ASAP/Wf-XML 2.0 Cookbook

Keith D Swenson
Fujitsu Software Corporation

OVERVIEW

Wf-XML is a protocol for process engines that makes it easy to link engines together for interoperability. Wf-XML 2.0 is an updated version of this protocol, built on top of the Asynchronous Service Access Protocol (ASAP), which is in turn built on Simple Object Access Protocol (SOAP).

This article is for those who have a process engine of some sort, and wish to implement a Wf-XML interface. At first, this may seem like a daunting task because the specifications are thick and formal. But, as you will see, the basic capability can be implemented quickly and easily. This article will take you through the basics of what you need to know in order to quickly set up a foundation and demonstrate the most essential functions. The rest of the functionality can rest on this foundation. The approach is to do a small part of the implementation in order to understand how your particular process engine will fit with the protocol.

ASSUMPTIONS

It is assumed that you have a Business Process Management System (BPMS) onto which you are trying to fit this protocol. The specific design of that BPMS, the philosophy behind it, the technology it is built on, does not matter.

All BPM Systems are assumed to have a collection of process definitions that can be referred to by unique name or ID. How those processes are defined does not matter, nor does it matter how those definitions are stored and retrieved. The process definition may, but does not need to, have an external representation. All that is necessary is that the definition can be addressed using a unique ID.

It is assumed that there is a way to start a process instance for a particular process definition and that there is a set of data values that needs to be provided at the time the process is started. It is assumed that each process instance has a unique ID of some sort that can be used to access the current state of the process. Again, it does not matter how that process instance is stored, or what technology is used to retrieve it. All that is necessary is that there is a unique ID that can be used to access the current state.

It is assumed that the system in question has a way to send and receive SOAP messages. The above assumptions are sufficient for implementing the basic factory/instance roles of the protocol.

It is assumed that BPM systems will have a way to send a SOAP message and go into a persistent wait state such that SOAP messages at any point in the future could reactivate and continue the process. This will allow the

BPM system to start a remote process instance and wait for its completion in the observer role of ASAP.

The above is the minimum capability that is necessary to interact via ASAP or Wf-XML. These interactions can then be extended with additional parts of the protocol depending upon the capabilities of the BPM system.

- If the BPM system offers a way to list all the instances, then there is a command to allow the observer (or other client) to browse the currently running process instances.

- If the BPM system offers a way to list all the process definitions, then Wf-XML can be used for browsing the process definitions.

- If the BPM system has a way to add new process definitions, then Wf-XML can be used to send and register a new process definition.

- If the process instance is composed of a set of currently running activities, then Wf-XML can provide a way to list these activities and to access their currently running state.

These latter capabilities are optional extensions that can be made, but are not required for the basic interaction.

CAVEAT

This article is being written before either ASAP or Wf-XML 2.0 are formally ratified. That means that the specs may change in ways that make this article obsolete. Keep in mind as you read this article that the names of the operations may be slightly different and the names and arrangement of the tags may change from what is presented here. But the general concepts are sure to be the same and you will still find the general implementation strategy to be helpful. Before attempting any interoperability tests, please check with the official specification to be sure that your implementation is valid.

ASAP PRIMER

An Asynchronous Service Access Protocol (ASAP) has two sides. In the diagram below, the client side (Observer) is on the left, while the service side (Factory and Instance) is on the right. You will probably want to implement both sides, but it is recommended that you start by implementing the service side first. This immediately makes your BPM system a service for other systems.
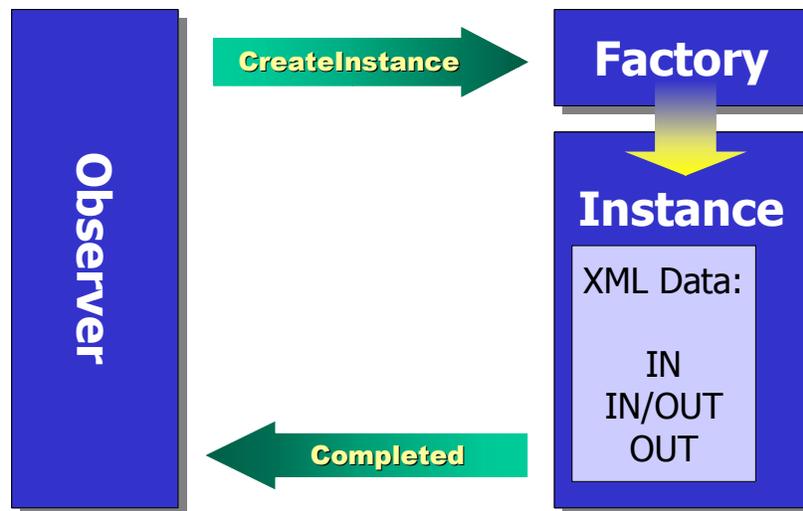
First implement the factory operation: CreateInstance that accepts details from the observer (with an immediate response back to the observer). The service runs for a while and, when finished, it makes a Completed request back to the observer (accepting an immediate response back). If you are using a process engine, this is enough to demonstrate the basic ability to invoke remotely via ASAP, and to return the result of the operation later.
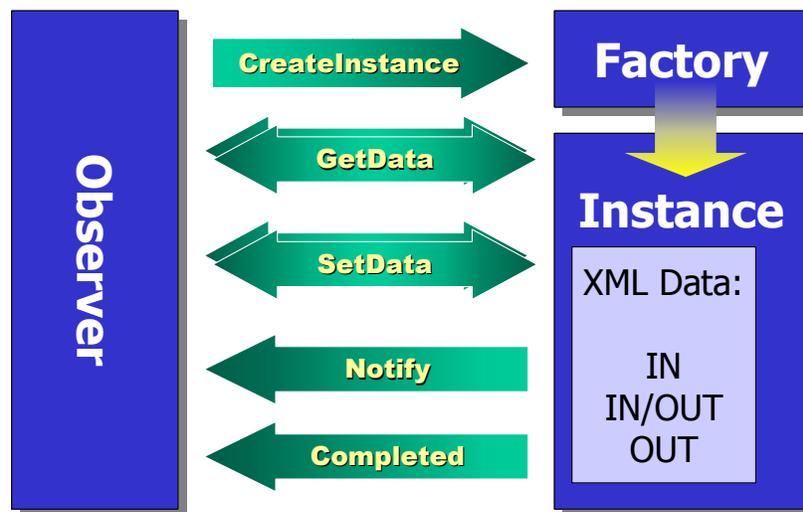
Next, we add some additional functions to the Instance resource. The Get-Data method will allow the observer to retrieve the current state of the instance, allowing for polling interactions. The SetData method will allow the observer to change the data values later. Since many services may not allow changing of data after the start, it is acceptable to simply return a fault message (exception). Finally, if the Service Instance goes through a number of state changes before it completes, it could make Notify operation requests back to the observer.

