A Unified Model for Service- and Aspect-Oriented Software Development

Nabor C. Mendonça and Clayton F. Silva

Abstract—There is a recent interest in integrating the service- and aspect-oriented software development paradigms. However, most of the approaches proposed thus far rely on a non-service based aspect model that severely restricts the language choices either for implementing the service-oriented applications, the aspects, or both. This paper presents a unified development model for service- and aspect-oriented applications. In the proposed model, aspects are also implemented as loosely-coupled web services (called aspectual services) that can be weaved dynamically into runtime interaction events captured between service consumer and service provider applications over the network. Compared to existing approaches, the aspectual service model offers two main benefits: it improves the conceptual integrity of the development process, since no separate aspect model is necessary; and it allows aspects to be freely implemented and deployed in a language-independent way. The model provides a conceptual basis for a new aspect language tailored for service-oriented applications, which is currently under development.

Index Terms—Service-oriented development, aspect-oriented programming, aspectual services, separation of concerns.

I. 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 are characterized by one or more service providers that declaratively describe their services’ operations in a standard, machine-readable format, and by one or more service consumers that, based on the available service descriptions, can dynamically discover, select, and invoke services of their interest.

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

Software developers are rapidly embracing the SOC paradigm, particularly in the web environment, where it is mostly used in the form of a suit of non-proprietary technologies collectively known as web services [2]. These technologies include SOAP [3], an XML-based service interaction protocol, WSDL [4], an XML-based language for describing service interfaces, and UDDI [5], a service repository for dynamic service discovery and location.

Several middleware systems are now available that implement the web services model for a variety of programming languages and software development platforms [2]. However, the distributed and loosely-coupled nature of web service-based applications raises a series of non-functional concerns that are difficult to modularize using those middleware systems alone. Typical examples include concerns pertaining service discovery and selection, logging, security, adaptability, and reliability. Because a non-functional service-based concern affects both web service providers and consumer, its implementation is usually scattered across (and thus tangled with) the functional components of several applications. As the number of non-functional concerns that have to be addressed by those applications increases, their implementation and subsequent evolution become exceedingly difficult.

The fact that some software concerns are inherently difficult to modularize, using traditional functional decomposition techniques, is the driving force behind the recent aspect-oriented programming (AOP) paradigm [6]. AOP introduces a new type of abstraction – called aspect – for explicitly separating such types of concerns (called cross-cutting concerns), and provides new composition mechanisms for weaving aspects back into functional components at well-defined join points.

Since non-functional service-based concerns can also be seen as cross-cutting concerns in light of the AOP philosophy, recently there has been a growing interest in integrating these two paradigms [7][8][9][10][18]. However, most of the solutions proposed thus far rely on a non-service based aspect model that severely restricts the language choices either for implementing the service provider and service consumer applications, their crosscutting aspects, or both.

In our view, the implementation choices imposed by the above solutions are not only restrictive but also counterintuitive with respect to the heterogeneous and loosely-coupled nature of the SOC paradigm. Moreover, such
restrictions may render the above solutions impractical in many real-world development scenarios, where web services are used to integrate independently developed legacy applications for which no aspect weaving mechanism is likely to be available. Consequently, there is an evident need for solutions that unify the SOC and AOP paradigms, and that can be used in a broader range of service-based applications and development scenarios.

In this paper, we present a unified model for the development of service- and aspect-oriented applications. Contrary to earlier attempts to integrate these two paradigms, in our model aspects are also implemented as loosely-coupled web services, called aspectual services, whose operations can be weaved dynamically into service-level interaction events captured over the network at execution time. As with traditional web services, aspectual services can be freely implemented and deployed, the only restriction being that they should provide a previously agreed service interface. The aspectual services model thus offers two important benefits to web service-based application developers: first, it improves the conceptual integrity of the development process, since the service model is used both for implementing the service-oriented applications and the aspects (hence, no separate aspect model is necessary); second, it allows the modular implementation of cross-cutting service-based concerns in a way that is completely independent of any specific programming language or software development environment.

Our unified model provides a basis for the design of a novel aspect language tailored for service-oriented development. This language, called Web Service Aspect Language (WSAL), is currently being implemented as an extension of XML, which will allow aspectual services specifications to be easily described and processed using existing XML authoring and manipulation tools.

