Proposed Changes to BPML Working Draft

August 16, 2002

Authors:

Assaf Arkin, Intalio

Status of this Document

This document presents a list of proposed changes to the June 24 BPML 1.0 working draft. These changes are based on feedback provided following the release of the June 24 working draft, as well as previous feedback that we were unable to incorporate into the June 24 working draft.

These changes are intended to complete the BPML specification in the scope of addressing executable business processes.

Comments on this document and discussions of this document should be sent to bpmi-dev@bpmi.org.

Table of Contents

1. Global Model and Connectors

The global model is used to define the manner in which multiple inter-connected but independent processes interact with each other. Such processes are loosely coupled and typically deployed in different locations. Most commonly the global model is used to illustrate an end-to-end process involving multiple partners and systems.

The base requirements for representing a global model are currently met by the WSCI 1.0 specification. For the purpose of modeling an end-to-end process, one could use the WSCI definition of a global model.

We have outlined four additional requirements from a global model:

- The explicit definition of roles within the model, such that roles can be referred to from external definitions.
- Composition of global models such that global models can be built incrementally and combined in various ways.
- Partial (i.e. open) global models, such that incomplete definitions can be saved and exchanged between applications.
- Support for multiple process languages including WSCI, BPML and others.

Due to time constraints, these requirements cannot be addressed within the scope of BPML 1.0. We may consider a separate spec to specifically address this issue. Our recommendation at this point would be to remove the global model definition from the BPML 1.0 specification and refer implementers to the WSCI 1.0 specification.

Connectors are used by the global model for expressing inter-process communication. Within a context, they are used as a mechanism for intra-process communication.

The ability to model and perform intra-process communication is essential in order to model synchronization in parallel flows, support complex compositions of processes, and support complex transitions between activities.

An alternative to the use of connectors is the introduction of signals (see elsewhere in this document) and two new primitives for raising a signal and synchronizing on a signal. This model has two added benefits. First, it provides a clear distinction between inter- and intra- process communication, such that the use of ports is never confused. Second, it provides better facilities for intra-process communication, such as support for multiple consumers and producers, and deadlock avoidance.

The introduction of signals will remove the dependency on connectors and allow us to remove section 7 from the BPML 1.0 specification.

2. Process identity

In order to distinguish between process instances derived from the same process definition, each instance requires a unique identifier value. The identifier is used for the purpose of correlating incoming and outgoing messages, for instance query and management, and even for advanced control flows.

To prevent conflicts, instance identifiers must be unique across all instances. UUIDs are a form of identifier that is universally unique across space and time and is cost effective to generate. In some cases it is desired for the identifier to take a simpler form than that of a UUID, e.g. a sequential order number, or a social security number.

Currently BPML does not mandate what form of identifier is used, but recommends the use of UUIDs. We would like to allow a process definition to indicate the name of the property holding the instance identifier (default being bpml:instance), the type of the identifier value (default being xsd:string) and the key space in which the identifier is known to be unique (default being UUID). This element is optional, and if absent, the default behavior is assumed.

The syntax for this element is given as:

```
<identity
  name = QName : bpml:instace
  type = QName : xsd:string
  keyspace = anyURI : uuid-ns>
  Content: (documentation?)
</identity>
```

This change affects sections 3.2.3, 3.2.5 and 4.5.

3. Property definitions

Currently a property can be defined to have a type that is an XML Schema element, simple type or complex type.

There is no clarification in the case that a property is instantiated from a message. For that purpose, one must allow a property to be defined with the same structure as the message by referencing the WSDL message definition.

In order to support the communication of ports, e.g. for the purpose of allowing callbacks in asynchronous message exchanges, one must be able to define a property that holds an end-point URI. Type checking is only possible if the property is defined to hold end-point URIs for ports of a specific type, by referencing the WSDL port type definition.

The property and parameter definitions will change to include two new attributes: message (QName) and portType (QName). This change affects section 3.2.4 and section 4.2.1.

In addition, we would like to remove implicit property definition from the BPML specification. Although in many cases implicit property definition saves the need to explicitly define properties, it does not allow for strict type checking and can easily lead to abuse, as properties are added/removed by changing external definitions (e.g. selectors) without proper indication or validation.

4. Support for interface

The current draft does not indicate what it entails to declare that a process supports a WSCI interface.

This change affects section 3.2.3.

5. Functions

Currently the specification lists BPML XPath functions but does not indicate why they are required. A short introduction is called for. Minor corrections have been pointed out for the section dealing with instance functions. Additional functions must be introduced in order to support signals.

The introduction affects section 4.4.

Fixes are applied to section 4.6.

New section for signals will be added

6. Renaming output parameters

A shortcoming of the call activity is in forcing output parameters to be assigned to properties with the same name in the context of these activities. Where reusable processes are called, these names may conflict with similarly named properties in the activity's context, requiring redundant local context definitions and assignments.

