Aspect-Oriented Web Service Composition with AO4BPEL

Anis Charfi* and Mira Mezini

Software Technology Group
Darmstadt University of Technology
{charfi,mezini}@informatik.tu-darmstadt.de

Abstract. Web services have become a universal technology for integration of distributed and heterogeneous applications over the Internet. Many recent proposals such as the *Business Process Modeling Language (BPML)* and the *Business Process Execution Language for Web Services (BPEL4WS)* focus on combining existing web services into more sophisticated web services. However, these standards exhibit some limitations regarding modularity and flexibility. In this paper, we advocate an *aspect-oriented* approach to web service composition and present AO4BPEL, an aspect-oriented extension to BPEL4WS. With aspects, we capture web service composition in a modular way and the composition becomes more open for dynamic change.

Keywords: Adaptive web service composition, aspect-oriented programming, separation of concerns, BPEL.

1 Introduction

When the Web first emerged, it was mainly an environment for publishing data. Currently, it is evolving into a service-oriented environment for providing and accessing not only static pages but also distributed services. The Web Services [1] framework embodies the paradigm of Service-Oriented Computing (SOC) [2]. In this model, applications from different providers are offered as services that can be used, composed and coordinated in a loosely-coupled manner.

Web services are distributed autonomous applications that can be discovered, bound and interactively accessed over the Web. Although there can be some value in accessing a single web service, the greater value is derived from assembling web services into more powerful applications. Web service composition does not involve the physical integration of all components: The basic components that participate in the composition remain separated from the composite web service. As in *Enterprise Application Integration (EAI)*, specifying the composition of web services means specifying which operations need to be invoked, in what order, and how to handle exceptional situations [1].

* Supported by the German National Science Foundation (DFG) as part of the PhD program "Enabling Technologies for Electronic Commerce" at Darmstadt University of Technology.

L.-J. Zhang and M. Jeckle (Eds.): ECOWS 2004, LNCS 3250, pp. 168–182, 2004.

Several composition languages have been proposed e.g., WSCI [3], BPML [4] and BPEL4WS [5]. These languages are process-based and have their origins in *Workflow Management Systems* (WFMS) [6]. A process defines the logical dependencies between the web services to be composed by specifying the order of interactions (control flow) and rules for data transfer between the invocations (data flow). In this paper, we identify two major problems of current composition languages.

The first problem concerns the modularity of the composition specification. A real-life composite web service usually offers several composite operations, each of which is specified as a business process that in turn aggregates other more elementary operations. Such hierarchical modularization of the composition specification according to the aggregation relationships between the involved business processes might not be the most appropriate modularization schema for some aspects of the composition that address issues such as *classes of service, exception handling, access control, authentication, business rules, auditing, etc*. The code pertaining to these concerns often does not fit well into the process-oriented modular structure of a web service composition, but rather cuts across the process boundaries. Without support for modularizing such *crosscutting concerns* [7] (this term is used to describe concerns whose implementation cuts across a given modular structure of the software) their specification is scattered around the processes and tangled with the specification of other concerns within a single process. This makes the maintenance and evolution of the compositions more difficult. When a change at the composition level is needed, several places are affected – an expensive and error-prone process.

The second problem that we identify with process-oriented composition languages concerns support for dynamic adaptation of the composition logic. Such languages assume that the composition logic is predefined and static, an assumption that does not hold in the highly dynamic context of web services. In fact, new web services are offered and others disappear quite often. In addition, the organizations involved in a web service composition may change their business rules, partners, and collaboration conditions. This motivates the need for more flexible web service composition languages, which supports the dynamic adaptation of the composition.

In order to tackle these limitations, we propose to extend process-oriented composition languages with aspect-oriented modularity mechanisms. The aspect-oriented programming (AOP for short) paradigm [9] provides language mechanisms for improving the modularity of *crosscutting concerns*. Canonical examples of such concerns are authorization and authentication, business rules, profiling, object protocols, etc. [10]. The hypothesis underlying AOP is that modularity mechanisms so far support the hierarchical decomposition of software according to a single criterion, based e.g., on the structure of data (a.k.a. object-based decomposition) or on the functionality to be provided (a.k.a. functional decomposition). Crosscutting modularity mechanisms [7] supported by AOP aim at breaking with this tyranny of a single decomposition [11] and support modular implementation of crosscutting concerns. Furthermore, with support for *dynamic weaving* [12, 13, 14], aspects can be activated/deactivated at runtime. In this way, aspects can also be used to adapt the application's behavior dynamically.

In this paper we present an aspect-oriented extension to BPEL4WS (BPEL for short) and show how this extension is useful for both improving the modularity of web service composition specifications and supporting dynamic adaptations of such compositions. Our approach is not specific to BPEL, though, and can be applied to any process-oriented composition language that supports executable business

processes. BPEL was chosen as the basis technology merely because it is becoming the standard language for web service composition.

