BPELJ: BPEL for Java

A Joint White Paper by BEA and IBM

March 2004

Authors

Michael Blow (BEA)
Yaron Goland (BEA)
Matthias Kloppmann (IBM)
Frank Leymann (IBM)
Gerhard Pfau (IBM)
Dieter Roller (IBM)
Michael Rowley (BEA)

Copyright Notice

.

Table of Contents

Introduction

The Web Services Business Process Execution Language (BPEL) is a programming language for specifying business processes that involve Web services. BPEL is especially good at supporting long-running conversations with business partners. Even before the standard is formally released it is becoming clear that BPEL will be the most widely adopted standard for business processes involving Web services.

BPEL is geared towards "programming in the large", which supports the logic of business processes. These business processes are self-contained applications that use Web services as activities that implement business functions. BPEL does not try to be a general-purpose programming language. Instead, it is assumed that BPEL will be combined with other languages, which are used to implement business functions ("programming in the small").

This white paper proposes a combination of BPEL with Java, named **BPELJ**, that allows these two programming languages to be used together to build complete business process applications. By enabling BPEL and Java to work together, BPELJ allows each language to do what it does best.

The following is a table that lists the kinds of tasks that are best suited to BPEL's strengths along side of a list of some of the tasks within a process that would be best accomplished with Java.

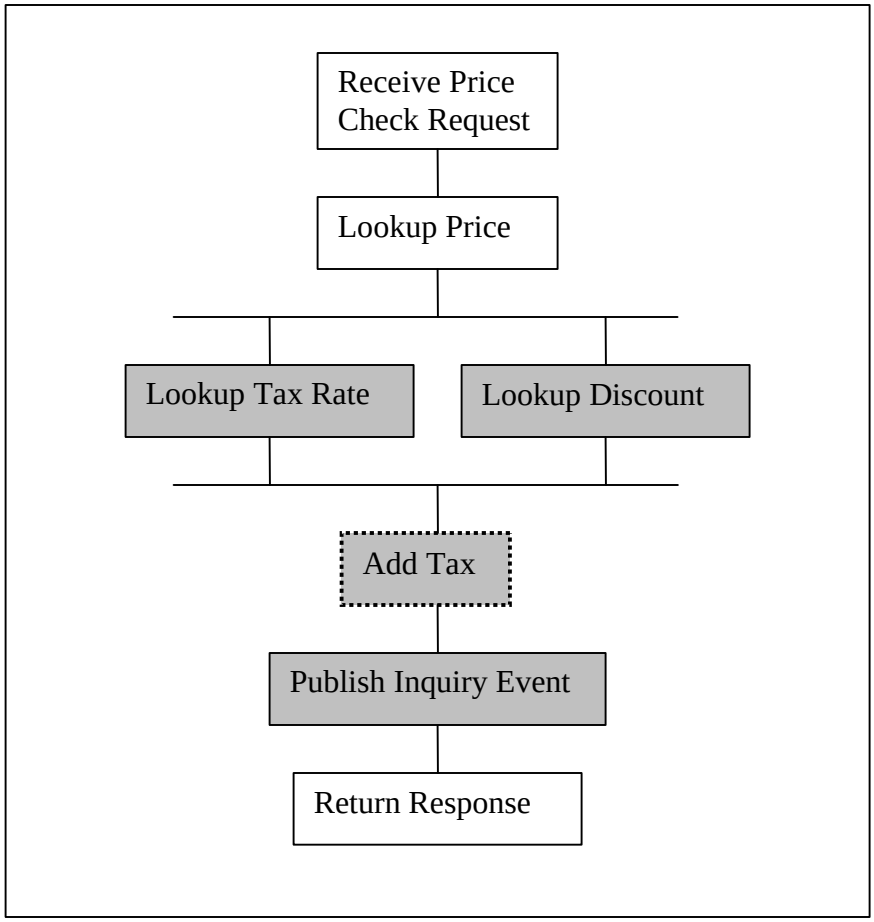
BPEL	Java
• Describe the logic of business processes • Maintain multiple long-running units of execution that are also interruptible • Selectively compensate completed activities of long-running units of execution in case of failures • Resume work where left because of failures to minimize work to be redone • Route incoming messages to the right place within the right process • Accept one of a variety of possible expected incoming message types. • Define a set of activities that should occur at a designated time and in pre-defined order. • Send messages to Web services.	• Calculate a value to be inserted into a document. • Construct a document to be sent to a web service using information from other documents and variables. • Deconstruct a document that has arrived – finding important values, converting them and then inserting them into other documents. • Calculate a value that will be used to affect the flow of control within the business process (e.g. loops and branches) • Perform side-effects without having to create Web services

BPELJ enables Java and BPEL to cooperate by allowing sections of Java code, called Java *snippets,* to be included in BPEL process definitions. Snippets are expressions or small blocks of Java code that can be used for things such as: loop conditions, branching conditions, variable initialization, Web service message preparation, logic of business functions etc. BPELJ

introduces a few minor changes to BPEL as well as several extensions in order to fit BPEL and Java conveniently together. (The changes to BPEL are listed in the appendix.) However, if any of these changes are not accepted, BPELJ will use existing features of BPEL, with a somewhat more awkward result.

BPELJ extensions are introduced via extension points defined in the BPEL standard to provide the new functionality. A BPELJ process will execute on any platform that supports the BPELJ extensions to BPEL. Note especially, that BPELJ does *not* include any extensions that are required for all BPELJ processes, so any BPEL process is a valid, executable BPELJ process.

