IBM

Business Process with BPEL4WS: Learning BPEL4WS, Part 3

Activities and the in-memory model

Level: Introductory

Matthew Duftler (duftler@us.ibm.com), Software Engineer, IBM
Rania Khalaf (rkhalaf@us.ibm.com), Software Engineer, IBM

01 Oct 2002

The recently released Business Process Execution Language for Web Services (BPEL4WS) specification is positioned to become the Web services standard for composition. This series of articles aims to give readers an understanding of the different components of the language, and teach them how to create their own complete processes. The previous parts of the series gave an overview of the language, and took readers through creating their first simple process. This part will cover each of the activities in more detail. We will also cover how the various BPEL4WS constructs may be represented and manipulated in memory.

Introduction

Now that we've gone over the language basics in Parts 1 and 2 of this column, and we have created a simple example (see Resources), let's go over how to use each of the activities in more detail. In this article, we will provide detailed descriptions of each of the BPEL4WS activities. We will also describe the in-memory representation employed by BPWS4J (IBM's implementation, available on alphaWorks) to represent BPEL4WS processes, and give an example illustrating the model's use.

Basic activities

Basic activities are the simplest form of interaction with the outside world. They are non-sequenced and individual steps that interact with a service, manipulate the passing data, or handle exceptions.

Web services interactions

There are three activities a process can use for interacting with the outside world: `<invoke>`, `<reply>`, and `<receive>`. As we saw in the previous articles, the interactions occur with the partners of the process using these three activities. By specifying a portType, operation, and partner, each of these activities identifies the Web services call it belongs to.

The `<invoke>` activity is used by a process to make invocations to Web services provided by partners. In addition to the portType, partner, and operation, the invoke specifies input and output containers, for the input and output of the operation being invoked. An invocation can be either synchronous (request/response) or asynchronous (one-way). In the latter case, only an input container is required.

A business process provides services to its partners through a pair of `<receive>` and `<reply>` activities. The receive represents the input of a WSDL operation provided by the process. If the process needs to send back a reply to the partner who sent the message, then a reply activity is necessary. Multiple reply activities may be defined in the process to answer that partner's call; however, only one matching `<reply>` may become active at any one time. The matching of the appropriate reply activity is done at runtime, when the process looks for such an activity that is ready to run and has the same portType, operation, and partner as the `<receive>`. For example, the process may place the received message in a certain container, containerA, to be sent back if a certain condition is met, and the message from another container, containerB otherwise. In this case, the receive will have two links with the condition on them; these links will go to two reply activities, only one of which will activate and send the correct message back to the partner.

Process lifecycle
A business process instance may only be created if a message is sent to specially marked `<receive>` or `<pick>` activities.

The receive activities specify that they are able to create a process instance by setting the *createInstance* attribute to true. The `<pick>` activity will have an onMessage element with that same attribute set to true. `<pick>` is explained in more detail later on in this article. The first receive activity thus marked that gets a message will create the instance, and the rest will then just be treated as regular receive activities in that instance.

At this point you may wonder how you can create a second instance. Correlation is used to figure out which instance an incoming message is meant for. All receives and onMessages that can start a process must have the same correlation set. So when a message comes in, the process checks to see if it contains a correlation set that matches that of an existing instance. If it does it sends it there. Otherwise, it will create a new instance based on the receive activity that matches the operation, portType and partner information of the incoming message. Correlation in BPEL will be discussed in detail later in this series of articles.

Manipulating data: the <assign> activity

The `<assign>` activity can be used to copy data from one container to another, as well as to construct and insert new data using expressions. The use of expressions is primarily motivated by the need to perform simple computation (such as incrementing sequence numbers) that is required for describing business protocol behavior. Expressions operate on message selections, properties, and literal constants to produce a new value for a container property or selection.

Each `<assign>` activity contains one or more `<copy>` elements; each `<copy>` element contains exactly one `<from>` element and exactly one `<to>` element. There are various forms of the `<from>` and `<to>` elements, and the most important thing to remember is that the value to be copied from the source (the `<from>`) to the destination (the `<to>`) must be type-compatible.

One of the simpler forms of `<from>` simply specifies a container by name. When using a source that specifies just a container, an entire message will be copied; this means that the destination must also specify just a container.

A more complex form of `<from>` specifies a container and a part within that container. When using this form, the destination must also specify both a container and a part.

A third form of `<from>` specifies a general expression to be evaluated using XPath. This expression can be anything XPath will allow, provided it returns an XPath value type (string, number, or Boolean). When using this form, the destination must specify a container and a part.

In short, messages can overwrite other messages, and message parts can overwrite other message parts. Since expressions do not return entire messages, they can only be used to overwrite message parts.