The remainder of the paper is organized as follows. Sec. 2 gives a short overview of how web service compositions are expressed in BPEL and discusses the limitations of this approach. Sec. 3 gives an overview of our aspect-oriented extension to BPEL and discusses how the limitations identified in Sec. 2 are addressed by it. We report on related work in section 4. Sec. 5 concludes the paper.

2 Process-Based Web Service Composition

In this section, we shortly introduce web service composition with BPEL as a representative for process-oriented web service composition languages and then consider its limitations.

2.1 Introduction to BPEL4WS

BPEL is a workflow-based composition language. In traditional workflow management systems, a workflow model represents a business process that consists of a set of basic and structured activities and the order of execution between them [15]. BPEL introduces control structures such as loops, conditional branches, synchronous and asynchronous communications. The building blocks of business processes are activities. There are *primitive* activities such as <invoke> and *structured* activities that manage the overall process flow and the order of the primitive activities. *Variables* and *partners* are other important elements of BPEL. Variables are used for data exchange between activities and partners represent the parties that interact with the process. Executable BPEL processes can run on any BPEL-compliant execution engine such as BPWS4J [16]. The execution engine orchestrates the invocations of the partner web services according to the process specification.

For illustration, Listing 1 shows a simple BPEL process from [16]. This process returns a string parameter back to the client whenever the operation *echo* is called. It consists of a <*sequence*> activity that contains two basic activities. The activity <*receive*> specifies that the process must wait until the client calls the operation *echo*. The activity <reply> specifies that the process has to send the message contained in the variable *request* to the client.

```
<process name = "echoString" …/>
 <variables>
   <variable name="request" messageType="StringMessageType"/>
 </variables>
 <partners>
   <partner name="caller" serviceLinkType="tns:echoSLT"/>
 </partners>
 <sequence name="EchoSequence">
   <receive  partner="caller" portType="tns:echoPT"
             operation="echo" variable="request"
```

```
            createInstance="yes" …/>
   <reply   partner="caller" portType="tns:echoPT"
            operation="echo"    variable="request"
            name="EchoReply"/>
 </sequence>
</process>
```

Listing 1. A simple process in BPEL4WS

2.2 Limitations of BPEL4WS

BPEL exhibits two major shortcomings: (a) lack of modularity for modeling crosscutting concerns and (b) inadequate support for changing the composition at runtime. In the following we elaborate on each of them.

2.2.1 Lack of Modularity in Modeling Crosscutting Concerns. As already mentioned in the introduction, a hierarchical modularization of the web service composition according to the aggregation relationships between the involved business processes might not be the most appropriate modularization schema for aspects of the composition that address issues such as classes of service, exception handling, access control, authentication, business rules, auditing, etc. Let us illustrate by the example of a simple travel service shown in Figure 1 how the code pertaining to these concerns cuts across the process boundaries and is not modularized. Our example travel web service provides the operations *getFlight* and *getHotel* which are specified as BPEL business processes (schematically represented by the vertical bars in Figure 1). A production web service usually provides several composite operations targeting different market segments.

Now, let us consider auditing [17] - an essential part of any workflow management system concerned with monitoring response times, logging, etc. An organization that composes external partner services is interested in measuring the response times of its partners because the response time of its service depends on those of the partners. The code needed for performing various auditing tasks will be scattered around the processes for composite operations, tangled in all these places with other concerns pertaining to these operations. This is because BPEL does not provide means to express in a modular way (in a dedicated module) at which points during the execution of various processes of a composite web service to gather, what auditing information. The resulting composition definition becomes complex and difficult to reason about it.

Similar modularity deficiencies can also be observed if we consider the calculation of the price for using the composite travel service in Figure 1. The code for the price calculation can certainly be encapsulated in some externalized web service. Both operations *getFlight* and *getHotel* in Figure 1 do indeed share such a common billing web service, *ws1*. What is not encapsulated, though, is the decision about "where" and "when" to trigger the billing functionality, i.e., the protocol that needs to be established between the operations *getFlight* and *getHotel* and the billing service. The code that is responsible for invoking the billing service is not modularized in one place: It crosscuts the modular process-based structure of the composition, as

illustrated by the grey area cutting across the boundaries of the vertical bars (processes) in Figure 1. As a consequence, if the billing service *ws1* is replaced by some other billing service *ws2*, or the billing policy changes, we would have to change both process specifications for *getFlight* and *getHotel*. The problem is much more critical if we have more than two composite operations, which is to be expected in real complex web services. In general, one has to find out all the code that pertains to a certain concern and change it consistently. This is unfortunate especially because business rules governing pricing policies can be expected to change often.