In addition to making it possible to use Java to do the computational work of a business process, BPELJ also makes it possible to use BPEL to orchestrate long-running interactions with J2EE components. There is a lot of business logic that is currently deployed in Java components and BPELJ makes it possible to create business processes that include these components as well as Web services within the same business process. So, for example, consider the simple business process picture below. In this process, the white boxes are for web service invocations and the shaded boxes are invocations of Java methods. The box with dashed lines represents an activity that contains a snippet of embedded Java code. This example process will be explained in more detail below.



BPEL 1.1 introduced the concept of a *partner link type* that can be added to a WSDL file to declare the interfaces that two collaborating business partners can expect from each other. These partner link types make use of WSDL port types as the means of specifying interfaces. A BPEL process declares the long running conversations that it expects to be involved in by defining *partner links* that use these types. In order to allow processes to use Java resources in addition to Web services, BPELJ makes it possible to create partner link types whose interfaces are defined using Java interfaces rather than WSDL port types. Partner links within a BPELJ process that make use of these new partner link types are referred to as *Java partner links.*

One of the primary differences between a Java interface and a Web service port type is that the methods of a Java interface can accept arbitrary Java objects as parameters, whereas a Web service is defined to accept XML. In order to make effective use of Java partner links, a process must be able to prepare Java objects that can be passed as input parameters and to be able to accept Java objects as return values. BPELJ accomplishes this by allowing process variables to be declared using Java interface types, in addition to the XML Schema types and WSDL message types that BPEL uses for variables. A snippet within a BPELJ process can then create a Java instance, assign it to a BPELJ variable, call a number of its methods to properly prepare it, and then send the object as a parameter to a method provided by a Java partner link. The object

that is sent as the return value of the method can then be stored in another variable, prior to its use elsewhere in the process.

A BPELJ process definition may be a stand-alone document or it may be embedded within a Java file as an annotation of a Java class. Annotations of classes appear within block comments above the class declaration, although in J2SE 1.5 and future versions of Java, it will be possible to specify such annotations with an explicit syntax that does not require the annotation to be in a comment. When a BPELJ process is defined in the annotation of a class the class is referred to as a *process class*. There are no restrictions on the interface of the process class and there are also no resulting restrictions on the BPELJ process definition. Snippets in such a BPELJ process have access to the package visible fields and methods of the process class for any variable defined to be of that class.

Example

The example process pictured in the introduction is a process to provide product price checks. There are no branches or loops in the process, but the process does make use of a mix of Web service partner links, Java partner links, in-lined Java snippets. Here is what each step does:

1. **Receive Price Check Request**: receives the Web service request that initiates the process.
2. **Lookup Price**: looks up the price from a backend Web service.
3. **Lookup Tax Rate and Lookup Discount**: Two parallel activities – one that looks up the tax rate for the customer's state by calling a method on an EJB and the other that looks up the customer's discount. The state of the customer is accessed using XPath.
4. **Add Tax**: is an activity that contains in-line Java code (referred to as a *snippet activity*). It multiplies the subtotal by the tax rate and sets the total in the response.
5. **Publish Inquiry Event**: publishes the response that will be sent to the customer (which includes information about the request) to a JMS topic whose purpose is to publicize the fact that a customer is looking up the price of a product, for any person or system that might care to track such inquiries.
6. **Return Response**: sends the response document to the customer via a Web service call.

The BPELJ for this example looks like the following:

```
<process name="PriceQuote" expressionLanguage="http://jcp.org/java"
  bpelj:package="com.mycom.processes"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  targetNamespace="http://mycom.com/bp/PriceQuote"
  xmlns:bpelj="http://schemas.xmlsoap.org/ws/2003/03/business-process/java"
  xmlns:pq="http://mycom.com/ws/external/PriceQuote">

  <partnerLinks>
    <partnerLink name="client" partnerLinkType="pq:priceQuoteLT" myRole="provider"
                 partnerRole="requestor"/>
    <partnerLink name="backend" partnerLinkType="pq:priceQuoteLT" myRole="requestor"
                 partnerRole="provider"/>
    <partnerLink name="rateLookup" partnerLinkType="bpelj:com.mycom.ejb.RateLookup"/>
    <partnerLink name="inquiryTopic"
                 partnerLinkType="bpelj:javax.jms.TopicPublisher"/>
  </partnerLinks>

  <variables>
    <variable name="request" type="pq:priceRequest"/>
```

```
  <variable name="response" type="pq:priceResponse"/>
  <variable name="taxRate" type="xsd:float"/>
  <variable name="discount" type="xsd:float"/>
  <variable name="jmsMessage" type="bpelj:javax.jms.TextMessage"/>
</variables>

<correlationSets>
  <correlationSet name="client"
                  algorithm="http://somestandards.org/astandard/mechanism"/>
</correlationSets>

<sequence>
  <receive name="Receive Price Check Request" partnerLink="client"
           operation="requestPriceASync" createInstance="true" >
    <correlations>
      <correlation set="client" initiate="yes"/>
    </correlations>
    <output part="request" variable="request"/>
  </receive>
  <invoke name="Lookup Price" partnerLink="backend" operation="requestPriceSync">
    <input part="request" variable="request"/>
    <output part="response" variable="response"/>
  </invoke>
  <flow>
    <invoke name="Lookup Tax Rate" partnerLink="taxLookup"
            operation="lookupTaxRate">
      <input part="request" variable="request">
        <!—- get the state from "request" and pass just that. -->
         <query expressionLanguage="http://w3c.org/xpath1.0">
           //buyer/address/state
         </query>
      </input>
      <output variable="taxRate"/>
    </invoke>
    <invoke name="Lookup Discount Rate" partnerLink="rateLookup"
            operation="lookupDiscount">
      <input part="request" variable="request"/>
      <output part="response" variable="discount"/>
    </invoke>
  </flow>
  <bpelj:snippet name="Calculate Total">
    <bpelj:code>
      response.setTaxRate(taxRate);
      response.setDiscount(discount.getRate());
      float subtotal = response.getSubtotal();
      subtotal = subtotal * (1 – discount.getRate());
      response.setSubtotal(subtotal);
      float taxes = subtotal * taxRate;
      float total = subtotal + taxes;
      response.setTax(taxes);
      response.setTotal(total);
      // Prepare the text message to be sent in the next activity.
      jmsMessage = p_inquiryTopic.getSession().createTextMessage(response);
    </bpelj:code>
  </bpelj:snippet>
  <invoke name="Publish Inquiry Event" partnerLink="inquiryTopic"
          operation="publish">
    <input part="message" variable="jmsMessage"/>
  </invoke>
  <invoke name="Return Response" partnerLink="client" operation="returnResponse">
    <correlations>
      <correlation set="client" initiate="no"/>
    </correlations>
```

```
      <input part="response" variable="response"/>
    </invoke>
  </sequence>
</process>
```