Wf-XML

Wf-XML extends the ASAP protocol by adding some additional capabilities between business process management systems. Such engines usually have a way to install and remove process definitions (factories). Wf-XML then adds a new resource called a container resource. On this resource you can invoke the operations of ListFactories in order to discover the factories that are installed on the container. Then you can create new Factories by supplying a new process definition. The factory resource is extended by the ability to retrieve the process definition and the ability to change the process definition to a new one as supplied. A version-numbering scheme is provided to keep track of the changes.
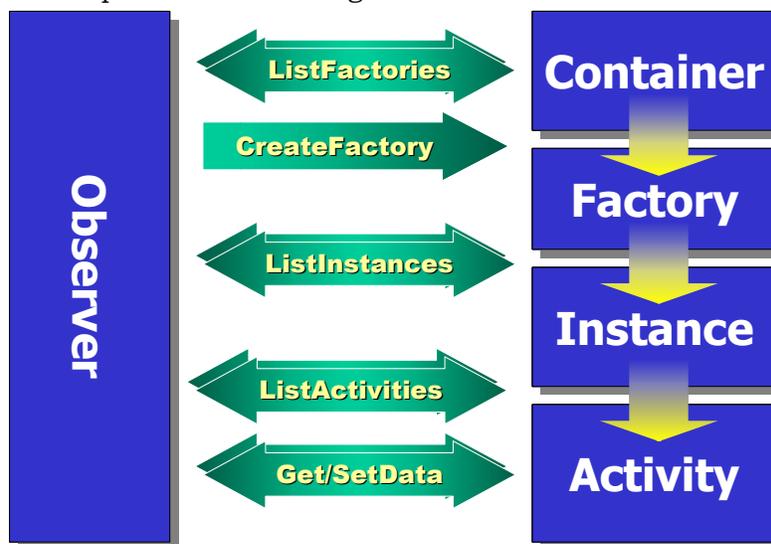
Wf-XML also adds the concept of an activity resource in order to give additional information about the process instance that is running. With ASAP we know that the process is open and running. When ListActivities is called, a list of the currently active activities is returned, telling you what step the process is currently at, so that you can monitor the advancement of the process.

Level 1 – Create and Complete (Server)

This section describes how to implement the server side of the CreateInstance to Completed cycle.

Factory Resource Address

Each process definition is represented as a *factory resource*. If you have 113 process definitions, you will have 113 factory resources. SOAP requests are made to the factory resource in order to do things like start instances and list all instances. Those new to the ASAP protocol will find this a little uncomfortable at first. You might ask, "Why not just make a single operation that you pass the name of the process definition to?" In fact, this is what

you will be doing, but we represent the factory as a web resource for a very important reason.

Consider how the Web works. When you access a document on the Web, you can use a single URL value. Encoded into the URL are the name of the machine to make the request to and the address of the document on the machine. Some addresses will invoke servlets that retrieve the document from a special place, or generate it on the fly. You, as the receiver of the document do not need to be concerned how the URL was composed. All that is important is that everything that the server needs to know in order to deliver the document is encoded into the URL. A link can be placed in a page. That link contains the URL, and by clicking on the link the document is retrieved.

When process engines are linked using ASAP, the URL of the factory will contain all the information necessary to locate the factory. For a BPM engine, this is easy because each process definition has a unique name or ID. It does not matter whether the ID is a numeric value, or simply a unique name, as long as there is a string of characters that uniquely identifies the particular process definition. Some systems may need to combine two or more values to make a unique ID. For example, a system that offers process definition versions may need to combine the name and the version in order to make a unique ID. The address of a factory can be composed two different ways. In both ways, there is a 'base URL' which can be thought of as the address of the handler that receives the SOAP message, and the process definition ID as an extension.

The process definition ID can be encoded as a URL parameter. Consider the case below where the address of a servlet is given, followed by examples of process definition IDs:

```
1   Servlet:   http://server:8080/bpm/factory.jsp
2   Factory:   http://server:8080/bpm/factory.jsp?id=84352
3   Factory:   http://server:8080/bpm/factory.jsp?id=Purchase+Order
4   Factory:   http://server:8080/bpm/factory.jsp?id=Expense&version=1.3
```

One of the advantages of using URL parameters is that you can easily use multiple values, and the typical servlet engine will parse the values automatically. Because the order of the values does not matter, you gain flexibility.

The second way is to map the handler as part of the path and then extend the path with the details of the unique ID. For example:

```
1   Servlet:   http://server:8080/bpm/factory/
2   Factory:   http://server:8080/bpm/factory/84352
3   Factory:   http://server:8080/bpm/factory/Purchase+Order
4   Factory:   http://server:8080/bpm/factory/Expense/1.3
```

It does not really matter how you construct the URL, so pick a technique that makes the most sense for the technology used to send and receive SOAP messages.

PROCESS INSTANCE ADDRESS

The CreateInstance operation will be called on the factory resource and will cause a process instance to be created. In order to allow the process instance to be accessed later to check status or perform other operations, the

process instance needs an URL. Since BPM systems are designed to access process instances on demand, they always have some form of unique ID for retrieving them. Again, you will have some form of handler at a base address, which is then extended with the specifics for the process instance ID.

```
1   Servlet:   http://myserver:8080/bpm/instance.jsp
2   Instance:  http://myserver:8080/bpm/instance.jsp?id=84352
```

You may find yourself asking the question: "Why form a whole URL for the instance? Why not just use the factory address and pass the instance ID as a field in the message?" There are two answers. The first is one of packaging. We want to provide a single value back to the caller. Any number of different values can be packed into a single URL, using well-defined rules. So you may, if you wish, extend the factory address with the process instance ID. But you may also decide to do it other ways. If the server takes on the full responsibility of packing the values together by returning a complete URL, and of course parsing the values apart when used, then the server can use any scheme to encode any amount of data into the URL. The second reason is that the instances and the factory may not be served from the same host machine. This may be used as a primitive form of load balancing, or there may be a technical reason for running a process on a particular host. An infinite number of URL addresses are readily available for free so we use this approach to hide the technical requirements of the system from the caller.