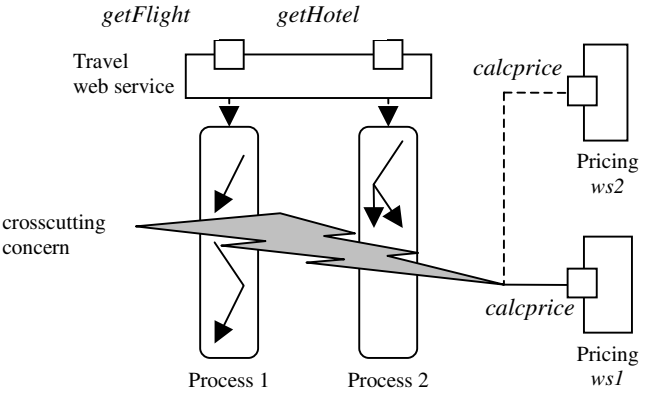


Fig. 1. Crosscuts in process-based web service composition

In general, business rules are typical examples of crosscutting concerns [18] in web service compositions. In the very competitive business context worldwide, business rules evolve very often (new partners, strategies…). Currently business rules are not well modularized in BPEL process specifications. Thus, when new business rules are defined, they get scattered over several processes. The resulting application is badly modularized, which hampers maintenance and reusability [18].

Again, the problem is that implementing business rules effects, in general, sets of points in the execution of the web services which transcend process boundaries. At present, BPEL does not provide any concepts for crosscutting modularity. This leads to tangled and scattered process definitions: One process addresses several concerns and the implementation of a single concern appears in many places in the process definition. If crosscutting concerns were well separated, process designers could concentrate on the core logic of process composition and process definitions become simpler. One would be able e.g., to exchange security policies without modifying the functional part (*independent extensibility*).

2.2.2 Changing the Composition at Runtime. When a BPEL process is deployed, the WSDL files of all the services participating in the composition must be known and once a process has been deployed, there is no way to change it dynamically. The only flexibility in BPEL is *dynamic partner binding*. This static view of the world is inherited from traditional workflow management systems from which the process-oriented web service composition model emerged; WFMS exhibit a major deficiency,

namely the inadequate support of *evolutionary* and *on-the-fly* changes demanded by practical situations [19].

However, web service compositions implement cross-organizational collaborations, a context in which several factors call for *evolution* of the composition, such as changes in the environment, technical advances like updating of web services, addition or removal of partner web services, and variation of non-functional requirements. Some of these changes require dynamic adaptation, i.e., the composition must be open for dynamic modification, which is not possible in BPEL.

One might argue that if a runtime process change is required, we just have to stop the running process, modify the composition, and restart. This is actually the only way to implement such changes in BPEL. However, this is not always a feasible solution for several reasons. First, stopping a web service may entail a loss of customers. Second, especially in B2B scenarios, a composite operation may run very long, which may prevent us from stopping it. If we stop such a long-running transaction, we must roll back or compensate all previously performed activities. Third, the modification of the composition schema on a case-by-case basis leads to all kinds of exceptions. Last but not least, several situations require runtime ad-hoc derivation from the original planned business process. In general, such derivations are necessary if users are involved in making decisions and also if unpredictable events occur [19]. Such changes should affects only some process instances at runtime.

To illustrate the issues, consider a travel agency portal which aggregates information from partner web services (airline companies and hotel chains). The resulting travel portal is a composite web service exposing three composite operations as shown in Figure 2. Each of these operations is specified as a business process in BPEL e.g., the operation *getTravelPackage* aggregates two airline web services and a hotel web service.

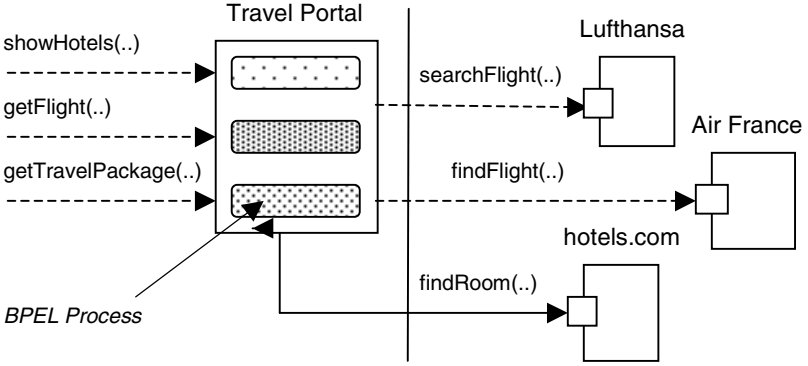


Fig. 2. A Travel portal as a composite web service

Assume that we want to add car rental to the composite operations *getTravelPackage* and *getFlight*. This way, when the client requests a flight or a travel package, she also gets propositions for car rental. One can also envisage variations of this adaptation, where such an offer is only made to frequent customers

or to those clients that have specified interest in the offer in their profiles. In this case, the adaptation should be effective only for specific business process instances.