The rest of paper is organized as follows. The next section presents our unified aspectual service model in details. Section III gives an overview of WSAL and reports on the current status of its implementation. Section IV covers related work. Finally, Section V concludes the paper with a summary of our results and with some directions for future work.

II. THE ASPECTUAL SERVICE MODEL

A software development model provides a conceptual framework upon which one can reason about the main concepts introduced by a given software development paradigm in a way that is independent of any particular implementation technology. The aspectual service model, whose definition borrows largely from the theory of aspects proposed by Chavez and Lucena [11], encompasses five interrelated (sub)models: (i) the component model, (ii) the join point model, (iii) the advice model, (iv) the aspect model, and (v) the weaving model. We describe each of these models in the following subsections.

A. The Component Model

The component model is a fundamental part of our unified model. It defines the abstractions through which a system can be decomposed in terms of a certain kind of component. These include core concepts, such as components and their interaction and composition mechanisms, rules that constrain elements of those concepts, and a set of general principles [11].

In our model, a component is represented by any application that provides or consumes a web service. Component interactions in turn are represented by all service request and service response messages that the component applications exchange at runtime. As it is the case with the SOC paradigm, in our model components are seen as “black boxes”, i.e., their implementation and execution environments are inaccessible except to their own developers. This is an important restriction, since it makes it possible to decouple the weaving mechanism from any particular aspect implementation technology. In addition, this restriction affects both the join point model and the weaving model, as we will describe later.

With regard to component composition, even though a number of web service composition mechanisms have already been proposed (e.g., BPEL4WS [19]), none of them is yet established as a web service standard. For this reason, we do not – at least not at this stage – explicitly include any web service composition mechanism in our component model.

B. The Join Point Model

A join point is a well-defined point in the structure or execution flow of a component where aspects can be applied to. The join point model thus defines the elements of the component model semantics that aspects coordinate with [6]. 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 we treat service components as black boxes, there are no static join points in our model. Another consequence of that restriction is that dynamic join points can only be specified in terms of runtime interaction events that can be monitored outside the components’ own execution environment. In our model, such events correspond to the exchange of SOAP [3] messages between service consumer and service provider applications over the network.

The types of dynamic join points defined in our model reflect the general structures of the SOAP messages that are typically exchanged between service-oriented applications. Those structures in turn are based on the properties typically defined in the web services’ WSDL descriptions [4]. A WSDL description defines the abstract properties of a service and the properties that associate the service with a particular transport protocol. These elements include the operations provided by the service, the parameters and general structure of the request and response messages of each operation, the service’s network location, amongst a number of other properties related to the SOAP traffic.

Specifically, our join point model defines five types of join points, namely namespace, message part, service operation,
service location, and client location. These are described below.

1) name space: A name space join point identifies one or more XML name spaces, by means of their Universal Resource Identifiers (URIs), among those defined in a web service’s WSDL description.

2) message part: A message part join point identifies one or more elements that compose any of the input or output messages used by the operations defined in a web service’s WSDL description.

3) service operation: A service operation join point identifies one or more operations among those defined in a web service’s WSDL description.

4) service location: A service location join point identifies one or more URLs where a web service is provided.

5) client location: A client location join point identifies one or more network addresses where the consumer of a web service resides. This type of join point may not be applicable to network scenarios where the intercepted SOAP/HTTP traffic is originated from network proxies.

Although certainly not comprehensive, this set of join point types is flexible enough to allow join points to be specified for any particular SOAP message, either in terms of its contents (using the name space, message part and service operation types) or in terms of properties related to its underlying transport protocol (using the service location and client location types). In both cases, the attribute values of each join point can be defined using regular expressions, thus giving aspectual service developers greater flexibility in identifying the service-based interactions events of their interest.

Like in most existing AOP approaches, our join point model also defines a pointcut element as a named combination of multiple join point elements, possibly of different types. The pointcut element includes a pattern attribute whose value (provided as regular expressions) specifies the context for the join point. Finally, the join point elements in a pointcut can be combined in arbitrary ways, e.g., using conjunction and disjunction operators.

