University of Stuttgart
Germany

Institute of Architecture of Application Systems

BPEL'n'Aspects: Adapting Service Orchestration Logic

Dimka Karastoyanova, Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{karastoyanova,leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{INPROC-2009-54,
  author = {Dimka Karastoyanova and Frank Leymann},
  title = {{BPEL'n'Aspects: Adapting Service Orchestration
           Logic}},
  booktitle = {Proceedings of 7th International Conference on Web
           Services (ICWS 2009)},
  year = {2009},
  pages = {222 - 229},
  doi = {DOI 10.1109/ICWS.2009.75},
  publisher = {IEEE Computer Society}
}
```

BPEL'n'Aspects: Adapting Service Orchestration Logic

Dimka Karastoyanova, Frank Leymann
IAAS, University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{Karastoyanova | Leymann}@iaas.uni-stuttgart.de

Abstract

The need for flexibility in process-based applications, in particular during their execution, places the demand for enabling adaptability of processes. AOP is considered to be one of the approaches to flexibly switch on and off functionality on per-instance basis in applications during their execution; analogously, this paradigm can be applied in a BPEL environment to enable adaptation of running orchestrations. In the presented approach we strive towards reuse of as much concepts and technology already available in a Web service (WS) environment as possible. We combine standard BPEL, the publish/subscribe paradigm and WS-Policy so that WS operations play the role of aspects with respect to BPEL processes. We present the syntax for such aspects as an extension of the WS-Policy framework. We introduce the architecture of the supporting infrastructure and a prototypical implementation. The approach draws on the combined benefits of service orientation and the AOP paradigm to improve the state-of-the-art techniques for flexibility of service orchestrations in a non-intrusive manner.

1. Introduction

Business process modeling is based on the flexible two-level programming approach [10], where business logic is specified separately and independently from the discrete functions implementing single process activities (also called tasks). In a service-oriented environment, functions are provided as services. Currently Web services (WSs) are the technology widely used for applications utilizing service-oriented architectures (SOA) in a standardized manner [14]. The de facto standard for specifying executable processes that use WSs is BPEL [21]; since BPEL allows for recursive composition, all BPEL processes (a.k.a. orchestrations) are also exposed as WSs. This is, BPEL us used to implement the orchestration logic, while WSs are the component model for using applications implemented in any programming language (including BPEL).

The main motivation for our work is the need to improve flexibility of service compositions. In general, process adaptability with respect to process logic, organizational structure, services used, and infrastructure is needed. Process logic adaptability has been dealt with by academic research in terms of process model evolution and process instance migration [1, 6]. Thus process logic adaptability has been enabled for the design and run time life cycle phases of processes, but the approaches used are intrusive and require changes in the languages for describing processes and/or the process execution environments, i.e. the process engines. Organization structure adaptation is covered by BPEL4People. Service adaptability is addressed by the service middleware (a.k.a. bus) and declarative deployment [9], but needs more to handle failures of services referenced by the composition and to adapt the service selection criteria, as [20] works out. Infrastructure adaptation is enabled by the utility model in general and dynamic provisioning in particular. Still, to apply these approaches requires extensions to the process languages, which precludes adapting legacy (BPEL) process models and also hampers reuse. Reuse is a paramount requirement, since apart from being able to adapt to changes in the environment the processes, the underpinning technologies and the supporting infrastructures need to facilitate reuse of legacy applications (including processes) and preserve investment in technologies and skills. In other words, the approaches enabling flexibility need to be *non-intrusive* with respect to existing technologies and infrastructure in order to gain acceptance. These requirements are not yet met by the existing flexibility approaches, as related work shows.

In this paper we argue that applying the AOP paradigm to WS and BPEL environments boosts process flexibility and is in the same time non-intrusive and fosters reuse, high degree of modularity and configurability of processes. Utilizing the capabilities of existing technologies and their implementations as well as computing paradigms successfully applied in industry preserves the relevance of the approach to industrial applications. Thus organizations can profit from both their existing infrastructure and the agility in their reactions to the dynamics of present-day markets.

Our work is based on the analogy we identify between publish/subscribe systems and the AOP paradigm. The contributions of this work are based on the following mapping between AOP concepts and BPEL environments:

- The programs to be enhanced with additional functionality are BPEL processes
- A BPEL engine is the interpreter of the BPEL processes and notifies life cycle events of process elements/activities
- The functions to be weaved into these processes, i.e. the aspects, are Web service operations. Such services may be executed before, after or instead of activities in processes and may overwrite the values of transition conditions and variables. These are the desired adaptations on process models or instances.
