Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation.

Francisco Curbera, Rania Khalaf, William A. Nagy, and Sanjiva Weerawarana
IBM T.J. Watson Research Center

BPEL4WS: Workflows and Service Components

BPEL4WS [1], also known as BPEL for short and soon to renamed WS-BPEL, provides the most complete realization to date of the workflow execution model in the context of a service-oriented architecture. As opposed to other process language specifications, BPEL builds natively on the Web service component model defined by the Web Services Description Language (WSDL), and models every activity in the process in terms of Web services interactions. Service-oriented architectures introduce a set of very distinctive abstractions that result in significant modifications of the basic workflow model, such as is described in [2]. In particular, the "software as a service" approach results in a componentized view of software applications, and the application of workflow as a component composition mechanism. BPEL naturally supports a multi-party interaction model, and BPEL processes present themselves as service components to other Web services.

BPEL results from the innovative merge of two approaches to workflow, as exemplified by the process algebraic view of XLANG [3] and the graph oriented view of WSFL [4], resulting in a sophisticated execution model that inherits the power of structured programming with the directness and flexibility of graph models. BPEL includes extensive support for exception handling, which is at the core of the blending of the algebraic and graph execution models, see [5].

BPEL's new model of process-oriented composition that creates numerous challenges for implementers. This paper describes the architecture of the BPWS4J [6] implementation of BPEL, and shows how it addresses these challenges. The BPEL specification implemented in BPWS4J is version 1.1, which has since been submitted to the OASIS consortium for standardization. The deliverable will be renamed WS-BPEL and will be versioned 2.0.

This paper discusses the implementation challenges that BPEL execution semantics present to implementers, as well as the overall architecture and design solutions that the BPWS4J provides.

An Overview of the BPEL Language

BPEL process expresses its relationship with its partners using typed connectors known as 'partnerLinks'. A BPEL process can invoke their operations using the <invoke> activities. Additionally, the process itself exposes one or more Web service interfaces that

it provides to its partners. PartnerLinks can be one-sided or two-sided. A one-sided partnerLink is one in which either the process acts as a pure client of the partner's service or the partner acts as a pure client of the process. A two-sided partnerLink is one where the process both invokes the partner's service, and itself gets invoked by that partner. Therefore, BPEL provides a recursive model for composing Web services.

The unit of execution in BPEL is the 'activity'. There are two kinds of activities: (1) Simple activities, such as those for invoking Web services (<invoke>), receiving and replying to the operations that the process itself exposes (<receive> and <reply>), throwing faults (<fault>), waiting (<wait>) and so on. (2) Structured activities that contain other activities nested within them on which they impose control and to which they provide common properties. Examples of complex activities include strict sequencing (<sequence>), parallelism (<flow>), nondeterministic choice (<pick>), and others. The <scope> is an important BPEL structured activity that provides fault handlers, event handlers, compensation handlers and scoped variable definitions to the activities nested with in. In addition to the control imposed by structured activities, BPEL also allows control links to connect activities at any level of nesting within a <flow>. An activity that is the target of multiple links will also have a 'joinCondition'. Once all incoming links have been evaluated and the activity has control from its parent activity, then the 'joinCondition' gets evaluated. The activity can start running if the joinCondition evaluates to true. If the 'joinCondition' fails, a 'joinFailure' fault is thrown by the activity.

BPEL's exception handling is reminiscent of Java's exception handling mechanism: An exception thrown from an activity is thrown up the scope hierarchy, disabling all other activities in each scope as it goes, until a matching fault handler is found. Additionally, a set of links will be fired with a negative value. These are any links leaving the faulting activity and any links leaving disabled activities in the scope to which the fault has been thrown whose targets are outside of that scope.

BPEL allows multiple instances of the same process definition to run simultaneously, providing a 'correlation' mechanism to handle routing of messages to the correct instances.

BPWS4J Components

The BPWS4J implementation includes three main components:

1. A runtime process model together with a corresponding parser and a writer.
2. A BPEL process container, which provides a runtime and deployment environment for BPEL processes.
3. An event driven flow engine/process interpreter that executes BPEL within the BPWS4J runtime environment.