C. The Advice Model

The join point model defines the abstractions for specifying where a given aspect should be applied in terms of the elements of the component model. The advice model in turn defines the abstractions for specifying what additional behavior an aspect should introduce at the specified join points. Similarly to other AOP models, our model defines a separate advice element for specifying aspect behavior. However, in contrast to existing models, where advice implementations are bound to a particular programming language or development platform, in our model advice are provided in the form of loosely-coupled web service operations that can be invoked dynamically by the weaving mechanism once an specified join point has been identified. Therefore, aspect developers are free to implement their advices using any implementation technology, as long as they make them available via a common SOAP-based interface.

We call aspectual service any web service that provides advice operations for one or more aspect specifications.

Our advice model defines an aspectual service (or a service) element as a set of related advices. In addition, this element identifies the actual web service that provides the implementation for those advices. To this end, the service element includes two special attributes, name and location, whose values specify the web service's name and URI, respectively.

Like join points, advices are also typed in our model. 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 defined in our model are derived from the different types of interaction events that may occur between service consumer and service provider applications at runtime. Three basic types of events are considered: service requests, service responses, and service failures (or exceptions). Based on these events, we define the following seven types of advices: before request, upon request, after response, upon response, after exception, upon exception, and around. Below we describe each of those advice types. Moreover, to better illustrate their semantics, we also compare them with some of the advice types available in AspectJ [12], a well-known AOP extension of the Java language.

1) before request, after response, after exception: These three advice types add crosscutting behavior by invoking the advice’s specified operation before a corresponding service request, service response or service exception message (which has been intercepted matching the advice’s specified join points) reaches its target destination. They all leave the contents of the intercepted message unchanged.

2) upon request, upon response, upon exception: An upon request advice acts upon the contents of an intercepted service request message matching the advice’s specified join points, by having the advice’s specified operation modifying the request message’s SOAP contents before it reaches its target service provider. Similarly, an upon response or upon exception advice acts upon the contents of a corresponding service response or service exception message intercepted matching the advice’s specified join points, by having the advice’s specified operation modifying the contents of the intercepted message before it reaches its target consumer application.

3) around: An around advice replaces the behavior of a requested service operation matching the advice’s specified join points. This is done by having the advice’s specified operation generating a new SOAP response message to be delivered to the requesting consumer application, instead of the message that would be produced by the target service provider.

In terms of semantics, the before request, after response,
after exception, and around advice types of our model are similar to the before, after returning, after throwing, and around advice types, respectively, of AspectJ. The semantics of the upon request, upon response, and upon exception advice types, in turn, are derived from special cases of the around advice in AspectJ. More specifically, the upon request advice type corresponds to the around advice in AspectJ, in the case that the last statement of the advice body is an invocation of the original (advised) method, i.e., in the case that the advice’s last statement corresponds to a call to the proceed construct of AspectJ. Similarly, the upon response advice type corresponds to the around advice in AspectJ, in the case that proceed is called as the first statement of the advice body. Finally, the upon exception advice type corresponds to the around advice in AspectJ, in the case that proceed is invoked as the first statement of the advice body, but the invocation fails (i.e., the advised method throws an exception).

In addition to providing the expected semantics for advice operations, advice types also constrain the kinds of context information that will be available to a given advice operation. Specifically, for advice types before request, upon request, and around, the available context information will include the contents of the intercepted request message, along with some important network properties related to the message’s underlying transport protocol, such as the location of the message’s source and target applications. For the other types of advice, the available information will further include the contents of the intercepted response message (in the case of after response and upon response advices) or the contents of the intercepted exception message (in the case of after exception and upon exception advices). Context information is particularly useful to around advices that need to invoke the original service operation, such as in advices that implement caching or service selection concerns; in such situations, to invoke the original operation the advice simply forwards the intercepted request message to its original destination.

It should be noted that implementing advices as external web service operations is likely to have a significant impact on the performance of the weaving mechanism. For this reason, our model introduces two special advice attributes, namely context and mode, which can be used to reduce the performance impact of invoking aspectual service operations as part of the weaving process. The context attribute is used to further restrict the kinds of context information that the weaving mechanism should pass on to an advice’s aspectual service operation, so as to reduce the communication overhead. This is a useful feature since, depending on the nature of the advice, not all context information available at a certain service interaction join point may be necessary. For example, an advice that measures service reliability would be interested only in the status (i.e., either success or failure) of each requested service operation, and hence could ignore completely the actual contents of the SOAP messages exchanged between the service provider and its service consumers.