- A component executing the actual weaving, i.e. the weaver, enables the inclusion of additional functions/activities (WS operations) into running processes. The weaver thus executes the calls to weaved-in WSs upon a notification of the engine of an event pointed to by a pointcut in an aspect.

The mechanisms and infrastructure implementing this mapping, presented later in the paper, are easy to deploy, generic by design and can be used with any BPEL engine implementation. The approach relies on a messaging infrastructure implementing the publish/subscribe communication mode since the approach maps pointcuts to subscriptions to appropriate process life cycle events. From the point of view of business users it provides the necessary flexibility and agility with respect to reaction to changes in the business and regulation policies, customer satisfaction and faster response to the dynamic market, while preserving existing investment.

The advantages of this approach over existing ones sum up to: (1) dynamic adaptation of processes is enabled as a result of the support for dynamic weaving of aspects during process runtime and on a per-instance basis; (2) the approach is much more generic than other existing approaches for weaving functionality in BPEL since the aspects in our approach are WS operations rather than any other language-specific implementation; (3) since extending the workflow language (e.g. BPEL) is not a generic enough solution, this work provides a general purpose mechanism to specify when which function must be included in a process, based on the standard WS-Policy framework; (4) the approach and the infrastructure can be used with legacy BPEL processes and BPEL execution environments.

The paper is organized as follows. We begin with background information on technologies and paradigms used in this work. Section 3 presents an overview of the BPEL'n'Aspects approach, section 4 describes its concrete realization in terms of WS-Policies and attachments. The architecture of the BPEL'n'Aspects infrastructure is introduced in section 5; the prototypical implementation is presented in section 6. Section 7 gives a summary of related work. We discuss directions for future work and summarize the contributions in the last section.

2. Background Information

2.1 AOP Paradigm

Aspect Oriented Programming (AOP) is an established paradigm that enables describing and separating crosscutting system concerns in a modular and highly reusable manner [8]. AOP supports switching on and off new behavior at a specific point of program execution, while maintaining the system well modularized. AOP is already supported by many software vendors, which is an evidence of its success. Many AOP languages are already available, for example AspectJ, AspectC++, JBoss AOP, Aspect#, Jasco, Spring. AOP is being applied to support flexibility and adaptability of applications/services by allowing to switch on and off orthogonal functions depending on the user requirements. The terms used in this paper come from the AOP field and in particular from AspectJ and are defined as follows [7]: A *joinpoint* is an identifiable point within the execution of a program (e.g. the invocation of a method) where new behavior may be included. The set of possible joinpoints for a component model is called joinpoint model [4]. A *pointcut* is a language construct used to select specific joinpoints for inclusion of new behavior (e.g. all invocations of the "charge credit card" method) and thus allows specifying the particular points in the execution of the original program where the new code is to be inserted. An *advice* is the new behavior to be included at a joinpoint and contains the new code to be executed (e.g. a tracing feature or "store debit" method). Additionally, an advice specifies whether it is to be executed before, after or around (i.e. instead of) the joinpoint. An *aspect* is a unit encapsulating a pointcut and an advice. It specifies the new functionality to be included and the place in the execution of the original program where this functionality is to be inserted. A *weaver* is the functionality that combines the code encapsulated in aspects with the code of the original program. There are different weaving mechanisms [7] that can be classified in two groups – static and dynamic. Dynamic weaving enables the interchangeability or deactivation of aspects during program execution, while static weaving disallows such capability, i.e. once defined aspects cannot be deactivated or exchanged.

2.2 Publish/Subscribe Concepts

Publish/subscribe (or pub/sub) mechanisms are about delivery of information to recipients that are not necessarily known in advance. Often, this information represents signals – so-called *events* – about the occurrence of situations of interest to a third-party. The

format of the delivered information is called a *notification message*. A source of a notification message is called a *producer*, and a target of a notification message – a *consumer*. The act of transporting a notification message from a producer to a consumer is called notification. The original source of a notification message is referred to as publisher. The case in which the notification message is sent directly from the publisher to a consumer is called direct notification. Often, the publisher sends the message to an intermediary called *broker*, and the broker broadcasts the notification message to the consumers. This case is referred to as brokered notification. A *subscription* is an artifact that represents the interest of a consumer for a certain kind of notification message, i.e. an event of interest. Pub/sub paradigm enables decoupling of publishers and subscribers in terms time and space, which is a major advantage.