TRANSPORT STRATEGY

The above examples use 'http' addressing, meaning that the XML of the SOAP message will be passed over http. ASAP and Wf-XML are not limited to http addresses; they can be delivered in any way that a SOAP message can be delivered. That being said, initial implementations typically use http because it is easy to work with and debug. Also, all implementations of Wf-XML are sure to support http transport of SOAP messages, hence interoperability with other systems is assured if you support this transport.

Some features are optional when using http. This document is simplified somewhat by the assumption that you will be using http. Just keep in mind if you use a different transport you may need to implement the protocol a slightly different way. Please check the specification for details.

CONTEXT DATA AND RESULT DATA

Before implementing the first operation, you need to decide a structure for the context data and result data. The context data is a structure used for sending data to the factory or process instance. Think of it as the input data. If your BPM engine supports updating the process variables, then this structure is also used for the SetProperties operation. The Result data is an XML structure for communicating data in the other direction. The Completed operation and Notify operations contain the Result data XML structure.

Most BPM services store the context data in a set of *process variables*. How the BPM system stores the data does not matter. The ASAP clients of your server will never see your variables directly. They only see the XML repre-

sentation of them. Therefore it is common to describe the context data XML structure as being the process variables, but your job is to accept this XML and then save the values as process variables. Furthermore, at the end of the process, or when requested, you need to be able to read the process variables and produce the Result data XML structure.

Why are there two different structures: context and result? There is a need to have values that can be set, values that can be read, and values that can be both set and read. Using XML schema to define structure, there is no convenient way to indicate what parts of the structure are IN variables, OUT variables, or IN/OUT variables. IN variables are in the context data structure. OUT variables are in the result data structure. IN/OUT variables are in both structures, and by convention will be found at the same XPath location in both structures. It is possible that all variables are IN/OUT, so the context and result structures can be identical.

The ASAP specification leaves it up the implementation to define these XML structure. Any XML structure that can be described by XML Schema is allowed. Most BPM systems allow different process definitions to define different sets of variables, so it is insufficient to define a single context data structure for the entire BPM system. Many BPM systems have, for each process, a set of name/value pairs. The simplest way to construct the context data XML structure is to generate tags using the name of the variable that contain the value of the variable. If your BPM system offers typed variables, then be sure to consider which XML Schema type the value most closely matches, or use String if type checking is not needed. If the variable itself contains a complex record structure, you may want to think about how best to map this structure into XML. If the variable contains XML directly, be sure to put the XML in the message as XML, and not encoded into a string, so that the variable can be easily extracted without having to parse the result multiple times.

You will find it most convenient to parse the request or response message into a DOM tree. The context data branch of the tree can be easily iterated through. For each tag, look for a corresponding variable and set the value. For generating response data, iterate though your variables; and generate a DOM tag element with the name of the variable and the contents as the value of the variable. If you write this in a generalized way it will work for all process definitions automatically.

Be careful though. It is possible that your BPM system allows variables to be named in ways that are not allowed as XML tag names. For example, you may allow space characters in variable names, but XML tag names may not contain spaces in them. This means that you need to somehow encode the variable name into an acceptable tag name. Experience has shown that it is often possible to just use a simplified version of the variable name where any offending characters are simply stripped out of the name. This is not a reversible conversion, since it is possible for more than one variable name to be simplified to the same tag name. Human nature is such that variables usually have names that are different enough so that the simplified names are still unique within the process. Generally, it is sufficient to validate the process definition by iterating through the variables when it is

first installed, checking that the simplified names are all unique within the process—simply throw an exception if they are not, and let the process designer fix the problem.

SECURITY

Security of information systems involves authentication, authorization and privacy. One must take care that a Wf-XML or ASAP implementation does not become a way around the built in security mechanisms of a BPM system.

Privacy is the simplest requirement to meet. Those requiring privacy must be able to send and receive the SOAP messages over an SSL link—usually by using HTTPS (but SOAP allows other ways to send a message and privacy can be ensured by using SSL). If you are sending SOAP messages over SMTP, then you may need to encrypt the contents of the email message to ensure privacy. All of these solutions are out of the scope of the protocol.

Authentication: The first rule is that every SOAP request MUST be authenticated. Never allow unauthenticated requests into the server. Usually server 1 (client) authenticates to server 2 (service) through the normal means so that server 2 is assured that the call is coming only from server 1.

Authorization: Server 2 then makes available only the information that server 1 needs to know. Once you have this set up you need to be concerned about who is using server 1 that can cause server 1 to invoke a service on server 2. If server 2 gives privileged information to server 1, then you must assume that any user of server 1 can get that privileged information. It is best if server 2 only gives information to server 1 that would be acceptable to be given to any user of server 1. This is not always possible. The server is, after all, acting on behalf of another, and may need access to restricted information to accomplish its task. If server 2 allows server 1 access information that is not generally accessible, then you must be sure that server 1 guards the information with the same rules that server 2 guards the information with.

It is not my goal to solve the problem in this article, but simply to raise the issue here that if you have access control on information, you need to think carefully on how to maintain the same access control now that you have two servers cooperating. This has very little to do with the protocol, but the implementation needs to be careful that the protocol does not become a backdoor to privileged information.