The top of this business process starts with two attributes of note:

1. `expressionLanguage="http://jcp.org/java"`, which says that any expressions that don't say otherwise are written in Java.
2. `bpelj:package="com.mycom.processes"`, which says that any embedded Java code should be considered to be in the `com.mycom.processes` package.

The <`partnerLinks`> section includes two partner links ("`rateLookup`" and "`inquiryTopic`") that are for interacting with Java resources. The `rateLookup` partner link is for an EJB, while the `inquiryTopic` is a JMS topic. Both of these partner links only specify a single role (since there are no asynchronous callbacks), so they can be specified by only naming the Java interface of the one role.

The last variable in the <`variables`> section is named "`jmsMessage`". Its type is a standard JMS TextMessage object, which is specified by using a QName with a prefix that identifies the `bpelj`: namespace, and a local part that is specifies a class ("`bpelj:javax.jms.TextMessage`").

All of the invoke activities use a proposed new BPEL syntax that allows for the assignment of input message parts and output message parts using <`input`> and <`output`> child elements of invoke. This is more convenient than creating and populating temporary variables that hold WSDL message types, and also makes it possible to use the same syntax to pass multiple parameters to a Java method when the invoke is for a Java partner link.

The "Lookup Tax Rate" invoke activity is the first activity that calls a Java method. In this case, the <`input`> element uses an XPath expression to lookup a string representing the state of the customer, which it passes in to the `lookupTaxRate` method of the `rateLookup` EJB.

Following the flow there is a <`bpelj:snippet`> activity. This is a BPELJ specific activity that is composed of in-line Java code. The code can access all of the BPEL variables that are in scope at the location of the activity. BPEL variables that are defined with XML types are mapped to Java for the sake of the snippet code using SDO (although the XML binding technology can be changed by using an `xmlBinding` attribute).

The snippet activity ends with a line that sets the value of the `jmsMessage` BPEL variable by using a simple Java assignment statement. That variable is then used by the last invoke activity to publish the message to a JMS topic.

These are a few of the things that can be done with BPELJ. The rest of the paper will describe all of these, as well as other constructs, in more detail.

Snippets

Java snippets are Java expressions or statement blocks that are embedded in a BPEL process definition. Snippet code may assume that it is running in a J2EE container. Snippets may be used:

- as the body of snippet activities;
- for variable initialization;
- anywhere BPEL expressions may be used (the following is from the BPEL spec):
  - Boolean-valued expressions (transition conditions, join conditions, while condition, and switch cases)
  - Deadline-valued expressions ("until" attribute of onAlarm and wait)
  - Duration-valued expressions ("for" attribute of onAlarm and wait)
  - General expressions (assignment)

Snippets have access to all variables that are in scope at the location of the snippet. The variables are represented as regular Java variables. Changing such a variable within a snippet causes the variable to be changed in the process once the snippet is complete.

Snippets also have access to partner links and correlation sets and the status of incoming links as if they were local variables. Each of these constructs will be covered in later sections.

Here is an example snippet that is simpler than the example in the introduction. It simply reads two BPEL variables (`taxRate` and `subtotal`) and then sets a variable called `salesTax`.

```
<variables>
   <variable name="taxRate" type="xsd:float">
   <variable name="subtotal" type="xsd:float">
   <variable name="salesTax" type="xsd:float">
</variables>
...
<bpelj:snippet>
  <bpelj:code>
    salesTax = subtotal * taxRate;
  </bpelj:code>
<bpelj:snippet>
```

Because the variables are represented as local variables, there is no problem if there are BPEL variables of the same name in different scopes. Variable names are disambiguated by applying BPEL's scoping rules at the location of the snippet. The types of the variables will be based on the XML binding technology that is in scope, which is discussed more in the section below titled "Binding XML Variables to Java". The example above assumes that simple types from XML Schema are mapped to Java types according to the rules defined by SDO.

In the example above the Java code is inside of a new kind of BPEL activity called a *snippet activity*. Since `<bpelj:snippet>` is a BPEL activity, in addition to having a `<bpelj:code>` child element, it can have other standard BPEL standard elements, such as `<target>` element or a `joinCondition` attribute.