It should be noted that the adaptations of the composition logic such as the one illustrated here require, in general, a semantic adaptation mechanism to bridge the *compositional mismatch* [15]. In *component composition languages* [20] this kind of adaptation is enabled by *glue code*. Similarly, when we compose web services, we have to adapt their interfaces by writing glue code. BPEL supports only *data adaptability* by means of the <*assign*> activity. This activity allows one to create new messages or modify existing ones, using parts of other messages, XPath expressions, or literal values. Data adaptability deals with the structure of data. Glue code support is still missing in BPEL.

3 Aspect Oriented Web Service Composition

Aspect-Oriented Programming (AOP) [9] is a programming paradigm explicitly addressing the modularization of crosscutting concerns, which makes AOP the technology of choice to solve the problems discussed in Sec. 2. While it has been mostly applied to object-oriented programming, it is applicable to other programming styles [21], including the process-oriented style. We propose to use aspects as a complementary mechanism to process-oriented web service composition and argue that the definition of dynamic aspects at the BPEL level allows for more modularity and adaptability.

3.1 Introduction to Aspect-Oriented Programming

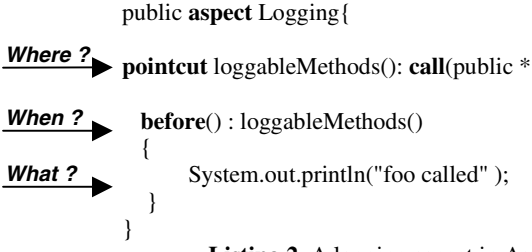
AOP introduces a new unit of modularity called *aspect* aimed at modularizing crosscutting concerns in complex systems. In this paper, we will use the terminology of AspectJ [22], the most mature AOP language today. In this terminology, there are three key concepts of AOP: *join points*, *pointcuts* and *advice*. Join points are points in the execution of a program [22]. In object-oriented programs, examples of join points are method calls, constructor calls, field read/write, etc.

In order to modularize crosscuts, a means is needed to identify related join points. For this purpose, the notion of a *pointcut* is introduced – a predicate on attributes of join points. One can select related method execution points, e.g., by the type of their parameters or return values, by pattern matching on their names, by their modifiers, etc. Similar mechanisms are available to select sets of related setter / getter execution points, sets of constructor calls / executions, exception handlers, etc. Current AOP languages come with predefined pointcut constructs (pointcut designators) in AspectJ.

Finally, behavioral effect at join points identified by a pointcut is specified in an *advice*. The advice code is executed when a join point in the set identified by the pointcut is reached. It may be executed before, after, or instead, the join point at hand, corresponding to *before*, *after* and *around* advice. The code specified in a before, respectively after advice is executed before, respectively after the join points in the associated pointcut have executed. With the around advice the aspect can control the

execution of the original join point: It can integrate the further execution of the intercepted join point in the middle of some other code to be executed around it.

An aspect module consists in general, of several pointcut definitions and advice associated to them. In addition, it may define state and methods which in turn can be used in the advice code. Listing 2 shows a simple logging aspect in *AspectJ*, which defines a pointcut *loggableMethods* specifying **where** the logging concern should be integrated into the execution of the base functionality – in this case, the interesting join points are the executions (the *call* pointcut designator) of all public methods called *bar*, independent of the class they are defined in, their return type, as well as the number and type of the parameters (wildcards * and ..). The aspect also defines an advice associated to the pointcut *loggableMethods* that prints out a logging message *before* any of the points in *loggableMethods* is executed. The advice specifies **when** and **what** behavior must execute at the selected join points.



Listing 2. A logging aspect in AspectJ

Integrating aspects into the execution of the base functionality is called *weaving*. In static AOP approaches, as e.g., in AspectJ, at compile-time/load-time pointcuts are mapped to places in the program code whose execution might yield a join point at runtime. The latter are instrumented to add calls to advice and eventually dynamic checks that the identified places in code do actually yield a join point at runtime [23]. In dynamic AOP [12, 13, 14] languages, aspects can be (un)deployed at application runtime, the behavior of which can thus be adapted dynamically.