Since it is necessary to enable XPath expressions to access information from the process, BPEL4WS introduces several XPath extension functions. The extension functions are defined in the standard BPEL4WS namespace, *http://schemas.xmlsoap.org /ws/2002/07/business-process/*, and the prefix *bpws* is associated with this namespace.

An example of one of these functions is shown below.

```
bpws:getContainerData("containerName", "partName", "locationPath"?)
```

In this example, where *containerName* is the name of a container, *partName* is the name of a part within that container, and *locationPath* is an optional absolute location path within the specified part.

The `bpws:getContainerData()` function can be used by any XPath expression wishing to retrieve data from a container within the process, as if it were a built-in XPath function.

Note: In addition to copying messages, parts, and expressions, the `<assign>` activity can be used to copy service references to and from partner links.

Other basic activities

Faults can be signaled in BPEL4WS by a `<throw>` activity. In order for you to be able to use fault handlers to eventually catch and handle that fault, the language requires that the fault have a globally unique QName. An optional container may be added to point to where data related to the fault may be found. For example, you may have a `<throw>` activity that signals a certain kind of fault, and then a fault handler that has a `<reply>` that sends a partner information about a fault. That reply would use the container specified in the fault activity.

The <terminate> activity can be used to immediately abandon all execution within the business process instance that executes the terminate activity.

The <wait> activity allows the process to wait for a specific time interval or until a certain deadline is reached.

The <empty> activity does nothing. It may be used if you need to catch and suppress a fault.

Structured activities

Structured activities prescribe the order in which a collection of activities take place. They describe how a business process is created by composing the basic activities it performs into structures that express the control patterns, data flow, handling of faults and external events, and coordination of message exchanges between process instances involved in a business protocol.

As described earlier in the article series, the <sequence> activity contains one or more activities that are executed sequentially. The activities are executed in the order in which they appear within the <sequence> element. When the final activity in the <sequence> has completed, the <sequence> activity itself is complete.

The <switch> activity functions much like the switch construct that occurs in many traditional programming languages. There is an ordered list of one or more conditional branches, defined by <case> elements, followed by an optional <otherwise> element. Each <case> branch specifies a Boolean XPath expression, and the expressions are evaluated in the order in which they appear (the conditional expressions are evaluated using much the same logic described for general expressions in the earlier section on <assign>). The first <case> element whose Boolean expression evaluates to true has its child activity performed. If no <case> element's condition holds true, the child activity of the <otherwise> element is performed. If no <otherwise> element is specified, there is an implied <otherwise> that contains an <empty> activity. When the selected branch completes, the <switch> activity is complete.

The <while> activity repeatedly performs its child activity until the specified Boolean condition no longer evaluates to true. The condition is evaluated as an XPath expression.

The <pick> activity contains a set of event handlers. Each handler contains one activity, which may run after the pick has started and the event that the handler is waiting for occurs. The handlers include alarm handlers which specify a duration or deadline, and message handlers (onMessage) which wait for messages from a particular partner, portType, and operation triplet. The message handlers are able to create a process instance in the same way as a receive, as described in the section about process lifecycle above (the createInstance attribute is used). Only the first event handler to receive its event will run, and the <pick> will complete once that handler's activity completes.

The <scope> activity provides fault and compensation handling capabilities to the activities nested within it. Scopes will be covered in more detail in the upcoming article on fault-handling and compensation.

The <flow> construct provides the ability to run activities in parallel, as well as to define guarded links. It may contain an arbitrary number of activities. When a flow is started, all the activities in it are ready to run unless they have incoming links that have not yet been evaluated. A flow defines a set of links whose source and target activities must be nested within it.

The links put their own constraints on how the activities of a process are set to run. Once an activity completes, it evaluates any conditions on the links leaving it. If no condition is defined and the activity has completed normally, then the conditions all evaluate to true. If the activity has faulted, or could not be run, then it sends all its links out false. On the other side, the activity that has links coming into it has to wait until it knows the value coming down all of its links before it can run. It also needs to have control from its enclosing activity. For example, if it is the second activity in a sequence and all of its links come in, it may not run if the first activity in that sequence has not yet completed. Once it has this control, and it knows the value of all its incoming links, it evaluates a condition known as the join condition. The join condition involves the state of the incoming links and must evaluate to true if the activity is to run. If it evaluates to false, then the activity signals a join failure and ends abnormally. The default join condition is true if the activity's implicit link (control from the parent) is true and any one of its explicit links is true. All the link conditions are XPath expressions.

BPWS4J model

Now that we've covered the language in a bit more detail, let's take a look at a mechanism for creating and representing in-memory representations of BPEL4WS processes. We call this set of APIs the BPWS4J model. The "model" includes a factory mechanism, and all the interfaces used to represent the BPEL4WS constructs.