A *snippet activity* contains a snippet that is a statement block that does not return a value. Snippet activities are generally used to perform simple transformations and mappings. More advanced functions are usually implemented as separate services and accessed via BPEL Partner Links or Java Partner Links, as discussed later in this document. However, there are many places within BPELJ where snippets may be used where they must return a value. One such place is variable initialization. A different process that needs to compute sales tax could possibly do it right in the initialization of the `salesTax` variable. If you assume that the `subtotal` and `taxRate` variables are defined in an outer scope, such an initialization snippet might look like this.

```
<variables>
   <variable name="salesTax" type="xsd:float">
     <bpelj:value>
       subtotal * taxRate
     </bpelj:value>
   </variable>
</variables>
```

In the example above the value was defined with an expression. Snippets that return values are written as Java expressions, by default. However, if statements are required, the snippet may be defined with a `style="block"` attribute, in which case the snippet should contain a statement block that returns a value. Although it isn't necessary for the above example, if it were written out using the block style, it would look like this:

```
<variables>
   <variable name="salesTax" type="xsd:float">
     <bpelj:value style="block">
       float tax = subtotal * taxRate;
       return tax;
     </bpelj:value>
   </variable>
</variables>
```

Each Java snippet is conceptually mapped to a separately generated method. This means that variables that are defined in one snippet cannot be seen by another snippet. It also means that a `return` statement within a snippet only causes the snippet to return. It does not cause any change to the control flow of the process. This is a different model from that used by the Java Server Pages specification (JSP).

A BPELJ `<process>` element may include a `bpelj:package` attribute that contains a Java package name. If it does, the snippets of the process are considered to be in the Java package identified by the attribute, so they may access the package visible Java constructs from that package.

Binding XML Variables to Java

XML variables are mapped to Java using according to the XML-to-Java binding technology specified with the `bpelj:xmlBinding` attribute on the variable declaration; or, to affect all variables in the process, the attribute may be specified on the process element itself. Here is an example that shows a process that sets the process-wide default XML binding to DOM Level 3,

and then uses that default for the `justificationDoc` variable, but overrides that XML binding to be JAXB 2.0 for the `po` variable.

```
<process name="purchaseOrderProcess" bpelj:xmlBinding="bpelj:DOM3" ...>
   <variables>
     <variable name="justificationDoc" type="lns:justificationDocument"/>
     <variable name="po" type="lns:POMsg" bpelj:xmlBinding="bpelj:JAXB20"/>
   </variables>
   <sequence>
     ...
     <bpelj:snippet>
       <bpelj:code>
         // Get the approver using JAXB accessor
         Approver approver = po.getApprover();
        // Get the approver's comments in the justification doc using DOM.
         NodeList commentNodeList =
             justificationDoc.getElementsByTagName("approverComment");
         ...
       </bpelj:code>
     </bpelj:snippet>
   </sequence>
 </process>
```

Variables with Java Types

BPEL currently allows variables to be defined using WSDL message types or XML schema element definitions or XML schema simple types. BPELJ adds to this the ability to specify variables whose types are defined using Java classes or interfaces. Java simple types are not added, since they would duplicate the functionality already provided by simple XML schema types. Variables that hold arbitrary Java types can only be used by Java snippets or for calls to Java partner links. They cannot be used by non-Java aware aspects of the process. So, for example, they cannot be assigned to WSDL message parts. However snippets may use data from Java-typed variables to set the contents of XML-typed variables.

BPELJ also makes it possible to initialize any variable using a Java snippet. Variables of Java types that are not explicitly initialized are implicitly initialized to null. Initialization of a variable occurs when the scope containing the variable declaration is first entered. In the case of process-level variables initialization happens when the process first starts. When multiple variables are defined for the same scope, the variables are initialized in the order they are defined. The initialization code for variables may access variables in enclosing scopes as well as previously declared variables of the same scope.

The following example shows three variables defined. The `customer` variable is initialized to null. The `orderStatusTopic` is initialized with a snippet that calls a method on a `jndiContext` variable from an outer scope. The `shipper` variable is then initialized with a snippet that creates a new object instance, using the `orderStatusTopic` variable as well as a `shipperID` variable that was also from an outer scope.

```
<variables>
  <variable name="customer" type="java:com.mycom.Customer"/>
```

```
  <variable name="orderStatusTopic" type="java:javax.jms.Topic">
    <bpelj:value>
      (Topic)jndiContext.lookup("orderStatusTopic");
    </bpelj:value>
  </variable>
  <variable name="shipper" type="java:com.mycom.Shipper ">
    <bpelj:value>
      new ShipperImpl(shipperID, orderStatusTopic)
    </bpelj:value>
  </variable>
<variables>
```

The only requirement on the Java types that may be used for Java variables is that they must implement the Serializable interface. The objects that are contained within Java variables are persisted with the business process, using the object's serialization as its persistent representation. When a snippet calls a method on an object or passes an object to another method the process must persist any changes that are made to the object.