WS-Notification [22] specifies both direct and brokered notification for WS environments. It is a set of specifications and is based on WSDL, SOAP and WS-Addressing. The WS-Notification specifications family identifies several roles a WS may play, including: publisher, notification producer, consumer, notification broker that implements the consumer and producer roles for brokered notification, subscriber and subscription manager. For each role a port type is defined to enable the communication according to the message exchange protocol specifications for brokered (WS-Brokered Notification) or non-brokered (WS-BaseNotification) notification; the formats for topics and exchanged messages are also defined. Implementations include Pubscribe (http://ws.apache.org/pubscribe/), WS-Messenger (www.extreme.indiana.edu/xgws/messenger/), ServiceMix (http://incubator.apache.org/servicemix/home.html), MUSE (http://ws.apache.org/muse/).

2.3 BPEL and BPEL Engines

BPEL provides a flexible model for service composition using a process-based approach. The constructs/activities in BPEL enable the definition of control flow, data and data manipulation, exception and event handling, compensation scopes etc. BPEL is centered on WSs and is designed to be extensive; available extensions cover: sub-process support (BPEL-SPE [24]), human involvement (BPEL4People, http://www.oasis-open.org/committees/bpel4people/charter.php), use of semantic WS (BPEL4SWS [19]) and of Java code (BPELj, http://www.ibm.com/developerworks/library/specification/ws-bpelj/), and support for other service technologies using BPELlight [18].

The BPEL code is deployed on a BPEL engine and then typically transformed into an engine-internal optimized representation. BPEL engines comprise several components. The so-called navigator uses the process internal representation and navigates over the process models to execute each process instance. It delegates the execution of interaction activities to the underlying middleware, the bus [9]. Each engine uses proprietary implementation of activities, their states and the event model that controls the state transitions. The navigator is usually built so that while executing process instances it changes the status of activities using a set of predefined events, i.e. it performs navigation steps. In our previous work the event models of several BPEL engines have been utilized to propagate information about the status of process instances to external components (audit trails, monitoring tools) as well as to influence the behavior of the process instances from the outside [15]. As a result we have created a modular, pluggable infrastructure that enables mapping of the internal engine event model to messages sent to external components that plug additional functionality into the engine without the necessity of changing the BPEL models, extending the BPEL language and/or the engine. For more details on the infrastructure architecture consult section 5 and [15].

3. BPEL'n'Aspects – The Approach

In this section we provide an overview of the BPEL'n'Aspects approach. It is based on the analogy between event notifications and reaction to them and the AOP paradigm. With this approach we strive towards reuse of as much concepts and technology already available in a WS environment as possible and thus maintain reusability of concepts and technologies, and compliance to legacy WS-based applications.

Processes are executed through interpreting of the process model by an underlying process engine [10]. Especially, a BPEL engine interprets process models specified in BPEL in terms of discrete navigation step. Any functionality that needs to be weaved into a running BPEL process can be interleaved with the original process logic before or after discrete navigation steps. Our approach supports dynamic adaptation of process logic which corresponds to run-time weaving in AOP terms.

The basic idea is to surface events occurring during navigation through a BPEL process model. These events signal that the BPEL engine (similar to a virtual machine) has reached an event of interest (or a joinpoint). Operations of services may be registered as subscribers to such events. Whenever an event happens, the operation of the registered service will be invoked - this is in fact the weaving. We use concepts known from the publish/subscribe paradigm [22] as the underpinnings of our mapping of the concepts of aspect orientation onto a BPEL environment (see also Figure 1):

- *Joinpoints* are mapped to specific language elements in BPEL. For example, invoke activities, transition

conditions etc. are supported joinpoints. All events notifying a state change of these elements during run time are potential joinpoints.

- *Advices* are mapped to WS operations. For example, the "discount calculation" operation of the "Discount Calculation Web Service" is specified as an "after" advice (see Figure 1). Thus, in contrast to other existing approaches, any WS can stand for the implementation/code of an aspect, which is very much in synch with the fundamentals of the WS technology, where services are a first class citizen; interoperability is promoted, too.
