World Wide Web (2007) 10:309–344
DOI 10.1007/s11280-006-0016-3

AO4BPEL: An Aspect-oriented Extension to BPEL

Anis Charfi · Mira Mezini

Received: 7 September 2004 / Revised: 31 July 2005 /
Accepted: 16 August 2005 / Published online: 27 March 2007
© Springer Science + Business Media, LLC 2007

Abstract Process-oriented composition languages such as BPEL allow Web Services to be composed into more sophisticated services using a workflow process. However, such languages exhibit some limitations with respect to modularity and flexibility. They do not provide means for a well-modularized specification of crosscutting concerns such as logging, persistence, auditing, and security. They also do not support the dynamic adaptation of composition at runtime. In this paper, we advocate an aspect-oriented approach to Web Service composition and present the design and implementation of AO4BPEL, an aspect-oriented extension to BPEL. We illustrate through examples how AO4BPEL makes the composition specification more modular and the composition itself more flexible and adaptable.

Keywords web service composition · workflow · separation of concerns · aspect-oriented programming · modularization · adaptation · BPEL

1 Introduction

Web Services [1] 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, combining existing Web Services into more sophisticated Web Services often brings more value [1]. There are two complementary facets of Web Service composition: orchestration and choreography [68]. They address different aspects of creating business processes from Web Services.

A. Charfi (✉) · M. Mezini
Software Technology Group, Darmstadt University of Technology, Hochschulstr 10, 64289 Darmstadt, Germany

Orchestration refers to an executable business process that interacts with other Web Services [68]. It represents control from one party's perspective. Several orchestration languages have been proposed such as the Business Process Modeling Language (BPML) [4] and the Business Process Execution Language for Web Services (BPEL, formerly known as BPEL4WS) [24]. BPEL is a process-oriented Web Service orchestration language, in which an executable workflow process defines the flow of control and data among the participant Web Services. An orchestration specifies the implementation of a composite Web Service.

Choreography is more collaborative than orchestration. Choreography languages such as the Web Service Choreography Description Language (WS-CDL) [84] describe interaction protocols between multiple parties from a global viewpoint [68]. They track the message sequences, typically the public message exchanges that occur between Web Services, rather than a specific business process executed by a single party. A Web Service choreography specifies an interoperable peer-to-peer collaboration between Web Services regardless of implementation details. Web Service choreographies are thus not directly executable.

In this paper, we focus on Web Service orchestration languages and more specifically on BPEL. We identify two problems. First, we argue that BPEL lacks means to modularize *crosscutting concerns* [57], i.e. concerns that cut across the modular structure of BPEL processes (e.g., logging, persistence, and security). Second, we observe that BPEL does not provide appropriate support for dynamic adaptation of the composition.

In a complex Web Service that results from the composition of other Web Services, several operations are specified by means of executable business processes, which in turn aggregate other Web Service operations. This hierarchical modularization of the composition specification, according to the aggregation relationships between the Web Services involved, is a powerful tool to master the complexity of composite Web Services. However, it is not the most appropriate modularization schema for crosscutting concerns such as logging, persistence, security, and business rules [9, 21, 28, 31, 55, 70, 85]. Code pertaining to these concerns does not fit well into the process-oriented modular structure of a composite Web Service, because it cuts across the process boundaries. Even if one could capture the core logic for these concerns in external middleware services, the protocol for invoking the latter would remain scattered all over the processes and tangled with the specification of other concerns, thereby making the maintenance and evolution of the composite services more difficult. When a change at the composition level is needed, several places are affected; this makes changes expensive and error-prone.

Process-oriented composition languages such as BPEL lack support for dynamic adaptation of the composition. These languages originate from workflow management systems [36] and assume that the composition logic is predefined and static. This assumption does not hold in the often highly dynamic context of Web Services. In fact, a running BPEL process may require runtime adaptation because of unexpected situations and failures such as the unavailability of a partner Web Service, variations of the quality of service properties of a partner, or changes in regulations and collaboration conditions. This calls for more flexible Web Service composition languages that support dynamic adaptation (especially in the case of long-running processes).

In order to address these limitations, we propose to extend process-oriented composition languages with aspect-oriented mechanisms. The Aspect-Oriented Programming (AOP) paradigm [50] provides language support for improving the modularity of crosscutting concerns. Furthermore, with *dynamic weaving* (i.e., the dynamic integration of aspects with the process [8, 10, 67]), the application's behavior can be adapted at runtime.

The contributions of this paper are threefold. First, we present the shortcomings of BPEL with respect to crosscutting concern modularization and dynamic adaptation. Second, we analyze the benefits of applying aspect-oriented techniques to Web Service composition. Third, we present the design and implementation of AO4BPEL, an aspect-oriented extension to BPEL, and illustrate through examples how it solves the deficiencies mentioned above.

The remainder of this paper is organized as follows. In Section 2, we study how Web Service compositions are expressed in BPEL and discuss the limitations of this approach. In Section 3, we present the design of AO4BPEL and illustrate through examples how the limitations identified in Section 2 are addressed by AO4BPEL. In Section 4, we describe the prototype implementation of AO4BPEL. In Section 5, we report on related work. Finally, in Section 6, we conclude the paper and outline directions for future work.

2 Composing Web Services with BPEL

Web Service composition is about combining existing Web Services into a more sophisticated Web Service. Process-oriented Web Service composition languages such as BPEL [24] and BPML [4] define a workflow process [36], which specifies a set of interactions between the participant Web Services. The process also specifies the control flow (i.e., the ordering of the interactions) and the data flow (i.e., the data that is exchanged between the participant Web Services) underlying the composition. In this section, we introduce BPEL, a well-known example of process-oriented Web Service composition language. Next, we use a scenario to illustrate the lack of modularity and adaptability in BPEL.

2.1 Introduction to BPEL

BPEL is a workflow-based Web Service composition language in which the composition is also exposed as a Web Service. BPEL 1.0 was proposed in July 2002 by BEA, IBM, and Microsoft. In April 2003, BPEL 1.1 was submitted to OASIS for standardization via the Web Services BPEL Technical Committee [60]. The output of this committee has been renamed WS-BPEL 2.0 [5] for compliance with the naming conventions of Web Service specifications.

BPEL differentiates *primitive* activities and *structured* activities. Primitive activities are atomic whereas structured activities are composite. A BPEL process typically starts with a *receive* activity, which blocks waiting for a client request. The *invoke* activity calls an operation on a partner Web Service. The *reply* activity sends a response to the client. The parties that interact with the composition (i.e., the clients and Web Services) are called *partners*. A *partner link* is a typed connector between two WSDL port types. It specifies two roles: one played by the composition and the other by the partner [48]. The process data is held in *variables*; each variable has a message type that is defined in the WSDL file of the composition or one of its partners. The *assign* activity is used to modify the content of a variable.

Structured activities such as *sequence*, *flow*, and *switch* act as containers for other activities. They structure the latter according to predefined control flow patterns such as sequential execution, concurrency, and conditional branching. In the *flow* activity, additional ordering constraints can be expressed by using *links* (i.e., control flow edges connecting a *source* activity with a *target* activity). A *link* specifies that the target activity can only execute after the source activity completes; it can also have a boolean *transition condition* attached to it [48].

BPEL also defines *fault handlers*, *compensation handlers*, and *event handlers*. These handlers are attached to either the process or a *scope* activity; the latter provides context for the activities nested within it. If an error arises inside the scope, the corresponding fault handler is executed. Compensation handlers are used to undo already successfully completed activities. Event handlers specify an action that should be taken when a certain event occurs, e.g. when a messages comes in (*message events*) or a timer goes off (*alarm events*).

As several instances of the same process may be running concurrently, a mechanism is needed for routing Simple Object Access Protocol (SOAP) [83] messages to the correct process instance. BPEL introduces *correlation sets* to correlate messages with process instances by matching specific message parts with the process variables.

BPEL combines two styles of workflow process modeling. The *structured style*, which is derived from process calculus [40, 59], defines complex control constructs such as BPEL structured activities. The *graph style* defines the control flow by using

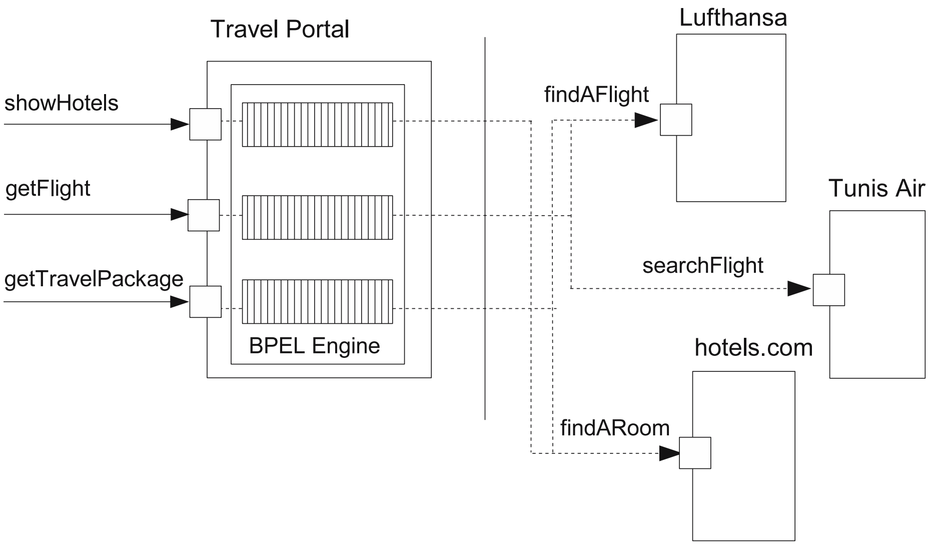

Figure 1 Example of a composite web service: a travel portal.

edges between activities. The control flow is explicit in the graph style, whereas it is implicit in the structured style. These two styles have a similar expressiveness [48].

Executable BPEL processes can run on any BPEL-compliant orchestration engine (e.g., BPWS4J [41]). The engine orchestrates the invocations of the partner Web Services according to the process specification.

For illustration, consider a travel portal that aggregates travel information from airline companies and hotel chains. The result of the aggregation is a composite Web Service exposing three operations (see Figure 1). Each of these operations is specified by a BPEL process (e.g., the operation *getTravelPackage* aggregates Lufthansa's Web Service and the hotels.com Web Service). The horizontal bars represent BPEL processes and the dashed lines show the partner operations that are invoked from each process.

Listing 1 shows the process that implements the operation *getTravelPackage*. As input, it takes a departure date, a return date, a departure city, and a destination city; it returns a string description of a vacation package. The process response is held in the variable *clientresponse* (line 9). This process also declares three *partner links* that respectively connect the composition with the client, the Lufthansa Web Service, and the hotels.com Web Service (lines 3–5). The main activity of this process is a *sequence* activity (lines 15–38).

```
 1  <process name="travelPackage" .../>
 2   <partnerLinks>
 3    <partnerLink name="client" partnerLinkType="clientPLT" .../>
 4    <partnerLink name="flight" partnerLinkType="LufthansaPLT" .../>
 5    <partnerLink name="hotel" partnerLinkType="hotelPLT" .../>
 6   </partnerLinks>
 7   <variables>
 8    <variable name="clientrequest" messageType="tns:findPackageRequest"/>
 9    <variable name="clientresponse" messageType="tns:findPackageResponse"/>
10    <variable name="flightrequest" messageType="fs:findAFlightRequest"/>
11    <variable name="flightresponse" messageType="fs:findAFlightResponse"/>
12    <variable name="hotelrequest" messageType="hs:findARoomRequest"/>
13    <variable name="hotelresponse" messageType="hs:findARoomlResponse"/>
14   </variables>
15   <sequence name="packageSequence">
16     <receive name="receiveClientRequest"
17           partnerLink="client" portType="tns:travelServicePT"
18           operation="getTravelPackage" variable="clientrequest" createInstance="true"/>
19     <assign> ...</assign>
20     <invoke name="invokeFlightService"
21          partnerLink="flight" portType="LufthansaPT" operation="findAFlight"
22          inputVariable="flightrequest" outputVariable="flightresponse"/>
23     <invoke name="invokeHotelService"
24          partnerLink="hotel" portType="HotelPT" operation="findARoom"
25          inputVariable="hotelrequest" outputVariable="hotelresponse"/>
26     <assign >
27      <copy>
28      <from variable="flightresponse" part="findAFlightReturn"></from>
29       <to variable="clientresponse" part="flightInfo"></to>
30      </copy>
31      <copy>
32      <from variable="hotelresponse" part="findARoomReturn"></from>
33       <to variable="clientresponse" part="hotelInfo"></to>
34      </copy>
35     </assign>
36     <reply name="replyToClient" partnerLink="client" portType="tns:travelServicePT"
37         operation="getTravelPackage" variable="clientresponse" />
38    </sequence>
39 </process>
```

Listing 1 The travel package process

In this paper, we will focus on BPEL because it is expected to become the standard for Web Service composition [5] and it is already widely accepted in both academia and industry. A number of application servers already provide support for BPEL (e.g., IBM WebSphere [38], Oracle Application Server [42], and Microsoft BizTalk Server [58]).