The runtime process model provides an in-memory process representation of a BPEL process, and contains full information about the process definition. Here we will not pay

special attention to this component *per se*, except to indicate its central importance in supporting the operation of the BPEL container and the BPEL interpreter. The runtime process model is used by the BPEL container to direct the deployment of processes, as a basic template to create new process instances, and as a reference to route incoming messages to specific process instances. The process interpreter also uses the model to control the execution of a process. The process container and the process interpreter will be reviewed in more detail in the following sections. Before that, we will take a look at the overall BPWS4J runtime architecture.

Runtime Architecture Overview

The BPWS4J runtime architecture is sketched in Figure 1.

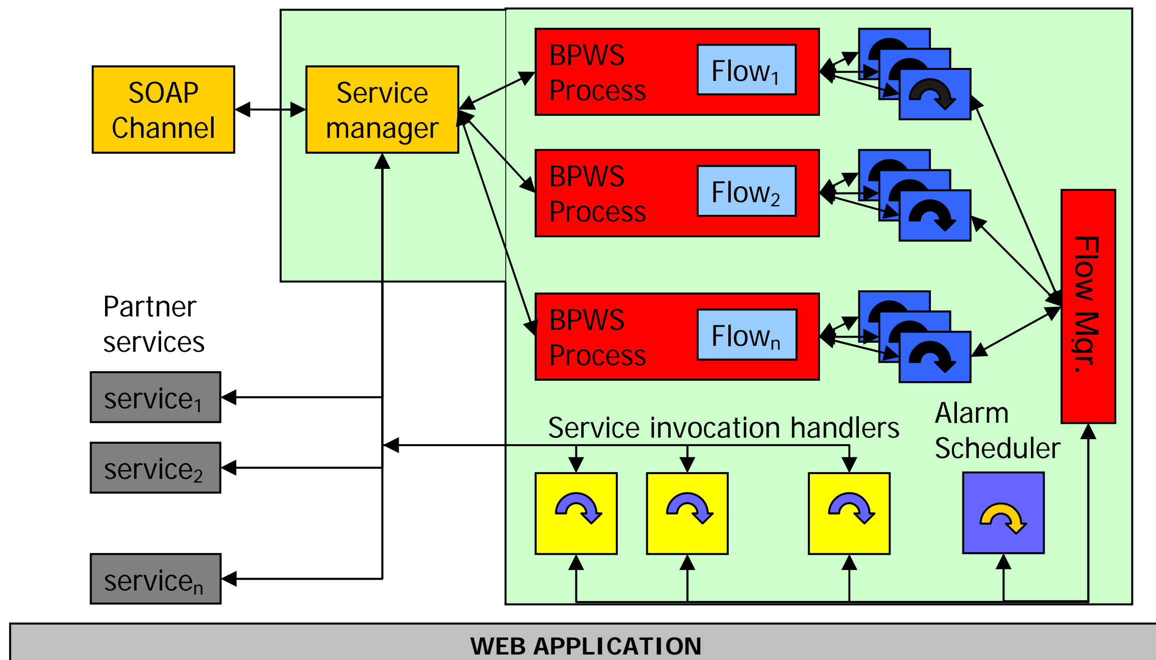

Figure 1: Runtime architecture of BPEL implementation. Each 'BPWSProcess' is a single process definition. On the right are running instances of that process. Messages are handled by the flow manager and the service manager.

Each individual BPEL process model is deployed in the BPWS4J engine as a separate Web application, reflecting the fact that SOAP over HTTP is the supported incoming invocation protocol (with several different SOAP over HTTP implementations being supported.) All instances of a process model are thus handled by the same Web application. The process container serves incoming and outgoing requests. The service manager dispatches incoming invocations to individual flow instances of deployed processes, and sends outgoing ones to process instance partners. The identification of partner services (i.e. services that are able to send messages to a process instance or received them from it) is under full control of the service manager as well. Flow instances, on the other hand, execute under the control of the flow manager, which can request different kinds of services from the process container (e.g. service invocation handlers and the alarm scheduler.)

The BPEL Process Container

We have seen in the prior section that the BPEL process container provides several crucial services to assist in the execution of BPEL processes. Three of these services merit special attention.

1. Process deployment and binding.
2. Process instance lifecycle management.
3. Incoming message routing and service invocation support.