- *Pointcuts* are mapped to subscriptions for navigation events occurring when a workflow engine interprets BPEL processes. For example, a pointcut may select the "Invoke Calculation" invoke activity of the "Order Placement" process model. This means that a service (advice) will be executed once the BPEL engine fires the event stating that the "Invoke Calculation" invoke activity has been executed (if the "after" advice type is used).
- *Aspects* are mapped to packages coupling subscriptions for navigation events (pointcuts) and operations of WSs (advices). For example, an aspect specifies that the "discount calculation" operation of the "Discount Calculation Web Service" is to be executed after the "Invoke Calculation" activity of an instance of the "Order Placement" process (see Figure 1).
- *Weaving* is mapped to signaling and observing navigation events and invoking operations of Web services upon occurrence of events of interest. For example, when navigating through an instance of the "Order Placement" process model the environment will execute the "discount calculation" operation of the "Discount Calculation Web Service" after having performed the "Invoke Calculation" activity and thus enhance the process instance with additional functions not modeled in its process model.

Joinpoint models depend on the component models used for the implementation of applications, e.g. in BPEL all language elements are potential candidates for joinpoints. Reaching a joinpoint in a BPEL process is expressed in terms of events the BPEL engine needs to generate and notify. The events in a process which are relevant for process adaptability enablement are those that make up the joinpoint model in this work and are tied to appropriate BPEL constructs [17], e.g. ActivityReady, ActivityExecuted, ActivityExecuting, Link_Evaluated Scope_Compensated, etc.

We allow implementing advices through WS operations at concrete ports, not port types. Allowing a subscription to events on behalf of an operation, for which the port is unknown at the time of weaving, would require a service bus with the capability of discovering appropriate ports dynamically. Work on dynamic service discovery exists and is not in the focus of this work.

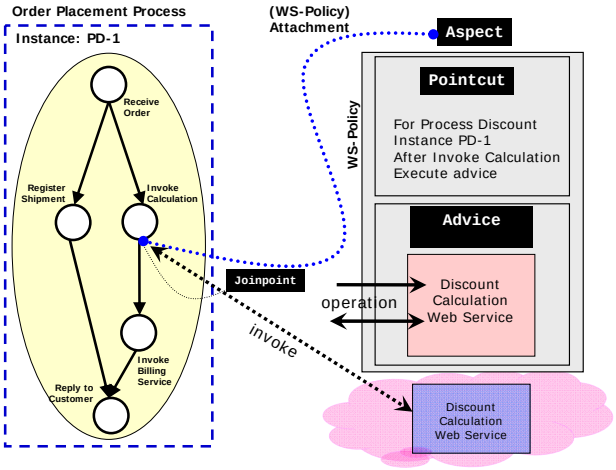

Figure 1: Weaving Services into Processes

3.1 Attaching Aspects to BPEL Processes

We use an attachment mechanism to define where in the process (model or instance) an advice (a WS) is to be interleaved (see Figure 1). In this way we specify which event, i.e. which construct and its state within a certain process model, will trigger an interaction with which WS/advice. This attachment may be specified (1) for all instances of the subject process model, (2) only for a specific instance, or (3) for a subset of instances (e.g. all instances in which the value of a variable is under a value relevant for the business logic, or all instances of a process that have been created after a concrete date, etc.).

3.2. Advice Types

An advice defines whether the WS will be executed instead of a construct, i.e. instead of an activity, instead of a transition condition, or in addition to a construct – before or after. The concrete syntax is presented in the next section. Obviously, the advice types makes sense only with certain process instance events, e.g. calling a WS instead of an activity after the activity has completed (event "Activity_Executed") is not reasonable; these restrictions are taken into account when identifying the events for the event model [17, 16] and hence the joinpoint model used.

3.3. Aspect Life Cycle

The life cycle of an aspect in our approach starts with the creation of the aspect. This phase is followed by the explicit deployment of the aspect on the infrastructure. For this the aspect and the specification of the attachment

to process model or instance is used and results in a subscription defined for a particular event. Attaching an aspect to any process instance will be allowed only if the process instance has not reached the state, whose corresponding event notification has been identified as relevant for weaving in this particular aspect. This is the moment at which an aspect is weaved into the process model or one or more of its instances. The aspect is executed as often as the event for which the aspect subscribes is intercepted. The life cycle of an aspect ends with its undeployment, i.e. explicit detachment from the target artifact. Typically, whenever an aspect is undeployed, the processes that have already initiated the execution of the aspect are allowed to finish this WS call. Other modes of operation are also possible. The aspect life cycle can be controlled by a validity time interval in the aspect definition. This in turn will lead to a timed subscription for the event signaling that the joinpoint has been reached. Note that the processes being adapted are in the execution phase of their life cycle whenever aspects are executed.