2.2 Limitations of BPEL

To motivate the need for mechanisms for crosscutting modularity and dynamic adaptation, we will study some concerns whose implementation would benefit from such mechanisms.

2.2.1 Case studies

In this section, we will investigate the implementation of some example crosscutting concerns in the travel portal scenario defined in Section 2.1. Concretely, we will consider data collection for billing, data persistence, measurement of activity execution time, logging, and business rules. We will show that the implementation of these concerns cuts across the process dimension and cannot be expressed in a modular way using BPEL constructs. In conformance with the definition given in [51], we use the term *modular* to mean "in a localized manner and with well-defined explicit interfaces to the rest of the composition logic."

Data collection for billing: In our travel agency scenario, assume that Lufthansa charges a fee for using its Web Service. Several pricing models can be considered. One possibility is to charge only clients who use the Web Service more than 100 times per day. Another pricing model is to charge clients who use the Web Service to search for flights without booking any flights subsequently. Yet other models are conceivable.

When such pricing policies are enforced, the travel agency will get a bill from Lufthansa, but it has no easy means to check whether the bill is accurate. To verify the bill, the travel agency can count how many times Lufthansa's Web Service has been called from within any BPEL process that is deployed on the orchestration engine; all process instances must be taken into account.

To do so, the programmer has to examine the process definitions and find out where the operation *findAflight*[1] is called. In our example, this operation is invoked from the processes implementing *getFlight* and *getTravelPackage*. Both processes must be changed as shown in Listings 2 and 3. After each invocation of Lufthansa's Web Service, we call the counting Web Service to increment the invocation counter. The counting Web Service provides the operation *increaseCounter*, which takes an integer parameter as input and adds it to the invocation counter. For each process, we also have to add a new partner link for the counting Web Service (line 3) and a variable (line 5) to be used as input by the new *invoke* activity. In order to assign the

[1]In a more realistic scenario, other operations of the Lufthansa Web Service could also be called across the process boundaries; but for the sake of clarity, let us assume there is only one.

correct value to that variable (the input parameter of the operation *increaseCounter*), we use an *assign* activity that is put together with the *invoke* activity into a *sequence* activity (lines 11–20).

```
 1  <process name="travelPackage" .../>
 2     ...
 3     <partnerLink name="CounterWS" partnerLinkType="CounterPLT" .../>
 4     ...
 5     <variable name="increaseRequest" messageType="increaseCounterInput"/>
 6     ...
 7    <sequence name="packageSequence">
 8     <receive name="receiveClientRequest" .../>
 9      <assign>...</assign>
10      <invoke name="invokeFlightService" operation="findAFlight" .../>
11      <sequence name="collect billing data">
12       <assign>
13        <copy>
14         <from expression="1"/>
15         <to variable="increaseRequest" part="increaseBy"/>
16        </copy>
17       </assign>
18       <invoke partnerLink="CounterWS" portType="CounterPT"
19             operation="increaseCounter" inputVariable="increaseRequest" .../>
20     </sequence>
21
22      <invoke name="invokeHotelService" operation="findARoom" .../>
23      ...
24     <reply name="reply" partnerLink="client" .../>
25   </sequence>
26 </process>
```

Listing 2 Collecting billing data in the travel process

```
 1  <process name="flightProcess" .../>
 2     ...
 3     <partnerLink name="CounterWS" partnerLinkType="CounterPLT" .../>
 4     ...
 5     <variable name="increaseRequest" messageType="increaseCounterInput"/>
 6     ...
 7    <sequence name="flightSequence">
 8     <receive name="receiveClientRequest" .../>
 9     ...
10     <invoke name="invokeLufthansa" operation="findAFlight" .../>
11     <sequence name="collect billing data">
12       <assign>
13        <copy>
14         <from expression="1"/>
15         <to variable="increaseRequest" part="increaseBy"/>
16        </copy>
17       </assign>
18       <invoke partnerLink="CounterWS" portType="CounterPT"
19             operation="increaseCounter" inputVariable="increaseRequest" .../>
20     </sequence>
21
22     <invoke name="invokeTunisAir" operation="searchFlight" .../>
23      ...
24     <reply name="reply" partnerLink="client" .../>
25   </sequence>
26 </process>
```

Listing 3 Collecting billing data in the flight process

The discussion so far illustrates how the data collection for billing is a crosscutting concern. The BPEL code addressing that concern is scattered across several processes. There is no single module that encapsulates (1) the logic that belongs to the data collection concern and (2) the specification of the interface of the latter with the rest of the travel portal logic. The data collection logic includes the new *sequence* activity, the declaration of partner links, and the variables involved in realizing the collection of data (i.e., what to do for collecting data). The interface of the data collection concern with the rest of the travel portal logic involves the specification of the points in the execution where data collection should be triggered.

Due to the lack of a module that encapsulates the data collection concern, we have to add the same *sequence* activity after each of the *invoke* activities that call Lufthansa's Web Service; we also have to insert the same partner link and variable in each of the affected processes. The code pertaining to the data collection concern is scattered across several processes and tangled with process activities that address other concerns. Consequently, the process definition becomes complex and evolves into a monolithic block of activities. It is not clear which partners, variables, and activities pertain to which concern and where the logic for a given concern is executed.

Measurement of activity execution time and logging: We may also need data collection for management and auditing purposes [79]. An organization that runs Web Service-based workflows is probably interested in measuring the execution time of some process activities. To measure the execution time of a BPEL activity *X*, we set up a special auditing Web Service and invoke an appropriate operation to start a timer before each occurrence of *X*. We also need to invoke another operation to stop the timer after each occurrence of *X*.

In the case of BPEL, logging is more important than in programming languages such as Java because most BPEL debugging tools available to date are still quite primitive, unlike Java development tools. The only possibility is to debug the orchestration engine, which is usually not feasible in practice because it is overly complex and the source code of the engine is not available. Assume that the BPEL processes do not return the expected output to the client in the travel agency scenario. In order to identify the cause of the problem, we want to have a log file that shows, for each process, the input data received from the client (the *receive* activity) as well as the input and output data for the invocations of the partner Web Services. Here too we will need redundant process modifications at various places, to call the logging Web Service and copy the data from the process variables into the input variables of the logging operations.

Auditing concerns such as the measurement of activity execution time and logging are also crosscutting. The scattering and tangling problems in this case are even worse than with the data collection concern, because the calls for execution time measurements and logging are required before and after each monitored activity. This crosscutting issue is illustrated in Figure 2. The vertical bars represent the processes specifying *getFlight*, *showHotels*, and *getTravelPackage*. The horizontal bars represent BPEL code that belongs to crosscutting concerns such as auditing and persistence.

Persistence: BPEL data is transient: when a process instance terminates, all data held in process variables is lost. In a number of scenarios, this is not acceptable (e.g., if the travel agency requires the data of the booked flights and accommodations to be stored in a persistent way). One solution for supporting data persistence in BPEL consists in setting up a Web Service for persistence that provides operations to store the values of variables in a file (see the *File Persistence* Web Service in the top right

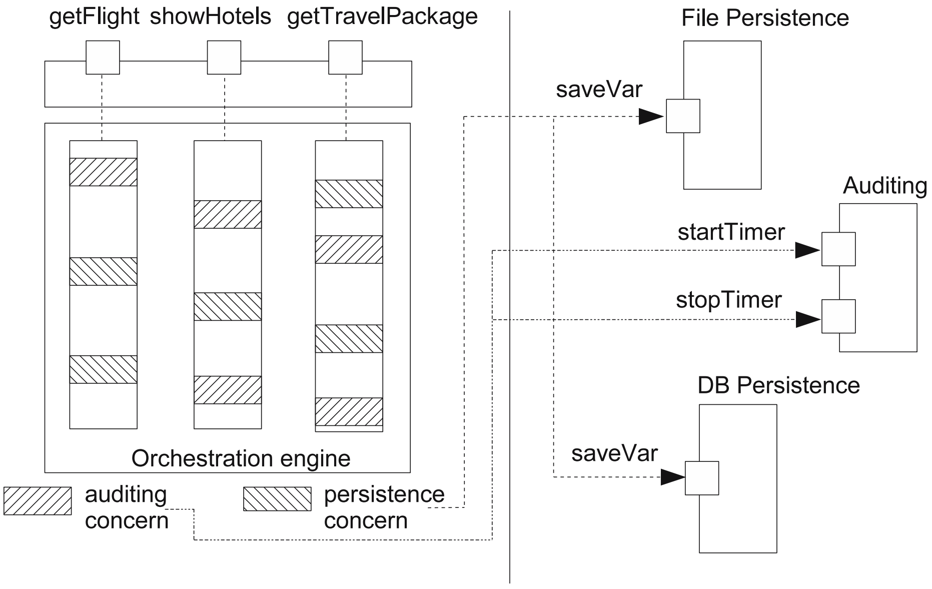

Figure 2 Crosscuts in process-oriented Web Service composition.

corner of Figure 2). The *File Persistence* Web Service is invoked whenever the value of a variable changes. Assume that we want the flight and hotel data to be persistent in the three processes shown in Figure 2. To this end, we have to insert *invoke* activities that call the *File Persistence* Web Service after invoking the flight Web Service and the hotel Web Service, respectively. This change again affects several locations in the three processes.

The code that encodes the decision about where and when to trigger the persistence functionality, establishes a protocol between the BPEL processes for *getFlight*, *showHotels*, *getTravelPackage* and the *File Persistence* Web Service. Hence the corresponding code is not modularized in one place: it crosscuts the modular process-based structure of the composition, as illustrated in Figure 2. Now, assume we replace the *File Persistence* Web Service by another Web Service that implements database persistence. The latter would have a different interface because for database persistence, the operation *saveVar* requires the table and column names as input parameters, whereas for file persistence, it requires the file name. Consequently, to switch from file persistence to database persistence, we have to change the process specifications for *getFlight*, *getTravelPackage*, and *showHotels* and to deal with the problems of crosscutting changes.

Business rules: Business rules [28] are another example of crosscutting concerns, especially those encountered in composite Web Services [14]. In the very competitive business context worldwide, business rules can change frequently [82] due to new partnerships, changing strategies, mergers, etc. Currently, business rules are not well modularized in BPEL process specifications [14]. Thus, when new business rules are defined, they get scattered over several processes. They also get buried in the process code. The problem is that implementing business rules affects, in general, sets of points in the execution that transcend process boundaries.

2.2.2 The need for crosscutting mechanisms

BPEL 1.1 does not support mechanisms for expressing 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 (as represented by the horizontal bars in Figure 2). Given constructs for modularizing crosscutting concerns, process designers would be in a better position to make design decisions as to what to consider as core composition logic and which crosscutting concerns to separate in well-defined modules. A number of criteria could drive these design decisions [52]: the extent of the crosscutting nature of features, the expected requirements for maintainability and change, etc. The separation of a concern makes it possible to modify the code pertaining to it without changing all the composition (*independent extensibility*) and also to reuse that code in other compositions.

The point here is not that the concerns mentioned in the previous section should be separated in any case, but rather that their crosscutting nature requires new modularization mechanisms. Whether, when, and how to use these mechanisms is a matter of design and methodology. But designers should be able to capture some concerns in a modularized way if they wish to do so.

One could argue that concerns such as the measurement of activity execution time or persistence could be addressed in a middleware layer below BPEL. However, if the concern at hand is inherently related to BPEL constructs, we claim that it cannot be supported in underlying middleware. For execution time measurement, if we consider only *invoke* activities, the response time of a Web Service can be monitored at the SOAP engine level. But if we want to know the execution time of a certain part of the process (e.g., for a *scope* or a *flow*), this information is unknown to underlying middleware.

Similar arguments apply to persistence. In fact, implementing persistence in a middleware layer below BPEL means that the incoming and outgoing SOAP messages need to be persistent. However, at the SOAP level, we lose the connection between messages and variables because SOAP does not know about BPEL variables and which BPEL activities may read or write them. As a result, if a BPEL process invokes the same operation of a partner several times and persistence is required only for one invocation, it is not possible to identify the messages that must be persistent.