The solution is to add a simple means for renaming output parameters and assigning them to the named property. This will be supported for both call and compensate activity. In addition, it can be supported for the action activity in order to rename properties obtained from selectors, or even to indicate which selectors are used.

This change affects sections 6.1.1, 6.1.3, 6.4 and 6.6.

7. Faults

The current draft is not complete with regards to the handling of faults. It does not cover how the content of a fault is produced by the fault activity, how the content of a fault is assigned to a property upon receipt, how BPML faults map to WSDL fault, and how fault codes relate to each other.

These clarifications will affect sections 6.9, 5.1.2, 5.2.5 and 4.2.3.

8. Signals

Signals are used for intra-process communication. Signals provide an alternative mechanism to the use of WSDL operations with additional semantics that may not be expressed directly through inter-process connectors.

In the current draft, WSDL operations are used for both inter-process and intra-process communication, with the exception that intra-process communication occurs through connectors and such ports are not publicly exposed. There is no easy way to determine which operations occur on internal ports and handled by context connectors, and which operations occur on public ports and must be exposed as WSDL services.

The introduction of signals creates a distinct separation between messages that are exchanged with independent services, defined using WSDL operations and performed using BPML actions, and between messages that are exchanged between activities within a process flow, removing the confusion. Inter-process communication will always be defined using WSDL ports, while intra-process communication will always be defined using signals.

In addition, because signals are exchanged only within a well defined context, additional semantics can be introduced that are essential for modeling intra-process communication, such as publish/subscribe semantics, detection and handling of blocked activities, and synchronization conditions.

A signal is a form of message that can only be sent within a particular context. Signals cannot cross context barriers, a signal that is defined within a context cannot be communicated outside that context. Signals are never used for loosely coupled interactions.

A signal has a name that distinguishes it from other signals defined in the same context. A signal can be used to carry data or to identify the signal out or multiple signals with the same name. The structure of a signal is defined it as a composition of one or more message parts, using the WSDL abstract message definition. The syntax for a signal definition is:

```
<signal
  name = NCName
  message = QName>
  Content: (documentation?)
</signal>
```

There are two operations that can be performed on a signal. These are not defined as WSDL operations, but rather as two BPML activities for simplification and additional semantics. These activities are called raise and synch.

The synch activity blocks until the signal is raised before proceeding. This is the most generic form of synchronization. The raise activity is used to raise a signal and, if the signal carries any data, associate values with the signal. A signal can be raised any number of time. The raise activity has the following syntax:

```
<raise
  name = NCName
  signal = NCName>
  Content: (documentation?, timeConstraint?, output*)
</raise>
```

Synchronization is a blocking activity that waits for the signal to be raised one (or more) times. The synch activity will proceed once the signal has been raised, whether the signal is raised before or during the execution of this activity.

A condition can be used to qualify the signal being raised. If the signal is raised and the condition evaluates to false, the synch activity will not complete but will continue blocking until the signal is raised again. The condition is evaluated each time the signal is raised to determine if the synch activity can respond to that signal.

If the signal carries any data, the data can be accessed in the condition, e.g. to identify a particular signal out of a set of signals with the same name. If the synch activity responds to the signal, it will assign the content of the signal to properties in its context, allowing the data to be used by downstream activities.

The synch activity can wait for the signal to be raised once, or a specified number of time. For example, if ten parallel flows were initiated and the synch activity waits for the first five to complete, it will do so by waiting for the signal to be raised five times before completing.

The synch activity has the following syntax:

```
<synch
  name = NCName
  signal = NCName
  count = ( QName | #one ) : #one
  condition = XPath>
  Content: (documentation?, input*)
```

```
</synch>
```

At some points a process may need to synchronize with two distinct signals, either by combination or alternatively. The process may want to synchronize on a signal under a time constraint, or alternative to a message or fault. For that purpose, the choice activity can be used with the introduction of a new event handler called onSynch. The syntax for this event handler is:

```
<onSynch>
  Content: (documentation?, synch, {activity set})
</onSynch>
```

The onSynch event handler can also be used in exception handling, e.g. a signal can be used to abruptly terminate a child or parent process. Such signals can be standardized as part of the BPML specification.

The synch activity and onSynch event handler can also be used for instantiation of a process upon the raising of a signal. As with messages, the all and choice activities can be used to define instantiation from a combination of signals. The instantiation type for such a process would be 'signal'. Signals are always generated from within a context and not directly from external entities. A process is instantiated once for applicable signal or set of signals.

Relationship Between Synch and Join

Raising a signal in order to instantiate a process is an alternative way to using the spawn activity. However, we believe that the spawn activity is more straightforward to use and combines well with the call activity.