3.4. Advanced Functions

Aspects can be applied collectively as a group in an "all or nothing" manner – *composite aspects*. This avoids having some aspects already applied and executed, while others are still being attached in the case that the aspects really belong together. A composite aspect can only be attached if none of its pointcuts has been passed.

Compensating for the dynamically introduced activities is always crucial in flexible workflows. The same is valid for *compensation of the aspects* introduced here. Since the weaved-in functionality is not foreseeable during process modeling, it is not reasonable to expect that compensation handlers be defined in the process models. The services/activities that are dynamically incorporated into the main process logic should be described so as to contain reference to their compensating services; thus the compensation of the dynamically introduced functionality would be enabled.

Data exchange and data dependencies among processes and aspects requires novel mechanisms for dealing with (i) data produced by an aspect that is not needed by a process, (ii) data type and format mismatch between process variables and input or output of a WS, (iii) data unavailable in the process but needed by included functions. Data dependencies are also an issue during fault handling and compensation.

4. Realizing Aspects as Policies

We use WS-Policies [23] to represent aspects and WS-Policy Attachments as the means for associating aspects with BPEL processes. An attachment allows to associate policies with an artifact (<AppliesTo> in Listing 1) that already exists. Especially, a policy can be associated or dissociated at any time during the lifetime of the artifact. Using attachments, thus, enables highly dynamic scenarios of defining aspects for (existing) BPEL processes and attaching them to process models or instances during their execution. An artifact is identified by a so-called domain expression, i.e. a domain-specific means of unique identification. The associated policy is either directly included in the attachment (<Policy>) or referenced (<PolicyReference>). Referencing policies enables reuse; in our case reuse of aspect specifications. Reuse is facilitated by the use of Policy Attachments in the first place.

```
<wsp:PolicyAttachment>
 <wsp:AppliesTo>   <DomainExpression/>+
 </wsp:AppliesTo>
  (<wsp:Policy>...</wsp:Policy>|
<wsp:PolicyReference>...
                </wsp:PolicyReference>)+
</wsp:PolicyAttachment>
```

Listing 1. Schema of Policy Attachment

In this work domain expressions are identifiers of process instances. Process instances can be specified by indicating unique process instance identifiers (PIIDs) or by specifying the name of the process model, all of whose instances are thus identified. The domain expression together with the process artifact element defines the language used as quantification mechanism in AOP terms, i.e. the pointcuts. Note, that any other identification mechanisms could be invented like: all instances created in certain period of time, all instances of a process model in a certain state etc. Concrete identification mechanism is not the focus of our work we only specify the basic ones (Listing 2):

```
<a4B:Processes> (<a4b:Model name="…"/>? |
  <a4b:Instance ID="…"/>*) </a4b:Processes>
```

Listing 2. Domain expression example

Aspects are defined as policies, to be precise as assertions as defined by the WS-Policy framework. An assertion is a domain-specific specification of certain behavior; domain-specific assertion specifications are identified by a separate namespace (in this work it is xmlns:a4b="http://www.iaas.uni-stutgart.de/iaas/a4b"). In Listing 3, the <a4b:Advice> element is the container for the specification of the WS to be weaved in. This service is identified by an endpoint reference. The operation to be used as provided by this service is specified in the <a4b:Operation> element. The message sent to the service may be materialized from the process context (all the variables in a BPEL process comprise its context); for this purpose an <a4b:InputTransform> element may be

specified. Thus it is possible to state that, for example, an aspect executed before an activity needs to modify the input variable for this activity. Similarly, the response message may have to be folded into the process context, which can be specified via the <a4b:OutputTransform> element. The <a4b:When> element specifies whether the service must be run before, instead or after executing the specified process artifact, as identified by the <a4b:ProcessArtifact> element and its attributes for type and corresponding identifier. Note that the identification mechanism depends on the type of artifact; e.g. an activity may be identified uniquely by its name, while a transition condition is identified by an expression or its name. One could consider including WS-Policy operators in order to narrow the specific pointcut with logical operators that may use variable values as selection criteria (e.g. a customer order is over a threshold that makes him eligible for a discount).