In the vein of the seminal papers on the end-to-end argument in system design [71] and on open implementations [53], we argue that the decision as to what concerns to capture within middleware, and which ones to capture at the application level, is a design consideration. It makes more sense to capture certain concerns at the application end. For instance, this is why the Enterprise Java Beans component model [27] provides means to express application-level access control within bean implementations, in addition to the security mechanisms provided by any EJB application server.

In conclusion, it is important to provide application designers with means to capture crosscutting concerns at the application level in a well-modularized way. The implementation of middleware can also benefit from such crosscutting modularization mechanisms [29, 30, 69, 87].

2.2.3 The need for runtime changes of the composition

Web Service composition languages such as BPEL inherited a static view of the world from workflow management systems, which did not properly support evolutionary and dynamic changes [37]. However, composite Web Services implement cross-organizational collaborations, a context in which several factors call for composition evolution (e.g., changes in the environment, variations of non-functional properties, and unpredictable events). Some of these changes require dynamic modifications of the composition.

Previous work on *adaptive workflows* [11, 12, 46] focused on providing flexible workflow models that support change. While theoretically applicable to BPEL, the concepts and results of adaptive workflows have not yet found their way into BPEL workflows. When a BPEL process is deployed, the WSDL files of all Web Services participating in the composition must be known; once a process has been deployed, there is no way to adapt it dynamically. The only flexibility provided in BPEL is *dynamic partner binding*, which allows partners to be mapped to specific Web Services at runtime.

To illustrate the issues with process changes in our travel agency scenario, assume that we want to add some business logic to invoke a car rental Web Service in the three BPEL processes. When one client requests a flight, an accommodation, or a travel package, one also gets propositions for car rental. We can also have variations to this adaptation; for instance, an offer is made only to frequent customers, or to those clients who have expressed interest in the offer in their profiles. In this case, the adaptation should be effective only for specific process instances.

The required dynamic change is a crosscutting concern because it arises in many places across different processes. Unlike the examples cited in the previous section, which address *non-functional crosscutting concerns*, the car rental business logic is a *functional crosscutting concern* (i.e., related to the process business logic).

Dynamic changes in the composition are often problematic because BPEL lacks mechanisms to add code for crosscutting functionality to existing processes in a non-obtrusive and modularized manner at runtime. There are other situations where ad-hoc deviation from the originally planned business process is required at runtime [37] (e.g., if users are involved in making decisions or if unpredictable events occur). One could argue that if a runtime process change is required, we just have to stop the running process, modify the composition, and restart. This is currently the only way to implement such changes in BPEL. However, this is not always a viable solution: if we stop a long-running process, we must roll back or compensate all previously performed activities. For example, consider an ordering process that checks whether the requested product is available, performs payment, and finally calls a shipping Web Service. If we stop that process just before calling the shipping Web Service, the resulting state is inconsistent.

In addition to dynamic changes of the composition logic, it may also be necessary to flexibly switch on and off some non-functional concerns such as persistence, auditing, logging, or authentication at runtime.

3 Aspect-oriented web service composition

Aspect-Oriented Programming (AOP) [50] was designed explicitly to address the modularization of crosscutting concerns, which makes it particularly suitable for solving the problems discussed in Section 2. While it has been mostly applied to object-oriented programming to date, it is also applicable to other programming styles [20], 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. In this section, we will first introduce the key concepts of AOP using AspectJ [49], perhaps the most mature AOP language to date. Subsequently, we will present the design and implementation of AO4BPEL, our aspect-oriented extension to BPEL. Finally, we will show how AO4BPEL can be used to address the problems outlined in Section 2.

3.1 Introduction to aspect-oriented programming

Before the advent of AOP, modularity mechanisms supported the hierarchical decomposition of software according to a single criterion, based for instance on the structure of data (*object-based decomposition*) or the functionality to be provided (*functional decomposition*). Crosscutting modularity mechanisms [57] implemented in AOP aim at breaking with the "*tyranny of a single decomposition*" [78] and support a modular implementation of crosscutting concerns.

There are several approaches to aspect-oriented software development. They were analyzed and classified into different categories by Masuhara and Kiczales [57]. In this paper, we refer to what they call the *pointcut-advice model* when we talk about AOP.

AOP introduces a new unit of modularity, called *aspect*, which aims at modularizing crosscutting concerns in complex systems by using *join points*, *pointcuts* and *advices*. Join points are well-defined points in the execution of a program [49]. Which points in the execution of a program are considered as join points is determined by the join point model of an aspect-oriented language. As AspectJ is an aspect-oriented extension of Java, its join point model defines points in the execution of object-oriented programs: method calls, constructor calls, field read/write, etc.

In order to modularize crosscuts, it is necessary to identify related join points. For this purpose, we use a pointcut, i.e. a predicate on attributes of join points. One can select related method execution points based on the type of their parameters or return values, on matching patterns in the names or modifiers, etc. Similar mechanisms are available to select sets of related setter and getter execution points, sets of constructor calls and executions, exception handlers, etc. Current AOP languages come with predefined pointcut constructs (known as *pointcut designators* in AspectJ [49]).

Finally, a common behavioral effect for a set of related join points is specified in an *advice* associated with the pointcut. An advice is a piece of code that is executed whenever a join point in the set identified by a pointcut is reached. It may be executed before, after, or instead of the join point at hand; this corresponds to *before*, *after* and *around* advice types. With an 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.

Advice code also has access to the execution context at join points that trigger its execution. In addition to identifying relevant points in the program execution, a pointcut can also declare what part of the join point context is made available to the advice. For instance, AspectJ supports language-specific constructs for exposing the target of a method call, the arguments passed to the call, etc.

An aspect module consists of several pointcut definitions and advices associated with them. In addition, it can define state properties and methods, which in turn can be used within the advice code. Listing 4 shows a simple logging aspect in AspectJ. This aspect defines a pointcut, *loggableMethods*, which specifies where the logging concern should join the execution of the base functionality. In this case, the interesting join points are the calls (the *call* pointcut designator) to all methods named *bar*, independent of the class they are defined in, their return type (the wildcard "*" is used to abstract over the class/method names and return types), as well as the number and types of the parameters (the symbol ".." serves to abstract over parameters of the call). The object that executes the method call is exposed to the advice using the pointcut designator *this*; it is bound to the object *o*.

```
public aspect Logging
{
  //where ?
  pointcut loggableMethods(Object o): call(* bar(..)) && this(o);

  //when ?
  before(Object o): loggableMethods(o)
  {
  //what ?
  System.out.println("bar called from object " + o.toString());
  }
}
```

Listing 4 A logging aspect in AspectJ

This aspect defines a pointcut, *loggableMethods*, and an associated advice that prints out a logging message before executing any of the join points matched by the pointcut. The advice specifies what behavior to execute, and when to execute it, at the selected join points. It can also use the context of the join point by calling the method *toString* on the target of the current method call.

The *Logging* aspect enables the logging concern to be well modularized in a separate module. If the logging functionality is required in other places, we just have to modify the pointcut definition. In a non-AOP object-oriented solution, it would be necessary to go through all locations where logging is required and modify the classes appropriately. With aspects, conversely, a single piece of code needs to be changed, and the logging functionality can be switched on and off without modifying the rest of the application code.

Integrating aspects into the execution of the base functionality is called *weaving*. The reverse operation is called *unweaving*. In *static AOP* approaches such as AspectJ, weaving happens at compile time or load time. In *dynamic AOP* approaches [8, 10, 67], aspects can be deployed and undeployed at runtime. Thus, dynamic AOP allows aspects to adapt the behavior of applications to changes in the requirements and runtime environment [72].

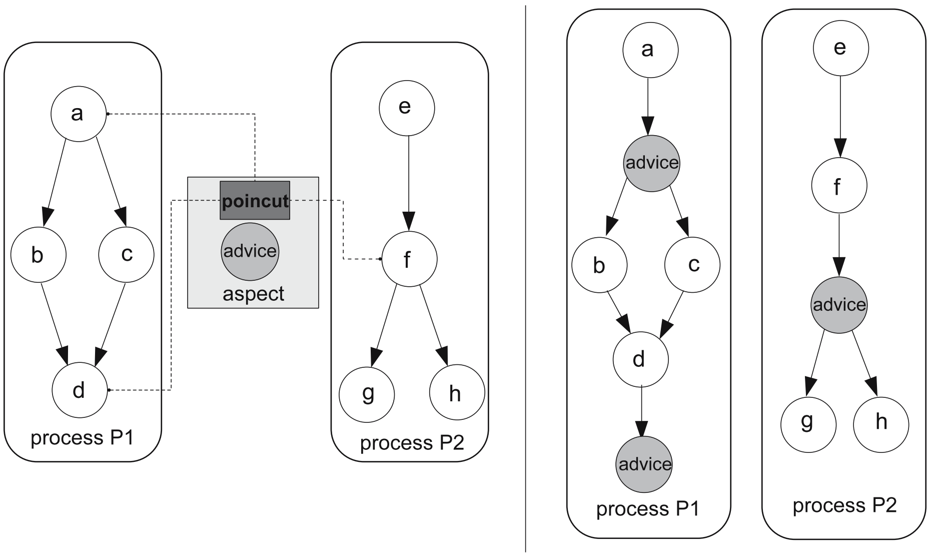

Figure 3 The after advice and workflow graphs.

3.2 Aspectual workflows with AO4BPEL

We will now present the basic concepts of *aspect-oriented workflow languages* and introduce AO4BPEL, which implements these concepts. Next, we will elaborate on the join point model, the pointcut language, and the relationship between advices and BPEL constructs. Then, we will study some examples of AO4BPEL aspects.

3.2.1 Graph formalism

To express aspect-oriented workflows [15], we use a graph formalism. The workflow is represented by a directed acyclic graph where nodes represent activities and edges represent control flow dependencies between activities.[2] A directed edge between two nodes specifies that the activity corresponding to the end node can only start after the activity corresponding to the start node completes. A node activity can be either an atomic node or a sub-graph of nodes. Node nesting is allowed only if it does not lead to cycles in the graph.

For illustration, consider the process *P1* on the left-hand side of Figure 3. This process defines four activities. The activities labeled *b* and *c* can start only when the activity *a* completes. As there is no edge connecting *b* and *c*, these activities can execute in parallel. The activity labeled *d* can only start when activities *b* and *c* complete.

In this graph formalism, join points are graph nodes representing the execution of an activity. Similarly, in AO4BPEL, join points are well-defined points in the execution of process activities; these activities can be primitive or structured. An essential

[2]In this discussion, for the sake of simplification, we do not consider data flow dependencies.

property of the aspect-oriented paradigm is *quantification* [52]. For Web Service composition, quantification means that the pointcut language must support the selection of points across several processes. In the graph formalism, the pointcuts provide means to select a set of nodes in the graph; they are represented as rectangles with outgoing connections to the graph nodes they select. For illustration, consider the pointcut of the aspect in Figure 3, which selects the nodes *a* and *d* in the process graph of *P1* (left-hand side) and the node *f* in the process graph of *P2* (right-hand side).

3.2.2 Join point model and pointcut language

If we map our graph formalism onto AO4BPEL, pointcuts are means for referring to sets of activities that may span several processes. At these join points, crosscutting functionality should be executed. The attributes of a process or an activity can be used as predicates to choose relevant join points. For example, to refer to all invocations of a partner Web Service, we use the attributes *partnerLink* and *portType* of the *invoke* activity.

As BPEL process specifications are XML documents, XPath [19] is a natural choice for the pointcut language. Pointcut languages are about addressing and selecting activity nodes, and XPath is suitable for that. AO4BPEL aspects are also specified in XML. The content of the *pointcut* element is an XPath expression that selects those activities where the execution of additional crosscutting functionality will be integrated. As XPath expressions can span different processes, the pointcut language of AO4BPEL can express crosscutting structure across several processes. The pointcut element has a boolean *contextCollection* attribute that indicates whether the join point context is exposed to the advice or not (see Section 3.2.3). XPath set operators such as the union operator "|" can be used to combine pointcuts that select different types of activity.

AO4BPEL supports not only join points that correspond to the execution of activities, but also *internal join points* [17]. These are points in the interpretation of an activity rather than at the composition specification level. For instance, internal join points are required to get access to the SOAP message that corresponds to a messaging activity. They break down the interpretation of a messaging activity into several points and allow the aspect to express statements such as "*before a SOAP message is sent out in the course of interpreting a messaging activity, get that message and do this and that.*" These join points allow the AO4BPEL programmer to access the SOAP messaging layer. They are especially relevant to middleware concerns, in order to check the compliance of a received message with certain policies of the composite Web Service, or to add some headers to a SOAP message (according to WS-Reliability [61], WS-Security [62], etc.).