Deployment

BPEL process definitions specify the set of partners that interact with the process at the abstract interface level only (via WSDL port types.) The BPEL language itself does not determine how each of these partner services is actually bound (i.e. it does not specify which service instantiation or what protocol should be used in the interaction.) The goal of this design is to enable the reuse of processes in different deployment environments and the use of different interaction technologies. Only in the case of application level dynamic binding (i.e. when service references are explicitly sent to the process) can the specification be fully completed from within a process. The deployment of a BPEL process consequently requires the provision of the specific identities of partner services or the specification of the runtime strategy that should be followed to discover them.

The BPWS4J engine inspects the in-memory compiled process model to determine which partners need to be bound and which ones don't, based on whether the interaction is initiated by the process or by the partner service. Only the former needs to be resolved at deployment time. At this time, BPWS4J supports only the static binding of these partners, but additional binding strategies are currently being developed.

Lifecycle

BPEL provides an "implicit lifecycle" model for process instances. Process instances are created by the reception of specially designated application level messages (i.e. "startable" invocations,) and are destroyed when the last activity of the process instance completes its execution. In addition, process instances are identified by the value of certain application specific message fields (correlation sets), rather than explicit process identifier tokens.

Supporting the implicit lifecycle model requires that the process container (the service manager unit in particular) decide, upon receiving an application message, whether the message needs to be routed to an existing process instance or that a new instance should be created. The service manager unit of the process container implements a complex decision algorithm which takes into account the operation being invoked by the message, the process definition settings (to determine if a "startable" operation is being targeted), and the possible values of correlation fields present in the message (to check for possible matches to existing instances.) When the appropriate conditions are met, a new process

instance, which will be used to hold instance specific execution state, is created by cloning an instance of the in-memory runtime process model.

Message routing

The routing of incoming messages is the process by which correlation values are extracted from incoming messages and, if present, used to identify the target process instance. The BPEL correlation mechanism allows a single process instance to be identified by more than one correlation set; moreover, each use of a single operation may be associated to different correlation sets. The logic of the matching algorithm is for this reason particularly complex, requiring the service manager to select and test all correlation sets that might be attached to message receive activities of a specific Web service operation. A set of possible error conditions arise from the flexibility of the BPEL correlation mechanisms, most of which are not documented in the BPEL specification, since it carefully limits itself to defining the behavior of individual process instances and avoids entering into the design and operation of BPEL engines. Some of these error conditions are: messages that don't match a process instance but which don't satisfy the conditions for staring a new instance (BPWS4J logs and drops these with no further action,) messages matching more than one process instance (BPWS4J will deliver the message to the first matching instance, ignoring the rest), etc.

Messages matching a particular instance are posted to the corresponding instance input queue. In addition, process instance invocation requests arising from the instance execution are sent to an invocation request queue and served by a pool of invocation threads. Outgoing invocations take advantage of the multi-protocol support provided by the Web Services Invocation Framework [7], allowing the process instances to make invocations using J2EE protocols such as IIOP, JMS or native Java calls, and potentially others (as more protocols are added as to in the WSIF project; see also [8]).

An Event Driven Process Interpreter

The event-driven process interpreter, here on referred to simply as the interpreter, is the piece of the runtime that executes an instance of a process model. As noted earlier, the interpreter is not aware of the outside world, and uses the Process Container for all of its external interactions.

A process model is compiled into a "runnable" process object, BPWS4JprocessRT, for each instance of that process model. The structure corresponds nearly one-on-one with the original process model, with the execution semantics of BPEL encoded in the different kinds of activities and constructs. As in the BPEL model, there are both simple and complex activities. Complex activities, including scopes, control the activities enclosed within them. Each instance runs with its own thread, with parallelism simulated through the event driven architecture. In order to understand the navigation model used, one must first take a look at the lifecycle of a BPWS4J activity.

Observe that BPWS4J does not support modification of a process model or process instances after deployment. Dynamic modification of workflow applications and even running workflow instances remains an active research area.