```
<a4b:Aspect Id="..."?>
  <a4b:Advice name="..."?>
    <a4b:When type="before|instead|after"/>
    <wsa:EndpointReference>...
                  </wsa:EndpointReference>
    <a4b:Operation name="..."/>
    <a4b:InputTransform>...
                  </a4b:InputTransform >?
    <a4b:OutputTransform>...
                 </a4b: OutputTransform>?
  </a4b:Advice>
  <a4b:Pointcut>
   <a4b:ProcessArtifact type="activity
   |transitionCondition |..."
    identifier="..."/>   </a4b:Pointcut>
<a4b:Aspect>
```

Listing 3 Schema of an Aspect

Specifying aspects as assertions allows combining aspects into an overall policy based on the grammar defined in [23] and collectively attaching them to BPEL process models (composite aspects support). When aspects are assigned to artifacts the supporting infrastructure must verify if the joinpoints specified in the pointcuts of the aspects have already been passed. If this is the case, attaching an aspect to an artifact must be disallowed.

5. Infrastructure Architecture

The infrastructure we architected to enable the BPEL'n'Aspects approach comprises four major components (see Figure 2). The *BPEL engine* is the component executing the BPEL processes. For each process instance the engine tracks the status of all elements of the corresponding process model. It also delegates the invocation of services that implement process activities to a piece of middleware, called the *Service Bus* [9]. Weaving is mapped onto features of a *broker*. The broker (i) manages the deployment and undeployment of aspects, (ii) manages and publishes the notifications of events of interest happening in the process engine for which a corresponding aspect is defined, (iii) interacts with the Bus in order to delegate the invocation of services (aspects) and takes care of returning the results of WS invocations (i.e. aspect execution) to the concrete artifacts (process instance) and thus (iv) performs the weaving of aspects into processes. The *aspect management tool* is used to create, edit, delete, deploy, and undeploy aspects and aspect groups. The tool generates the WS-Policy attachments and the domain expressions defined in this work whenever a user specifies which service needs to be dynamically weaved-in for a particular process model or instance.

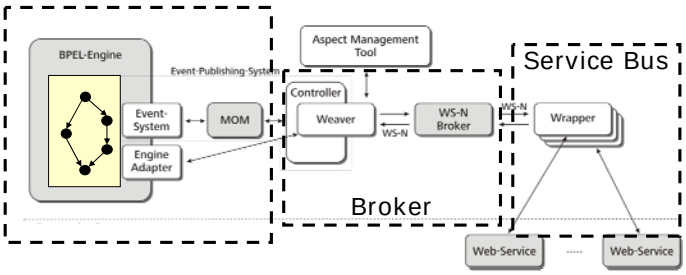

Figure 2: Conceptual architecture and prototype

The engine is able to signal navigation events to the broker, i.e. the engine is a publisher, and the broker is able to receive the event notifications and hence plays the subscriber role. Thus the weaving is realized in part by subscribing the broker for events published by the engine. Additionally, the broker manages the subscriptions of the aspects/WSs, hence it plays the roles of subscription manager and notification publisher with respect to the WSs to be invoked, which are in turn the subscribers. The broker has to pass back the responses of WSs to the engine. The response message can be returned to the particular artifact, for which the aspect/WS has been executed using notifications again. In this case the broker plays the role of a notification produces/publisher, while the engine is the notification consumer. The engine then dispatches the message to the appropriate artifacts [15]. This functionality is also part of the realization of the weaving.

Attaching/detaching aspects (i.e. policies) to elements in BPEL processes (and process instances) results in interpreting the attachment and creating or deleting a subscription to process instance events according to the pointcut specified in the aspect. After attaching an aspect, the created subscription at the side of the broker enables it to receive a notification message signaling that the joinpoint identified by the pointcut has been reached.

Then the aspect implementation (a WS) is invoked and the result returned to the engine.