For capturing internal join points, AO4BPEL introduces two pointcut designators: *soapmessagein* and *soapmessageout*. These pointcut designators are used with pointcut designators that capture activities. Because the pointcut specifications that result from combining BPEL-level and interpretation-level pointcut designators intercept points in the execution of different layers, we call them *cross-layer pointcuts*.

The *soapmessagein* pointcut designator works in conjunction with *invoke* and *receive* activities. When used in conjunction with *invoke*, it captures the join point where a SOAP message has been received by the engine as a response for an *invoke*.

When used with *receive*, *soapmessagein* captures the join point where a SOAP message matching a *receive* activity arrives at the orchestration engine.

The *soapmessageout* pointcut designator works in conjunction with *invoke* and *reply*. When used with *reply*, it captures the join points where the orchestration engine has generated the SOAP response message and will send this message; in conjunction with *invoke*, it captures the join points where the respective SOAP request message has been generated and is ready to be sent.

The join point model described so far is *activity driven*; this is the model currently used in AO4BPEL. Other join point models are conceivable for workflow languages, depending on the perspective from which the workflow is considered [15]. A *control-flow driven* join point model selects points in the execution based on certain control flow and transition patterns such as conditional routing, concurrency, split, join, etc. A *data-flow driven* join point model focuses on process variables (e.g., initialization or assignment of a new value). A *participant driven* join point model looks at the process definition from the perspective of partner interactions. A combined use of these join point models may increase the expressiveness of the AO4BPEL pointcut language [65]. However, a thorough investigation of the usefulness of such an increased expressiveness in the context of Web Services is outside the scope of this paper.

3.2.3 Advice

In AO4BPEL, an *advice* is a BPEL activity that executes before, after, or instead of a join point. In the case of process-level join points, the advice is logically inserted into the parent activity of the join point at hand. AO4BPEL supports the before, after, and around advice types and several combinations thereof (with *soapmessagein* and *soapmessageout*).

Advice semantics. To understand advice semantics, we can use the graph formalism presented earlier. The effect of the around advice is to replace any join point node captured by the pointcut associated with the advice, by another node that may contain the join point node. The advice of the aspect on the left-hand side of Figure 4 is a graph of nodes. The special node *JP* refers to the join point to replace. The process P2 in the middle shows a high-level view of the effect of deploying the aspect on the graph of P2. The process P2 on the right-hand side of Figure 4 reveals the internals of the advice activity.

The logical effect of a before or an after advice can be explained in the same way as the around advice. For a before or an after advice, we can replace the join point at hand by a *flow* activity that contains both the advice and the join point. For a before advice, a *link* goes from the advice to the join point activity; for an after advice, a *link* goes from the join point activity to the advice. The right-hand side of Figure 3 illustrates the effect of deploying the aspect with an after advice on the graphs of P1 and P2.

A sub-flow node in the graph formalism corresponds to the *flow* activity in BPEL. Hence, for AO4BPEL, the logical effect of the around advice is to replace the join point activity at hand by a *flow* activity that contains the advice and the join point activity. If the join point activity is the source or target of some links, the *flow* becomes the source or target of these links. AO4BPEL introduces a special activity

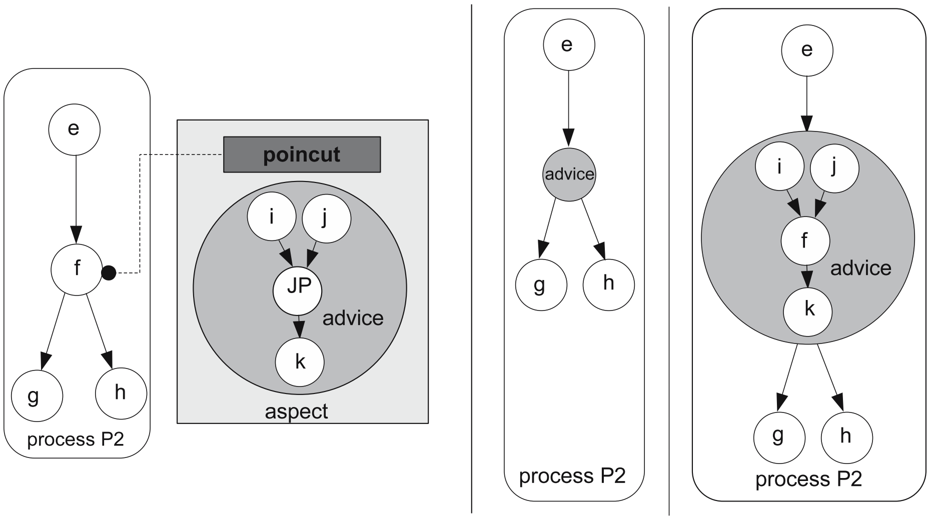

Figure 4 The around advice and workflow graphs.

named *proceed*, which can be used inside the around advice as a placeholder for the join point activity. When the advice activity is interpreted, the *proceed* activity integrates the execution of the join point activity in the middle of the advice activity.

With the around advice, arbitrary *links* can be defined between the advice and the join point according to several workflow patterns [80, 86]. This makes it possible for AO4BPEL to support more advice types beyond those available in AspectJ; for instance, it is possible to implement a *parallel advice* using the around advice, and appropriate links between the advice activity and the join point activity.

In addition to pointcut and advice declarations, an aspect can also define partner links, variables, fault handlers, compensation handlers, event handlers for alarm events, and correlation sets.

Advices and faults. Faults may occur during the execution of the advice activity, which can affect the parent process. However, as the aspect is an add-on to existing processes, it must take care of all the faults that may occur within the advice. For this reason, each advice in AO4BPEL is embedded in a *scope* activity that defines an empty *catchall* fault handler. The purpose of the *scope* and the default *catchall* fault handler is to ensure that faults never propagate from the advice to the process. In fact, variables, fault handlers, compensation handlers, and event handlers of the aspect are implicitly attached to that scope.

If the aspect does not define appropriate fault handlers for the faults that may be thrown during the advice execution, the fault is handled by the *catchall* handler and the process continues executing normally. Moreover, if the aspect defines top-level fault handlers, these handlers are not allowed to throw a fault. This can be checked at aspect deployment time using static analysis.

When a fault occurs inside the advice activity and the default *catchall* fault handler is executed, the outgoing links of the advice are traversed normally because the advice scope handles the fault. As a result, the *scope* ends normally and the values of its outgoing links are evaluated as usual. In particular, links leaving from the enclosing *flow* are also traversed normally.

Advice and compensation. Long-running BPEL processes may not complete in a single atomic transaction. Consequently, if a fault occurs during the execution of such a process, it is necessary to reverse the effects of the activities that were completed so far. In BPEL, the construct for undoing work is the *compensation handler*, which is associated with a scope.

In AO4BPEL, the advice is inserted into the enclosing scope of the current joint point. So, if some child activities of that scope are to be compensated for some reason (e.g., in a fault situation), the advice activity may also need to be compensated. The compensation handler of the advice (if it exists) should be called in the same way as the compensation handlers of the child activities of the advice's parent scope.

This is enforced as follows: when the advice is woven with the process, the weaver gets a reference to the parent scope of the advice and adds the advice's compensation handler (if it exists) to the list of compensation handlers managed by the parent scope. When the advice is unwoven, its compensation handler is removed from that list. In this way, the compensation handler can be called correctly as defined by the BPEL specification.

Advices and correlation sets. Correlation sets are BPEL constructs for routing SOAP messages to the correct process instance by mapping parts of those messages to parts of the process variables.[3] To motivate the relationship between advices and correlation sets, consider an advice that calls a partner Web Service via an *invoke* activity. The *invoke* calls either an aspect-local partner or a partner of the parent process. In both cases, when the partner response message comes in, the engine must somehow identify the corresponding aspect that made the invocation. However, at runtime, an advice does not have an identity of its own because it is triggered during the execution of several instances of the processes selected by the pointcut. How to correlate messages with the advice using the correlation sets defined in the aspect?

We solved this problem as follows. If the aspect defines its own correlation sets, the AO4BPEL implementation adds them to the parent processes, which allows the engine to route messages correctly (i.e., to the process instance with the woven advice). Otherwise, if the advice calls a partner of its parent process, the aspect can use the correlation sets of the parent.

Context collection and reflection. In most cases, the advice activity is not completely decoupled from the process, and the advice may require context information or data from the current join point activity (and in some situations also from its parent process). *Context collection* provides a means to extract and use the join point context inside the advice. However, as a pointcut typically selects different types of activities, more generic constructs are required for passing context. For example, for a data validation pointcut that selects a *receive* activity and an *invoke* activity, the corresponding advice needs to access the input variables of these join point activities in a generic way (i.e., without specifying the name of each variable).

[3]In WSDL, message parts describe the logical units of a message. In BPEL, variables act as containers for messages and thus also have different parts.

AO4BPEL provides some special constructs that can be used by the advice to access the join point context. The variable *ThisJPActivity* is a reflective variable that holds information about the current join point activity such as process name, activity name, activity type, partner link, port type, and operation name. The last three are only set if the join point is a messaging activity (i.e., *invoke*, *reply*, and *receive*).[4] For example, assume that a pointcut of a logging aspect captures all *invoke* activities with a certain port type and operation attributes. The variable *ThisJPActivity* can then be used by the logging advice to log the name of the process from which the invocation has been made. The advice can access the context of the parent process using the construct *ThisProcess(x)* where *x* is the name of a variable, partner link, or correlation set. In order for the advice to use the join point context, the pointcut must expose this context (i.e., the attribute *contextCollection* must be set to true).

When the pointcut selects only messaging activities, two other variables can be used to collect the data context of the join point: *ThisJPInVariable* and *ThisJPOutVariable*. They refer to the variables used by the join point activity. Each of them can be used inside the advice code as a variable name (e.g., with *invoke*). *ThisJPInVariable* is an alias to the input variable of an *invoke* activity, or to the variable of a *receive* activity. *ThisJPOutVariable* is an alias to the output variable of an *invoke* activity, or to the variable of a *reply* activity. The individual parts of those special variables can be accessed by name or more generically by index (e.g., *firstpart*, *lastpart*, *partN* where *N* is the index).

In the context of internal join points, AO4BPEL provides two special variables called *soapmessage* and *newsoapmessage*. The first variable is used to expose the SOAP message at the join points captured by the pointcut designators *soapmessagein* and *soapmessageout*. The SOAP message is typically sent by the advice to a middleware service that processes the message (e.g., by adding some specific headers for security [17]). The variable *newsoapmessage* contains the SOAP message that has just been processed.

Advice restrictions. Aspects are only meant to modify the implementation of a composite Web Service and not its interface. Therefore, we should avoid using the messaging activities *receive*, *pick*, *reply*, and event handlers for message events in the advice. Otherwise, messaging activities and event handlers for message events would require a matching operation in the WSDL file of the composite Web Service. This means that the WSDL interface of the composite Web Service would have to be modified, unless the matching operations were already present. This would require a change in already published interfaces, which in our view must be avoided.

Using the *terminate* activity in the advice leads to the termination of the process instance, which may be dangerous. Nevertheless, there are use cases where the advice must be allowed to send a fault message to the client using the *reply* activity and terminate the process instance. For example, it may be necessary to add support for data validation to the processes presented in the travel agency scenario. If the dates that the client passes to the *receive* activities of those processes are not valid (e.g., the return date is before the departure date), we should not call any partner Web Service. The appropriate behavior is to send a fault message to the client and

[4]The *pick* activity with *onMessage* branch can be used in an equivalent way to *receive*. Therefore it is also a messaging activity.

terminate the process instance. We leave it for implementors to decide whether to allow or disallow the activities *reply* and *terminate*.

Advice deployment. AO4BPEL supports both *process-level* and *instance-level* aspect deployment. With process-level aspect deployment, which is the default case, the advice applies to all instances of a process. With instance-level aspect deployment, only some process instances are affected by the advice. This feature is especially useful when aspect activation depends on runtime data such as the value of an activity variable. For example, in a stock quote scenario, we activate a pricing aspect for each process instance according to the pricing strategy selected by the customer; depending on the customer identifier, a pay-per-request aspect or a flat-payment aspect may be activated. Correlation sets may be used to identify process instances for instance-level aspect deployment.

3.3 Examples

In this section, we will revisit examples from Section 2 to show how AO4BPEL aspects capture crosscutting concerns. Due to space limitations, only some of these concerns are presented here.