RECEIVING: CREATEPROCESS (ON FACTORY)

The first operation you will create will be the one that creates a process instance within a BPM system. An example message is given below. Implementing the CreateProcess handler entails receiving this message, parsing the XML into a DOM, and then doing the appropriate operations with the elements.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5      <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
```

```
6      <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8      <as:ResponseRequired>Yes</as:ResponseRequired>
9     </as:Request>
10   </env:Header>
11   <env:Body>
12    <as:CreateInstanceRq>
13     <as:StartImmediately>Yes</as:StartImmediately>
14     <as:ObserverKey>http://sender.com/observer.jsp</as:ObserverKey>
15     <as:Name>Expense Approval for Jones</as:Name>
16     <as:Subject>Expense Approval for Jones</as:Subject>
17     <as:Description>This is a …</as:Description>
18     <as:ContextData xmlns:m123="http://rec.com/contextschema.jsp?id=213">
19      <m123:Name>Jenny B Jones</m123:Name>
20      <m123:amount>$317.45</m123:amount>
21      <m123:city>San Jose</m123:city>
22      <m123:zip>95134</m123:zip>
23     </as:ContextData>
24    </as:CreateInstanceRq>
25   </env:Body>
26  </env:Envelope>
```

Line 5: The sender may or may not be the same value as the observer. Currently, the only reason Sender Key is needed is to include it in the response message.

Line 6: This is the address that the sender was trying to deliver to, so this should be the address of the factory. If you wish, you can compare this value to the assumed source URL for the SOAP messages, and, if different, indicate some kind of error. Initially, though, use this value only to copy into the response.

Line 7: The request ID is needed only to copy back into the response so that the response can be accurately matched with the request. This really should not be necessary when using http protocol, but if you receive a request ID, simply copy that value into the response message.

Line 8: The easiest implementation is to always send a response to every request. When using http this is the desired implementation anyway. For initial implementations ignore this value and always return a response. Later, when you are getting ready to certify a completed implementation, revisit this value.

Line 13: The 'StartImmediately' option is for the relatively rare case that someone might want to create a process instance without starting it, and then use a ChangeState operation to start it later. Most of the time, systems will simply wait until they are ready to start the process instance, and perform the create and start in the same call. So most of the time this will be 'Yes'. For the initial implementation, ignore this value, or if you wish to be proper, check that the value is 'Yes' and return a fault (exception) if it is anything else.

Line 14: This is the observer key and it MUST be stored someplace, even for the simplest implementation of the protocol. Ultimately, the BPM engine should have a special place to record this for every process instance. On the assumption that you are creating this Wf-XML interface for an existing engine, and that you do not have the luxury of changing the BPM engine, then consider using a process variable to store this string value. If this is not

possible, then the interface layer can store this by maintaining a persistent map from process instance ID to observer ID. This option is less than desirable because it is more effort to keep the process instances and observers consistent. However you store it, it is absolutely required at process end, that this observer key can be retrieved in order to call the Completed operation on the observer.

Line 15: If your BPM engine has a way to name process instances, use this value for the name, otherwise you can ignore it.

Line 16: If your BPM engine has a subject for each process instance, use this value for the subject, otherwise you can ignore it.

Line 17: If your BPM engine has descriptions for the process instances, use this value for the description, otherwise you can ignore it.

Line 18: This is the context data that is the initial data for the process. See the section on Context data above. Note that the context data tags are in a different namespace from the other tags. This must be done in order to prevent name clash and to allow variables with any name to be represented. The namespace is often associated with an XML Schema definition, and in this case that is the schema of the context data we discussed before. Remember that each factory (each process definition) has a unique definition of the context data. In this example, the name space identifier includes the ID of the process definition. This is not necessary, but it is a reminder that the schema depends on the factory. Similarly the URL includes the ID in such a way that it will be possible later to implement a servlet to return the XSD file for that process definition for a validating parser. It is not necessary to actually generate the XSD in this initial version, but now the pattern is set.

Lines 19-22: This data must be parsed out of the XML and converted into whatever form is suitable for starting a process instance on your BPM system according to that particular process definition.

That should be all the information that you need to create and start a process instance. Make the appropriate calls on your BPM engine to instantiate the process before generating the response. A sample response is given below, with a similar discussion after it.

```
1   <?xml version="1.0">
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5      <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6      <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8     </as:Response>
9    </env:Header>
10   <env:Body>
11    <as:CreateInstanceRs>
12     <as:InstanceKey>http://rec.com/instance.jsp?id=456 </as:InstanceKey>
13    </as:CreateInstanceRs>
14   </env:Body>
15  </env:Envelope>
```

Line 5: Copy this from the request.

Line 6: Copy this from the request.

Line 7: Copy this from the request.

Line 12: Generate this URL for the Instance resource using the process instance ID and whatever encoding style you decided upon (according to the section on process instance address).

And there you are, you are done with your first ASAP operation.

SENDING: COMPLETED (TO OBSERVER)

Upon receiving an ASAP request to start an asynchronous server (process instance), you make a commitment to send a Completed message when that process instance completes. Different BPM systems will offer different degrees of flexibility in making this come about. Your system might offer a way for an external program to register a "call back" upon the termination of a process instance, in which you can call the code to send the response. If this is not available, you might have to include an activity at the end of the process that invokes the code to send the completed message. If there is no way to cause the BPM system to proactively do something at the end of the process, you might have to resort to having the Wf-XML interface-layer poll the process instance and, when it detects that the status has changed, send the Completed message. Ultimately, upon completion of a process, the BPM system should somehow automatically check to see if an observer URL exists, and send a completed message to it.

Once you have decided how to invoke it, the coding of the completed message is quite straightforward. Next is an example of the completed message, followed by line descriptions.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://rec.com/instance.jsp?id=456</as:SenderKey>
6       <as:ReceiverKey>http://sender.com/observer.jsp</as:ReceiverKey>
7      </as:Request>
8     </env:Header>
9     <env:Body>
10     <as:CompletedRq>
11      <as:InstanceKey>http://rec.com/instance.jsp?id=456</as:InstanceKey>
12      <as:ResultData xmlns:m123="http://rec.com/resultschema.jsp?id=213">
13       <m123:amount>$317.45</m123:amount>
14       <m123:Approved>Yes</m123:Approved>
15      </as:ResultData>
16     </as:CompletedRq>
17    </env:Body>
18   </env:Envelope>
```

Line 5: The Sender Key will be the URL of the process instance.

Line 6: The Received Key will be the observer URL you recorded for this process instance. Note, there is no reason to specify ResponseRequired since the default for this tag is 'yes'. And, there is no reason to specify a request ID since you are using http and the response will be synchronous.

Line 11: It is critical that the Instance URL be specified here. The observer system may be observing many different remote subprocesses. This field is used to determine which one of them is now completed.