Conditions

One of the most common uses for snippets is for specifying Boolean conditions for BPEL constructs that require them, such as: while conditions, switch case conditions, transition conditions and join conditions. In the following example, a list of items is assumed to have been returned from some Java component as a `java.util.Collection`. If the component that returned the list of items were a web service, it would probably return an XML document, but when Java components are used it is more likely that the list is returned as a Java collection. The while loop that follows uses an iterator to access all of the `Item` objects in the collection, so the loop makes use of the `iter.hasNext()` method as its loop condition.

```
<process ... expressionLanguage="http://jcp.org/java">
  <variables>
    <variable name="iter" type="java:java.util.Iterator">
      <bpelj:value>items.getIterator()</bpelj:value>
    </variable>
    <variable name="item" type="java:com.acme.Item"/>
  </variables>
  <while>
    <condition>iter.hasNext()</condition>
    <sequence>
      <bpelj:snippet>
        <bpelj:code>item = (Item)iter.next();</bpelj:code>
      </bpelj:snippet>
      ...
```

In this example the `<condition>` element does not need to be specially marked as containing a Java snippet. This is because the `expressionLanguage` attribute of the process specifies that all expressions will use Java, unless they specify otherwise. If the expressionLanguage in the process had been something other than Java, then it would have been possible to put the `expressionLanguage=" http://jcp.org/java"` attribute on the condition element itself.

Join Conditions

Join conditions may also be specified with snippets. Such conditions also have access to the link status of incoming links. The status of each incoming link is represented as a local Boolean variable whose name is the name of the link. The following is part of the example that is used in the BPEL specification to introduce join conditions. Since this is in the BPEL specification, it uses XPath 1.0 to represent the join condition.

```
  <invoke name="settleTrade"
    <condition>bpws:getLinkStatus('buyToSettle') and
               bpws:getLinkStatus('sellToSettle')</condition>
    <target linkName="buyToSettle"/>
    <target linkName="sellToSettle"/>
    <source linkName="toBuyConfirm"/>
    <source linkName="toSellConfirm"/>
  </invoke>
```

The following is an example of the same join condition using a Java snippet to specify the join condition.

```
  <invoke name="settleTrade">
    <joinCondition>buyToSettle && sellToSettle</joinCondition>
    <target linkName="buyToSettle"/>
    <target linkName="sellToSettle"/>
    <source linkName="toBuyConfirm"/>
    <source linkName="toSellConfirm"/>
  </invoke>
```

BPEL only allows the XPath expressions that are within join conditions to access link status. The XPath cannot access BPEL variables or any other aspect of the process state. The same restriction applies to snippets used within join conditions. The snippets expressions can only access the Boolean parameters that represent the status of incoming links.

Java Partner Links

BPELJ makes it possible to define partner link types that use Java interfaces instead of WSDL port types as the means of specifying the types of the partners. As with regular partner links, Java partner links may be used by the process both to invoke the operations of a partner as well as to receive operation requests from partners. Java partner links make it possible for a BPEL process to use Java components in addition to Web services, along with the ability to pass any serializable Java object as an operation parameter or return value.

Specifying a Java partner link type is done in a WSDL file, as with other partner link types, but the roles are defined using fully qualified class names to define their types. Here is an example of such a partner link definition.

PortType Definition in WSDL

```
<plnk:partnerLinkType name="PurchasingLT">
  <plnk:role name="seller">
```

```
    <plink:portType name="bpelj:com.them.Seller"/>
  </plnk:role>
  <plnk:role name="buyer">
    <plink:portType name="bpelj:com.us.Buyer"/>
  </plnk:role>
</plnk:partnerLinkType>
```

The declaration of a partner link that uses a Java partner link type cannot be distinguished from a regular partner link type. So it might look like this.

```
<partnerLinks>
   <partnerLink name="purchasing"
        partnerLinkType="lns:PurchasingLT"
        myRole="buyer"
        partnerRole="seller"/>
</partnerLinks>
```

Web service partner links can be defined with only one role when there is no need for asynchronous messages to be sent to the initiating partner. The same is true for Java partner links. In this case, BPELJ allows the `partnerLink` to specify its `partnerLinkType` with a reference to the Java interface of the one role (this is the style used in the introductory example).

```
<partnerLinks>
   <partnerLink name="purchasing" partnerLinkType="bpelj:com.them.Seller"/>
</partnerLinks>
```

The object that implements the interface specified in a Java partner link may be in the same VM as the business process or it may be remote (an EJB, for example). If it is remote, parameters are passed using RMI calling semantics (Java serialization, by value). Otherwise, parameters are passed using Java method invocation semantics (by reference).

An `<invoke>` that uses a Java partner link passes parameters with child `<input>` elements. The input element is a proposed extension to BPEL itself, as its value is not limited to BPELJ. The following Web service uses an invoke call with `<input>` elements to pass two variables, one for each part of the input message:

```
  <invoke partnerLink="Purchasing" operation="initiatePurchase">
    <input part="Requisition" variable="req"/>
    <input part="PO" variable="po"/>
  </invoke>
```

This syntax has the advantage that it can be used, without modification, to invoke Java methods through Java partner links. While the use of `<input>` elements is merely a convenience for Web service invocations (since it saves the step of creating a message variable), it is necessary for Java invocations. Java partner links are defined in terms of Java interfaces, so there are no WSDL message types defined for them. Consequently, it is not possible for a Java `<invoke>` to use a single BPEL `inputVariable` attribute to refer to a variable that contains all of the parameters.

The child <input> elements for a Java invoke must be listed in the same order as the parameters of the method being called. The part names, if they are specified, must be the same as the names of the parameters in the method definition. An input element may specify a query to be run against the variable before passing it as a parameter by specifying the query in a <query> child element. It is also possible to pass a role of a partner link by specifying both a partnerLink and a role attribute, instead of a variable attribute.

