Extending BPEL with transitions that can loop

ActiveVOS linksAreTransitions BPEL Extension

AN ACTIVE ENDPOINTS PAPER

AUTHOR: DR MICHAEL ROWLEY

© 2009 Active Endpoints Inc. ActiveVOS is a trademark of Active Endpoints, Inc. All other company and product names are the property of their respective owners.
## Content

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Why BPEL doesn’t allow cycles</td>
<td>3</td>
</tr>
<tr>
<td>These goals are the reason that backward links were disallowed</td>
<td>3</td>
</tr>
<tr>
<td>Proposal</td>
<td>4</td>
</tr>
<tr>
<td>Semantics of Transitions</td>
<td>6</td>
</tr>
<tr>
<td>Mutually Exclusive Conditional Outbound Transitions</td>
<td>8</td>
</tr>
<tr>
<td>Backward compatibility with BPEL 2.0</td>
<td>9</td>
</tr>
<tr>
<td>Compatibility with BPMN 2.0</td>
<td>9</td>
</tr>
<tr>
<td>About Active Endpoints</td>
<td>11</td>
</tr>
</tbody>
</table>
Introduction

This document describes an extension to WS-BPEL 2.0 to support transitions, which are similar to BPEL’s links, except that unlike links the directed graph can include cycles. With this extension, BPEL supports the free-form control flow that is allowed by the BPMN 2.0 notation, without taking on all of the complexity of BPMN 2.0 execution language. People familiar with BPMN should find that BPEL with this extension results in an environment where models execute with the semantics that they would expect.

Why BPEL doesn’t allow cycles

There are two goals that motivate most of BPEL's control flow constraints:

1) Prevent infinite synchronization waits

2) Prevent multiple simultaneous executions of a single activity within a single process instance.

There are 2 constructs which violate the second goal -- parallel for-each and scope event handlers -- but they are designed to create walled-off areas that avoid leaking their multiple execution semantics into other areas of the process (e.g. you can't create a link that crosses one of these boundaries).

These goals are the reason that backward links were disallowed.

Consider the following process:
With existing BPEL link semantics, activities can only execute after their incoming link status has been determined (links start in an "unset" state). So, if the back link was not prevented by a static check, it would cause the process to deadlock, since Activity1 would be waiting for Activity3 and Activity3 would be waiting on Activity1.

If the links had BPMN-style semantics, where the firing of a transition causes the target activity to execute, then Activity2 could end up having 2 executions going at the same time. BPMN allows this, but I would consider it to be very bad practice to actually do it. The problem is that once you have two executions (two tokens) on a single path there is no way to join those tokens with a gateway (i.e. wait for them both to finish). The only way to wait for multiple tokens to complete on a single path is to wait for the end of an entire embedded sub-process.

The other downside of allowing multiple tokens on a single path is that it is really hard for users to understand what will happen, and for operators to understand what is happening or what has happened.

And finally, when multiple tokens are allowed, the merge semantics for an AND merge can easily wait indefinitely. When all the tokens aren’t at the merge, there usually is no way to tell that a token won’t eventually arrive on the inbound transitions that doesn’t currently have tokens. This can cause embedded subprocesses to never complete, since they only complete when all tokens are consumed.

This proposal provides the ability to transition backward in a process, but with restrictions that prevent multiple tokens on the same path. The no-cyclic-links restriction of BPEL is overly restrictive for achieving this goal.

Proposal

The proposed extension slightly modifies the semantics for links. We could call the resulting construct a "sequence flow" to align with BPMN, but since "flow" means something different for us and our users, we will call them "transitions".

When the source of a transition completes, then if the transition condition evaluates to true (or there is no condition) then the transition fires and if the target of the activity is an activity with no other incoming transitions, the activity will execute.
Almost every legal control flow with BPEL links will have the same behavior with transition semantics (more on that later). The way that links are made to behave like transitions is with a process-level extension attribute of ext:linksAreTransitions="true" (where the extension is mustUnderstand="yes").

The graph of activities and transitions may be cyclic, but there will be a restriction that there cannot be a transition whose source is an activity within one thread of a fork whose target is an activity that precedes the fork. This will be defined more formally below. The picture shown in the previous section of this document would not be allowed in this restriction. Including the fork more explicitly, as shown below, makes the illegal pattern a little clearer.

Definition: Thread. In the proposal below, a thread can be defined using BPEL constructs as the immediate child of a flow and all of the activities under it (if the activity is a container). This is sometimes called a "thread of control" or a "control path". It is technically different from a Java or operating system thread, although it does bear some similarities to those constructs.
Definition: Fork. A fork is any activity, gateway or event that has multiple outgoing transitions, where more than one of the outgoing transitions may fire. Parallel for-each and event handlers are also considered to be forks.

Fork Restriction: For any fork (F) in the process, any transition path from F that leads back to F must pass through a node where all of the threads created at F have been provably remerged into a single thread.

We don't believe that people will be sorry to see this restriction. People don't purposely create processes that have multiple simultaneous executions of the same activity (other than in explicit parallel constructs, such as parallel for-each and event handlers).

Semantics of Transitions

When the source of a transition T1 completes without throwing an exception, the transition condition of T1 is evaluated and if it is true then the transition gets a token. If the target of the transition has only one inbound transition, then the token is moved to the target node and the target node is ready to execute.

For the purposes of execution semantics, any kind of control flow into a node is treated as a transition into that node. This means that transitions are introduced by BPEL’s constructs such as sequence, if, while, repeat-until, for-each, handlers, etc, as well as the AEI extensions of break and continue.