Activity Lifecycle

The event driven model described above relies on a well-defined model for activity lifecycles. Activities follow a simple activate-enable-run-complete cycle that can be terminated (disabled) at any point due to different forms of fault processing, resulting in the activity entering the disabled state. An activity is activated once it receives control from its enclosed complex activity, and becomes enabled once the status of each incoming link is known and the join condition is true. On the other hand, the existence of control loops in BPEL results in the need to "revive" activities according to the loop controlling conditions. The BPWS4J activity lifecycle is represented in Figure 2.

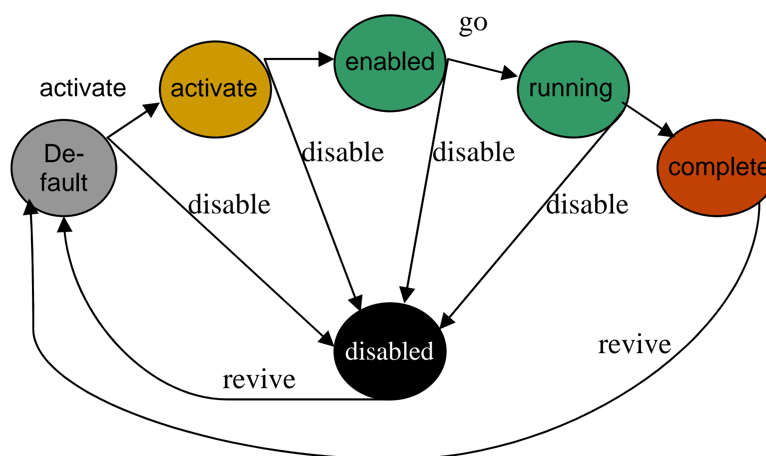

Figure 2: Lifecycle of an activity.

Events and Navigation

Once an instance is created, it runs and listens to events from the Process Container. Events in the interpreter exist at three levels: the process level, the scope level, and the complex activity level. Only the process exchanges events with the Process Container. These events include outgoing requests for invocations and incoming results, requests for and results of assignment values that are sent out to an XPath module, outgoing replies, and notifications of activity termination. A message queue is used to hold incoming messages until they can be routed to appropriate receive activities within that instance, with checks in place to ensure the conditions in the specification regarding conflicts in

matching messages to receive activities and in matching reply activities to their correct prior receive activity. At the scope level, two more events are handled: faults and links. The two are intertwined, as we shall explain further. Finally, at the complex activity level, events are listened from and fired to the enclosed activities to control them as shown in Figure 3.

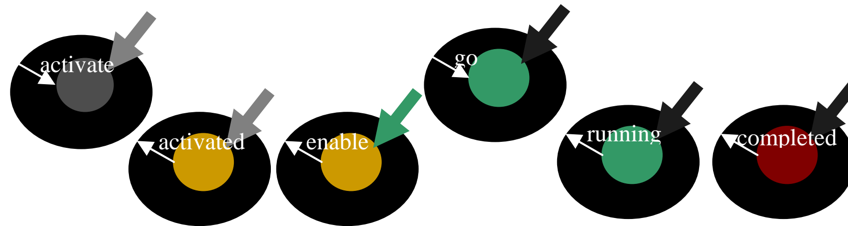

Figure 3: Lifecycle events between a complex activity (black) and a nested activity (colored circle). The thick arrows represent a link whose target is the nested activity. Note that disabled events are sent to the enclosing scope, not necessarily the containing structured activity. In addition, activities inside a loop receive "revive" events as well.

An activity starts running in BPWS4J once a combination of two things happens: an activity receives control from its enclosing activity and each of its incoming links has fired such that the join condition is true. This is the first step needed for BPWS4J to natively encode BPEL's combination of graph and structured process modeling. Activities related through links in BPEL are joined using event wiring, where an activity with outgoing links fires link events, and activities that are the target of links register as listeners for these events with the scopes of the source activities.

Scopes, Links, and Faults

BPEL describes how faults are thrown and handled while processing. One of the built-in faults is known as a "joinFailure" and occurs if the join condition of an activity evaluates to false once it has been activated and the status of each of its incoming links is known. In BPEL, the unit of fault handling is the scope. Scopes have fault handlers attached, and once a fault occurs it is thrown up the scope hierarchy until a scope is found with a handler for it. Once such a scope is found, the execution of all activities within it is stopped, all links leaving it fire negatively, and navigation is passed to the fault handler's activity. In BPWS4J, this corresponds to having the faulting activity fire a fault event that goes up the scope hierarchy, all activities in the handling scope are disabled which includes having any state variables reset and having their links fire negatively, and starting the activity within the fault handler.

