Isolating Process-Level Concerns using Padus

Mathieu Braem[1], Kris Verlaenen[2], Niels Joncheere[1], Wim Vanderperren[1], Ragnhild Van Der Straeten[1], Eddy Truyen[2], Wouter Joosen[2], and Viviane Jonckers[1]

[1] System and Software Engineering Lab (SSEL), Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{mbraem,njonchee,wvdperre,rvdstrae,vejoncke}@vub.ac.be
[2] DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{kris.verlaenen,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract. Current workflow languages for web services suffer from poor support for separation of concerns. Aspect-oriented software development is a well-known approach to improve this. In this paper, we present an aspect-oriented extension for the WS-BPEL language that improves on current state-of-the-art by introducing an explicit deployment construct, a richer joinpoint model, and a higher-level pointcut language. In addition, the supporting technology is compatible with existing WS-BPEL engines.

Classification. Business process modeling and analysis, processes and service composition

1 Introduction

Over the last years, *web services* [1] have been gaining a lot of popularity as a means of integrating existing software in new environments. By composing a number of basic web services, new web services can be created that provide more advanced functionality. These compound web services can then be reused in even other web services, which further facilitates software reuse.

Originally, the only way to compose web services was by manually writing the necessary glue-code in programming languages such as C and Java. It quickly became clear, however, that a composition of web services is more naturally captured by dedicated *workflow languages* [2] than by general-purpose programming languages.

Today, the most popular workflow language with regard to the composition of web services is the *Business Process Execution Language* (WS-BPEL) [3]. WS-BPEL builds on the foundations of WSFL [4] and XLANG [5], and can be used to specify both executable business processes and abstract business processes. Executable processes model the behavior of one participant in a composition (i.e. *orchestration*), while abstract business processes specify the externally visible behavior of a composition (i.e. *choreography*). WS-BPEL processes are platform- and transport-independent, and are expressed using XML.

1.1 Separation of Concerns

In this paper we improve the modularization capability of WS-BPEL in order to provide a better separation of concerns [6] in the workflow specification. In WS-BPEL (and other workflow languages, for that matter) a large number of concerns (such as authorization and billing) cannot be cleanly separated from the main functionality of the workflow specification. WS-BPEL processes suffer from a problem that is named the “tyranny of the dominant decomposition” [7]. A WS-BPEL process can only be decomposed according to the control flow of the process, and concerns that do not align with this decomposition end up scattered across the process specification and tangled with one another. For example, billing requires invoking some billing service each time before and after a certain functionality in the process is provided. This makes it difficult to add, modify, or remove such concerns. Also, because WS-BPEL processes must be specified in a single XML file, complex processes give rise to large XML files which can become difficult to understand, maintain and evolve.

To solve the above problem, we propose to apply *aspect-oriented* decomposition and composition mechanisms to WS-BPEL. Aspect-oriented software development (AOSD) [8] has been gaining a lot of popularity as a means of improving separation of *crosscutting concerns* in software. Examples of such *crosscutting concerns* are security concerns such as access control and confidentiality [9], debugging concerns such as logging [10] and timing contract validation [11], and business rules such as billing [12]. The goal of AOSD is to achieve a better separation of concerns, by allowing crosscutting concerns to be specified in separate modules called *aspects*, so that adding, modifying or removing these concerns does not require changes to the rest of the system.

Traditional aspects consist of two main parts: *pointcut* definitions and *advices*. Points in the program execution where an aspect can be applied (e.g. method invocations in object-oriented programming) are called *joinpoints*. Pointcuts select sets of joinpoints where aspects should be applied; these pointcuts can be expressed using declarative pointcut languages. An advice specifies the concrete behavior that should be executed at certain joinpoints — typically before, after or around the original behavior of the joinpoints. Inserting the behavior defined by aspects at the correct locations in the main program is called *weaving*.

Initial research on AOSD has concentrated on applying its principles to the object-oriented programming paradigm. However, as motivated by Arsanjani *et al.* [13] and others [14–16], AOSD has a lot of potential in a web services context, too.

1.2 Web Service Composition in Telecom

The research described in this paper is part of a larger research project, which is named WIT-CASE and is performed in collaboration with Alcatel, and which addresses composition of web services on a telecom service delivery platform. We will therefore illustrate the motivation for our approach by providing examples from within this context. Typical use cases for a telecom service delivery platform

include setting up and executing a multi-party conference call. Such use cases mostly have the same general characteristics. For example, the platform needs to check whether the user is allowed to access the functionality he has requested before providing this functionality (authorization), and the user needs to be billed for his usage according to some billing scheme (billing).