Using Partner Links from Snippets

In addition to BPEL variables, snippets may also reference partner links as if they were local variables to the snippets whose name is the name of the partner link prefixed by "p_". The name has to be prefixed because partner links can have the same names as variables. A partner link is an object with accessors for the link's partnerRole and myRole, where only the partner role may be changed. If prefixing the partner link name with "p_" causes it to collide with another name that is in scope, then the partner link is not available to the snippet. Here is the API for a partner link:

```
public interface PartnerLink<PartnerType, MyType>
{
   MyType getMyRole();
   PartnerType getPartnerRole();
   void setPartnerRole(PartnerType partner);
}
```

The interface is written using J2SE 1.5 generics because it helps document the intent of the API. The MyType generic parameter is the interface that is defined for myRole of this partner link. Similarly, the PartnerType generic parameter is the interface defined for the partner role. For WSDL-based partner links, both of these generic parameters are a mapping of the EndPointReference XML element to Java using the xmlBinding of the process. This API in J2SE 1.4 and earlier would just use Object in the places where the generic parameters are used.

The following is an example that compares an invoke on a Java partner link as it is accomplished with an <invoke> activity with the same invoke performed within a snippet.

Invoking from an <invoke> activity

```
  <invoke partnerLink="purchasing" portType=" "
          operation="initiatePurchase">
    <input part="Requisition" variable="req"/>
    <input part="PO" variable="po"/>
  </invoke>
```

Invoking from a snippet

```
  <bpelj:snippet>
    <bpelj:code>
      p_purchasing.getPartnerRole().initiatePurchase(req, po);
    </bpelj:code>
  <bpelj:snippet>
```

Both forms are useful. The snippet form will typically be used from within hand-generated code, where the call itself is likely to be in the midst of other code. The `<invoke>` activity form is more likely to be used by tools, since its use is much more constrained and thus more amenable to being mapped to GUI constructs.

The partner role of a partner link may be assigned either by configuring the BPELJ process, or may be set at runtime using either an `<assign>` activity, or may be set from within a snippet. Assigning the partner role from within a snippet can be accomplished by calling the `setPartnerRole()` method of the partner link variable. Note that, unlike variables, simply assigning an object to a partner link variable is not allowed.

Variables that hold Java objects have methods that can be called from within snippets and Java partner link variables have methods that can also be called, but the two constructs are different. Here is a list of the differences:

- It is possible to receive a message from a Java partner link, but not from a Java variable.
- Java partner links can be assigned through external configuration, whereas Java variables must be assigned by the process.
- Java partner links have an externally managed lifecycle. In other words, it is not possible for a process to create the object that implements a Java partner link, but it is possible for it to create objects that are stored in Java variables.

BPEL Correlations with Java

BPEL provides a correlation set mechanism that can be used for routing messages to the right location within a process. Invokes through Java partner links can almost use the BPEL mechanism without modification. However, BPEL's correlation sets make use of property alias definitions, where the property alias is defined on a message type. Java partner links do not have message types, so there needs to be some other way of specifying the property aliases.

The solution is to allow property aliases to be defined for XML Schema types and Java interface types in addition to message types. Properties on Java types can be defined either with a snippet, or with XPath. XPath can be defined to run on a Java object by assuming that each Java object represents an XML node with a child element for each JavaBean-style accessor. Below are three property alias definitions, all three of which are based on a PurchaseOrder.

Property Alias on an XML Schema type using XPath

```
<bpelj:propertyAlias propertyName="POID" type="lns:PurchaseOrder"/>
  <bpelj:query>//POID</bpelj:query>
</bpelj:propertyAlias>
```

Property Alias on a Java class using XPath

```
<bpelj:propertyAlias propertyName="POID" type="bpelj:com.me.PurchaseOrder"/>
  <bpelj:query>//POID</bpelj:query>
</bpelj:propertyAlias>
```

Property Alias on a Java class using a snippet

```
<bpelj:propertyAlias propertyName="POID" type="bpelj:com.me.PurchaseOrder"/>
  <bpelj:extractValue arg="po">
     po.getPOID()
  </bpelj:extractValue>
</bpelj:propertyAlias>
```

Because none of these property aliases is defined on a message type, the use of the property aliases has to differ from standard BPEL as well, since the `parts` must be specified at the place where the correlation set is referenced. Here is an example that defines a correlation set that uses the POID property and then uses it for an invoke.

```
  <correlationSets>
    <correlationSet name="PurchaseOrder" properties="POID"/>
  </correlationSets>

  <invoke partnerLink="purchasing" portType="java:com.mycom.Buyer"
          operation="initiatePurchase">
    <input part="poPart" variable="po"/>
    <correlations>
      <correlation set="PurchaseOrder" initiate="yes" pattern="out"
                   parts="poPart">
    </correlations>
  </invoke>
```

The only thing that is not standard BPEL in the above example is the part attribute specified in the correlation element. BPEL doesn't need it because the parts are specified in the propertyAlias definition.

In the future, we expect that BPEL will introduce the concept of *opaque correlations* whose values are chosen by the execution framework. If the above example used such an opaque correlation, then the only difference would be that the `correlationSet` definition would have an `opaque="true"` attribute rather than a `properties` attribute. This way no property or property aliases would need to be defined.

Snippet Access to Correlation Sets

As with partner links, snippets may access correlation sets as if they were local variables whose name is the name of the correlation set, prefixed by "c_". Object referenced by the correlation set variable would be have the following interface.

```
public interface CorrelationSet
{
    public void initialize(java.util.Map propertyValues);
    public Object getProperty(String propertyName);
}
```

The initialize method can only be called on correlation sets that have not already been initialized (either by BPEL or by Java). The correlation set variables cannot be the target of an assignment, nor can the properties of a correlation set be changed from a snippet. The object returned by the

getProperty call is the Java object for the property's XML value, according to the XML binding that is in scope.