The *Counting* aspect in Listing 5 collects the necessary billing data. It counts how many times the operation *findAFlight* of Lufthansa's Web Service is invoked from inside any process. This aspect declares a partner Web Service, *CounterWS*, which supports the operation *increaseCounter* for incrementing a counter. This operation takes an integer parameter as input, to be added to the total number of invocations. The pointcut *Lufthansa Invocations* (lines 10–12) captures all invocations of the operation *findAFlight* from any process deployed in the orchestration engine. The after advice (lines 13–25) associated with that pointcut increases the total number of invocations by one.

```
 1 <aspect name="Counting">
 2  <partnerLinks>
 3    <partnerLink name="CounterWS" partnerLinkType="CounterPLT"
 4                   myRole="caller" partnerRole="counter"/>
 5  </partnerLinks>
 6  <variables>
 7    <variable name="increaseRequest" messageType="increaseCounterInput"/>
 8  </variables>
 9  <pointcutandadvice>
10   <pointcut name="Lufthansa Invocations" contextCollection="true">
11     //process//invoke[@portType="LufthansaPT" and @operation="findAFlight"]
12   </pointcut>
13   <advice type="after">
14    <sequence>
15     <assign>
16      <copy>
17       <from expression="1"/>
18       <to variable="increaseRequest" part="increaseBy"/>
19      </copy>
20     </assign>
21     <invoke partnerLink="CounterWS" portType="CounterPT"
22           operation="increaseCounter"
23               inputVariable="increaseRequest" outputVariable="increaseResponse"/>
24    </sequence>
25   </advice>
26  </pointcutandadvice>
27 </aspect>
```

Listing 5 The Counting aspect

Listing 6 shows a monitoring aspect for measuring the execution time of certain process activities. The pointcut (lines 13–15) of this aspect matches the *invoke* activities that call the operations *findARoom* and *findAFlight*. This aspect uses an around advice, which calls the operations *startTimer* (line 24) and *stopTimer* (line 29) of an auditing Web Service respectively before and after executing the join point activity. The advice integrates the execution of the join point activity by using the special activity *proceed* (line 27). The operations of the auditing Web Service take an input parameter that identifies the monitored activity. This parameter is set to the name of the current join point activity, which is taken from the reflective variable *ThisJPActivity* (line 20). While this example illustrates the measurement of the execution time of *invoke* activities, the pointcut can easily be extended to monitor the execution time of structured activities.

These two aspects show how crosscutting concerns can be separated from the core composition logic. The collection of billing data/monitoring of execution time functionality is not intertwined with the process specification any more. The logic for collecting data is now explicit, and so is the decision as to where and when to collect which data during the execution of the business processes. In this way, we could define aspects with different implementations of a certain concern and weave the appropriate ones according to the context. Also, the concerns that are modularized in the form of aspects can be plugged in and out as needed.

```
 1  <aspect name="PerformanceMonitor">
 2   <partnerLinks>
 3     <partnerLink name="AuditingWS" partnerLinkType="AuditingPLT"
 4               myRole="caller" partnerRole="measurer"/>
 5   </partnerLinks>
 6   <variables>
 7     <variable name="startTimerRequest" messageType="startTimerInput"/>
 8     <variable name="startTimerResponse" messageType="startTimeOutput"/>
 9     <variable name="stopTimerRequest" messageType="stopTimerInput"/>
10     <variable name="stopTimerResponse" messageType="stopTimeOutput"/>
11   </variables>
12   <pointcutandadvice>
13    <pointcut name="monitored activities" contextCollection="true">
14       //invoke[@operation="findAFlight"] | //invoke[@operation="findARoom"]
15    </pointcut>
16    <advice type="around">
17     <sequence>
18       <assign>
19        <copy>
20              <from variable="ThisJPActivity" part="name"/>
21              <to variable="startTimerRequest" part="activityName"/>
22            </copy>
23           </assign>
24      <invoke partnerLink="AuditingWS" portType="AuditingPT"
25            operation="startTimer" inputVariable="startTimerRequest"
26            outputVariable="startTimerResponse"/>
27      <proceed/>
28      <assign>...</assign>
29      <invoke partnerLink="AuditingWS" portType="AuditingPT"
30            operation="stopTimer" inputVariable="stopTimerRequest"
31            outputVariable="stopTimerResponse"/>
32     </sequence>
33    </advice>
34   </pointcutandadvice>
35  </aspect>
```

Listing 6 The aspect for monitoring execution time

To highlight the advantages of AO4BPEL over other solutions, recall the processes of *getTravelPackage* and *getFlight* shown in Listings 2 and 3. In order to keep track of the invocations of Lufthansa's Web Service without AOP, we had to insert the same activity (i.e., the *sequence* containing the *assign* and *invoke* activities), as well as the corresponding variable and partner link, into different processes; this change could only be done statically, and the functionality of a specific crosscutting concern could not be switched on and off at runtime.

By injecting non-functional concerns (e.g., data collection for billing or execution time monitoring) into the process, we already support a simple way of changing the composition. AO4BPEL also allows the composition logic to be changed at runtime. In Section 2, a requirement was to add car rental business logic to the processes specifying *getTravelPackage* and *getFlight*. For achieving this goal in AO4BPEL, we define the aspect *AddCarRental* (see Listing 7).

The *AddCarRental* aspect declares a composite pointcut (lines 11–15) that captures the *reply* activities of the processes of *getTravelPackage*, *getFlight*, and *showHotels*. Assume that these operations return a string result with the vacation package information, the flight information, or accommodation information. The advice (lines 16–35) is a *sequence* activity that executes before the join point activity. At the beginning of the advice, the input parameters of the car rental Web Service are taken from the parent process using the context collection variable *ThisProcess(clientrequest)* (line 20). Next, the operation *getCar* is invoked (line 25). After that, the *assign* activity (lines 27–33) is used to modify the data returned by the operations *getTravelPackage*, *getFlight*, and *showHotels*. The output variable of these operations is accessed generically using the context collection variable

```
 1  <aspect name="AddCarRental">
 2   <partnerLinks>
 3     <partnerLink name="CarPortal" partnerLinkType="CarPLT"
 4               myRole="caller" partnerRole="carWS"/>
 5   </partnerLinks>
 6   <variables>
 7     <variable name="getCarRequest" messageType="getCarInput"/>
 8     <variable name="getCarResponse" messageType="getCarOutput"/>
 9   </variables>
10   <pointcutandadvice>
11      <pointcut name="about to reply" contextCollection="true">
12         //process[@name="travelProcess"]//reply[@operation ="getTravelPackage"] |
13         //process[@name="flightProcess"]//reply[@operation="getFlight"] |
14         //process[@name="hotelProcess"]//reply[@operation="showHotels"]
15      </pointcut>
16      <advice type= "before">
17        <sequence>
18         <assign>
19                <copy>
20                   <from variable="ThisProcess(clientrequest)" part="deptDate"/>
21           <to variable="getCarRequest" part="startDate"/>
22          </copy>
23            ...
24         </assign>
25         <invoke partnerLink="CarPortal" portType="CarPT" operation="getCar"
26                      inputVariable="getCarRequest" ouputVariable="getCarResponse"/>
27         <assign>
28          <copy>
29           <from expression="concat(bpws:getVariableData(ThisJPOutVariable),
30                             bpws:getVariableData(getCarResponse))"/>
31           <to variable="ThisJPOutVariable"/>
32          </copy>
33         </assign>
34        </sequence>
35      </advice>
36   </pointcutandadvice>
37  </aspect>
```

Listing 7 The car rental aspect

ThisJPOutVariable (line 29). The *from* element of the *assign* activity uses the XPath string function *concat* to concatenate the string content of the reply's variable with the string content of *getCarResponse* (line 29).

The *AddCarRental* aspect solves the problem of dynamic change outlined in Section 2. This change specifies a business rule that has been encapsulated in a modular and separate manner as an aspect. As a result, if the business rule changes, we only have to activate or deactivate the appropriate aspect at runtime.

This aspect can be activated via the *aspect deployment tool* of our implementation prototype. This tool asks the programmer to specify the XML file with the aspect definition and the WSDL file of the car rental Web Service. The aspect can be activated dynamically while the respective processes are running, thereby applying the adaptation behavior at runtime.

3.4 AO4BPEL, middleware concerns and the Web Service stack

The BPEL specification does not specify how middleware concerns such as security, persistence, and reliable messaging should be addressed; each implementation can approach these concerns in its own way. In other words, each BPEL engine must be modified on a case-by-case basis. For instance, we first modify the engine to support security, then we modify it to support persistence, etc. With AO4BPEL, we have a more generic solution to the problem of adding support for middleware concerns to the BPEL engine. In fact, if the engine supports aspects, we can add support for persistence, security, auditing, etc. by writing aspects, without modifying the engine itself. By enabling the engine to support aspects (which requires only one change), we provide extensibility points that can be used to extend the engine without modifying it.

When talking about middleware concerns, it is necessary to discuss the relationship between AO4BPEL and the Web Service protocol stack. Several specifications in this stack address issues such as security [62], reliable messaging [61], etc. These specifications are layered on top of SOAP and define new messages and message headers for each concern. There are also specifications that address quality of service and management issues in Web Services [2, 47].

Web Services can publish their requirements, capabilities, and preferences to interested parties using WS-Policy [73], a generic framework for specifying Web Service policies in XML. Policies are attached to the WSDL file of a Web Service [75]. Several domain-specific policy languages leverage WS-Policy (e.g., WS-SecurityPolicy [45]).

One may argue that given the WS-* specifications, there is no need for addressing concerns such as persistence, security or auditing in AO4BPEL aspects. However, Web Service specifications such as WS-Security and WS-Reliability are based upon SOAP and WSDL, and thus do not cover the security and reliable messaging requirements of BPEL processes. In fact, some middleware concerns are inherently related to BPEL constructs and must therefore be dealt with at the BPEL level [16–18]. We have already explained in Section 2 why persistence and auditing in BPEL processes have to be addressed at the BPEL level rather than in underlying middleware. We will now elaborate on reliable messaging in the context of composite Web Services.

Some of the reliable messaging requirements in BPEL need to be addressed at the process level, e.g. the ordered delivery of messages that belong to messaging activities and go to different partners in a sequence [18]. To illustrate this requirement, consider a *sequence* activity that contains two one-way (or asynchronous) *invoke* activities, which call two different partners. As the two *invoke* activities are in a sequence, the programmer can assume that the SOAP message of the first *invoke* arrives at the first partner service before the SOAP message of the second *invoke* arrives at the second partner service. We call a sequence that fulfills this requirement a *reliable sequence*.

WS-Reliability [61] and WS-ReliableMessaging [32] cannot cope with this requirement for three reasons. First, these specifications support reliable messaging only between two parties, and we have three in this scenario (the composition and the two partners). Second, as WSDL is stateless, there is no way to specify that some shared context for reliable messaging is required for the two partner invocations. Third, we cannot just attach reliable messaging policies to the WSDL files of the two partners, because at the WSDL level we lose the information about the sequential execution of the partner invocations.

The point here is that reliable messaging for BPEL is not the same as reliable SOAP messaging, BPEL security is different from SOAP message security [17], and transactions in the context of BPEL [77] are different from Web Service transactions [66] in general. These differences are important because from the BPEL perspective, we see not only WSDL interfaces and SOAP messages, but also the internal implementation of the composition.

Note also that aspects are not meant to replace policies. The Web Services that are involved in a Web Service composition and the composite Web Service itself should use WS-Policy to describe their requirements and capabilities. Aspects come into play at a different level: the implementation level of the composition. Unlike policies, they are not visible from outside. AO4BPEL aspects can help enforce some policies, but do not replace them.

3.5 Aspects and BPEL extensibility

BPEL supports extensibility by allowing namespace-qualified attributes to appear in any BPEL element, and by allowing elements from other namespaces to appear within BPEL-defined elements. Extensibility can be used to introduce new elements and attributes in BPEL [34] or to attach policies [75].

This extensibility mechanism cannot be used to solve the problems of modularity and flexibility mentioned in Section 2, for several reasons. First, language extensions (e.g., for supporting persistence or security) require many changes: at each BPEL activity where a certain concern is required, we have to put extensions or attach policies. In AO4BPEL, this problem is solved by the pointcut concept, which allows the selection of sets of activities at once. Second, approaches that are based on BPEL extensibility [75] embed extensions directly in the BPEL code. As a result, one cannot switch on and off a certain concern in a flexible manner because the extensibility elements or attributes are tightly coupled with the process specification. Third, with AOP, we provide a more generic way to integrate new concerns in the composition because solutions that are based on BPEL extensibility would introduce new extensions to BPEL, or to the policy assertion language, for each BPEL-specific concern (e.g., persistence or security).