Line 13-14: The Result Data must be read from the process variables and put into the XML as per the mapping decided upon in the above topic, "Context Data and Result Data."

Send this XML to the observer URL and receive the response like the one below.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5       <as:SenderKey>http://rec.com/factory.jsp?id=213</as:SenderKey>
6       <as:ReceiverKey>http://sender.com/observer.jsp</as:ReceiverKey>
7      </as:Response>
8     </env:Header>
9     <env:Body>
10     <as:CompletedRs/>
11    </env:Body>
12   </env:Envelope>
```

The only thing that you need to do is to make sure that the CompletedRs tag exists, as shown on line 10. Merely the fact that a response was successfully received, should be sufficient to assume the message was delivered. If you receive an error, or for any other reason do not receive a successful response, you should keep retrying to send until it is successful. This retrying behavior is best used in conjunction with the headers from the WS-Reliability specification from OASIS to avoid sending duplicate messages by mistake.

By implementing the above, you are ready for the Level 1 interoperability demonstration and a service.

LEVEL 2—REMOTE SUBPROCESS (CLIENT)

Level 2 implements the client side of the CreateInstance to Completed cycle—providing the capability to invoke a remote asynchronous service and wait for it to complete. To do this, you need a process engine of some sort that is able to send a SOAP message and then wait for a SOAP request that can cause the process to continue. It is assumed that this will be done as part of a process and that the details are stored in a process instance. The process instance, then, plays the observer role in the protocol. In order to keep the concepts straight, we will call the observer process instance (the waiting process) the "parent" process instance. The new process instance (that is invoked in the other service) will be called the "child" process instance, or the sub-process instance.

When configuring a process to be able to create a subprocess, the process designer must provide two things: (1) the URL of the factory of the subprocess, and (2) a way to map data from the parent into the child and vice versa.

DATA MAPPING

The parent process instance holds some data in process variables and it needs to give some data to the remote factory in order to create the sub-process. We cannot assume that the schema of these is the same. There will need to be a translation mechanism. Values from the parent process

instance should be collected and transformed to produce values for the subprocess. How this is done is completely up to you. One simple approach is to have the creator of the parent to child link provide an XSLT transform script that produces the "context data" part of the message, which can then be sent in the CreateProcess. If you are expecting data to come back from the subprocess, then another transform will need to be provided. There are some graphical-mapping-tools which, given two schemas, can generate mappings both ways.

OBSERVER ADDRESS

As far as the service side of the protocol is concerned, you can create the Observer URL in any way you wish. What you need to keep in mind when defining this URL is that you are going to receive the Completed SOAP request at this address. It will be highly convenient to include the process instance ID of the parent process that is waiting for this event. Then, when the request comes, the handler can easily gain access to the correct parent process instance and make use of information stored there to correctly handle the request. In a very real sense, the parent process instance is really the observer of the invoked process, so the observer URL should be the process instance URL.

In some cases, you can have the process instance observing multiple sub-processes simultaneously. It might be convenient to code more information into the observer URL. For instance, if there is a particular node in the process that represents the remote sub-process request, you could include the ID of that node in the observer URL. With ASAP and Wf-XML you have complete freedom to include any information into the observer URL, whatever you need in order process the arriving requests.

SENDING: CREATEPROCESS (TO FACTORY)

This is, of course, the same command described at the top of the article, but now it is described from the client perspective, and includes considerations for constructing this message and receiving the response. Again, a sample message follows.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp?parent=44</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      </as:Request>
8     </env:Header>
9     <env:Body>
10     <as:CreateInstanceRq>
11      <as:ObserverKey>http://sender.com/observer.jsp?parent=44
     </as:ObserverKey>
12      <as:Name>Expense Approval for Jones</as:Name>
13      <as:Subject>Expense Approval for Jones</as:Subject>
14      <as:Description>This is a …</as:Description>
15      <as:ContextData xmlns:m123="http://rec.com/contextschema.jsp?id=213">
16       <m123:Name>Jenny B Jones</m123:Name>
17       <m123:amount>$317.45</m123:amount>
18       <m123:city>San Jose</m123:city>
19       <m123:zip>95134</m123:zip>
20      </as:ContextData>
21     </as:CreateInstanceRq>
22    </env:Body>
23   </env:Envelope>
```

Line 5: This is the URL of your handler

Line 6: This is the URL of the factory specified by the process designer.

No need to specify a request ID. Omit Response required, which is 'yes' by default, and omit StartImmediately, which is also 'yes' by default.

Line 12: Part of the data mapping needs to include a specification of the name of the subprocess.

Line 13: Part of the data mapping needs to include a specification of the subject of the subprocess.

Line 14: Part of the data mapping needs to include a specification of the description of the subprocess.

Line 15-20: Data mapping needs to be able to generate the context data part of the message from the parent process variables. You should anticipate the following response.

```
1    <?xml version="1.0">
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp?parent=44</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      </as:Response>
8     </env:Header>
9     <env:Body>
10     <as:CreateInstanceRs>
11      <as:InstanceKey>http://rec.com/instance.jsp?id=456 </as:InstanceKey>
12     </as:CreateInstanceRs>
13    </env:Body>
14   </env:Envelope>
```

The only item of data that is important is line 11 the InstanceKey. This needs to be saved in the parent process instance.