In the rest of the section we present more details on the infrastructure architecture. To enable the communication between the broker and the engine, we extended the generic process engine architecture [15]. The engine has been augmented with a component, called controller, which is responsible for notifying life cycle events about process instance constructs and reacting to external events appropriately (according to predefined protocols); the controller therefore implements the logic for the reaction to these events, too. The controller is a pluggable component allowing for plugging in domain/protocol-specific controller implementations. The *weaver* is one such domain-specific controller implementation; note that the weaver is a part of the broker conceptual component. The weaver is a consumer of the navigation events produced by the engine and filters out the events irrelevant for the domain it supports. In our case the weaver subscribes only to events that are directly mapped to the joinpoint model used, i.e. to joinpoints relevant for process adaptation. For example, it registers only for the messages signaling life cycle events pertaining to a single activity after which an advice needs to be executed in a single process instance and not for all other life cycle events related to the same process instance; for this purpose a pub/sub middleware (MOM) implementation is utilized.

The filtering of the (process instance and activities) life-cycle messages is done at the broker, because only the messages that are subscribed to by aspects need to be notified (by the weaver). Some of the events published by the engine block the execution of process instances to ensure that the advice/WS is executed before the next activity is performed. The classification of events (incoming, outgoing, blocking) is presented in [17]. Details about the custom controller, i.e. the weaver, are also presented in [15].

The weaver facilitates maintainability, and can be easily applied with the internal event model of any BPEL engine. Any additional events/joinpoints can be included since it is inherently extensible.

The communication between the weaver and the WSs is enabled in terms of a pub/sub infrastructure; the actual service calls are done by the so-called wrappers. For each of the pointcuts of the deployed aspects a topic is created (by the Aspect Management Tool). Any time an event, for which a topic is defined, is notified by the engine, the weaver publishes the message on the corresponding topic; then the wrapper being the subscriber to that topic executes the service invocation [16, 15]. For each of the pointcuts (i.e. topics) there is only one wrapper that consumes the messages. The wrapper serves as a gateway to the weaver for service calls and thus enables its decoupling from the service invocation functionality. Upon response from a WS the wrapper publishes the message on a single topic where the responses of all WS calls are published; the weaver is a subscriber.

6. Prototype

In previous projects we have extended the open-source ActiveBPEL engine (www.activebpel.org) with an event publishing framework, so that life cycle events can be propagated to external components like audit trails, monitoring tools, and others [15, 20]. Additional extensions were needed to support the presented approach through handling of incoming events notified by the weaver to the engine; for this aspect-oriented programming using AspectJ has been utilized (to ensure modularity and non-intrusiveness of changes). The MOM implementation used is the ActiveMQ (http://activemq.apache.org/) JMS implementation. The filtering of events is done using selectors as enabled by JMS. In addition, the weaver and the aspect management tool have been built from scratch. We have chosen to use WS-Notification for the communication between the weaver and the wrappers, for the benefits of a standardized solution. We leverage the WS-Notification implementation provided by WS-Messenger, which also provides the piece of bus infrastructure responsible for the service calls. The weaver is a J2EE Web application. The aspect management tool is a standalone Java application. It provides a Swing-based GUI to facilitate management, deployment and undeployment of aspects on the infrastructure in a user-friendly manner. It utilizes the Apache Neethi (http://ws.apache.org/commons/neethi) implementation of the WS-Policy framework for editing and storing policies.

7. Related Work

Substantial amount of research has been done in applying aspect-oriented techniques in BPEL environment. For example, the AO4BPPEL language is an aspect-oriented extension for BPEL which permits aspects to be included in BPEL processes at runtime [3]. Each BPEL activity is considered a potential joinpoint, pointcuts are specified by XPath expression, while advices are implemented as BPEL activities or Java methods calls. In comparison, the BPEL'n'Aspects approach presented is not restricted to only BPEL code for the advice implementations, but rather allows for the use of any WS. Additionally, we avoid extending BPEL and thus enable reuse of legacy BPEL processes. The authors of [5] apply AOP to adapt and extend a BPEL engine with new features like tracing, debugging and new language constructs using the so-called engine aspects. In addition, process aspects are used to enable dynamic

weaving of BPEL code into BPEL processes or instances. Furthermore, the focus of this approach is the enactment of monitoring of processes, and not merely process adaptation. Since this work also focuses on weaving aspects implemented in BPEL only, we argue that weaving in any kind of implementation using WSs is the more generic and realistic approach. In [2] BPEL processes are annotated with rules for monitoring in order to control functional and non-functional properties. Static weaving is carried out by a BPEL pre-processor.