However, we emphasize again that aspects and policies can co-exist. In fact, the current version of AspectJ supports Java annotations, which can be seen, to a certain extent, as equivalent to attaching policies to BPEL activities. As discussed in [52], annotations can be used in pointcuts as a means to select points in the program where advices have to be executed. In [52], these two mechanisms are investigated for separating design decisions and absorbing change. Some guidelines are given as to how and when to use these mechanisms in isolation or combination [52].

There was a discussion in the OASIS WS-BPEL Technical Committee about introducing a *call* activity to invoke a sub-flow [63]. But even with sub-flow constructs, the problems of code scattering and tangling still exist. In fact, the sub-flow constructs correspond roughly to the procedure abstraction often found in conventional programming languages. These constructs aim at improving the hierarchical structure of a process specification, but do not address crosscutting issues. A *call* activity could be used, for instance, to capture the logic needed for a crosscutting concern (including partner links and variables). However, it would not capture the decision as to when and where that concern joins the process in a localized and explicit way: a call to the sub-flow implementing some given crosscutting functionality is required in several places, so it is still scattered. AOP enables a more flexible control over when to call the sub-flow (i.e., the advice) and whether the call must be switched on and off at runtime. The relationship between procedure calls and pointcut advice, viewed as two kinds of abstraction mechanisms, is studied throughly in [52].

3.6 Complexity

One could argue that with AOP, the system becomes difficult to understand and debug. In fact, this is the case with all modularization mechanisms: when the system is decomposed according to several dimensions, functionality is spread over several modules. However, in a non-AOP monolithic design, the situation is not really better because the code for each concern is intertwined with the business logic of the process. Aspects are not responsible for adding complexity to the composition: the complexity is already there. It can be dealt with in three ways.

First, the crosscutting concerns can be manually integrated in the process definition, with clearly negative consequences on modularity (see Section 2). In such an approach, understanding and debugging the code of crosscutting concerns is even harder than with aspect-oriented techniques [51].

Second, we can rely on middleware to deal with crosscutting concerns. For instance, the separation of concerns was one of the main motivations for adopting component technologies such as Enterprise Java Beans [27]. This approach also poses problems because the beans cannot be understood, tested, and debugged outside the container, and testing or debugging them within the container is very complex [44].

Third, we can use aspect-oriented techniques. People from the enterprise application development community [10, 43] and others [69] have identified aspects as a means to avoid problems with the container-based modeling of crosscutting concerns.

Complexity can also be reduced by making AOP systems more understandable and predictable, e.g. using aspect visualization tools [21]. To this end, our prototype implementation of AO4BPEL provides aspect visualization support, so that the composer can see which aspects affect which processes.

3.7 Static analysis

Is static analysis affected by the use of aspects? The latest draft of the WS-BPEL specification [5] has a section on all mandatory checks that must be performed by compliant implementations [63]. BPEL implementations perform static analysis to prevent errors. As mentioned in Section 3, we rely on static analysis to check that AO4BPEL restrictions are fulfilled, for instance to verify that certain activities are not used by the advice or that a top-level fault handler of the advice does not throw a fault.

The effect of aspects on static analysis is more subtle, particularly in the presence of dynamic weaving. The problem is that dynamically woven aspects can break invariants that have been checked during static analysis, before the processes affected by the aspect start running. This is an issue that we plan to address in depth in future work. A thread of research that we intend to follow is the work by Krishnamurthi et al. [54], which addresses modular verification of advice code. Adapting this work to process analysis in the presence of aspects means that the BPEL engine would maintain a graph-based representation of the process and would store assertions that must hold at the nodes of the activity graph. When aspects are deployed, the engine makes sure that none of the assertions exposed is violated by triggering a deployment time analysis of the advice code.

4 Implementation

AO4BPEL can be implemented as an extension of any BPEL engine. As a proof-of-concept, we have developed a prototype implementation of AO4BPEL on top of IBM's orchestration engine BPWS4J [41], the only engine that was available at the time we started this work. The current implementation supports three process-level join points (*invoke*, *reply*, and *receive*) in addition to the internal join points (*soapmessagein* and *soapmessageout*). We chose messaging activities as join points because they represent the interaction points of the composition with its partners. These are the points where it is especially interesting to add crosscutting functionality in order, for instance, to support non-functional properties of the interactions. The current implementation supports the advice types mentioned in Section 3.2.3.

4.1 Architecture of the AO4BPEL engine

The architecture of our prototype implementation is shown in Figure 5. This generic architecture can be reused for other BPEL engines. The AO4BPEL implementation extends a BPEL engine with an *aspect runtime* component that builds a wrapper around the BPEL interpreter. The AO4BPEL engine calls the aspect runtime to check if there is an advice before and after the interpretation of each activity.[5] To this end, the BPEL interpreter passes metadata about the current activity (process name, activity name, parent activity, activity type, activity attributes, variables, etc.) to the aspect runtime. The latter holds a list of all currently deployed aspects and

[5]The performance overhead caused by these checks is discussed in Section 4.4.

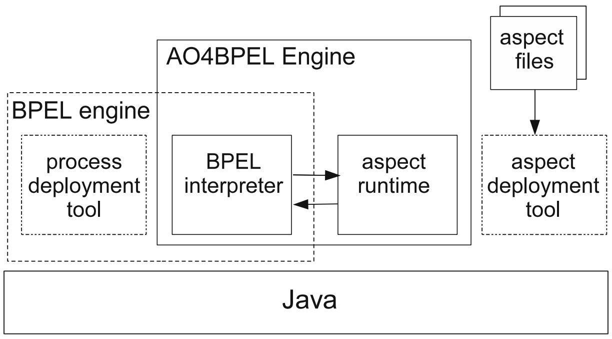

Figure 5 Architecture of our aspect-enabled Web Service composition platform.

checks if any pointcut declaration matches the activity that is currently executed by the BPEL interpreter. If it is the case, the aspect runtime executes the advice. Some metadata exposes the join point context to the advice.

We implemented the aspect deployment tool (see Figure 5) as a web application based on Java Server Pages [26], as the process management tool of BPWS4J. This tool allows the deployment, undeployment, and listing of all currently deployed aspects. The process-list view of the BPWS4J deployment tool has been extended to show the aspects that match each deployed process. This enables the composer to better understand and predict the process behavior.

4.2 Dynamic weaving

The aspect runtime is the dynamic weaver of AO4BPEL. Two strategies can be used to implement dynamic weaving. First, one can modify the in-memory representation of the process inside the engine [23] (e.g., to insert the advice activities). However, such an implementation requires the process execution to be suspended at several places before the process representation is modified. Moreover, it is not possible to apply an advice to only certain process instances, and unweaving aspects at runtime is technically very difficult. The second weaving strategy modifies the interpretation flow by checking if any aspect matches the current join point activity. If there is a match, the respective advice activity is executed.

For a better understanding of the second strategy, it is important to consider the activity lifecycle implemented by the BPEL engine at hand. For example, BPWS4J follows an *activate-enable-run-complete* lifecycle [25], which can be terminated when different faults occur. Further to a fault, the activity enters the *disabled* state. In such a lifecycle, the checks for aspects that match the current join point activity are performed before the *run* phase and after the *complete* phase in the activity lifecycle. In our prototype, we adopted the second strategy of dynamic weaving because it offers native support for AOP concepts (aspects, pointcuts and advices) as first-class entities in the orchestration engine, on a par with processes and activities.

The resulting activity lifecycle model is depicted in Figure 6, which was taken from [25] and slightly modified to illustrate advice weaving. The figure shows that the around advice is implemented in a similar way to the before advice (i.e., both are woven in the transition from *enabled* to *running*). Unlike the before advice, the join point activity is disabled and may be revived later if the around advice uses the *proceed* activity.

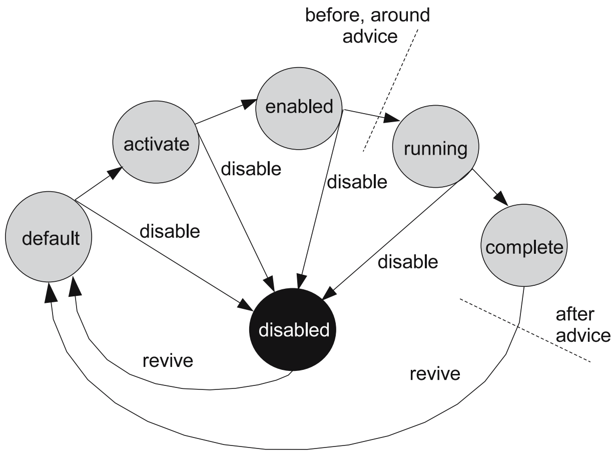

Figure 6 Advice weaving and the activity lifecycle.

From the viewpoint of a dynamic weaver, what happens before the activity is activated or after it completes at the process level is irrelevant. The AO4BPEL weaver is not concerned with constructs of the process metamodel (join conditions, transition conditions, links, etc.) because AO4BPEL has a dynamic weaver embedded in the engine itself, as opposed to a source-level weaver. From the weaver's perspective, what matters is that an activity matched by a pointcut is being interpreted (i.e., a join point activity is about to enter the *running* state or exit the *completed* state). This also means that the co-existence of the graph and algebraic styles in BPEL is not an issue for the weaver.

4.3 Other tasks of the aspect runtime

The aspect runtime can be implemented so that it executes the advice by itself. However, it is better to delegate advice execution to the BPEL interpreter: this approach favors reuse and leads to a simple aspect runtime that can be implemented easily. Nevertheless, several steps are required before the BPEL interpreter can execute the advice.

First, context collection and reflection constructs need to be resolved. If the advice uses say *ThisJPInVariable*, this variable name should be replaced by the name of the input variable of the join point activity. This is implemented as follows. Whenever an activity is matched by a pointcut whose *contextCollection* attribute is set to true, the engine collects the context data of the current join point and then resolves the special constructs based on the collected data. Second, the partners, variables, compensation handlers, and correlation sets of the aspect are attached to the parent process of the current join point activity. Finally, the advice activity can be executed by the BPEL interpreter as if it were an activity of the parent process.

For better performance, the aspect runtime does not evaluate XPath pointcut expressions for each BPEL activity executed by the engine. Instead, it evaluates pointcut expressions only when a new process or aspect is deployed or undeployed. This evaluation operates on the XML process documents and returns a set of matching activities for each pointcut expression. Then, the metadata of the returned activities is stored in an internal data structure of the aspect runtime. As a result, when a process activity is executed, the aspect lookup can be performed faster without costly XPath evaluations. Our prototype uses Xalan-J [3] to evaluate pointcut expressions. Xalan-J provides the class *org.apache.xpath.XPathAPI*, which is very simple to use and does not require much preliminary configuration.

To support instance-based aspect deployment, the aspect runtime manages mappings between process instances and aspects. This allows the aspect runtime to decide whether a given process instance is affected by the aspect.

Another task of the aspect runtime is to handle aspect ordering and aspect interaction problems. When several advices match the same activity, the aspect runtime executes them in the following order: before, around, before soapmessageout, around soapmessageout, around soapmessagein, after soapmessagein, after. However, if two advices with the same type share a join point, they are supposed to be independent and execute concurrently. In many cases, there are dependencies between the aspects that must be taken into account. Therefore, we intend to extend AO4BPEL with constructs to declare aspect precedence and aspect dependencies.

4.4 Performance overhead for aspect checks

Additional checks are needed during the process interpretation to test whether a given process execution point is affected by some aspect. Clearly, this induces some performance overhead. However, this overhead is negligible when compared with the execution time of a messaging activity such as *invoke*, which carries out a costly interaction via the Internet. To check this, we ran the travel process 10 times respectively on the BPWS4J engine (without aspect support) and on our prototype (with additional checks for aspects), without any aspects deployed. The average process execution time[6] was 313 ms in the first case, 316 ms in the second case. This means that the overhead caused by additional checks for aspects is about 1% of the process execution time.

We also compared the performance of two versions of the travel process. In the first version, the counting functionality is directly integrated into the process. We added an *invoke* activity that calls the counting Web Service, and we deployed the resulting process on BPWS4J. In the second version, the counting functionality was defined in a separate counting aspect that was deployed on the AO4BPEL prototype. The average process execution time was 367 ms for the first version of the process, 385 ms for the aspect-oriented version. This means that the execution time for the additional *invoke* activity on BPWS4J is 54 ms, whereas the execution time of the counting advice on our prototype is 69 ms. The difference (15 ms) is due to the aspect machinery.