The mode attribute, in turn, is only applicable to before and after advices. It is used to indicate to the weaving mechanism that the advice’s specified aspectual service operation should be invoked asynchronously, i.e., the weaving mechanism should not block waiting for the advice operation to complete execution. The idea is that, since before and after advice operations do not return any information back to the weaving mechanism, they could – in principle at least – be executed in parallel with the rest of the weaving process (and therefore much more efficiently), without compromising their expected semantics. The decision on whether a given before/after advice operation can safely be invoked in asynchronous mode will be up to the developer specifying the aspect.

D. The Aspect Model

An aspect represents a system property that has to be satisfied simultaneously in multiple system components, and that is not part of the components’ primary functionalities [6]. In our model, an aspect element localizes the specification of a set of join points (or pointcuts), and a set of addional behaviors (advices) to be combined at the specified join points. More specifically, an aspect specification associates each of its advice elements with a set of service interaction join points as well as with one of the aspectual service operations provided by the aspect’s specified web service. This way, aspectual service operations may be (re)used by different aspect specifications in different ways.

An aspect specification provides all the necessary information to configure the dynamic weaving mechanism, so that it will intercept any SOAP message satisfying the specified join points, and invoke the appropriate aspectual service operations once those join points have been identified.

E. The Weaving Model

Weaving is the process of composing aspects with their affected components at a set of specified join points [11]. Our weaving model is based on a dynamic weaving mechanism, which intercepts SOAP messages at the network-level, and invokes the appropriate aspectual service operations (advices) once particular service-based interaction events have been identified. The weaving mechanism can be deployed at a network region closer to the target client applications or the target service providers, or at any network region in between, according to the needs of application developers. The exact network location chosen will affect the behavior of the weaving mechanism. For example, when the weaver is located closer to specific client applications, it will only act upon those SOAP messages originated from or received by those applications, independently of the location of their target web services. Similarly, when the weaver is located closer to some specific service providers, it will only act upon those SOAP messages that are destined to or originated from those services, independently of the location of their client applications. This process is illustrated in Fig. 1.

Since no modifications are necessary to the program structure of either service consumers or service providers, the
weaving mechanism is completely independent of any implementation technology. This is a key feature of our model, as it gives aspectual service developers greater flexibility in choosing the implementation technologies (e.g., programming language, development platform, web service middleware) that best suit their needs and preferences.

The weaver also is responsible for compiling the necessary context information specified for each advice type, and for implementing the synchronization mechanism needed to concurrently invoke before and after advice operations when defined with the asynchronous invocation mode attribute.

III. THE WEB SERVICE ASPECT LANGUAGE

This section gives an overview of the Web Service Aspect Language (WSAL) and reports on the current status of its implementation.

A. WSAL Overview

WSAL implements the unified aspectual service model presented in the previous section. Its XML-based syntax has the benefit of making it possible to take advantage from existing XML-based technologies, such as XPath [13], to facilitate the specification of aspectual service elements, as defined in the model. The language schema was defined following strictly the aspectual services model’s elements and their attributes, as presented in the previous section. This means that, to be syntactically correct, an aspectual service specification must be a valid WSAL document.

Each aspectual service specification is validated through a customized XML parser, which processes the WSAL document and generates a set of configurations files to be used by the aspect weaving mechanism (or weaver). The weaver uses those files to dynamically deploy one or more SOAP intermediaries over user-defined network locations. These intermediaries are configured so as to intercept the SOAP traffic according to the information provided in the original WSAL document, and, if necessary, to invoke the specified aspectual service operations. The SOAP intermediaries also are responsible for compiling the appropriate context information to be passed as input parameters to each invoked operation, and for implementing the concurrency mechanism necessary for performing asynchronous service invocations.

B. WSAL Specification Examples

We now show two examples of aspectual service specifications written using WSAL. These examples are meant to illustrate the use of WSAL for specifying crosscutting concerns in real-word service-oriented applications. The two examples illustrate the specification of aspects for monitoring service reliability and for client-side service authentication, respectively.

1) A Service Reliability Monitoring Aspect

Fig. 2 shows a WSAL document specifying an aspect to monitor the reliability of a given web service. This aspect is used to measure the percentages of valid and failure responses returned by the service.