Any activity with the "suppressJoinFailure" attribute set to true is compiled away such that it is wrapped in a scope that has an empty fault handler for join failures. The resulting behavior is noted to be equivalent in the BPEL specification. This is the second step needed for BPWS4J to natively support BPEL's combination of graph and structured process modeling: it is natively able to handle dead-path elimination, a key phenomenon

for graph-oriented process models that always synchronize on joins, through its handling of BPEL faults. For a discussion of dead-path elimination, and the usage of join failures and their suppression in BPEL, see [5].

BPWS4J and the Execution of Grid Applications

BPEL is a general purpose process definition language, and as such it is perfectly able to execute most types of workflow applications. At the same time, BPEL does not provide specific mechanisms for dealing with specific requirements of Grid workflows. These can be approached, however, by taking advantage of techniques developed outside the BPEL specification itself. We briefly discuss here some of these requirements. We refer the reader to [12] and [13] for a more detailed treatment of the use of BPEL in Grid scenarios.

The dynamic evolution f workflow models is a potentially critical requirement in Grid applications, where the conditions under which a workflow application or even a particular workflow instance executes and evolve very rapidly and require adaptive response. Examples of these are weather tracking applications. Dynamic evolution in Grid scenarios is essentially different from those traditional considered in the case business applications, where the need to modify a process is typically associated with very long running processes. Here, manual intervention is typically the preferred approach; processes are typically stopped, modified and restarted without significant impact on the overall execution of a running instance. In Grid scenarios, however, automatic adaptation of processes is usually the only possible approach, the time scales on which the changes need to be accomplished much shorter. Tight integration between planning and scheduling software with the workflow engine is required to provide this form of dynamic adaption. BPWS4J is not designed to allow dynamic adaption, but it could be enabled by architecting direct access to its in-memory representation of a process definition (though an appropriate API). A set of safeguards would need to be in place to ensure the safety of dynamic process instance manipulation.

The need to access massive data sets is another characteristic of Grid applications. BPEL4WS, built on top of the abstract application model of WSDL allows for efficient serialization to be plugged at deployment time into any process model. BPWS4J however, is not fully enabled to take advantage of this feature of the language. BPWS4J supports client side invocation using alternative serializations and protocols through the use of the Web Services Invocation Framework (although it is limited for this reason to available WSIF protocol providers); on the server side, however, BPWS4J currently only supports SOAP based invocations.

It is possible however to leverage the use of endpoint references to allow for passing data references between applications and processes. Endpoint references (see [9]) can be issued by the host of a large data set to provide allow applications to selectively retrieve data as needed for the execution of a process instance; the Web Services Resource Framework in particular supports XPath-like queries of data sets modeled as XML Schema typed resources. The use of these approaches is fully supported by the BPEL specification, and by the BPWS4J engine.

Some fundamental aspects of a production-level workflow engine, such as process instance level monitoring and scheduling are not addressed at this point in BPWS4J, which is not a production level engine but a prototype focused on capturing the execution semantics of the BPEL language.

Related Work

There is very little published material regarding the internals of workflow engine implementations. In the case of BPEL engines, there is no literature addressing the challenges imposed by the language semantics on engine implementers. For obvious reasons, most software vendors regard this information proprietary and confidential. The integration of a BPEL process engine into a Java Enterprise (J2EE) framework is by far the most common approach among middleware vendors; this approach is discussed at a high level, with focus on the programming model implications, in [10].

Academic efforts are more widely publicized, again with little or no coverage of the specifics requirements that BPEL execution semantics place on workflow engines. For a recent contribution to the literature see for example [11], where a new workflow language is introduced that attempts to support the complete set of workflow patterns previously identified by some of the authors, and a complete discussion of the design of an implementation of that language is presented. In the Grid space, a few reports on workflow engine implementation exist, typically focusing on high level design of capabilities and on programming model issues; implementation architecture details are often hard to find. An interesting taxonomy of Grid workflow systems is provided in [17]. Discussion of implementation issues can be found in [14], [15] and [16].