Scopes may also have outbound transitions that are tied to boundary events (including faults). These event transitions get a token when the boundary event occurs while the scope is executing. Prior to placing a token on the outgoing transition, the scope is terminated. Note that this is not currently possible in BPEL for events, but it is exceptionally useful to be able to say that any time some event occurs during the execution of a scope, the scope should stop and control flow should be directed to a part of the process.

If the target node of a transition has more than one incoming transition then it is considered to be a merge node.
A merge node may include a BPEL joinCondition, which is an XPath that includes Boolean variables representing named incoming transitions (BPEL control flow constructs don’t provide named transitions). The joinCondition is evaluated when at least one incoming transition has a token and when there are no tokens upstream of the merge node other than the tokens on its incoming transitions. A token is upstream of the merge node if there is a valid directed path from the token to the merge node through any valid BPEL control flow construct. Note that, due to the fork restriction above, no token can exist upstream of a transition that already has a token. The joinCondition can be used to achieve the semantics of the various BPMN gateway types, with some restrictions, which will be discussed in detail below.

OR merges can’t fail, but AND merges and Complex merges can fail when some of their inbound transitions don’t have tokens and never will get a token. Merges that fail will consume any inbound tokens. This has the same effect as BPEL’s dead-path elimination, due to the fact that merge condition evaluation starts when there are no upstream tokens.

Having failed merges consume inbound tokens is important for the following use case:
In this case the process can take either or both of two paths: A & B, but it also has an activity “Coordinate” that should only happen when both A & B happen. In this scenario, the parallel merge before the “Coordinate” activity will sometimes get only one token, but it should not prevent the last inclusive merge from ever happening, which is what BPMN semantics would do. If we removed the arrow from “Coordinate” to the inclusive merge and replaced it with an end event, that wouldn’t help BPMN, since the subprocess that contains all of this can only complete when all of the tokens have been consumed, and there would always be a token on one transition into the parallel merge.

**Mutually Exclusive Conditional Outbound Transitions**

The fork restriction for transitions requires that you can tell where there are forks within a process, but how do you tell if conditional transitions out of an activity cause a fork? If the transition conditions aren’t mutually exclusive, it is a fork, but it can be impossible to determine that expressions are mutually exclusive. For example, one outbound transition might have a condition of “$x/foo/@bar” while another outbound transition has a condition of “$x/foo/@baz”. It could be that the user knows that “@bar” and “@baz” can never both be on <foo>, but the system doesn’t know that.

There is a new property on activities called "mutexTransitions", or "mutually exclusive outbound transitions" in its long form. Since this property affects the semantics of the control flow, it should have some visual representation (e.g. filled in diamonds at the beginning of transitions).

The property is false by default. If the user gets an error because the system finds a violation of the fork restriction due to a node that isn’t really a fork, then the user can turn on the mutexTransition property to remove the error.

At runtime, if an activity marked with mutexTransitions has more than one transition condition evaluate to true, then a fault is thrown (which will typically suspend the process).
Backward compatibility with BPEL 2.0

For any valid BPEL process, transitions have the same semantics as links, as long as the following are also true:

1) suppressJoinFailure is true
2) conditional links don't have target activities that are in the middle of a sequence.
3) No join conditions exist that negate all inbound links.

Because the process is valid, we also know that the links do not form a cycle, so we don't have to worry about backward compatibility for processes with cyclic links.

The conditions above are the common case. Addressing each in turn:

1) We have yet to see a case where someone wanted a fault to be thrown when a join condition was not met. Note that throwing a fault on a failed synchronization is not even one of the patterns in the Workflow patterns page (and there are several patterns that neither BPMN nor BPEL support).
2) If you have a conditional link (T) that targets B in a sequence of A – B – C, then if the condition on T evaluates to false B will be skipped but C will still run! There is no equivalent semantics in BPMN (or any other workflow language we’re aware of), and if anyone did this, they would be very surprised at the resulting semantics.

Compatibility with BPMN 2.0

Transition merge semantics align very well with BPMN’s merge semantics for inclusive gateways. For parallel gateways, there is a slight difference, since BPMN would wait indefinitely, while we will fail the merge and then swallow the token.

It also aligns fairly well with complex gateways, but in BPMN a complex gateway has a joinCondition where the transitions variables are ints that represent a count of the tokens on each incoming transition. In our case, an incoming transition can have at most one token, so the transition variables can be Booleans.

There is a difference with exclusive gateways. In BPMN an exclusive gateway is an “uncontrolled merge”, so if multiple inbound transitions have tokens they will all get propagated through the gateway. However,
exclusive gateways are almost always used in situations where there can only be one token upstream of the merge, in which case the transition semantics given above match exactly.

People should not mind this difference (in fact we expect them to appreciate it), since people don’t purposely create processes so that multiple tokens will merge down to two tokens on the same path. If they create such processes by accident, it can be a real pain to find and fix.
About Active Endpoints

Active Endpoints (www.activevos.com) is the leading developer of visual orchestration systems. VOS empowers line of business project teams to create applications using services and industry standards, making their businesses more agile and effective. Active Endpoints’ ActiveVOS promotes mass adoption of SOA-enabled applications by focusing on accelerating project delivery time with a standards-based, easy to use system. Active Endpoints is headquartered in Waltham, MA with development facilities in Shelton, CT.

To find out how Active Endpoints can help your business, visit http://www.activevos.com, call +1 781 547 2900 and press 1 for Sales, or email us at info@activevos.com.