The aspect uses a join point element of type name space to define that only SOAP messages whose contents refer to the name space "urn:ServiceB" should be monitored. In this

```
<aspect id="ReliabilityMonitorAspect" ...>
  <pointcut name="monitoredResponse"
    type="namespace"
    pattern="urn:ServiceB"/>

  <aservice name="ReliabilityMonitorService"
    location="http://localhost:8080/axis/services/Rel-Monitor">
    <advice type="afterResponse"
      pointcut-ref="monitoredResponse"
      operation="CountValidResponse"/>

    <advice type="afterException"
      pointcut-ref="monitoredResponse"
      operation="CountFaultResponse"/>
  </aservice>
</aspect>
```

Fig. 2. WSAL specification for a service reliability monitoring aspect.
example, the web service that implements the aspectual behavior provides two operations: CountValidResponse, which is invoked by an after response advice every time a SOAP response message matching the specified join point is intercepted; and CountFaultResponse, which is invoked by an after exception advice every time a SOAP exception message matching the specified join point is intercepted. The idea is that the aspectual service would also provide operations for informing the reliability of the monitored service, in terms of the percentage of valid responses in relation to total number of SOAP messages the service has returned, and for resetting the two counters.

We should call attention to the fact that, with minor modifications to the WSAL specification, that aspectual service could be easily reused to monitor different services and different service operations, independently of their implementation technologies. In addition, with that aspect in place the service monitoring concern would be completely isolated in the aspect specification and its corresponding aspectual service implementation, instead of being scattered throughout the implementation of different service consumer or service provider applications.

2) A Client Authentication Aspect

Fig. 3 in shows a WSAL document specifying an aspect for performing client-side authentication. The aspect is used to insert the appropriate client authentication credentials into SOAP requested messages sent to Google’s search web service [14], before those messages reach their final destination.

This aspect uses a join point of type service location to define that the credentials should only be inserted into SOAP request messages sent to the service located at the URL http://api.google.com/search/beta2. The actual modification of each intercepted message is performed by an aspectual service operation called AddCredential, which is invoked by an upon request advice every time a SOAP request message matching the specified join point is detected. After altering the SOAP contents of the message to accommodate the client credentials information, that operation will return the request back to the weaving mechanism (i.e., SOAP intermediary) so that the message can be forwarded to its original destination.

As it is the case with the previous example, with minor modifications to the WSAL specification this aspectual service could easily be reused to handle authentication information on behalf of different client applications, independently of their implementation technologies. Moreover, like with the service reliability monitoring concern, with this aspect enabled all information related to client authentication is completely modularized in the aspect specification and its corresponding aspectual service implementation. This level of modularization is beneficial in the sense that any modification required to the authentication process will be carried out exclusively on those components, without affecting the client applications directly. In fact, with this aspect in place client application developers may even be unaware of the authentication process. Such obliviousness of the authentication concern not only facilitates development of the application code, but also makes it easier to understand and maintain.

C. Implementation Status

As a first approach, we are using IBM’s WBI [15], a programmable web intermediary technology, to implement our aspect weaving mechanism.

Web intermediaries are computational elements deployed along the path of web transactions and that can act upon web data at different points: when a request is being issued by a client application; when the request passes through a proxy or firewall; when a response is being returned to a client application; or even to bypass the server completely, by generating a new response to a client request from scratch [16]. In the case of WBI, a web intermediary is a programmable proxy server designed to facilitate the development and deployment of intermediary applications. WBI makes it possible, by means of its programmable intermediary modules, to control the processing of web data between a client application and a web server. WBI offers five types of modules: HttpRequestEditor, HttpGenerator, HttpEditor, Monitor and autonomous. These five types of modules – collectively known as MEG (Monitor, Editor, Generator) modules – can be plugged together dynamically to compose a variety of different intermediary applications [17].

In our work we have extended WBI with SOAP support, so that we could take advantage of its sophisticated HTTP interception technology in the implementation of our aspect weaving mechanism. Our extension consisted of defining a new intermediary application specifically tailored to handle the incoming SOAP traffic using one or more customized MEGs. Each of those MEGs is automatically generated by the WSAL compiler based on the information provided in a given WSAL aspect specification.