These preliminary performance measurements seem to confirm that the overhead for supporting aspects is negligible compared with the cost of interactions with partners. The travel process that we used contains only two *invoke* activities. For more complex processes, the relative performance hit (i.e., the performance overhead caused by the use of aspects divided by the overall process execution time) will decrease.

[6]This is the time elapsed between the creation of the process instance (as a result of a request message matching the receive activity) and the termination of that process instance.

5 Related work

Most aspect-oriented languages available to date are extensions of object-oriented or imperative programming languages. In this paper, we use AOP for defining workflow processes, which is completely different. To the best of our knowledge, this is the first proposal to use AOP at the level of the workflow process definition language.

Several crosscutting concerns have been "aspectized" using AspectJ [21, 49, 55]. Rashid and Chitchyan [70] show how data persistence is implemented using AOP in a highly reusable manner, but they observe that application developers cannot be fully oblivious of the persistent nature of data. Bodkin [9] presents experiments on implementing authorization, authentication, auditing, and filtering in web applications using aspects. The use of aspect-oriented software design for application-level security has also been addressed by Win [85]. Fabry and Cleenewerck [31] present aspect-oriented domain-specific languages for advanced transaction management. All of these approaches use AOP to modularize crosscutting concerns in object-oriented Java applications. Similarly, AO4BPEL could be used to modularize these concerns in process-oriented BPEL workflows. However, these approaches need to be adapted to the specific context of Web Services. Unlike Java applications, middleware concerns such as security, persistence, and transactions in BPEL span two layers: the messaging layer (SOAP) and the process layer (BPEL). Moreover, middleware-related Web Service specifications such as WS-Security [62] and WS-AtomicTransaction [56] should be taken into consideration.

There is also ongoing research on applications of AO4BPEL aspects in the context of Web Service composition. A first application is an aspect-based lightweight process container for the integration of middleware services in BPEL Processes [16] (similar to Enterprise Java Beans). We implemented this container using aspects that call dedicated middleware services at certain points in the process execution. These middleware services provide support for non-functional properties of the composition such as security, persistence, and reliable messaging. In two other papers [17, 18], we describe a security service and a reliable messaging service that illustrate how aspects can be used to secure and make reliable the interactions between partners in a Web Service composition. A second application is presented in [14], where AO4BPEL aspects are used to implement business rules [82] in the context of Web Service composition.

Adaptive workflows [11, 12, 37, 86] provide flexible workflow models that support change. Similarly, AO4BPEL aims at making process-based composition more open for change. However, our major focus is on crosscutting dynamic changes, i.e. changes that affect several processes and several activities within the same process. Many concepts from adaptive workflows remain valid for our work, such as the verification and correctness of workflow modifications.

Several authors have already confirmed the importance of flexibility and adaptability for service composition. Benatallah et al. [7] proposed Self-Serv for dynamic and peer-to-peer provisioning of Web Services. In this framework, Web Service composition is specified declaratively using state charts. Self-Serv adopts a peer-to-peer orchestration model, whereby the responsibility for coordinating the execution of a composite service is distributed across several coordinators. This orchestration model provides greater scalability than a centralized model with respect to performance. Self-Serv introduces the concept of service communities, which are containers for alternative services. Separating the service description from the actual service provider increases flexibility. However, unlike our approach, Self-Serv is not compatible with BPEL; it is rather an alternative. In addition, Self-Serv does not support dynamic process changes such as adding new Web Service types to the composition.

Casati et al. proposed eFlow [12], a platform for specifying, enacting, and monitoring composite e-services (i.e., electronic services for e-business). This platform models composite e-services using graphs. It supports dynamic process modifications and distinguishes between *ad-hoc changes* (which apply to a single process instance) and *bulk changes* (which apply to many or all process instances). The main motivation for eFlow is supporting dynamic changes, whereas the main motivation for AO4BPEL is modularizing crosscutting concerns and crosscutting changes in workflow process specifications. The eFlow platform supports changes by migrating the process instances from a source schema to a destination schema. In AO4BPEL, we do not migrate the entire process instance from a source schema to a destination schema: we just weave a sub-process (i.e., the advice) at certain join points.

Dynamic AOP [8, 10, 67, 72] is widely recognized as a powerful technique for dynamic program adaptation. Hirschfeld and Kawamura [39] use dynamic aspects for integrating third-party services and debugging live mobile-communication systems. Schmidt and Assmann [74] use AOP for increasing the flexibility of workflows and model these workflows according to different perspectives (data flow, control flow, and resources). Bachmendo and Unland [6] leverage dynamic aspects for workflow evolution within an object-oriented workflow management system. Unlike ours, the approaches presented in [39], [74], and [6] do not use AOP at the process definition level and do not target the domain of Web Service composition. In addition, these approaches leverage dynamic AOP merely as an adaptation technique and do not use it as a modularization technique for crosscutting concerns, which it actually is.

This argument also applies to Courbis and Finkelstein [23], the only work we know that uses dynamic AOP to adapt Web Service composition. They focus on the dynamic adaptation and debugging of BPEL processes, but do not address the issues of crosscutting concerns in Web Service composition and how to modularize them using aspects. In our view, their approach is more *interceptor-based* than aspect-oriented because the pointcut expressions of their aspects cannot span several processes: if a given change is required in three processes, three aspects are needed. The implementation presented in [23] uses the Visitor design pattern [35]: the BPEL engine is implemented as a visitor of an abstract syntax tree representing a BPEL process. In order to weave aspects, their engine is suspended at some points and the process tree is transformed, which has serious implications on performance. Moreover, the implementation does not support instance-based aspect deployment and dynamic aspect deactivation, because it modifies the internal representation of the process. Unlike our proposal, [23] does not address the relationship between BPEL aspects and BPEL concepts such as fault handling, correlation, and compensation. It is also unclear how a visitor-based engine can deal with links and flows, parallel execution, asynchronous invocations and events, compensation, etc.

One general advantage of our approach over those presented in [6, 23, 74] is the support for *quantification* [33]. The pointcut language of AO4BPEL allows designers to capture join points that span several processes in a modular and abstract way. The other approaches cannot do that.

WSML [81] is a client-side Web Service management layer that realizes dynamic selection and integration of Web Services. It is implemented using the dynamic AOP language JAsCo [76] and Java. WSML offers a rich set of management functionalities, but unlike AO4BPEL, it does not support process-oriented Web Service composition.

Few authors use AOP in the context of Web Services. Ortiz et al. [64] use AspectJ to modularize and add non-functional properties to Java Web Services. This approach works only for Java, which seriously limits Web Service composition. In addition, it requires full control over the Web Service implementation. The Contextual Aspect-Sensitive Services platform (CASS) [22] is a distributed platform that targets the encapsulation of specific crosscutting concerns (coordination, activity lifecycle, and context propagation) in service-oriented environments. CASS introduces pointcut and advice languages geared toward low-level message processing such as message interception, message adaptation, and message forwarding. AO4BPEL provides a more powerful pointcut language that captures pointcuts at the message-level, the process-level, and across the two layers. This allows AO4BPEL to capture crosscutting concerns in general and not only messaging-level concerns.

6 Conclusion

In this paper, we have discussed limitations of Web Service composition languages, especially BPEL, with respect to modularity and adaptability. To address these deficiencies, we have introduced the idea of aspect-oriented workflow languages and presented the design and implementation of AO4BPEL, an aspect-oriented extension of BPEL that supports dynamic weaving. We have illustrated through examples how AO4BPEL improves the modularity and increases the flexibility of Web Service composition. Then, we have presented our prototype implementation of AO4BPEL on top of BPWS4J. The focus of this paper was on motivating the need for mechanisms for crosscutting modularity and introducing aspect-oriented concepts to BPEL. Design considerations as to which crosscutting concerns should be best captured as aspects in a particular situation are outside the scope of this paper.

This proposal could be improved in a number of ways. First, it would be useful to support several pointcut and advice declarations in a single aspect. Second, more research is needed in the context of instance-based aspect deployment. For instance, one could investigate whether correlation sets could help identify the process instances for which the aspect will be applied. Third, the pointcut language could be extended with pointcut designators that support other workflow perspectives beyond the activity perspective (e.g., the data flow and the participant perspectives). This would probably make the expression of certain concerns easier. The pointcut language could also allow temporal relationships to be expressed between activities; this would enable programmers to specify that an advice should execute when activity *a* executes and activity *b* has just completed its execution. Fourth, the effects of aspects and dynamic weaving on static analysis is a topic that requires further research. Fifth, more work is needed to address the problems of aspect composition and aspect interactions. How should the system behave when advices of the same type apply to a join point? In which order should the aspects execute when there are dependencies between them? AO4BPEL could be extended with constructs to express aspect ordering and aspect dependencies.

Acknowledgements We thank the anonymous reviewers and the editors, who helped us improve the quality of this paper through their valuable comments. Special thanks go also to Rania Khalaf, Fransisco Curbera, and Sven Kloppenburg for our fruitful discussions. An early version of this paper was published in the proceedings of ECOWS 2004 [13].

References

1. Alonso, G., Casati, F., Kuno H., et al.: Web Services: Concepts, Architecture, and Applications. Springer, Berlin Heidelberg New York (2004)
2. Andrieux, A., Czajkowski, K., Dan, A., et al.: Web Services Agreement Specification (WS-Agreement) Version 1.1. September 2004. Available at http://www.gridforum.org/Meetings/GGF12/Documents/WS-AgreementSpecification.pdf
3. Apache: Xalan-Java Version 2.6.0. Available at http://xml.apache.org/xalan-j/
4. Arkin, A.: Business Process Modeling Language (BPML) Version 1.0. June 2002. Available at http://www.bpmi.org/bpml.esp
5. Arkin, A., Askary, S., Bloch, B., et al.: Web Services Business Process Execution Language Version 2.0, Working Draft. August 2005. Available at http://xml.coverpages.org/WSBPEL-SpecDraftV181.pdf
6. Bachmendo, B., Unland, R.: Aspect-based workflow evolution. In: Proc. of the Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns. Lancaster, UK, August 2001
7. Benatallah, B., Sheng, Q.Z., Dumas, M.: The self-serv environment for web services composition. IEEE Internet Computing **7**(1), 40–48 2003
8. Bockisch, C., Haupt, M., Mezini, M., et al.: Virtual machine support for dynamic join points. In: Proc. of the 3rd International Conference on Aspect-oriented Software Development (AOSD), pp. 83–92, Lancaster, UK, March 2004. ACM, New York, NY, USA (2004)
9. Bodkin, R.: Application security aspects. In: Invited Talk at the Industry Track of the 4th International Conference on Aspect-Oriented Software Development (AOSD), Chicago, IL, USA, March 2005
10. Burke, B., Fleury, M., Brock, A., et al.: JBoss AOP Version 1.3.0", 2005, Available at http://aop.jboss.org
11. Bussler, C.: Adaptation in workflow management. In: Proc. of the 5th International Conference on Software Process (ICSP), Chicago, IL, USA, June 1998
12. Casati, C., Ilnick, S., Jin, L., et al.: Adaptive and dynamic service composition in eFlow. In: Proc. of the 12th International Conference on Advanced Information Systems Engineering (CAiSE), Stockholm, Sweden, June 2000, LNCS, vol. 1789, pp. 13–31. Springer, Berlin Heidelberg New York (2000)
13. Charfi, A., Mezini, M.: Aspect-oriented web service composition with AO4BPEL. In: Proc. of the 2nd European Conference on Web Services (ECOWS), Erfurt, Germany, September 2004, LNCS, vol. 3250, pp. 168–182. Springer, Berlin Heidelberg New York (2004)
14. Charfi, A., Mezini, M.: Hybrid web service composition: business processes meet business rules. In: Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC), New York, NY, USA, November 2004, pp. 30–38.
15. Charfi, A., Mezini, M.: Application of aspect-oriented programming to workflows. In: Proc. of 5èmes Journées Scientifiques des Jeunes Chercheurs en Génie Electrique et Informatique (GEI), Sousse, Tunisia, March 2005
16. Charfi, A., Mezini, M.: Middleware services for web service compositions. In: Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW), Chiba, Japan, May 2005, pp. 1132–1133. ACM, New York, NY, USA (2005)
17. Charfi, A., Mezini, M.: Using aspects for security engineering of web service compositions. In: Proc. of the 2nd IEEE International Conference on Web Services (ICWS), Orlando, FL, USA, July 2005, vol. I, pp. 59–66. IEEE Computer Society Press, Los Alamitos, CA (2005)
18. Charfi, A., Schmeling, B., Mezini, M.: Reliable messaging for BPEL processes. In: Proc. of the 3rd IEEE International Conference on Web Services (ICWS), Chicago, IL, USA, September 2006
19. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0", W3C Recommendation, 16 November 1999
20. Coady, Y., Kiczales, G., Feeley, M., et al.: Using aspectC to improve the modularity of path-specific customization in operating system code. In: Proc. of the 8th European Software