Callback Objects as an Alternative to Correlation Sets

Instead of using correlation sets, Java partners can communicate back to the process by taking advantage of the fact that the `myRole` attribute of a Java partner link holds an object that can be used as a callback object. The `myRole` object has the following properties:

- It implements the myRole interface for that partner link
- It can be passed remotely
- Calls on it are guaranteed to be routed to the right BPELJ process instance

We expect that this should be the most efficient way to send a message to a process from a Java partner. A callback object cannot, however, be used to initiate conversations. It also cannot route to a specific internal scope of a process, since its scope is the same as the scope of the partner link.

This following example shows an invoke to a Java partner. It passes the `myRole` endpoint of the Buyer partner link as the first parameter, which is then used by the Java partner to send back subsequent messages.

```
<invoke partnerLink="purchasing" operation="initiatePurchase">
  <input part="callback" partnerLink="purchasing" role="buyer"/>
  <input part="poPart" variable="po"/>
</invoke>
```

This example implements the initiatePurchase operation and sends back an asynchronous PurchaseOrderResponse.

```
package com.seller;
public class SellerImpl implements Seller
{
    public void initiatePurchase(com.buyercom.Buyer buyerCallback,
                                 PurchaseOrder po)
    {
        POResponse response = processPurchaseOrder(po);
        buyerCallback.acceptPOResponse(response);
    }
}
```

Here is the BPELJ that receives the response.

```
  <receive partnerLink="purchasing" portType="java:com.buyercom.Buyer"
           operation="acceptPOResponse">
    <output part="poResponse" variable =”poResp”/>
  </receive>
```

BPELJ as Java Class Annotation

Snippets work best only for small amounts of code, where each code snippet only needs to be used in a single place in the process. However, sometimes processes need to have somewhat

more complex Java code associated with them or there is Java code that needs to be used in more than one place in a process.

In both of these cases, the right solution is to introduce new methods. One possible solution to this would be to create a new class for such methods that is completely independent of the BPELJ process. However, code in the snippets and code in the external methods may be intended to be understood together, so introducing a totally separate class does not recognize this close association. If the BPELJ is defined in the annotation of a class (referred to as a *process class*), then the association is much more explicit.

A BPELJ process that is in the annotation of a process class would have the following effects on the BPELJ:
1) The name attribute of the BPELJ process could be left unspecified and the process would get the same name as the process class.
2) The Java import statements of the process class would also apply to the BPELJ, so they don't have to be repeated there.
3) The snippet code in the BPELJ would be considered to be in the same package as the process class, so if a variable is declared to be of the process class, the snippets could use package visible methods on that variable.

The following is a simple example that shows a BPELJ process as an annotation of a process class.

```
/**
* @bpelj:process process::
* <process name="purchaseOrderProcess" ...>
*   <variables>
*     <variable name="self" type="bpelj:PurchaseOrderProcess">
*        <bpelj:value>new PurchaseOrderProcess()</bpelj:value>
*     </variable>
*     <variable name="po" type="lns:POMessage"/>
*     <variable name="subtotal" type="xsd:int"/>
*   </variables>
*   <sequence>
*     ...
*     <bpelj:snippet>
*       <bpelj:code>
*         subtotal = self.computeSubtotal(po);
*       </bpelj:code>
*     </bpelj:snippet>
*   </sequence>
* </process>
**/
public class PurchasOrderProcess implements Serializable
{
  int computeSubtotal(PurchaseOrder po)
  {
     int subtotal;
     // code that computes the subtotal
     return subtotal;
  }
}
```

Java Deadline and Duration Expressions

For the same reasons that we expect that BPEL conditions syntax will be changed so that the condition is specified in an element rather than an attribute, we also expect that deadline and duration expressions will also be specified by an element.

Deadlines

When the expressionLanguage is Java, BPEL's <until> element should contain a snippet that returns an object of type java.util.Calendar. Here is an example.

```
<wait>
  <until>
    // uses a BPEL variable name "policy"
    new java.util.Calendar(policy.getDeadline())
  </until>
</wait>
```

Durations

When the expressionLanguage is Java, BPEL's <for> element should contain a snippet that returns String. The string should contain the lexical representation of the XML Schema *duration* type. The following example waits 12 hours.

```
<wait>
  <for>
    "PT12H"
  </for>
</wait>
```

As with other places where snippets can be used as expressions, the `<until>` and `<for>` elements may have a bpelj:`style="block"` attribute declaration, in which case the snippet should contain a statement block that ends with a return of the appropriate type, rather than an expression.

Faults and Exceptions

Code called from a Java snippet may generate a BPEL fault by throwing a javax.bpelj.BPELJException exception. The constructor of this exception requires the QName of the BPEL fault that should be thrown. The exception may also, optionally, contain an object to represent the fault data.

```
  <scope>
   <faultHandlers>
     <catch faultName="lns:cannotCompleteOrder" faultVariable="f">
       <!-- Do something else -->
     </catch>
   </faultHandlers>
   <bpelj:snippet>
      <bpelj:code>
```

```
        QName qname =
           new QName("http://manufacturing.org/wsdl/purchase",
                     "cannotCompleteOrder");
        // A generated class for a fault message document...
        FaultMessageDoc faultMsg =
             new FaultMessageDoc("We cannot complete your order");
        throw new javax.bpelj.BPELJException(qname, faultMsg);
     </bpelj:code>
   </bpelj:snippet>
 </scope>
```

However, BPELJ does not require that all Java faults be explicitly translated into a javax.bpelj.BPELJException. Doing so would require too much boilerplate code around every snippet. Instead, any Java exception that propagates into the BPELJ process other than the BPELJExceptions are mapped into a single designated fault whose name is `bpelj:fault`. The fault data for this fault is the original Java exception object.