More specifically, the weaving process comprises the following steps:

1. Identification of the plugin (aspect) to be loaded into WBI;
2. Identification of the plugin’s MEGs (i.e., Java classes

Fig. 3. WSAL specification for a client authentication aspect.
public class Security1 {

...*/

/*
 * This method uses Java’s XML DOM API to insert the required client credentials into the “key” element
 * of a SOAP request message (sent to Google’s search web service) received as an invocation parameter.
 */

public Document AddCredential (Document doc)  {
    try{
        final String tag = "key";
        this.document = doc;

        // locate the “key” element in the given XML document (request message)
        NodeList list = document.getElementsByTagName(tag);
        org.w3c.dom.Node element = list.item(0);
        org.w3c.dom.Node noAux = null;
        noAux = element.getAttributes().getNamedItem("xsi:null");
        noAux.setNodeValue(null);

        // insert the client authentication token required to access the Google search service
        element.appendChild(document.createTextNode("H/aREVQFH1KgRNztnrhD5xXqLQu2UNA"));

        /// return the modified document to the client authentication aspect
        return document;
    } catch (DOMException ex){...}
}
}

Fig. 4. Aspectual service operation for the client authentication aspect.

that inherit from the WBI’s original MEGs and that implement the semantics required for each advice type defined in the aspect) and of the MEGs’ execution and precedence rules (those rules specify the pointcuts upon which the advice will be executed and in which order);

3. Identification of the service operation that will be invoked for each advice;

4. Compilation of the necessary context information that will be passed to each advice as an invocation parameter;

5. Actual invocation of the aspectual service operations according to the invocation mode (synchronous or asynchronous) defined for each advice.

To establish an invocation standard for advices, in WSAL we follow the convention that all aspectual service interfaces, described in WSDL, must define a SOAP binding for the document literal invocation style. In this way, all aspectual service operations must implement an abstract interface which is composed of a single input parameter and a single output parameter, both of type XML Document.

As each MEG in WBI defines its on execution rules, we use those rules to specify the conditions upon which a MEG should (or should not) be instantiated at every client-server interaction captured by WBI. In other words, we use those rules to define in which context and upon which pointcuts aspectual service operations must be combined (or woven) into the HTTP-based SOAP traffic intercepted by WBI.

Once a WBI plugin has been properly configured to match the semantics of a given WSAL aspect, it is only necessary to deploy the plugin using WBI’s deployment tools to enable the aspect (i.e. to allow it to be woven into the intercepted SOAP traffic). In this way, WSAL aspects can be enabled/disabled dynamically, without the need to interrupt or reinitialize WBI or any of its affecting service-based applications.

To illustrate how this process would work in practice, here we describe a possible implementation for the client authentication aspect, whose WSAL specification is shown in Fig. 3. This aspect can be implemented using a single WBI MEG, of type Request Editor. This MEG can be programmed so as to intercept only SOAP request messages that match the conditions defined in the aspect’s pointcut (i.e., request messages sent to Google’s search web service). During execution, the MEG would retrieve the contents of each intercepted request message, and then synchronously invoke the appropriate aspectual service operation (associated with the aspect’s only advice of type upon request), passing the contents of the intercepted message (in the form of a generic XML document) as an invocation parameter.

After receiving the modified message body (now with the appropriate credentials to invoke the Google service) from the advice operation, the MEG would forward the request back to its original destination. In that particular case, the MEG would also have to adjust the SOAP request’s original HTTP transport message, so that that it would reflect the changes made in the message body (for example, to recalculate the message size in bytes).

Fig. 4 shows a possible implementation for the
AddCredential aspectual service operation, which is invoked by the client authentication aspect. In this implementation, we use Java’s DOM API to insert the required client credentials into the body of a given XML document (representing an intercepted SOAP request message that has been sent to the Google search web service), received as the operation’s only invocation parameter.

IV. RELATED WORK

Using SOAP intermediaries as a way to implement non-functional service concerns is not a novel ideal. In fact, it has been part of the SOAP specification since its early versions [3], and is now supported by most existing web service frameworks. The aspectual service model takes this idea one step ahead, since it relies on SOAP intermediaries not as the technology to implement non-functional concerns directly (which would bind the aspect implementations to the intermediaries’ underlying platform), but as the means to bring a loosely-coupled, service-based AOP discipline to the development of service-oriented applications.