RECEIVING: COMPLETED (ON OBSERVER)

You need to be able to receive this at any time. The handler will load the process instance, transform the result data, set the appropriate process variables, and deliver the proper event to satisfy the wait condition so that the process instance will continue processing.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5      <as:SenderKey>http://rec.com/instance.jsp?id=456</as:SenderKey>
6      <as:ReceiverKey>http://sender.com/observer.jsp?parent=44
    </as:ReceiverKey>
7     </as:Request>
8    </env:Header>
9    <env:Body>
10    <as:CompletedRq>
11     <as:InstanceKey>http://rec.com/instance.jsp?id=456</as:InstanceKey>
12     <as:ResultData xmlns:m123="http://rec.com/resultschema.jsp?id=213">
13      <m123:amount>$317.45</m123:amount>
14      <m123:Approved>Yes</m123:Approved>
15     </as:ResultData>
16    </as:CompletedRq>
17   </env:Body>
18  </env:Envelope>
```

Line 11: Check that the URL is from the anticipated process instance. Match it to the process instance URL given at the time of creation. Completed events from earlier invocations are possible; they need to be ignored.

Line 12-15: This is the result data that must be transformed and stored in variables in the parent process instance.

Send a message like this to confirm receipt.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5      <as:SenderKey>http://rec.com/factory.jsp?id=213</as:SenderKey>
6      <as:ReceiverKey>http://sender.com/observer.jsp?parent=44
    </as:ReceiverKey>
7     </as:Response>
8    </env:Header>
9    <env:Body>
10    <as:CompletedRs/>
11   </env:Body>
12  </env:Envelope>
```

With Level 1 and Level 2 implemented, you can demonstrate a full round trip: a process in one engine reaches a node that starts a second process in another engine; the second process reaches the end and sends a completion event; and the first process receives it and continues. Because the exchange is based in XML on an open specification, you can choose from a variety of vendor products on either side of the interchange.

LEVEL 3 – INTROSPECTION (SERVER)

At the Level 2 remote sub-process invocation is functional, the next important step for improving interoperability is to make it easier to set up the process that invokes the sub-process. The assumption is that a "process

designer" who is not a programmer designs the process. The average process designer is not expected to be able to write an XSLT script to transform the data from one process schema to the other. Many process design tools offer GUI based features for transforming data. A tool that offers these capabilities needs to be able to retrieve the schema of the remote service. This Level 3 section concentrates on adding ASAP capabilities that support the design tool.

THE CONTAINER RESOURCE URL

The container resource is needed to represent the fact that a single BPM system can contain many different process instances. The container then represents the BPM system as a whole. It will have a fixed URL address that can be used to ask questions of the system as a whole. For example, what are the process definitions (factories) that are already present in the system? This is the resource that you use for adding new process definitions to the collection.

There is not much thought that needs to be given to the URL of the container. Every BPM system installation will only need a single fixed URL. An example is given below:

```
1    Container: http://rec.com/container.jsp
```

The idea is to give the URL for the container to the process design tool. Using that URL the design tool can list all the factories for selection. Once a given factory is selected, a request can be made to retrieve the factory properties (including the schema of the context data structure and the result data structure) so that the process designer can pick the data mapping from a data transform tool. The design tool can also request a process definition in a standard format to let the process designer see what the process looks like.

RECEIVING: LISTFACTORIES (ON CONTAINER)

Here is an example of a request message you might receive:

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7       <as:RequestId>abc123</as:RequestId>
8       <as:ResponseRequired>Yes</as:ResponseRequired>
9      </as:Request>
10    </env:Header>
11    <env:Body>
12     <as:ListFactoriesRq/>
13    </env:Body>
14   </env:Envelope>
```

Line 12: the first child of the body tag always indicates the operation being requested.

Generate a response similar to the one shown below.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5      <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6      <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8     </as:Response>
9    </env:Header>
10   <env:Body>
11    <wf:ListFactoriesRs xmlns:wf="http://wfmc.org/wfxml/2.0">
12     <wf:Factory>http://rec.com/factory.jsp?id=10</wf:Factory>
13     <wf:Factory>http://rec.com/factory.jsp?id=21</wf:Factory>
14     <wf:Factory>http://rec.com/factory.jsp?id=22</wf:Factory>
15     <wf:Factory>http://rec.com/factory.jsp?id=108</wf:Factory>
16     <wf:Factory>http://rec.com/factory.jsp?id=184</wf:Factory>
17     <wf:Factory>http://rec.com/factory.jsp?id=213</wf:Factory>
18    </wf:ListFactoriesRs>
19   </env:Body>
20  </env:Envelope>
```

Given that your BPM system has a way to list the installed process definitions, you need only iterate through the process definitions, and generate a line for each one that includes the URL to the Factory. Please check the specification because at the time of this writing the format of the factory tag is not well defined.

RECEIVING: GETPROPERTIES (ON FACTORY)

The GetProperties command is structured the same for all resources, but the properties of the Factory are different from the properties of the instance. A detailed description follows.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5      <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6      <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8      <as:ResponseRequired>Yes</as:ResponseRequired>
9     </as:Request>
10   </env:Header>
11   <env:Body>
12    <as:GetPropertiesRq/>
13   </env:Body>
14  </env:Envelope>
```

Lines 3-10: Handle the header in the normal fashion.

Line 13: The only significant thing in the message is the presence of the GetPropertiesRq tag.

A response should be constructed along the lines of the example below.

```
1   <?xml version="1.0"?>
2   <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3    <env:Header>
4     <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5      <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6      <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8     </as:Response>
9    </env:Header>
```

```
10   <env:Body>
11    <as:GetPropertiesRs>
12     <as:Key>http://rec.com/factory.jsp?id=213</as:Key>
13     <as:Name>Expense Approval</as:Name>
14     <as:Subject>Expense Approval</as:Subject>
15     <as:Description>This is a …</as:Description>
16     <as:ContextDataSchema>
17      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
18       <xsd:sequence>
19        <xsd:element name="Name" type="xsd:string" minOccurs="0"/>
20        <xsd:element name="amount" type="xsd:string" minOccurs="0"/>
21        <xsd:element name="city" type="xsd:string" minOccurs="0"/>
22        <xsd:element name="zip" type="xsd:string" minOccurs="0"/>
23       </xsd:sequence>
24      </xsd:schema>
25     </as:ContextDataSchema>
26     <as:ResultDataSchema>
27      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
28       <xsd:sequence>
29        <xsd:element name="amount" type="xsd:string" minOccurs="0"/>
30        <xsd:element name="Approved" type="xsd:string" minOccurs="0"/>
31       </xsd:sequence>
32      </xsd:schema>
33     </as:ResultDataSchema>
34     <as:Expiration>P90D</as:Expiration>
35    </as:GetPropertiesRs>
36   </env:Body>
37  </env:Envelope>
```

Line 3-9: Handle the header like normal, echo these values from the request.

Line 12: Return the URL of the factory here, as a second check

Line 13: The name of the process definition

Line 14: The subject of the process definition, if there is one

Line 15: The description of the process definition, if there is one

Line 16-25: Context data schema. You have determined how to map the incoming requirements for a process into XML, this is the XML Schema of that XML. In this very simple example, each incoming variable is marked as a string type. If you have strong typing and can make a better match than string, then use that. The more meta information that is given about the structure requirements, the more the design tool can do to assure that your system gets it. By marking each element with minOccurs="0" you make each element optional—if present it will be used, if not, it will be ignored.

Line 26-33: Result data schema. Like the context data, but it describes the data that will be returned in the Completed message.

Line 34: Expiration. How long are you guaranteeing that process instance information will be available after the Completed message is delivered? Some systems will keep this information forever, so you can use the setting provided set for 90 days in this example. If it is unlikely that details of the process instance will still be available 90 days after completed, read the spec and XML Schema to determine the correct setting here. The client may need additional information from the process instance after receiving the Completed message. The factory specifies how long that data is going to

hang around (at a minimum). After this time, there is no guarantee that the service instance address will be good.

RECEIVING: GETDEFINITION (ON FACTORY)

WfXML adds a GetDefinition operation to the factory resource that returns the process definition in the specified format.