Both the authorization and billing concerns are typically crosscutting. Therefore, an aspect-oriented approach can improve the modularization of web service compositions on a telecom service delivery platform. Without support for AOSD, nearly every WS-BPEL process on our platform would start with some authorization code before executing its main functionality, and would perform some billing functionality before and/or after certain resources are used. This means that, when some part of the authorization or billing policies changes, all these processes need to be modified. The presence of more than one authorization or billing policy would even further complicate this situation.

If, on the other hand, support for AOSD is available, crosscutting concerns such as authorization and billing can be expressed separate from the processes' main functionality in dedicated aspects. If authorization or billing policies would change, this would only require changes to the corresponding aspects, and not to the main processes. If one would like to support more than one authorization or billing policy (e.g. fixed fee billing as well as duration billing), it is sufficient to simply implement an additional aspect.

In this paper, we propose an aspect-oriented programming extension for WS-BPEL, named Padus, in order to provide a better separation of concerns. The characteristics of the telecom service delivery platform and the goals of the WIT-CASE project have had a profound impact on the design and implementation of Padus. First of all, the overall workflow specification language should be sufficiently expressive and should support creation of higher-level composition primitives. Moreover, adding AOP support to WS-BPEL should be as less disruptive as possible to the existing tool chain and should introduce as less run-time performance overhead as possible. For these reasons we have chosen to follow an approach in which the design of Padus is based on a logic-based programming language (in order to increase expressive power and ability to construct higher-level composition primitives) and the implementation of Padus is based on a static transformation approach (in order to be compatible with existing tool chain and minimize run-time performance overhead).

The paper is structured as follows. Section 2 describes our AOP language for WS-BPEL, while section 3 describes how this language is implemented. A brief case study is provided in section 4. We present related work in section 5 and state our conclusions in section 6.

2 The Padus Language

We present Padus, an aspect-oriented extension to WS-BPEL, which aims to overcome its lack of support for modularization of crosscutting concerns. It allows introducing crosscutting behavior to an existing WS-BPEL process in a

modularized way. Developers can augment WS-BPEL processes with additional behavior at specific points during their execution. These points can be selected using a logic pointcut language, and the Padus weaver can be used to combine the behavior of the core process with the behavior specified in the aspects. Using Padus, the complexity of the core process can be controlled by specifying crosscutting concerns like security and billing in separate aspects.

In this section, we describe the design of the Padus language. We follow the template for describing AOP languages proposed in AOSD-Europe's survey on aspect-oriented programming languages [17]. We describe the language along five dimensions: the joinpoint model (section 2.1), the pointcut and advice languages (sections 2.2 and 2.3), the aspect modules (section 2.4), and the aspect deployment language (section 2.5).

2.1 Joinpoint Model

Joinpoints are well-defined points during the execution of a WS-BPEL process where extra functionality could be inserted. They are related to the activities that are provided in WS-BPEL. Table 1 lists the kinds of joinpoints that are available. Each type is related to a specific WS-BPEL activity, which can be easily deduced from the type's name. The joinpoint model does not only allow behavioral joinpoints but also includes structural joinpoints related to structural WS-BPEL activities (which contain one or more activities themselves).

Behavioral joinpoints		**Structural joinpoints**	
invoking	replying	sequencing	switching
receiving	assigning	looping ("while")	picking
throwing	terminating	flowing	scoping
compensating	doingNothing ("empty")		

Table 1. Types of joinpoints available in Padus

Joinpoints are associated with properties relevant to that particular joinpoint. Some of these properties are related to the attributes and elements of the corresponding WS-BPEL activity. For example, table 2 provides the attributes of "invoking" joinpoints. Additional properties specify, among others, in which WS-BPEL process or structural activity a joinpoint occurs. Dynamic properties, like in which process instance a joinpoint occurs and the value of certain variables, are defined too. Using these properties, one can more precisely select interesting joinpoints.