BPEL matches catch statements to faults by matching both the faults QName and the type of the fault variable. BPELJ extends this in the natural way, to allow Java type matching semantics when matching fault variable types with fault data. The following example shows multiple catch clauses that differ only in the Java type of the fault variables. The first fault variable whose type can contain the type of the exception object that is the fault data is the catch clause that will be run. In this case, the `AccessControlException` extends `SecurityException`, so the first catch clause will run.

```
<scope>
 <variables>
    <variable name="securityException" type="java.lang.SecurityException"/>
    <variable name="otherException" type="java.lang.Throwable"/>
 </variables>
 <faultHandlers>
   <catch faultName="bpelj:fault" faultVariable="securityException">
     <!-- Handle the security exception -->
   </catch>
   <catch faultName="bpelj:fault" faultVariable="otherException">
     <!-- Handle any other kind of exception. -->
   </catch>
 </faultHandlers>
 <bpelj:snippet>
    <bpelj:code>
       throw new java.security.AccessControlException();
    </bpelj:code>
  </bpelj:snippet>
</scope>
```

Transactions

Customers use J2EE application servers for building robust applications. One aspect of robustness is the support of ACID transactions. Consequently, BPELJ extends BPEL by adding support of ACID transaction to business processes. For that purpose, scopes can be flagged with an attribute: `acid="true"` to designate the scope as requiring an ACID transaction. This type

of scope is called an *ACID scope*. An acid scope has all of the properties of a serializable scope, as defined in BPEL, but adds the constraint that the BPEL process state will not be persisted until the ACID scope ends. ACID scopes also allow external Java resources to be included in the transaction of the process.

As with serializable scopes, a scope that has been marked with acid='true' may have child scopes but those child scopes must not be marked with acid='true'. Child scopes of an ACID scope implicitly run within the same physical transaction than the scope declaring `acid='true'`

Receive, pick and wait activities are not, in general, allowed within an ACID scope. However, exactly one receive activity or exactly one pick activity can be used as the first activity of an ACID scope. In this case the ACID transaction starts when receiving the message expected by the receive or pick, or when the onAlarm handler of the pick activity is triggered.

Non-transactional invokes may be forced to be part of a transaction by including them in an ACID scope. However, there is no guarantee that the compensation handler for such an invoke will be called if the transaction aborts after the invoke has completed. It is recommended that invokes be used inside transactions only when they are known to be at least idempotent, since the likelihood that the invoke gets retried is high. To be safe, it should also be acceptable if the process that caused the invoke gets rolled back and never successfully completes. Thus the service may have been called one or more times by a process that does not (due to rollbacks) think that the service has ever been called.

In the future, we expect that when BPEL integrates the work of WS-AtomicTransaction it will be possible to get truly ACID transactions that include calls to web services.

Fault Handlers in ACID Scopes

The fault handler of an ACID scope is run within the same transaction as the scope. The default fault handler rolls back the ACID transaction, which triggers retry. A custom fault handler commits when leaving the fault handler, which is true even if the fault handler throws another fault.

In case of a rollback the transaction is retried. To avoid a retry loop an additional attribute `retryCount` on the scope is required, e.g., `retryCount=5`. When the scope has been retried more then `retryCount` times then a BPELJ fault `"acidScopeRetryCountExceeded"` is thrown. The `retryCount` attribute may also be specified on the `<process>` element, in which case it applies to all scopes that do not override it.

Conclusion

BPELJ makes use of BPEL's extensibility mechanisms in order to allow developers to create business processes that use a mix of Web services and Java resources. It also makes it possible to use Java as a language for calculations and data manipulation, so that a single file can contain complete and cohesive description of an entire business process.

Acknowledgements

The authors wish to acknowledge the contributions from the following people:
Don Ferguson (IBM), Dieter Koenig (IBM), Martin Nally (IBM), Rich Rollman (BEA), Doug Wilson (IBM).

Appendix 1: Changes to BPEL

1. BPEL currently does not allow new activity types to be added, so it is not currently possible to have a `bpelj:snippet` element that is treated like a BPEL activity (e.g. can be the target of links). If this cannot be changed, BPELJ could overload the BPEL `<empty>` activity, although such an approach is much less desirable.
2. BPEL does not currently allow `<invoke>` activities to specify input message parts with a `<input>` child elements, nor does it allow output parts to be handled with `<output>` child elements. This capability can also be accomplished by introducing a new scope that defines a temporary variable, assigns it, and then uses it. However, using such a pattern would be much more difficult to use and understand than having `<input>` and `<output>` elements.
3. This paper briefly refers to the concept of a *opaque correlation*, which we think should be added to BPEL. If it is not added we believe that any BPEL process would be more difficult to use, not just BPELJ processes (Issue 96).
4. The examples in this paper assume that the portType attribute of BPEL <invoke> is made optional. It is not really necessary, since it is implied by the partnerLink specified. By leaving it off, the fact that a partnerLink is a Java partnerLink is invisible at the <invoke> call.