```
1  <?xml version="1.0"?>
2  <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3   <env:Header>
4    <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5     <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6     <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7     <as:RequestId>abc123</as:RequestId>
8     <as:ResponseRequired>Yes</as:ResponseRequired>
9    </as:Request>
10  </env:Header>
11  <env:Body>
12   <wf:GetDefinitionRq xmlns:wf="http://wfmc.org/wfxml/2.0">
13    <wf:Format>XPDL</wf:Format>
14   </wf:GetDefinitionRq>
15  </env:Body>
16 </env:Envelope>
```

The header is handled as in the previous samples. The child of the Body tag is the operation name and the Format tag holds the format. XPDL and BPEL are the formats known at this point in time, more will surely be developed in the future.

The response is shown next.

```
1  <?xml version="1.0"?>
2  <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3   <env:Header>
4    <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5     <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6     <as:ReceiverKey>http://rec.com/factory.jsp?id=213 </as:ReceiverKey>
7     <as:RequestId>abc123</as:RequestId>
8    </as:Response>
9   </env:Header>
10  <env:Body>
11   <wf:GetDefinitionRs xmlns:wf="http://wfmc.org/wfxml/2.0">
12    <xp:xpdl xmlns:xp="http://wfmc.org/xpdl/1.1">
13     <!-- include the XPDL definition here -->
14    </xp:xpdl>
15   </wf: GetDefinitionRs>
16  </env:Body>
17 </env:Envelope>
```

After implementing Level 3, it is possible for a design tool to introspect the process definition and to help link a parent process to a sub-process.

LEVEL 4 – PROPERTY AND NOTIFICATIONS (SERVER)

Level 4 is implementation to *support* getting and setting properties, and notification. This enables changes in data to be propagated from parent process to sub-process while both processes are running.

STATUS MAPPING

You need to decide how to expose the status of process instances as defined by your BPM system in terms of the status values defined by the spec. Here is a list of the values defined:

```
1   open.notrunning.suspended
2   open.running
3   closed.completed
4   closed.abnormalCompleted
5   closed.abnormalCompleted.terminated
6   closed.abnormalCompleted.aborted
```

It is not necessary to be able to be in all these states. Instead, it is important to consider all of the states that your BPM system naturally supports, and to figure out which of these states best expresses that state. Most of the time the process instance is in the 'open.running' state, and if your system does not have suspend, then this is the only state that you need to report.

If there is a suspended state that holds up operations, which can later be resumed, then indicate this with the 'open.notrunning.suspended' value. The commands for 'suspend' and 'resume' are not defined by the standard, hence you must use the 'ChangeState' operation. A ChangeState operation to this state is the same as a 'suspend' operation, while a ChangeState to 'open.running' is equivalent to a 'resume' operation.

The states starting with 'closed' are terminal states—there can be no transitions back to an open state. Clearly if the process ends normally, it will be in the 'closed.completed' state. Other terminal states should be mapped as well as possible. Read the specification for more help on this.

RECEIVING: GETPROPERTIES

GetProperties is used to retrieve properties as well as the current status of the process instance.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/instance.jsp?id=456 </as:ReceiverKey>
7       <as:RequestId>abc123</as:RequestId>
8       <as:ResponseRequired>Yes</as:ResponseRequired>
9      </as:Request>
10    </env:Header>
11    <env:Body>
12     <as:GetPropertiesRq/>
13    </env:Body>
14   </env:Envelope>
```

Lines 3-10: Handle the header as specified in the past.

Line 12: The presence of the GetPropertiesRq tag indicates the request type.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/instance.jsp?id=456 </as:ReceiverKey>
7       <as:RequestId>abc123</as:RequestId>
8      </as:Response>
9     </env:Header>
10    <env:Body>
11     <as:GetPropertiesRs>
12      <as:Key>http://rec.com/instance.jsp?id=456</as:Key>
13      <as:State>open.running</as:State>
14      <as:Name>Expense Approval for Jones</as:Name>
15      <as:Subject>Expense Approval for Jones</as:Subject>
16      <as:Description>This is a …</as:Description>
17      <as:FactoryKey>http://rec.com/factory.jsp?id=213</as:FactoryKey>
18      <as:Observers>
19       <as:ObserverKey>http://sender.com/observer.jsp</as:ObserverKey>
20      </as:Observers>
21      <as:ResultData xmlns:m123="http://rec.com/resultschema.jsp?id=213">
22       <m123:amount>$317.45</m123:amount>
23       <m123:Approved>Yes</m123:Approved>
24      </as:ResultData>
25     </as:GetPropertiesRs>
26    </env:Body>
27   </env:Envelope>
```

Lines 3-9: Handle the header as the past. If a RequestId is sent, then return it.

Line 12: Fill in the instance key.

Line 13: Map the internal process state into the state value as per the section above.

Line 14: If the process instance has a name, fill it in here, otherwise omit the tag.

Line 15: If the process instance has a subject, put it here, otherwise omit.

Line 16: If the process instance has a description, put it here, otherwise omit.

Line 17: Fill in the URL of the factory (process definition) that this instance belongs to.

Lint 18-20: When subscribe and unsubscribe are supported, there could be multiple observers, otherwise there will only be a single observer. Initially, it is not necessary to support subscribe and unsubscribe.

Line 22-23: Result data encoded as described in the Context and Result Data section.

RECEIVING: SETPROPERTIES

Used by clients to change context data. The structure is roughly similar to that of the CreateProcess command, and should be treated in a similar manner.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
```

```
6      <as:ReceiverKey>http://rec.com/instance.jsp?id=456 </as:ReceiverKey>
7      <as:RequestId>abc123</as:RequestId>
8      <as:ResponseRequired>Yes</as:ResponseRequired>
9     </as:Request>
10    </env:Header>
11    <env:Body>
12     <as:SetPropertiesRq>
13      <as:Subject>Expense Approval for Jones</as:Subject>
14      <as:Description>This is a …</as:Description>
15      <as:Priority>4</as:Priority>
16      <as:ContextData xmlns:m123="http://rec.com/contextschema.jsp?id=213">
17       <m123:Name>Jenny B Jones</m123:Name>
18       <m123:amount>$317.45</m123:amount>
19       <m123:city>San Jose</m123:city>
20       <m123:zip>95134</m123:zip>
21      </as:ContextData>
22     </as:SetPropertiesRq>
23    </env:Body>
24   </env:Envelope>
```