Engineering Conference (ESEC), Vienna, Austria, September 2001, pp. 88–98. ACM, New York, NY, USA (2001)
21. Colyer, A., Clement, A., Harley, G., et al.: Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley, Reading, MA (2005)
22. Cottenier, T., Elrad, T.: Dynamic and decentralized service composition with aspect-sensitive services. In: Proc. of the 1st International Conference on Web Information Systems and Technologies (WEBIST), Miami, FL, USA, May 2005
23. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, MO, USA, May 2005, pp. 69–77.
24. Curbera, F., Goland, Y., Klein, J., et al.: Business Process Execution Language for Web Services (BPEL4WS) Version 1.1", May 2003. Available at http://www-106.ibm.com/developerworks/library/ws-bpel/
25. Curbera, F., Khalaf, R., Nagy, W., et al.: Implementing BPEL4WS: the architecture of a BPEL4WS implementation. In: Proc. of the GGF 10 Workshop on Workflow in Grid Systems, Berlin, Germany, March 2004
26. Delisle, P., Luehe, J., Roth, M.: "Java Server Pages Specification, Version 2.1", Sun (2005)
27. DeMichiel, L.G.: Enterprise Java Beans Specification, Version 3.0", Sun, Java Specification Request 220 (2004)
28. D'Hondt, M., Jonckers, V.: Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In: Proc. of the 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, UK, March 2004, pp. 132–140.
29. Duclos, F., Estublier, J., Morat, P.: Describing and using non functional aspects in component based applications. In: Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 2002, pp. 65–75. ACM, New York, NY, USA (2002)
30. Eichberg, M., Mezini, M.: Alice: modularization of middleware using aspect-oriented programming. In: Proc. of the 4th International Workshop Software Engineering and Middleware (SEM), Linz, Austria, September 2004, LNCS, vol. 3437, pp. 47–63. Springer, Berlin Heidelberg New York (2004)
31. Fabry, J., Cleenewerck, T.: Aspect-oriented domain specific languages for advanced transaction management. In: Proc. of the 7th International Conference on Enterprise Information Systems (ICEIS), Miami, FL, USA, May 2005, pp. 428–432.
32. Ferris, C., Langworthy, D. (eds.): Web Services Reliable Messaging Protocol (WS-ReliableMessaging)", February 2005. Available at http://www-128.ibm.com/developerworks/library/specification/ws-rm/
33. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: Proc. of the OOPSLA Workshop on Advanced Separation of Concerns, Minneapolis, MN, USA, October 2000
34. Flechter, T., Furniss, P., Green, A.; et al.: BPEL and Business Transaction Management, Choreology submission to OASIS WS-BPEL Technical Committee", 2003. Available at http://www.oasis-open.org/committees/download.php/3263/
35. Gamma, E., Helm, R., Johnson, R., et al.: Design Patterns. Addison-Wesley, Reading, MA (1995)
36. Georgakopoulos, D., Hornick, M.F., Sheth, A.P.: An overview of workflow management: from process modeling to workflow automation infrastructure. Distributed and Parallel Databases **3**(2), 119–153 (1995)
37. Han, Y., Sheth, A., Bussler, C.: A taxonomy of adaptive workflow management. In: Proc. of the Workshop Towards Adaptive Workflow Systems, held in conjunction with the ACM Conference on Computer Supported Cooperative Work (CSCW), Seattle, WA, USA, November 1998
38. Herness, E.N., High, R.J., McGee, J.R.: WebSphere application server: a foundation for on demand computing. IBM Syst. J. **43**(2), 213–237 (2004)
39. Hirschfeld, R., Kawamura, K.: Dynamic service adaptation. In: Proc. of the 4th International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES), Tokyo, Japan, March 2004
40. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewoods Cliffs, NJ (1985)
41. IBM: The BPEL4WS Java Run Time Version 2.1", August 2002. Available at http://www.alphaworks.ibm.com/tech/bpws4j
42. Jain, P.: Oracle Application Server 10 g, Release 2 and 3 New Features Overview, August 2005. Available at http://www.oracle.com/technology/products/ias/pdf/1012_nf_paper.pdf

43. Johnson, R.: Introduction to the Spring Framework, May 2005. Available at http://www.theserverside.com/articles/article.tss?l=SpringFramework
44. Johnson, R. Hoeller, J.: J2EE Development without EJB. Wiley, New York (2004)
45. Kaler, C., Nadalin, A. (eds.): Web Services Security Policy Language (WS-SecurityPolicy) Version 1.1, July 2005. Available at http://www-128.ibm.com/developerworks/library/ws-secpol/
46. Kammer, P.J., Bolcer, G.A., Taylor, R.N., et al.: Techniques for supporting dynamic and adaptive workflow. Comput. Support. Coop. Work (CSCW) **9**(3–4), 269–292 (2000)
47. Keller, A., Ludwig, H.: The WSLA framework: specifying and monitoring service level agreements for web services. J. Netw. Syst. Manag. **11**(1), 57–81 (2003)
48. Khalaf, R., Mukhi, N., Weerawarana, S.: Service-oriented composition in BPEL4WS. In: Proc. of the 12th International World Wide Web Conference (WWW), Alternate Paper Tracks, Budapest, Hungary, May 2003
49. Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An overview of AspectJ. In: Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP), Budapest, Hungary, June 2001, LNCS, vol. 2072, pp. 327–353. Springer, Berlin Heidelberg New York (2001)
50. Kiczales, G., Lamping, J., Mendhekar, A., et al.: Aspect-oriented programming. In: Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP), Jyväskylä, Finland, June 1997, LNCS, Vol. 1241, pp. 220–242. Springer, Berlin Heidelberg New York (1997)
51. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, MO, USA, May 2005, pp. 49–58. ACM, New York, NY, USA (2005)
52. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, pointcut and advice. In: Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP), Glasgow, UK, July 2005, LNCS, vol. 3586, pp. 195–213. Springer, Berlin Heidelberg New York (2005)
53. Kiczales, G., Paepcke, A.: Open Implementations and Metaobject Protocols", tutorial, 1996. Available at http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf
54. Krishnamurthi, S., Fisler, K., Greenberg, M.: Verifying aspect advice modularity. In: Proc. of the 12th International Symposium on Foundations of Software Engineering (FSE), Newport Beach, CA, USA, October 2004, pp. 137–146. ACM, New York, NY, USA (2004)
55. Laddad, R.: AspectJ in Action. Manning (2003)
56. Langworthy, D. (ed.): Web Services Atomic Transaction (WS-AtomicTransaction), November 2004. Available at ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf
57. Masuhara, H., Kiczales, G.: A modeling framework for aspect-oriented mechanisms. In: Proc. of the 17th European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany, July 2003, LNCS, Vol. 2734, pp. 2–28. Springer, Berlin Heidelberg New York (2003)
58. Microsoft: BizTalk Server 2004 Architecture White Paper, December 2004, Available at http://www.microsoft.com/biztalk/techinfo/whitepapers/2004/architecture.mspx
59. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin Heidelberg New York (1982)
60. OASIS: Web Services Business Process Execution Language (WS-BPEL) Technical Committee. Available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
61. OASIS: Web Services Reliability (WS-Reliability) Version 1.1, November 2004. Available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm
62. OASIS: Web Services Security (WS-Security) Version 1.0", March 2004, Available at http://www.oasis-open.org/specs/index.php
63. OASIS: WS-BPEL Issues List, September 2005. Available at http://www.oasis-open.org/committees/download.php/14416/wsbpel_issues_list.html
64. Ortiz, G., Hernandez, J., Clemente, P.: How to deal with non-functional properties in web service development. In: Proc. of the 5th International Conference on Web Engineering (ICWE), Sydney, Australia, July 2005
65. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: Proc. of the 19th European Conference on Object-Oriented Programming (ECOOP), Glasgow, UK, July 2005, LNCS, Vol. 3586, pp. 214–240. Springer, Berlin Heidelberg New York (2005)
66. Papazoglou, M.P.: Web services and business transactions. World Wide Web **6**(1), 49–91 (2003)
67. Pawlak, R., Seinturier, L., Duchien, L., et al.: JAC: a flexible solution for aspect-oriented programming in Java. In: Proc. of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001, LNCS, vol. 2192, pp. 1–24. Springer, Berlin Heidelberg New York (2001)

68. Peltz, C.: Web services orchestration and choreography. Comput. J. **36**(10), 46–52 (2003)
69. Pichler, R., Ostermann, K., Mezini, M.: On aspectualizing component models. Software Practice and Experience **33**(10), 957–974 (2003)
70. Rashid, A., Chitchyan, R.: Persistence as an aspect. In: Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, USA, March 2003, pp. 120–129. ACM, New York, NY, USA (2003)
71. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Trans. Comput. Syst. **2**(4), 277–288 (1984)
72. Sato, Y., Chiba, S., Tatsubori, M.: A selective, just-in-time aspect weaver. In: Proc. of the 2nd International Conference on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, September 2003, LNCS, vol. 2830, pp. 189–208, Springer, Berlin Heidelberg New York.
73. Schlimmer, J. (eds.): Web Services Policy Framework (WS-Policy), September 2004. Available at ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf
74. Schmidt, R., Assmann, U.: Extending aspect-oriented-programming in order to flexibly support workflows. In: Proc. of the ICSE Aspect-Oriented Programming Workshop, Kyoto, Japan, April 1998
75. Sharp, C. (ed.): Web Services Policy Attachment (WS-PolicyAttachment), September 2004. Available at: ftp://www6.software.ibm.com/software/developer/library/ws-polat.pdf
76. Suvée, D., Vanderperren, W., Jonckers, W.: JAsCo:an aspect-oriented approach tailored for component based software development. In: Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, USA, March 2003, pp. 21–29. ACM, New York, NY, USA (2003)
77. Tai, S., Khalaf, R., Mikalsen, T.: Composition of coordinated web services. In: Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware), Toronto, Canada, October 2004, LNCS, vol. 3231, pp. 294–310. Springer, Berlin Heidelberg New York (2004)
78. Tarr, P., Ossher, H., Harrison, W., et al.: N degrees of separation: multi-dimensional separation of concerns. In: Proc. of the 21st International Conference on Software Engineering (ICSE), Los Angelos, CA, USA, May 1999, pp. 107–119. ACM, New York, NY, USA (1999)
79. Tosic, V., Ma, W., Pagurek, B., et al.: Web Service Offerings Infrastructure (WSOI)—a management infrastructure for XML Web Services. In: Proc. of the IEEE/IFIP Network Operations and Management Symposium (NOMS), Seoul, South Korea, April 2004, vol. 1, pp. 817–830.
80. van der Aalst, W.M.P., Barros, A.P., ter Hofstede, A.H.M., et al.: Advanced workflow patterns. In: Proc. of the 7th International Conference on Cooperative Information Systems (CoopIS), Eilat, Israel, September 2000, LNCS, vol. 1901, pp. 18–29. Springer, Berlin Heidelberg New York (2000)
81. Verheecke, B., Cibran, M.A., Suvée, D. et al.: AOP for dynamic configuration and management of web services in client applications. International Journal on Web Services Research (JWSR) **1**(3), 25–41 (2004)
82. von Halle, B.: Business Rules Applied: Building Better Systems Using the Business Rules Approach. Wiley, New York (2001)
83. W3C: Simple Object Access Protocol (SOAP) Version 1.1", May 2000. Available at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
84. W3C: Web Services Choreography Description Language (WS-CDL) Version 1.0", October 2004. Available at http://www.w3.org/TR/ws-cdl-10/
85. Win, B.D.: Engineering application-level security through aspect-oriented software development. Ph.D. dissertation, Department of Computer Science, K.U. Leuven, Belgium, (2004)
86. Wohed, P., van der Aalst, W.M.P., Dumas M., et al.: Analysis of web services composition languages: the case of BPEL4WS. In: Proc. of the 22nd International Conference on Conceptual Modeling (ER), Chicago, IL, USA, October 2003, LNCS, vol. 2813, pp. 200–215. Springer, Berlin Heidelberg New York (2003)
87. Zhang, C., Jacobsen, H.A.: Resolving feature convolution in middleware systems. In: Proc. of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 188–205, Vancouver, Canada, October 2004. ACM, New York, NY, USA (2004)