Several other approaches have also proposed integrating the AOP and SOC paradigms (e.g., [7][8][18]). However, they all rely on an aspect weaving mechanism that is tightly bound to, and thus require applications and aspects to be implemented in, a specific programming language or development environment. For instance, the approaches by Verheecke and Cibrán [7] and Baligand and Monfort [18] 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 Verspecht et al. [8] relies on a weaving mechanism specific to the .NET framework, and thus requires aspects and applications to be implemented in the context of .NET, with the languages choices restricted to those languages supported in .NET. In contrast to the above approaches, in the aspectual service model aspects are implemented as loosely-coupled web services that can be weaved dynamically into SOAP-based interaction events captured over the network. Although such a dynamic weaving approach reduces the weaving scope of our model to interaction events that occur outside the applications’ execution environment, it also makes it possible to implement aspects in a way that is completely independent from any specific programming language or development environment.

Two recent works follow a slightly different approach, by integrating aspect- and service-oriented concepts at the service composition level [10][9]. Both extend BPEL4WS [19], a web service-based process composition language, with new constructs for specifying process composition aspects, and with 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 works, like our work on the unified aspectual service model, also support aspect implementation in a language- and environment-independent way. However, their weaving mechanisms 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 makes their aspect implementations tightly coupled to the underlying BPEL4WS engine, which constrains their target applications to BPEL4WS processes only. On the other hand, our unified model offers no support to specify aspects in terms of web service composition constructs. Nevertheless, our notion of aspectual services may still apply to service-oriented applications developed as web service compositions, as long as the SOAP-based interactions between the composed applications can be intercepted over the network.

V. CONCLUSION

This paper presented a unified development model that attempts to integrate the aspect- and service-oriented software development paradigms. In the proposed model, aspects are implemented as loosely-coupled aspectual services whose operations can be weaved dynamically at service interaction join points specified in terms of network- and SOAP-based properties. In contrast to other existing service-based AOP approaches, aspectual services have the benefit that they can be freely implemented and deployed in a language-independent way.

As a validation effort, we have instantiated the model in the form of a novel aspect language tailored for service-oriented applications, called Web Service Aspect Language (WSAL). To offer evidence of the language’s capabilities, in this paper we have described two examples of how WSAL could be used to modularize crosscutting service concerns in real-world service-based applications. The first example illustrated an aspect to modularize service reliability monitoring concerns, while the second example illustrated an aspect to modularize concerns related to client-side service authentication. We have also reported on the current status of our effort to implement the language’s support tools, such as the compiler and the SOAP-based aspect weaving mechanism.

As future work, once the language support tools are operational, we plan to conduct a series of case studies to empirically investigate the strength and limitations of WSAL using real-world service-oriented applications. In particular, we intend to evaluate the flexibility of our approach to cover a broad spectrum of service-based development and composition scenarios, and to measure the impact of our SOAP-based dynamic weaving mechanism on important quality of service parameters, such as throughput and performance. Finally, we consider extending WSAL with a more semantic approach, in which we could define context-aware join points by further extending the regular expression evaluation mechanism.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their comments and corrections, which significantly improved the paper.
REFERENCES


Clayton F. Silva was born in Fortaleza – CE, Brazil, in 1976. He holds a B.Sc. degree (1998) in computer science from the State University of Ceará, Brazil, and a M.Sc. degree (2006) in applied informatics from the University of Fortaleza, Brazil. His research interests include service-oriented development, aspect-oriented programming, software maintenance, software reengineering, web technologies and mobile computing.

Prof. Mendonça is a member of the IEEE Computer Society and the Brazilian Computer Society. He has been a Program Committee member for the IEEE International Conference on Software Maintenance (ICSM) since 2005, and the ACM Symposium on Applied Computing (SAC) since 2006. He has been an external reviewer for several international journals, including IEEE Transactions on Software Engineering, International Journal of Automated Software Engineering, IEEE Proceedings: Software Engineering, and Journal of Software Maintenance and Evolution: Research and Practice.

Clayton F. Silva was born in Fortaleza – CE, Brazil, in 1976. He holds a B.Sc. degree (1998) in computer science from the State University of Ceará, Brazil, and a M.Sc. degree (2006) in applied informatics from the University of Fortaleza, Brazil. His research interests include service-oriented development, aspect-oriented programming, and software engineering in general.

Mr. Silva is a Certified Function Point Specialist and a member of the International Function Point Users Group.