2.2 Pointcut Language

A pointcut selects a specific set of joinpoints. Pointcuts can be used to specify the joinpoints where additional behavior should be inserted. The pointcut language of Padus is based on logic meta-programming [18, 19]. A pointcut can be seen as

Attribute	Type	Description
name	String	An optional name for the WS-BPEL activity
partnerLink	String	The partner link used by the invoke
portType	String	The port type used by the invoke
operation	String	The operation of the port type that is invoked
inputVariable	String	The message that should be sent
outputVariable	String	The variable that should contain the reply message

Table 2. Attributes of "invoking" joinpoints

```
invoking(Joinpoint, 'smsService', 'smsServicePT', Operation),
startsWith(Operation, 'send').
```

Listing 1. Simple pointcut that captures "invoking" joinpoints that invoke an operation of which the name starts with "send"

a collection of constraints on the type and properties of allowed joinpoints. In addition, a pointcut is able to expose certain information (e.g. argument values) so that the advice can exploit this.

The pointcut language defines a predicate for each type of joinpoint. The attributes of the predicate refer to the attributes of that specific type of joinpoint. Table 3 shows the exposed bindings of the invoking predicate. Only the version with the most variables is really required. The others can be written in function of the larger one. The predicates with less variables simply offer extra convenience.

Predicate binding	Description
invoking(Joinpoint, Name, PartnerLink, PortType, Operation, InputVariable, OutputVariable)	All allowed attributes
invoking(Joinpoint, Name, PartnerLink, PortType, Operation)	Input and output variable names not bound
invoking(Joinpoint, PartnerLink, PortType, Operation)	Only Partnerlink, PortType and Operation bound

Table 3. Bindings for the "invoking" predicates

By constraining the attributes of a joinpoint predicate, certain joinpoints can be selected. Pointcuts can combine these predicates with standard predicates that are available in Prolog [20], for comparing basic data types, searching lists, etc. Pointcuts can include negations, and predicates can be combined with conjunctions or disjunctions. The small example in listing 1 denotes a pointcut that covers all "invoking" joinpoints of operations on the `smsServicePT` port type of the `smsService` partner link of which the name of the operation starts with "send".

The pointcut language also offers predicates for constraining or exposing additional (possibly runtime) properties of joinpoints, like for instance the process

or process instance a joinpoint occurs in, etc. Table 4 gives an overview of some of these predicates.

Predicate	Description
inProcess(Joinpoint, Process)	Links a joinpoint with the process it is defined in.
inProcessInstance(Joinpoint, ProcessInstance)	Links a joinpoint with the process instance it occurs in.
variableValue(ProcessInstance, Name, Value)	Links the name of a variable to its value in a specific process instance.

Table 4. Predicates for constraining additional properties of joinpoints

Using a logic pointcut language offers significant advantages over more traditional approaches. The pointcuts can use the full power of unification on logic variables (by backtracking). Furthermore, since pointcuts are logic rules that cover joinpoints, new user-defined pointcuts can be reused in the definition of similar pointcuts. The logic engine supporting our pointcut language also allows writing recursive pointcut definitions. The base predicates available in the pointcut language have well chosen names, which can clearly express the intension of the pointcuts and improve readability.

2.3 Advice Language

The advice language is used to specify how the behavior at certain joinpoints defined by a pointcut should be altered. Similar to traditional aspect-oriented systems, advices can either be *added* to the original behavior, or can *replace* the original behavior. New behavior can be introduced by inserting it *before* or *after* certain joinpoints defined by the pointcut. An *around* advice must be used if existing behavior might need to be replaced.

In advices can be used to add behavior inside some activity, like for example add an extra concurrent activity to a flow activity. This cannot be simulated by before or after advices. In some cases, an around advice could be used as a workaround, but this would result in significant code duplication. The *in* advice can not only be used to add new activities in structural WS-BPEL activities, but also to customize the behavior of certain WS-BPEL activities, like for example adding variables to a scope, or adding flow links to any WS-BPEL activity. Table 5 gives an overview of all the situations where an in advice could be used.