3.2 Overview of AO4BPEL

Here we present AO4BPEL, an aspect-oriented extension to BPEL4WS, in which aspects can be (un)plugged into the composition process at runtime. Since BPEL processes consist of a set of activities, join points in our model are well-defined points in the execution of the processes: Each BPEL activity is a possible join point. Pointcuts in AO4BPEL are a means for referring to (selecting) sets of join points that span several business processes at which crosscutting functionality should be executed. The attributes of a business process or of a certain activity can be used as predicates to choose relevant join points. E.g., to refer to all invocations of a partner web service, we use the attributes *partnerLink* and *portType* of the activity <invoke>. Since BPEL processes are XML documents, *XPath* [24] – a query language for XML documents – is a natural choice as the pointcut language. In an AO4BPEL aspect, the element <pointcut> is an XPath expression selecting those activities where the

execution of additional crosscutting functionality will be integrated. XPath provides logical operators, which can be used to combine pointcuts.

Like AspectJ, we support *before*, *after* and *around* advice. An *advice* in AO4BPEL is an activity specified in BPEL that must be executed before, after or instead of another activity. The *around* advice allows replacing an activity by another (dummy) activity. Sometimes we need to define some advice logic which cannot be expressed in BPEL4WS. One could use code segments in a programming language like Java in Collaxa's JBPEL [25] for this purpose. However, this breaks the portability of BPEL processes, which is the reason for us to use what we call *infrastructural web services*. Such services provide access to the runtime of the orchestration engine. We set up a *Java code execution web service*, which invokes an external Java method in a similar way to Java Reflection. Each code snippet that is required within an AO4BPEL advice can be defined as a static method in a Java class.

Figure 3 sketches the overall architecture of our aspect-aware BPEL orchestration engine. The system consists of five subcomponents: the process definition and deployment tool, the BPEL runtime, the aspect definition and deployment tool, the aspect manager, and the infrastructural services. The core components are the *BPEL runtime* and the *aspect manager*. The *BPEL runtime* is an extended process interpreter. It manages process instances, message routing and takes aspects into account. The *aspect definition and deployment tool* manages the registration and activation of aspects. The *aspect manager* controls aspect execution.

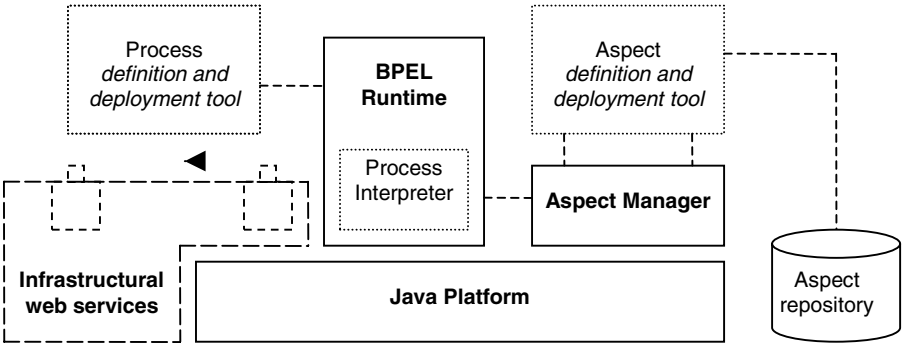


Fig. 3. Architecture of an aspect-aware web service composition system

In our first implementation, we intend to support only <invoke> and <reply> join points because basic activities represent the interaction points of the composition with the external partners. The most straightforward way to implement a dynamic aspect-aware orchestration engine is to extend the process interpreter function to check if there is an aspect before and after the interpretation of each activity. If this is the case, the aspect manager executes the advice and then returns control to the process interpreter. We believe that for the first prototype, the performance overhead induced by these local checks is negligible compared to the cost of interacting with an external web service.

3.3 Examples Revisited

In this section, we show how the examples from Section 2 are modeled in AO4BPEL.

3.3.1 Modularizing Non-functional Concerns. To illustrate the modularization of crosscutting concerns, consider the AO4BPEL aspect *Counting* in Listing 3 which collects auditing data: It counts how many times the operation *searchFlight* of Lufthansa has been invoked. The counting advice must execute after each call to that operation. Ideally, we have to save this information into a file in order to evaluate it later, i.e., we need to access the file system. This cannot be done in BPEL and some programming logic is necessary. The *Java code execution web service* comes into play here. This web service provides the operation *invokeMethod* which takes as parameters the class name, the method name, and the method parameters as strings. It supports only primitive parameter types i.e., integers and strings. When the process execution comes to a join point captured by the pointcut *Lufthansa Invocations*, the static method *increaseCounter* is called, which opens a file, reads the number of invocations, increments it and then saves the file to disk.

```
<aspect name="Counting">
<partnerLinks>
  <partnerLink name="JavaExecWSLink" …/>
</partnerLinks>
<variables>
 <variable name="invokeMethodRequest" …/>
</variables>
<pointcutandadvice type="after">
 <pointcut name="Lufthansa Invocations">
   //process//invoke[@portType ="LufthansaPT" and
   @operation ="searchFlight"]
 </pointcut>
 <advice>
  <sequence>
   <assign>
    <copy>
     <from>increaseCounter</from>
     <to variable="invokeMethodRequest" part="methodName"/>
    </copy>…
   </assign>
   <invoke partnerLink="JavaExecWSLink" portType="JavaExecPT"
           operation="invokeMethod"
           inputVariable="invokeMethodRequest"/>
  </sequence>
 </advice>
</pointcutandadvice>
</aspect>
```