Synching on a signal in order to wait for the completion of a process is an alternative to the join activity. In essence, the join activity performs synchronization by waiting for indication that a process instance has completed. We can define that an instantiated process retains a signal in the context from which it was instantiated and that such a signal can be used to synchronize with completion of the process.

Since the synch activity can be used to synchronize with a process while a process is doing work as well as upon completion, and can be used in event handlers and all/choice activities, it provides a more generic and capable mechanism. In fact, it replaces some of the extensions viewed for the join activity to support these capabilities. We may consider removing the join activity altogether.

Time-based Signal Raising

There are many cases where events are triggered at a particular time instant. Such events may be alternative to other activities in the normal control flow of the process, and can be supported by use of the choice activity and onTimeout event handler. Such events may be alternative to activities in the normal control flow, and can be supported by the exception construct and onTimeout event handler. It is also possible that such events could occur in parallel without disrupting the normal control flow.

One way to support such time-based events is to schedule the spawning of a nested process to a particular point in time. If the nested process must be spawned before all work in the context is completed, then the wait activity can be used followed by the spawn activity. If the nested process must be spawned regardless of when work in the context completes, the nested process can be spawned and the wait activity used to facilitate delay.

If a nested process must be spawned only while some work is being done in the context (e.g. to report significant delay in processing a request), then the use of the wait activity becomes impossible. Three alternative solutions emerge:

1. Use parallel flows, one of which waits conditionally for a timeout or a signal indicating completion of work in the context. If the first occurs, the process is spawned, otherwise, the process is never spawned.

2. Allow a process to be spawned with a time constraint by adding time constraints to the spawn activity. This is fairly easy to accomplish.
3. Allow a signal to be raised with a time constraint by adding time constraints to the raise activity, and use the signal to raise the process using instantiation of type signal.

We prefer the third approach since it can be put to use in multiple scenarios.

Raise/Synch Semantics and Blocking

A signal can be raised without having a synch activity to synchronize on it. This is an acceptable behavior, although it may seem redundant, it allows us to define additional signals that are raised not by BPML activities and may or may not be synchronized with, e.g. signals to indicate termination of a process, completion of transaction, etc.

Signals may be raised multiple times and may be synchronized with multiple times. Essentially, this defines a broadcast mechanism that sends the signal to multiple consumers, and also allows multiple producers to raise the same signal. For example, a kill signal can be raised by one of several activities in the parent process and synchronized with by the exception event handler of multiple nested processes.

It is not clear whether one needs additional semantics for the case of a single consumer or single producer, or whether one can use a form of analysis to determine the number of consumers and producers for a signal and match them, e.g. based on the pi-calculus model.

An activity may wait for a signal that is never raised. This may be part of the design of a process, where a signal is optionally raised, and the activity must have a way of detecting that the signal will not be raised during its execution. This is part of the semantics that are not readily available if WSDL operations were used.

We propose a simple algorithm that can be used to detect if an activity is blocking, waiting for a signal that is known to not be raised, including the possibility of a deadlock situation. Such an algorithm is easy and efficient to implement if one considers all that all possible transitions between activities are based on well specified BPML activities or the raising and synchronization of signals. If an activity expects a signal to be raised and said signal will not be raised in that context, then an exception is triggered and the activity can use the onFault event handler to address this case.

Misc

Signals crossing transaction boundaries are subject to transaction control. If raised from within an atomic transaction, the signal cannot be synchronized with outside that transaction until the transaction commits.

Signals between top-level processes are defined at the package level. They can only be used between processes that interact through signals and the spawn/call activities, as such processes are deployed within the same system and are tightly coupled. We have yet to determine how such a restriction can be defined.

Support for signal for process instantiation affects sections 3.2.1 and 3.2.3.

Support for defining signals in contexts affects sections 4.1.1 and 4.1.2.

Support for onSynch event handler affects sections 5.1, 6.5 and requires new section 5.1.6.

We will have to consider whether the join activity is required, and possibly modify the text for spawn and join activities to reflect them being a simplification of the signal mechanism.

9. Misc

- Fix to selector definition required in section 4.2.3.

- New section listing all BPML faults and their message structure.
- New section listing all pre-defined BPML messages and signals (non at the moment).
- New section including the BPML schema inside the BPML specification.
- Example to illustrate the use of BPML. Not all constructs will be covered by this example.
- How do we represent the contents of a message (single and multi part) as the value of a property?
- Need to clarify where instance property for transaction is being held. How do we access compensation of a transactional process that has completed? How do we access such compensation if a nested process from its parent? How do we access such compensation if we called/spawned that process?
- Make sure that a process can be spawned from within a transaction and will be spawned as part of the transaction, or can be spawned conditional to completion of the transaction.
-

10. Example