Advice code is defined in an XML element that specifies the type of the advice. A pointcut describes the points in the original process to which the advice applies. The extra behavior that should be inserted is specified using standard WS-BPEL elements. For before, after and around advices, this is a WS-BPEL activity. In advices can be used to insert other WS-BPEL elements too, as specified in table 5. For around advices, the `<proceed>` activity could be used to include the original behavior specified by the joinpoint. The pointcut's attributes

Joinpoint	Element	Description
all types	source	Add the activity as source of a flow link.
	target	Add the activity as target of a flow link.
flowing	activity	Add a new parallel activity to a flow.
	links	Add a new link to a flow.
switching	case	Add a new case to a switch.
	otherwise	Add the otherwise element to a switch.
picking	onMessage	Add a new message trigger to a pick.
	onAlarm	Add a new timeout trigger to a pick.
scoping	variable	Add a variable to a scope.
	correlationSet	Add a correlation set to a scope.
	faultHandler	Add a fault handler to a scope.
	compensationHandler	Add a compensation handler to a scope.
	eventHandler	Add an event handler to a scope.
assigning	copy	Add a copy to an assign.
invoking	correlation	Add a correlation element to an invoke.
	catch	Add a specific catcher to an invoke.
	catchAll	Add a generic catcher to an invoke.
	compensationHandler	Add a compensation handler.
receiving	correlation	Add a correlation element to a receive.
replying	correlation	Add a correlation element to a reply.

Table 5. Pointcuts where an in advice could be used

are exposed to the advice; these can be accessed in the advice by prefixing their name with the '$' character. Listing 2 shows an example of a before advice that logs all invocations of the `smsServicePT` web service. The extra behavior that is inserted is a sequence of two activities: first, the log message containing the invoked operation is created; then, this message is sent to the logging service.

2.4 Aspect Modules

An aspect represents one crosscutting concern. As such, aspects can contain several before, after, in and around advices. Listing 3 shows an example aspect that logs the start and end of all invocations of the `smsServicePT` web

```
<before joinpoint="Jp" pointcut="invoking(Jp, 'smsService', 'smsServicePT', Operation)">
  <sequence>
    <assign>
      <copy>
        <from>Logging invocation of operation $Operation</from>
        <to variable="logMsg" part="msg" />
      </copy>
    </assign>
    <invoke partnerLink="logging" portType="log:loggingPT"
            operation="logMessage" inputVariable="logMsg" />
  </sequence>
</before>
```

Listing 2. An advice that logs all invocations of the SMS service

```
1  <aspect name="logSMSInvocations">
2    <using>
3      <namespace name="xmlns:log" uri="logging.example.com" />
4      <partnerLink name="logging" partnerLinkType="log:loggingLT" />
5      <variable name="logMsg" type="log:logMsg" />
6    </using>
7
8    <pointcut name="smsInvocation(Jp, Operation)"
9              pointcut="invoking(Jp, 'smsService', 'smsServicePT', Operation)" />
10
11   <advice name="logMessage(Message)">
12     <sequence>
13       <assign>
14         <copy>
15           <from>$Message</from>
16           <to variable="logMsg" part="msg" />
17         </copy>
18       </assign>
19       <invoke partnerLink="logging" portType="log:loggingPT"
20               operation="logMessage" inputVariable="logMsg" />
21     </sequence>
22   </advice>
23
24   <before joinpoint="Jp" pointcut="smsInvocation(Jp, Operation)">
25     <advice name="logMessage('Invoking $Operation')" />
26   </before>
27
28   <after joinpoint="Jp" pointcut="smsInvocation(Jp, Operation)">
29     <advice name="logMessage('Invoked $Operation')" />
30   </after>
31 </aspect>
```

Listing 3. An example aspect logging the start and end of all SMS service invocations

service. The main sections of an aspect are the using declarations (lines 2–6), the pointcut (lines 8–9) and advice definitions (lines 11–22), and the actual advices (lines 24–30). To allow reuse of pointcuts and advices, aspects can include other aspect files.

Adding new behavior usually requires extending the information defined at process-level, too. For example, adding a new invocation to a process usually requires adding a partner link that specifies the interface of the new service, and a new variable that will contain the message that should be sent to that service. The <`using`> tag (lines 2–6) allows the definition of such information global to the process. It may include variables, partner links, partners, fault handlers, compensation handlers, event handlers and namespaces.

Pointcut expressions can be reused (lines 8–9) by giving them a name and specifying the parameters, which can either be further constrained when reusing the expression, or be referred to from inside an advice reusing the pointcut. Defining a pointcut expression like this generates a higher-level pointcut predicate that can then be used in other pointcut expressions.

