches is more limited in scope, being restricted to dynamic join points expressed in terms of interaction events captured by the BPEL4WS execution engine at the service composition level. This feature has the drawback that it tightly couples aspect implementation in those approaches to the BPEL4WS language.

On the other hand, WSAL design is independent from SOAP engines and business process modeling languages. The prototype aspect weaver for WSAL described in this paper relies on the WBI framework for message interception. However, other weavers can be implemented using different SOAP engines or providing their own interception frameworks.

7. Conclusions

This paper presented a novel aspect language, called Web Service Aspect Language (WSAL), which provides a natural mechanism for integrating aspect-oriented programming concepts into the context of service-oriented architectures. In WSAL, aspects are both implemented and deployed as loosely-coupled web services, which allow crosscutting concern implementations (or advices) to be dynamically woven into the SOAP traffic generated between service consumers and services providers. In contrast to most existing approaches that also aim at integrating these two emerging software development paradigms, the weaving process supported in WSAL is completely independent of any particular implementation technology. The paper also presented some details on the design and implementation of a prototype aspect weaver for WSAL, which has been developed based on a existing HTTP intermediary framework. Results of a preliminary performance evaluation of the weaver in a controlled setting were finally presented and discussed.

As regards future work, we envision the following research lines:

- Developing new versions of the weaver using different intermediary technologies. The idea is to evaluate empirically the characteristics of reusability and intermediary platform independence of our proposed weaver architecture.
- Conducting a more systematic evaluation of our tools, in both controlled and production settings. This would help us in fine tuning their performance to meet the QoS requirements of industrial-strength SOA applications.
- Developing new tools to support the creation, deployment and monitoring of WSAL aspects. These tools are vital to improve the quality of the WSAL development process and to make it more accessible to SOA developers in general.

References

1. M. P. Papazoglou and D. Georgakopoulos, Service-oriented computing, *Commun. ACM* **46**(10) (2003) 25–28.
2. OASIS, OASIS Reference Model for Service Oriented Architecture V1.0, 2006, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.
3. G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architecture and Applications* (Springer Verlag, 2004).
4. M. Gudin, M. Hadley, N. Mendelsohn, J. Moreau and H. F. Nielsen, SOAP Version 1.2, W3C Recommendation, 2003, http://www.w3.org/TR/soap12.
5. E. Christensen, F. Curbera, G. Meredith *et al.*, Web Services Description Language (WSDL) version 1.1, W3C Note, 2001, http://www.w3.org/TR/wsdl.
6. D. Bryan, V. Draluk, D. Ehnebuske, T. Glover *et al.*, Universal Description, Discovery and Integration (UDDI) version 2.04, 2002, http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm.
7. D. A. Menascé, QoS Issues in Web Services, *IEEE Internet Computing* **6**(6) (2002) 72–75.
8. E. Wohlstadter, S. Jackson, and P. Devanbu, DADO: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems, in *Proc. 25th IEEE/ACM Int. Conf. on Software Engineering* (*ICSE'03*), 2003, pp. 174–186.
9. OASIS, WS-Security Core Specification V1.1, 2004, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.
10. M. A. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui, A QoS broker based architecture for efficient web services selection, in *Proc. IEEE Int. Conf. on Web Services* (*ICWS'05*), 2005, pp. 113–120.
11. N. C. Mendonça and J. A. F. Silva, An empirical evaluation of client-side server selection policies for accessing replicated web services, in *Proc. 20th Annual ACM Symp. on Applied Computing* (*SAC'05*), *Web Technologies and Applications Track*, Santa Fé, New Mexico, USA, 2005, pp. 1704–1708.
12. G. J. Kiczales, L. A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, Aspect-oriented programming, in *Proc. 11th European Conf. on Object-Oriented Programming* (*ECOOP'97*), LNCS Vol. 1241 (Springer-Verlag, 1997), pp. 220–242.
13. B. Verheecke and M. A. Cibrán, AOP for dynamic configuration and management of web services, in *Proc. Int. Conf. on Web Services — Europe 2003* (*ICWS-Europe'03*), Erfurt, Germany, 2003.
14. A. Charfi and M. Mezini, Aspect-oriented web service composition with AO4BPEL, in *Proc. European Conf. on Web Services* (*ECOWS'04*), LNCS Vol. 3250 (Springer-Verlag, 2004), pp. 168–182.

15. C. Courbis and A. Finkelstein, Towards an aspect weaving BPEL engine in *Proc. 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (*ACP4IS'01*), Lancaster, UK, 2004.
16. M. Henkel, G. Boström, and J. Wäyrynen, Moving from internal to external services using aspects, in *Proc. 1st Int. Conf. on Interoperability of Enterprise Software and Applications* (*ICIESA'05*), Geneva, Switzerland, 2005.
17. B. Verheecke, W. Vanderperren, and V. Jonckers, Unraveling crosscutting concerns in web services middleware, *IEEE Software* **23**(1) (2006) 42–50.
18. N. C. Mendonça and C. F. Silva, Aspectual services: Unifying service- and aspect-oriented software development, in *Proc. 1st Int. Conf. on Next Generation Web Services Practices* (*NWeSP'05*), Seoul, Korea, 2005, pp. 351–356.
19. N. C. Mendonça and C. F. Silva, A unified model for service- and aspect-oriented software development, *Int. J. Web Services Practices* **2**(1–2) (2006) 59–67.
20. R. Barret and P. Maglio, Intermediaries: New places for producing and manipulating web content, *Computer Networks and ISDN Systems* **30**(1–7) (1998) 509–518.
21. R. Laddad, *AspectJ in Action* (Manning Publications Co., 2003).
22. Google, Google SOAP Search API (Beta), 2006, http://code.google.com/apis/soapsearch/reference.html.
23. G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design* (Addison Wesley, 2005).
24. D. B. Terry and V. Ramasubramanian, Caching XML web services for mobility, *ACM Queue* **1**(3) (2003) 70–78.
25. IBM Research, Web Intermediaries (WBI), http://www.almaden.ibm.com/cs/wbi/.
26. SUN Microsystems, Java Enterprise Ed. 5, 2006, http://java.sun.com/javaee/5/.
27. SUN Microsystems, SAAJ Project, 2007, https://saaj.dev.java.net/.
28. The Apache Software Foundation, WSIF Project, 2006, http://ws.apache.org/wsif/.
29. Microsoft Corp., .NET Framework, 2007, http://msdn.microsoft.com/netframework/.
30. SUN Microsystems, JAX-WS, http://jax-ws.dev.java.net/.
31. F. Baligand and V. Monfort, A concrete solution for web services adaptability using policies and aspects, in *Proc. 2nd Int. Conf. on Service Oriented Computing* (*ICSOC'04*), New York, NY, USA, 2004, pp. 134–142.
32. BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems, Business Process Execution Language for Web Services (BPEL4WS) version 1.1, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.