Line 3-10: Handle the header as specified in the past.

Line 13: If present, a new value for subject.

Line 14: If present, a new value for description.

Line 15: If present, a new value for priority.

Line 16-21: New values for the context data, if changes are allowed copy these into the process instance variables in the same manner as the CreateInstance operation.

Send a response back that is exactly the same as the response to GetProperties. (so it is not duplicated here).

RECEIVING: CHANGESTATUS

Change status can be used to transition a process into another state. There are no commands for the transitions themselves, such as 'suspend' and 'resume'. Instead change the state into the destination state that you want. There is no guarantee that you can transition from any state into any other state, so if asked to transition to a state where the transition is not allowed, you need only return a fault message indicating the failure.

Here is an example request message.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/instance.jsp?id=456 </as:ReceiverKey>
7       <as:RequestId>abc123</as:RequestId>
8       <as:ResponseRequired>Yes</as:ResponseRequired>
9      </as:Request>
10    </env:Header>
11    <env:Body>
12     <as:ChangeStateRq>
13      <as:State>open.notrunning.suspended</as:State>
14     </as:ChangeStateRq>
15    </env:Body>
16   </env:Envelope>
```

Lines 3-10: Handle the header as in previous examples.

Line 13: This tag is the only value you need to read. Determine what internal state the specified state maps to. If there is no corresponding internal state, return a fault message. If it is not possible to transition to that state from the current state, return a fault message.

Otherwise, execute the command to transition into the desired state. Below is a response message.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Response xmlns:aws="http://www.ASAP.info/spec/1.0/">
5       <as:SenderKey>http://sender.com/observer.jsp</as:SenderKey>
6       <as:ReceiverKey>http://rec.com/instance.jsp?id=456 </as:ReceiverKey>
7       <as:RequestId>abc123</as:RequestId>
8      </as:Response>
9     </env:Header>
10    <env:Body>
11     <as:ChangeStatusRs>
12      <as:State>open.running</as:State>
13     </as:ChangeStatusRs>
14    </env:Body>
15   </env:Envelope>
```

Lines 3-9: Handle the header in the normal manner.

Line 12: Specifies the actual state that was transitioned to. Note that this may not be the same as was requested because it may be a more detailed (specific) state.

SENDING: STATECHANGED

This is a SOAP request from the instance resource back to the observer resource notifying it that the state has changed. If your BPM system has the ability to proactively notify the interface layer on state changes, then consider sending this message to the observer.

```
1    <?xml version="1.0"?>
2    <env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
3     <env:Header>
4      <as:Request xmlns:as="http://www.asap.info/spec/1.0/">
5       <as:SenderKey>http://rec.com/instance.jsp?id=456</as:SenderKey>
6       <as:ReceiverKey>http://sender.com/observer.jsp</as:ReceiverKey>
7      </as:Request>
8     </env:Header>
9     <env:Body>
10     <as:StateChangedRq>
11      <as:State>open.running</as:State>
12      <as:PreviousState>open.notRunning.suspended</as:PreviousState>
13     </as:StateChangedRq>
14    </env:Body>
15   </env:Envelope>
```

Line 11: The current state.

Line 12: The state that it changed from, if available.

You may receive a response from this, but it can be safely ignored. There need be no guarantee of the delivery of this message.

CONCLUSION

The purpose of this article is to help those with a BPM system to start implementing a Wf-XML interface. Hopefully, this article has given you an un-

derstanding of the concepts, and a strategy for quickly implementing the necessary interactions. Levels 1 and 2 are sufficient for an interoperability demonstration that should always be done first. After the demonstration, an evaluation of the approach should be performed. Armed with the knowledge of how the protocol fits with your system, the implementation of the rest of the protocol should be relatively straightforward.

It is worth reflecting on the value of implementing the standard. A standard asynchronous service linking tool can browse to your service; it can pick up the details of what variables are expected to be sent to in upon starting, and also find out the details of what values will be returned at the end; using this, it can allow the user to map into and out of these structures; then the protocol allow the service to be started. At any point in time, this external program can ask for the status of your service. When anything changes in your process, you can send a message keeping that program up to date. Finally, when your service is completed, you can send the final notification.

Without a standard, every asynchronous service would do these things differently. Linking up would be not just be a matter of mapping the data values, but mapping the 8 or 9 key operations. Without the standard, the semantics of those operations may not be exactly the same. Small interface routines would have to be written to translate the meanings correctly. This is the approach that some vendors are promoting. In those cases, they simply stop at the point that they have a complete description of their interface, on the assumption that a programmer will be around to write the interface logic. ASAP and Wf-XML are designed to be used by non-programmers. This is the key.

Please look to the ASAP site and the WfMC for tools for the testing and support. ASAP can be found on the OASIS website at:

http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=asap

Wf-XML can be found at WfMC, and at the following discussion forum:

http://www.wfmc.org/
http://www.workflow-research.de/Forums/
        index.php?s=1cbbcce3cc5da8944dd340191c817831&act=SF&f=7

ACKNOWLEDGEMENTS