The extra behavior that should be inserted in before, after, around and in advices can be reused too (lines 11–22). The advice behavior is given a name and can be parametrized. These parameters can be referred to from inside the

```
<deployment>
    <!-- the following aspects need to be deployed for the selected processes -->
    <aspect name="..." process="..." id="..." />
    <aspect name="..." process="..." id="..." />
    ...
    <!-- the following precedence declarations are valid for the selected process
         or for all processes if no process is specified -->
    <precedence [process="..."] />
        <aspect id="..." [advice="before|after|around|in"] />
        <aspect id="..." [advice="before|after|around|in"] />
    </precedence>
    ...
</deployment>
```

Listing 4. Aspect deployment specification

advice code with their name (using the '$' prefix). The named advice behavior can be called from within advice code using the <advice> element (line 25 and line 29).

2.5 Aspect Deployment Language

A Padus aspect deployment specifies how aspects should be applied to the base processes and consists of two main parts: aspect instantiation and aspect composition. Aspect instantiation is responsible for instantiating and applying an aspect type to a concrete process. Processes are referenced using their name. It is also possible to select processes in a pattern-based manner using a logic language very similar to our pointcut language. As such, it is for instance possible to select only those processes that invoke a particular service or to select processes whose name starts with a given identifier. Listing 4 illustrates aspect deployment in Padus.

The second part of an aspect deployment, namely the aspect composition, is responsible for specifying the aspect precedence in case multiple aspects apply to the same joinpoint. In case no precedence is specified, the advice is executed in the order in which their corresponding aspects are specified. A precedence declaration overrides this default and is able to specify precedence on a per-advice-type basis. Aspect precedence for a before advice can thus be different than precedence for an after advice. The precedence is also able to vary over several deployments of the same aspect type, as it is bound to the aspect instance's ID and not to its type. Furthermore, the precedence specification can be limited to certain processes only, allowing a custom precedence specification for each process or group of processes if necessary. Similar to aspect instantiation, the process selection can be name-based or pattern-based.

3 The Padus Implementation

3.1 General Architecture

In existing literature on aspect-oriented execution models, two main approaches can be identified:

- **Static Weaving:** In a statically woven approach, the aspect and base-code are woven (i.e. merged) before run-time on either source or byte-code level. At runtime the aspects, like the base code, cannot be redefined, removed nor can new aspects be added.
- **Dynamic Weaving:** A dynamically woven approach uses dedicated techniques to allow weaving at runtime. This allows to dynamically add, remove and redefine aspects.

We opt for a statically woven approach for the execution model of the Padus language. Because the language is used to describe real-time processes in a telecom service delivery platform, performance is extremely important. In contrast to dynamic weaving, static weaving introduces no runtime overhead. Another important advantage is that it does not require a dedicated execution platform (i.e. a modified WS-BPEL engine in our case), which would otherwise seriously limit the applicability of the approach. Figure 1 illustrates the architecture of our weaver. A WS-BPEL process is transformed based on the aspect deployment descriptions. The result is again a regular WS-BPEL process that can be deployed on all WS-BPEL execution engines.

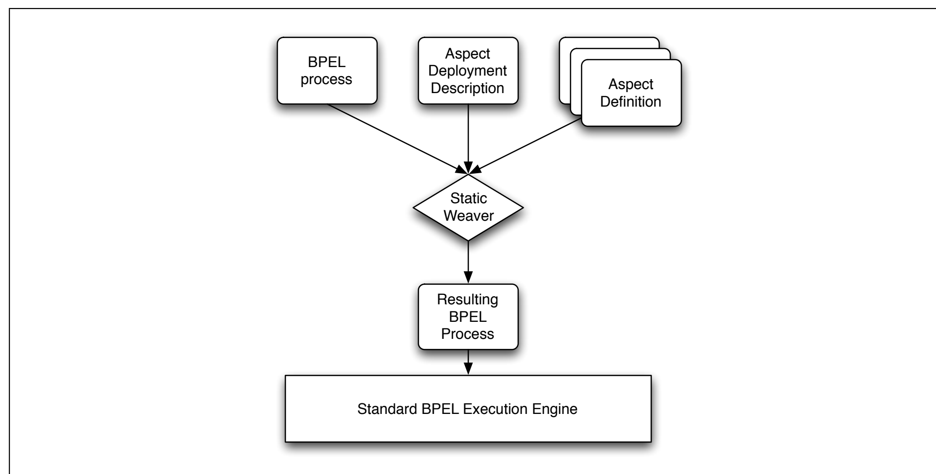

Fig. 1. Padus weaver architecture

3.2 Pointcut Matching and Document Transformation

In order to match the pointcuts and transform the target WS-BPEL specification, the following steps are taken:

- **Translation:** The WS-BPEL process is translated to a set of logic facts in the Prolog language. For every WS-BPEL activity, several facts are generated that define the equivalent activity in Prolog. There is also an explicit back-link to the nodes in the XML tree representation of the WS-BPEL process. This allows for a fast reverse translation process from any given activity to the concrete XML node.
- **Matching:** A logic engine (SWI-Prolog) is used to find all solutions for the pointcut rule. The result is a set of facts representing activities where the aspect is applicable. In case the pointcut defines conditions that are to be dynamically evaluated (such as `variableValue(ProcessInstance, Name, Value)`), partial evaluation is applied to only evaluate the static part of the pointcut. The dynamic part of the pointcut is inserted at the beginning of the advice. If it does not evaluate to true, the advice is not executed. Separating the dynamic part of a pointcut and inserting the conditional advice is independent from a concrete WS-BPEL process and the result might thus be stored for later deployments of the same aspect.
- **Joinpoint Identification:** All solutions for the rule are translated back to WS-BPEL activities using the explicit back-link generated in the translation process. The result is a set of joinpoints denoted by XPath in the WS-BPEL XML tree where the aspect should be woven.
- **Transformation:** An XML transformation engine (based on XSLT) is used to transform the WS-BPEL document at the joinpoints identified in the previous step. Depending on the concrete advice semantics, a different transformation is applied. For a before advice for instance, the advice process is inserted before the identified joinpoints. Non-WS-BPEL constructs in the advice, such as proceed, have to be translated to valid WS-BPEL activities as well. In case of proceed in an around advice for instance, the replaced behavior of the joinpoint is inserted instead of the proceed activity.

4 Case Study

In this section we show how our aspect language can be used to add billing to a multi-party conference call process. Two types of billing schemes are supported: a *fixed fee billing scheme* where the end user should pay a fixed price at the end of the conference call, and a *duration billing scheme* where the price is determined based on the duration of the conference call. Three aspects are used to represent these two billing schemes:

- A *generic billing aspect* (see listing 5) is used to define concepts common to both billing schemes: the billing service and message definitions (lines 2–6), the pointcuts representing the start and end of a conference call (lines 7–10), and an advice for invoking the billing service (lines 11–14).

```
<aspect name="Billing">
  <using>
    <namespace name="xmlns:bill" uri="my.billing.uri" />
    <partnerLink name="billing" partnerLinkType="bill:billingLT" />
    <variable name="billingMsg" type="bill:billingMsg" />
  </using>
  <pointcut name="confCallStarts(Jp)"
          pointcut="invoking(Jp, 'ConfCallService', 'confCallPT', 'createConfCall')" />
  <pointcut name="confCallEnds(Jp)"
          pointcut="invoking(Jp, 'ConfCallService', 'confCallPT', 'closeConfCall')" />
  <advice name="billService">
    <invoke partnerLink="billing" portType="bill:billingPT"
          operation="billService" inputVariable="billingMsg" />
  </advice>
</aspect>
```

Listing 5. Aspect defining generic billing concepts

```
<aspect name="FixedFeeBilling">
  <include name="Billing" />
  <after joinpoint="Jp" pointcut="confCallEnds(Jp)">
    <sequence>
      <assign>
        <copy>
          <from>1.5 EUR</from>
          <to variable="billingMsg" part="price" />
        </copy>
      </assign>
      <advice name="billService" />
    </sequence>
  </after>
</aspect>
```

Listing 6. Aspect implementing a billing scheme with a fixed fee

- The *fixed fee billing aspect* (see listing 6) introduces one advice that invokes the billing service with a fixed price at the end of the conference call.
- In the *duration billing aspect* (see listing 7), a first advice (lines 6–13) stores the start time of the conference call in a new variable (line 4), while a second advice (lines 14–25) uses this time to calculate the price of the conference call based on its duration and then invokes the billing service.

The logic needed for adding billing to the conference call process is now cleanly modularized and is not scattered across the basic control flow, which is very useful for keeping the complexity of the core functionality under control. Any of the two billing aspects can now be combined with the conference call process, or any other process, greatly improving reusability. The billing scheme can easily be modified afterwards too.