Listing 3. The counting aspect

This aspect shows how crosscutting concerns can be separated from the core of the composition logic. The monitoring functionality is not intertwined with the process definition. Moreover, we can define several monitoring aspects implementing different policies and weave the appropriate one according to the context.

3.3.2 Changing the Composition. In Section 2, we wanted to add car rental business logic into the composite operations *getTravelPackage* and *getFlight,* and argued that such an adaptation cannot be performed dynamically with BPEL. For achieving the same goal in AO4BPEL, the administrator defines the aspect *AddCarRental* shown in Listing 4. This aspect declares a pointcut, which captures the accommodation procurement activity in the *getTravelPackage* and the flight procurement activity in *getFlight*. The car rental activity must be executed after the join point activities referred to by the pointcut (after advice). This aspect also declares partner links, variables, and assignment activities. The <*assign*> activity is required to transform the data returned by the operations *getTravelPackage* and *getFlight.*

```
<aspect name="AddCarRental">
<partnerLinks>
 <partnerLink name="carRentalPortal" …/>
</partnerLinks>
<variables>
 <variable name="getCarRequest" …/>
 <variable name="getCarResponse" …/>
</variables>
<pointcutandadvice type= "after">
<pointcut name="accommodation procurement">
 //process[@name="getTravelPrcs"]//sequence[@name="FlightHotel"]
 //invoke[@portType ="HotelPT" and @operation ="findRoom"] or
 //process[@name="getFlightPrcs"]//flow[@name="FlightSearchFlow"]
</pointcut>
<advice>
 <invoke partnerLink="CarRentPortal" portType="carRentPT"
         operation="getCar" inputVariable="getCarRequest"
         ouputVariable="getCarResponse"/>
 <assign>
  …
 </assign>
</advice>
</pointcutandadvice>
</aspect>
```

Listing 4. The car rental aspect

The administrator must register the aspect with the BPEL execution engine. During the registration, the *aspect definition and deployment unit* of our BPEL engine requests the programmer to input the WSDL and the port address of the *car rental* web service. This step also requires the *partnerLinkTypes* that are used by the aspect. The aspect becomes active only after explicit deployment. The aspect activation can be performed dynamically while the respective process is running. This way, we apply the adaptation behavior at runtime.

One can conclude that this aspect tackles the problem outlined in Section 2. It allows for dynamic change. We specified the new business rule in a modular way as an aspect. If business rules change, we only have to activate/deactivate the appropriate aspect at execution time.

4 Related Work

Several research works recognize the importance of flexible and adaptive composition. *Self-Serv* [26] is a framework for dynamic and peer-to-peer provisioning of web services. In this approach, web service composition is specified declaratively by means of state charts. *Self-Serv* adopts a distributed decentralized, peer-to-peer orchestration model, whereby the responsibility of coordinating the execution of a composite service is distributed across several *coordinators*. This orchestration model provides greater scalability than a centralized one w.r.t. performance. In addition, Self-Serv introduces the concept of *service communities*, which are containers for alternative services. Separating the service description from the actual service provider increases the flexibility. However, unlike our approach, Self-Serv does not support dynamic process changes such as adding new web service type to the composition (cf. Sec.2).

Örriens et a.l [27], present a framework for *business rule driven* composition providing composition elements like *activities* and *composition rules*. Due to the separation of the activities from the specification of their composition, the latter can easily evolve by applying new composition rules. With AO4BPEL aspects, we achieve a similar effect of separating the main activities from their composition logic, while remaining compliant with BPEL4WS - a de-facto composition standard. In contrast to AO4BPEL, the approach presented in [27] does not support dynamic change and is rather geared towards the service composition lifecycle.

Adaptive workflow [19, 28, 29] provides flexible workflow models that are open for change. Similar to these works, AO4BPEL aims at making process-based composition more open for change. In addition, it addresses the issue of process modularity. We think that many concepts and results from adaptive workflow remain valid for our work e.g., the *verification and correctness of workflow modifications*.