Conclusions and Future Work

The development of the BPWS4J engine has provided extremely valuable insight into the deployment and runtime operation management of BPEL processes. The great flexibility allowed by BPEL provides many challenges to developers and process authors.
The deployment mechanisms, which allow extremely varied and dynamic models for partner resolution remain an open area of research; we are already extending the static deployment model offered in BPWS4J to incorporate dynamic discovery of partner information.
The sophistication and flexibility of BPEL correlation as the basic mechanism for message routing and process instance identification is fully supported in BPWS4J, as is the full implicit lifecycle model which is rooted on correlation sets. Implicit in the adoption of the WS-Addressing [9] specification, as the mechanism to encode partner references in BPEL, is new series of challenges that can profoundly affect how message routing occurs in BPE4LWS engines. Our experiences implementing BPWS4J suggest the desirability of introducing a set of best practices or process patterns to enable a fully BPEL-compatible but simplified environment. Fewer runtime errors, greater efficiency and less ambiguity would be desirable properties to be gained from that effort.
BPWS4J leverages the multi-protocol invocation capabilities of WSIF, but still limits incoming invocations to the SOAP over HTTP channel (although both the Apache SOAP

and Apache Axis SOAP engines are supported.) Enabling true multi-protocol incoming channels remains an open issue.

Acknowledgement

The BPEL engine described in this paper was architected and created as a group effort by the authors of this paper, Matthew Duftler, and Nirmal Mukhi. We also need to thank Alexander Slominski for his insight into extension of the BPWS4J engine for dynamic Grid workflow scenarios.

References

1. T. Andrews et al., Business Process Execution Language for Web Services Version 1.1, available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
2. F. Leymann, D. Roller, Production Workflow, Prentice Hall International, 2000.
3. S. Thatte, XLANG, available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
4. F. Leymann et al. Web Services Flow Language (WSFL 1.0), available at http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
5. F. Curbera et al. “Exception handling in the BPEL4WS language”, in Proceedings of the International Conference on Business Process Management, BPM 2003, Eindhoven, The Netherlands, Springer LCNS 2678, pp 276-290, June 2003.
6. W. A. Nagy, R. Khalaf et al.BPWS4J, available at http://www.alphaworks.ibm.com/tech/bpws4j
7. Web Services invocation Framework (WSIF), available at http://ws.apache.org/wsif/
8. N. Mukhi et al. “Service-oriented composition in BPEL4WS”, in Proceedings of the Twelfth World Wide Web Conference, Budapest May 2003.
9. A. Bostworth et al. Web Services Addressing (WS-Addressing), available at http://www-106.ibm.com/developerworks/webservices/library/ws-add/
10. M. Kloppmann et al. “Business process choreography in WebSphere: Combining the power of BPEL and J2EE”, IBM Systems Journal, 43, 2, November 2004.
11. W.M.P. van der Aalst et al. “Design and Implementation of the YAWL System”, Lecture Notes in Computer Science, Volume 3084, pp. 142 – 159, 2004.
12. A. Slominski, “On Using BPEL Extensibility to Implement OGSI and WSRF Grid Workflows”, GGF10 Grid Workflow Workshop, March 2004, available at http://www.extreme.indiana.edu/groc/ggf10-ww/.
13. F. Leymann, “Grid Infrastructure and Beyond”, Invited talk, GGF10 Grid Workflow Workshop, March 2004, available at http://www.gridforum.org/Meetings/ggf10/presentations/GGF10_FrankLeymann.pdf.
14. J. Cao et al. “GridFlow: workflow management for grid computing”, in Proceedings 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2003), pp. 198 – 205, May 2003.

15. S Krishnan et al. "GSFL: A Workflow Framework for Grid Services", preprint ANL/MCS-P980-0802, Argonne National Laboratory available at http://users.sdsc.edu/~sriram/publications/gsfl.pdf.
16. J Yu et al. "A Novel Architecture for Realizing Grid Workflow using TupleSpaces", in Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing, pp. 119-128, November 2004
17. J Yu et al. "A Taxonomy of Workflow Management Systems for Grid Computing" available at http://arxiv.org/ftp/cs/papers/0503/0503025.pdf.