The deployment descriptor in listing 8 specifies how the aspect should be instantiated. Here we apply the `FixedFeeBilling` aspect to the `ConferenceCall` process. Suppose a `SecurityCheck` aspect is used to make sure that only users that are allowed to end the conference call can actually do so. In this case, the `SecurityCheck` aspect should be applied first, to make sure that the billing only occurs if the conference call is actually terminated. Note that in this simple

```
<aspect name="DurationBilling">
  <include name="Billing" />
  <using>
    <variable name="startTime" type="xsd:time" />
  </using>
  <before joinpoint="Jp" pointcut="confCallStarts(Jp)">
    <assign>
      <copy>
        <from expression="func:getCurrentTime()" />
        <to variable="startTime" />
      </copy>
    </assign>
  </before>
  <after joinpoint="Jp" pointcut="confCallEnds(Jp)">
    <sequence>
      <assign>
        <copy>
          <from expression="func:calculatePrice(
            bpws:getVariableProperty('startTime'), '0.4 EUR')" />
          <to variable="billingMsg" part="price" />
        </copy>
      </assign>
      <advice name="billService" />
    </sequence>
  </after>
</aspect>
```

Listing 7. Aspect implementing a billing scheme based on duration

```
<deployment>
  <aspect name="FixedFeeBilling" process="ConferenceCall" id="ConferenceCallBilling" />
  <aspect name="SecurityCheck" process="ConferenceCall" id="ConferenceCallSecurity" />
  <precedence process="ConferenceCall" />
    <aspect id="ConferenceCallSecurity" />
    <aspect id="ConferenceCallBilling" />
  </precedence>
</deployment>
```

Listing 8. Aspect deployment specification

example the default precedence could be used to specify the right order in which the aspects should be applied too, but this might not be the case anymore if more processes and/or aspects were defined.

5 Related Work

AO4BPEL [14] is an aspect-oriented extension to WS-BPEL that allows for more modular and dynamically adaptable web service compositions. Each WS-BPEL activity is a potential join point. In contrast to Padus, AO4BPEL uses the lower-level XPath pointcut language. Pointcuts are too low-level and refer directly to paths in the document tree, which limits their reusability and makes them fragile with respect to evolution of the base process. Furthermore, their approach does not support an explicit aspect deployment construct nor allows for aspect reuse. While AO4BPEL allows for aspect addition and removal while processes are

running, supporting this requires a custom-made WS-BPEL engine, which is incompatible with the existing tool chain.

Courbis and Finkelstein [21] present an aspect-oriented language extension very similar to AO4BPEL. They also use XPath as a pointcut language and use a custom WS-BPEL engine for allowing dynamic aspect addition and removal. In contrast to AO4BPEL and Padus, however, the advice language is Java.

The Web Services Management Layer (WSML) [22] uses aspects implemented in JAsCo [23] to capture client-side web service management concerns such as billing, transactions, selection and caching. Compositions of web services are handled by traditional approaches such as WS-BPEL. The WSML is thus complementary to our approach: Padus is able to specify process specific aspects that reflect over the process definition while the WSML specifies service specific aspects independent of process details.

Previous research [18] already showed the advantage of using a logic language for both aspect declaration (defining pointcuts as logical queries) and weaver implementation (representing the program as logical facts) in the context of Smalltalk. The logic meta-programming approach to AOP also allows non-expert programmers to define their own high-level, domain-specific aspect languages.

6 Conclusions and Future Work

The paper presents an extension of WS-BPEL for allowing a better separation of concerns through aspect-oriented programming. The Padus language improves on existing approaches by:

- Providing a rich joinpoint model consisting of all WS-BPEL activities.
- Employing a higher-level logic-based pointcut language that makes the pointcuts less dependent on the concrete document structure. This makes the pointcuts less fragile with respect to evolution of the WS-BPEL process. Because of the higher-level pointcuts, reusing them becomes easier as well.
- Introducing the concept of an *in* advice to add new behavior to existing elements, which extends the expressiveness of the advice language.
- Providing an explicit deployment construct that allows to specify aspect instantiation to specific processes expressively using a logic language. Aspect composition is tackled by an expressive precedence specification that is able to vary depending on the aspect instances, advice types and concrete processes.
- Remaining compatible with the existing infrastructure.