Casati et. al, [29] present *eFlow*, which is a platform for specifying, enacting, and monitoring composite e-services (a predecessor of web services). eFlow models composite web services using graphs. It supports dynamic process modifications and differentiates *ad hoc change* (applies to a single process instance) and *bulk* change (applies to many or all process instances). Unlike AO4BPEL, eFlow supports change by migration of process instances from a *source schema* to a *destination schema*. This migration obeys several *consistency rules*. The advantage of our approach over eFlow is that we do not have to migrate the whole process instance from a source schema to a destination schema, we just weave sub-processes or advices at certain join points.

Dynamic AOP [12, 13, 14, 30] has been recognized as a powerful technique for dynamic program adaptation also by other authors. In [31], dynamic aspects are used for third-party service integration and for hot fixes in mobile communication systems. The idea of using AOP for increasing the flexibility of workflows was identified in [32], where the authors propose to model workflow according to different decomposition perspectives (dataflow, control flow, and resources). Along the same lines, *Bachmendo and Unland* [33] propose using dynamic aspects for workflow evolution within an object-oriented workflow management system.

There are two important differences between the approaches cited in the foregoing paragraph and the work presented here. First, none of the works presented in [31, 32, 33] directly targets the domain of web services. Second, all these approaches (mis)use dynamic AOP merely as an adaptation (patch) technique and do not use it as a

modularization technique for crosscutting concerns, which it actually is. The second argument also applies to [34] – the only work that intends to apply dynamic AOP to web service composition known to the authors of this paper. In [34], dynamic aspects are used to hot-fix workflow and to configure and customize the orchestration engine. This approach is more *interceptor-based* than really aspect-oriented. Similar to [32, 33], the proposal in [34] uses dynamic AOP merely as an adaptation technique and not as a modularization technique for crosscutting. The advantage of our approach over these works is the *quantification* [35] i.e., the pointcut language of AO4BPEL which allows us to capture join points that span several processes in a modular and abstract way. We illustrated the crosscutting nature of functional and non-functional concerns especially observable when several processes are considered.

WSML [36] is a client-side web service management layer, which realizes dynamic selection and integration of services. WSML is implemented using the dynamic AOP language JAsCo [37]. Our work on AO4BPEL and the work on WSML nicely complement each other, since WSML does not address the problem of web service composition, while AO4BPEL is not concerned with the middleware issues of web-service management.

5 Conclusion

In this paper, we discussed limitations of web-service composition languages with respect to modularity and dynamic adaptability. We argued that the overall static model of the web-service composition languages inherited from the workflow management systems is not able to address issues resulting from the highly dynamic nature of the world of web services. To address these limitations, we presented an aspect-oriented extension to BPEL4WS where aspects and processes are specified in XML. We discussed how our approach improves process modularity and increases the flexibility and adaptability of web service composition.

The major contribution of our work is to elucidate why AOP is the right technique to address the problems of BPEL by arguing that these problems emerge due to the lack of crosscutting modularity. BPEL processes are the counterpart to classes in OOP. That is, similar to OO which mainly supports hierarchical decomposition in objects that are composed of other simpler objects the process-oriented languages support hierarchical decomposition of systems into processes that are recursively composed of other processes. Such a single decomposition schema while appropriate for expressing the process structure in a modular way, is, however, not well-suited for expressing other concerns that cut across module boundaries. With its ability to quantify over given module boundaries [35] by means of pointcuts that span several processes, AO4BPEL is able to capture such concerns in a modular way. This is actually the key message put forward in this paper.

Currently, we are still working on a first prototype of the aspect-aware orchestration engine, which manages BPEL aspects and processes. In the future, we intend to enhance the interface for context passing between BPEL aspects and the base processes. We will also examine the correctness properties of process adaptation. Another direction for future research is to investigate whether semantic web and ontologies may enable semi-automatic generation of adaptation aspects in case of dynamic changes.

Acknowledgements. We would like to thank D. Karastoyanova, M. Haupt, and S. Kloppenburg for comments on earlier drafts of this paper. We also thank the anonymous reviewers for their suggestions for improving this paper.