The DySOA project [13] aims at enabling self-adaptive service systems by monitoring their QoS requirements dynamically. The work presented in [12] proposes the use of a rule driven approach for Business Collaborations Development. WS-Policies and AOP have been combined to enable flexible re-configuration of services in the absence of a service bus in [11], too. These approaches have different objectives than the ones we target in the present work.

8. Conclusions

In this paper we presented how the AOP paradigm can be applied to the WS technology in general and business process technology in particular to improve greatly their applicability in real world scenarios where flexibility is of utmost importance. The BPEL'n'Aspects approach utilizes the AOP paradigm and existing WSs and BPEL infrastructures for the purpose of improving the flexibility of BPEL processes, and in particular the adaptation of process logic. The non-intrusiveness, modularity, and maintainability features of AOP are preserved while discarding the necessity to change the BPEL language or the BPEL engine.

Unlike related research results, our approach is much more generic since the aspects we define are implemented by WSs only. We also rely on standards from the WS technology stack like WS-Notification for publishing events and WS-Policy for attaching aspects to process models or instances. The original process descriptions do not need to be modified to adapt, which is an enormous practical advantage over the existing approaches for flexibility of processes described in BPEL or using any other language. Additionally, this makes our approach applicable in any WS compliant industrial infrastructures due to its inherent support for interoperability.

In our future work we will mainly focus on refining the approach itself and its architecture and implementation with respect to composability of aspects, auditing and compensation of weaved-in functionality.

References

1. W.M.P. van der Aalst, et al. Adaptive Workflow: On the interplay between flexibility and support. In Proceedings of ICEIS'98, Setubal, Portugal, 1998. pp. 353-360
2. L. Baresi, S. Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In Proc. of ICSOC'2005, LNCS 3826.
3. A. Charfi, M. Mezini. Aspect-Oriented Web Service Composition. In Proceedings of ECOWS 2004.
4. C. v. F. G. Chavez, C. J. P. Lucena. A Theory of Aspects for Aspect-oriented Software Development. In Proceedings of Brazilian Symposium on Software Engineering, 2003.
5. C. Courbis, A. Finkelstein. Towards Aspect Weaving Applications. In Proceedings of ICSE 2005.
6. M. Reichert, P. Dadam. ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Information Systems 10(2), 1998.
7. J. Gradecki, N. Lesiecki. Mastering AspectJ. Wiley Publishing, 2003.
8. G. Kiczales. Aspect-Oriented Programming. In Proceedings of ECOOP'97, Finland, 1997.
9. F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. In Proceedings of ICSOC'2005.
10. F. Leymann, D. Roller. Production Workflow - Concepts and Techniques. PTR, 2000.
11. G. Ortiz, F. Leymann. Combining WS-Policy and Aspect-Oriented Programming. In Proceedings of ICIW'06, French Caribbean, February, 2006.
12. B. Orriens, J. Yang, M. Papazoglou. A rule driven Approach for Developing Adaptive Service Oriented Business Collaborations. In Proceedings of ICSOC'2005.
13. J. Siljee et al. DySOA: Making Service Systems Self-Adaptive. In Proc. of ICSOC'2005, LNCS 3826 Springer
14. S. Weerawarana et al. Web Services Platform Architecture, Prentice Hall, 2005.
15. R. Khalaf, et al. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. Proceedings of WESOA 2007 at ICSOC'07, 2007, Springer LNCS.
16. R. Schroth Konzeption und Entwicklung einer AOP-fähigen BPEL Engine und eines Aspect-Weavers für BPEL Prozesse. Thesis No. 2523, University of Stuttgart, 2006.
17. D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model, Technical Report Nr. 2006/10, University of Stuttgart, 2006.
18. J. Nitzsche et al. BPEL^{light}. In Proceedings of BPM 2007, Australia, 2007.
19. J. Nitzsche et al. BPEL4SWS. In Proceedings of AWESOME'07 at OTM 2007, 2007.
20. D. Karastoyanova, F. Leymann, J. Nitzsche, B. Wetzstein, D. Wutke. Parameterized BPEL Processes: Concepts and Implementation. In Proceedings of BPM 2006.
21. BPEL 2,0, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
22. Web Services Notification, 2004.
23. WS Policy, ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf
24. M. Kloppmann et al. WS-BPEL Extension for Sub-processes – BPEL-SPE. 2005.