Our aspect-oriented extension for WS-BPEL is an XML-based language and can be defined using an XML Schema [24]. But, similar to specifying a WS-BPEL process using a graphical notation (e.g. BPMN [25]), a more user-friendly graphical notation for aspects can be defined too. We already started on an extension of BPMN that supports the aspect-oriented idea and that can be translated to Padus aspects.

7 Acknowledgments

This research is partly funded by Alcatel Belgium and the Institute for the Promotion of Innovation Through Science and Technology in Flanders (IWT-Vlaanderen) through the WIT-CASE project.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V., eds.: Web Services: Concepts, Architectures and Applications. Springer-Verlag, Heidelberg, Germany (2004)
2. Du, W., Elmagarmid, A.: Workflow management: State of the art vs. state of the products. Technical Report HPL-97-90, Hewlett-Packard Labs, Palo Alto, CA, USA (1997)
3. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services version 1.1 (2003) `http://www.ibm.com/developerworks/library/ws-bpel/`.
4. Leymann, F.: Web Services Flow Language (WSFL 1.0). IBM (2001)
5. Thatte, S.: XLANG — web services for business process design. Microsoft (2001) `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.
6. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Comm. ACM **15**(12) (1972) 1053–1058
7. Ossher, H., Tarr, P.: Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In: Proc. 21st Int'l Conf. Software Engineering, IEEE Computer Society Press (1999) 687–688
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC (1997)
9. De Win, B., Joosen, W., Piessens, F.: Developing secure applications through aspect-oriented programming. In Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005) 633–650
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: Proc. ECOOP 2001, LNCS 2072, Berlin, Springer-Verlag (2001) 327–353
11. Vanderperren, W., Suvée, D., Jonckers, V.: Combining AOSD and CBSD in PacoSuite through invasive composition adapters and JAsCo. In: Proceedings of Net.ObjectDays 2003, Erfurt, Germany (2003) 36–50
12. D'Hondt, M., Jonckers, V.: Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In Lieberherr, K., ed.: Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004), ACM Press (2004) 132–140
13. Arsanjani, A., Hailpern, B., Martin, J., Tarr, P.: Web services: Promises and compromises. Queue **1**(1) (2003) 48–58
14. Charfi, A., Mezini, M.: Aspect-oriented web service composition with AO4BPEL. In Zhang, L.J., ed.: Proceedings of the 2nd European Conference on Web Services (ECOWS 2004), Erfurt, Germany, Springer-Verlag (2004) 168–182
15. Cottenier, T., Elrad, T.: Dynamic and decentralized service composition with Contextual Aspect-Sensitive Services. In: Proceedings of the 1st International Conference on Web Information Systems and Technologies (WEBIST 2005), Miami, FL, USA (2005)

16. Verheecke, B., Vanderperren, W., Jonckers, V.: Unraveling crosscutting concerns in web services middleware. IEEE Software **23**(1) (2006) 42–50
17. Brichau, J., Haupt, M.: Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe (2005)
18. De Volder, K.: Aspect-oriented logic meta programming. In Lopes, C., Kiczales, G., Tekinerdoğan, B., De Meuter, W., Meijers, M., eds.: Workshop on Aspect Oriented Programming (ECOOP 1998). (1998)
19. De Volder, K.: Type-Oriented Logic Meta Programming. PhD thesis, Vrije Universiteit Brussel (1998)
20. Deransart, P., Ed-Dbali, A., Cervoni, L., eds.: Prolog: The Standard Reference Manual. Springer-Verlag (1996)
21. Courbis, C., Finkelstein, A.: Towards aspect weaving applications. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, ACM Press (2005) 69–77
22. Cibrán, M.A., Verheecke, B., Jonckers, V.: Aspect-oriented programming for dynamic web service monitoring and selection. In Zhang, L.J., ed.: Proceedings of the 2nd European Conference on Web Services (ECOWS 2004), Erfurt, Germany, Springer-Verlag (2004)
23. Suvée, D., Vanderperren, W.: JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit, M., ed.: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003), ACM Press (2003) 21–29
24. Fallside, D.C., Walmsley, P.: XML Schema part 0: Primer second edition. W3C Recommendation 28 October 2004, World Wide Web Consortium (2004) `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`.
25. White, S.A.: Business Process Modeling Notation (BPMN) version 1.0 (2004) `http://www.bpmn.org/`.