An application first obtains a BPWSFactory instance via the static newInstance method of BPWSFactory. The newInstance method uses the following ordered lookup procedure to determine the BPWSFactory implementation class to load:

- Check the com.ibm.bpws.factory.BPWSFactory system property.
- Check the lib/bpws.properties file in the JRE directory. The key will have the same name as the above system property.
- Use the platform default value (will vary with implementations).

Note: There is also a static newInstance method that takes the fully-qualified class name of a factory implementation as an argument, in which case the above procedure is not employed.

Once a BPWSFactory instance is obtained, the method newBPWSProcess can be invoked to create a new BPWSProcess. Once that process is obtained, it serves as a factory that can be used to create the rest of the items that will make up the full process.

Listing 1 is an example that programmatically constructs a process containing `<receive>/<reply>`a sequence:

Listing 1. Constructing a <receive>/<reply> sequence

```
BPWSFactory factory = BPWSFactory.newInstance();
BPWSProcess process = factory.newBPWSProcess();
String tns = "urn:echo:echoService";
Containers containers = process.createContainers();
Container container = process.createContainer();
Partners partners = process.createPartners();
Partner partner = process.createPartner();
Sequence sequence = process.createSequence();
Receive receive = process.createReceive();
Reply reply = process.createReply();

process.setName("echoString");
process.setTargetNamespace(tns);

partner.setName("caller");
partner.setServiceLinkType(new QName(tns, "echoSLT"));
partners.addPartner(partner);
process.setPartners(partners);

container.setName("request");
container.setMessageType(new QName(tns, "StringMessageType"));
containers.addContainer(container);
process.setContainers(containers);

receive.setName("EchoReceive");
receive.setPartner(partner);
receive.setPortType(new QName(tns, "echoPT"));
receive.setOperation("echo");
receive.setContainer(container);
receive.setCreateInstance(Boolean.TRUE);
sequence.addActivity(receive);

reply.setName("EchoReply");
reply.setPartner(partner);
reply.setPortType(new QName(tns, "echoPT"));
reply.setOperation("echo");
reply.setContainer(container);
sequence.addActivity(reply);

sequence.setName("EchoSequence");
process.setActivity(sequence);
```

The in-memory representation created by the above sequence of steps should represent the process in Listing 2.

Listing 2. The in-memory model of the sequence steps

```
<process name="echoString"
         targetNamespace="urn:echo:echoService"
         xmlns:tns="urn:echo:echoService"
         xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/">

  <partners>
    <partner name="caller"
             serviceLinkType="tns:echoSLT"/>
  </partners>

  <containers>
    <container name="request"
               messageType="tns:StringMessageType"/>
  </containers>

  <sequence name="EchoSequence">
    <receive name="EchoReceive"
             partner="caller" portType="tns:echoPT"
             operation="echo" container="request"
             createInstance="yes"/>
    <reply name="EchoReply"
           partner="caller" portType="tns:echoPT"
           operation="echo" container="request"/>
  </sequence>

</process>
```

The pattern, just demonstrated, of using the process as a factory, and then adding the constructed item to the appropriate parent, is employed throughout the in-memory model.

The BPWS4J model can be used by tools, or by users who wish to programmatically create their processes, rather than reading or generating them.

Next time

In the next article in this series, we will explain how BPEL processes can be created visually using the BPWS4J visual editor. This will be followed by an article that illustrates the use of links and data manipulation by building on the loan approval sample we have already seen.

Resources

- Participate in the discussion forum.
- Please note that this articles refers to version 1.0 of the BPEL4WS specification. The latest version, BPEL4WS1.1, is now available, and an article describing the key differences between the two specifications will be available shortly.
- Check out the previous columns in the Business Process with BPEL column series.
- Read the full details of the Business Process Execution Language for Web Services, Version 1.0 specification.
- Download the Business Processes for Web services Java runtime (BPWS4J) from alphaWorks.

About the authors

Matthew J. Duftler is a software engineer in the Component Systems group at IBM T.J. Watson Research Center. He was one of the original authors of Apache SOAP, is the co-lead of JSR110, Java APIs for WSDL, and is a co-author of the IBM BPEL4WS engine, BPWS4J. You can contact Matthew Duftler at duftler@us.ibm.com.

Rania Khalaf is a software engineer in the Component Systems group at the IBM TJ Watson Research Center. She joined IBM in 2001 after having completed her Bachelors and MEng degrees from MIT. Rania is a co-author of the IBM BPEL4WS engine, BPWS4J available from alphaWorks. You can contact the author at rkhalaf@watson.ibm.com.