References

1. G. Alonso, F. Casati, H. Kuno,V. Machiraju. *Web Services: Concepts, Architectures, and Applications*. Springer, 2004.
2. M. P. Papazoglou. *Service-Oriented Computing: Concepts, Characteristics and Directions.* 4th Int. Conference on Web Information Systems Engineering (WISE'03), Italy, 2003.
3. A. Arkin et al., *Web Service Choreography Interface* 1.0, W3C, 2002.
4. A. Arkin et al., *Business Process Modeling Language*- BPML 1.0, 2002.
5. T. Andrews et al., *Business Process Execution Language for Web Services* 1.1, May 2003.
6. D. Georgakopoulos, M. Hornick, A. Sheth. *An Overview of Workflow Management: from process modeling to workflow automation infrastructure.* Distributed and Parallel Databases, April 1995.
7. H. Masuhara, G. Kiczales. *Modeling Crosscutting in Aspect-Oriented Mechanisms.* In
8. Proceedings of ECOOP2003, LNCS 2743, pp.2-28, Darmstadt, Germany, 2003.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. *Aspect-oriented Programming*. In ECOOP'97, LNCS 1241, pp. 220-242, 1997.
10. R. Laddad. *AspectJ in Action*. Manning Publications, 2003.
11. P. Tarr, H. Ossher, W. Harrison, S.M. Sutton. *N degrees of Separation: Multidimensional separation of concerns.* Proc. ICSE 99, pp. 107-119, 1999.
12. C. Bockisch, M. Haupt, M. Mezini, K. Ostermann. *Virtual Machine Support for Dynamic Join points.* Proceedings of the 3rd AOSD conference, Lancaster, UK, 2004.
13. R. Pawlak, L. Seinturier, L. Duchien, G. Florin. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java.* Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Japan, 2001.
14. B. Burke, M. Flury. *JBoss AOP*, http://www.jboss.org/developers/projects/jboss/aop.jsp.
15. R. Khalaf, N. Mukhi, S. Weerawarana. *Service-Oriented Composition in BPEL4WS.* WWW2003 conference, Budapest, Hungary, 2003.
16. The IBM BPEL4WS Java™ Run Time, http://www.alphaworks.ibm.com/tech/bpws4j.
17. V. Tosic, W. Ma, B. Pagurek, B. Esfandiari . *Web Services Offerings Infrastructure(WSOI) - A Management Infrastructure for XML Web Services*. Proc. of NOMS 2004, Seoul, 2004.
18. M. D'Hondt, V. Jonckers. *Hybrid Aspects for Weaving Object-Oriented Functionality and Rule-Based Knowledge*. Proceedings of the 3rd AOSD conference, Lancaster, UK, 2004.
19. Y. Han, A. Sheth, C. Bussler. *A Taxonomy of Adaptive Workflow Management.* CSCW'98 Workshop on Adaptive Workflow, USA, 1998.
20. F. Achermann, O. Nierstrasz. *Applications = Components + Scripts — A Tour of Piccola.* Software Architectures and Component Technology, Kluwer, 2001.
21. Y. Coady, G. Kiczales. *AspectC*, http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.
22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. *An overview of AspectJ*. In Proceedings of the ECOOP 2001, Budapest, Hungary, 2001.
23. E. Hilsdale, J. Hugunin. *Advice Weaving in AspectJ*. Proceedings of the 3rd AOSD conference, Lancaster, UK, 2004.
24. J. Clark. *XML path language* (XPATH), 1999 http://www.w3.org/TR/xpath.
25. Collaxa BPEL Server, http://www.collaxa.com

26. B. Benatallah, Q. Sheng, M. Dumas. *The Self-Serv Environment for Web Services Composition.* IEEE Internet Computing, January / February 2003.
27. B. Orriëns, J. Yang, M.P. Papazoglou. *A Framework for Business Rule Driven Web Service Composition*. ER (Workshops), Chicago, USA, 2003.
28. C. Bussler. *Adaptation in Workflow management.* Proceedings of the Fifth International Conference on the Software Process, CSOW, Illinois, USA, June 1998.
29. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, M.Shan. *Adaptive and Dynamic Service Composition in eFlow.* In Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE), Sweden, June 2000.
30. Y. Sato, S. Chiba, M. Tatsubori. *A Selective Just-in-Time Aspect Weaver.* Proceedings of the GPCE 03 conference, LNCS 2830, Erfurt, September 2003.
31. R. Hirschfeld, K. Kawamura. *Dynamic Service Adaptation.* 4th International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, Tokyo, Japan, 2004.
32. R. Schmidt, U.Assmann. *Extending Aspect-Oriented-Programming in order to flexibly support Workflows.* AOP Workshop, ICSE 98, USA, 1998.
33. B. Bachmendo, R. Unland. *Aspect-based Workflow Evolution.* Workshop on AOP and separation of concerns, Lancaster, UK, 2001.
34. C. Courbis, A. Finkelstein. *Towards an Aspect-Weaving BPEL-engine.* ACP4IS Workshop, 3rd AOSD conference, Lancaster, UK, 2004.
35. R.E. Filman, D.P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Advanced Separation of Concerns Workshop, OOPSLA 2000, Minneapolis, USA, 2000.
36. B. Verheecke, M. Cibran. *AOP for Dynamic Configuration and Management of Web Services.* International Conference on Web Services Europe 2003, Erfurt, 2003.
37. D. Suvee, W. Vanderperren, V. Jonckers. *JAsCo: an aspect-oriented approach tailored for component based software development.* 2nd AOSD conference, Boston, USA, 2003.