Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation.

Francisco Curbera and Rania Khalaf, IBM T.J. Watson Research Center

BPEL4WS: Workflows and Service Components

BPEL4WS [1] provides the most complete realization to date of the workflow execution model in the context of a service oriented architecture. Service oriented architectures introduce a set of very distinctive abstractions that result in significant modifications of the basic workflow model as outlined in pre-existing workflow model, see [2] for example. 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.

In addition, BPEL4WS results from the innovative merge between 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. BPEL4WS includes extensive support for exception handling, which is at the core of the blending of the algebraic and graph execution models, see [6].

The result of all of this is a new model of process oriented composition but also an important challenge to implementers. This position paper describes the architecture of the BPWS4J [5] implementation of BPEL4WS, and shows how it addresses this challenge.

BPWS4J Components

The BPWS4J implementation includes three main components:

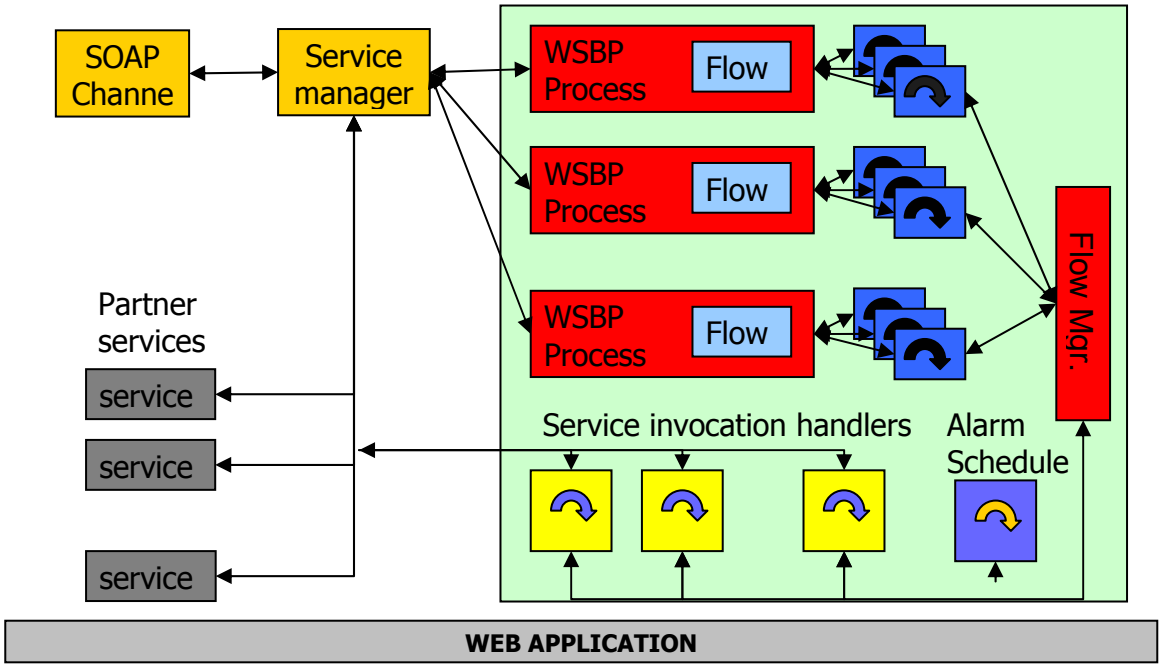
1. A runtime process model together with a corresponding parser and a writer.
2. A BPEL4WS process container, which provides a runtime and deployment environment for BPEL4WS processes.
3. An event driven flow engine or process interpreter which executes BPEL4WS within the BPWS4J runtime environment.

The runtime process model provides an in-memory process representation of a BPEL4WS 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 instantiate new process instances, and as a reference to route incoming messaged to specific process instances. It is also used by the process interpreter to control the execution of a process. The process container and the

process interpreter will be reviewed in more details in the following sections. First we will take a look at the BPWS4J runtime architecture.

Runtime Architecture Overview

The BPWS4J runtime architecture is sketched in Figure 1.



The execution of a BPEL4WS process is implemented as a Web application, reflecting the fact that SOAP over HTTP is the supported incoming invocation protocol (several different SOAP over HTTP implementations are supported, however.) All aspects of Web service interaction, both at deployment time and runtime are handled. Incoming and outgoing requests are served by the process container. The service manager dispatches incoming invocations to individual flow instances of deployed processes, or sends outgoing ones to process instance partners. The identification of partner services (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 (service invocation handlers and alarm scheduler.)

The BPEL Process Container

We have seen in the prior section that the BPEL process container provides several crucial services to assist 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 (WSDL port types.) The BPEL language itself does not determine how each of these partner services are actually bound, to what services or what protocols should be used in the interaction. The goal of this design is to enable reuse of processes in different deployment environments and the use of different interaction technologies. Only the case of application level dynamic binding (when service references are explicitly sent to the process) can be fully specified by a process. The deployment of a BPEL process consequently requires the provision of the specific identities of partner services or the runtime strategy that should be followed to discover them.

The BPWS4J engine inspects the in-memory compiled process model to determine what 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 need to be resolved at deployment time. At this point BPWS4J only allows static deployment of these, but additional binding strategies are currently being developed.

Lifecycle

BPEL4WS has a model of "implicit lifecycle" for process instances. Process instances are created by the reception of specially designated application level messages ("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 an explicit process identifier token.

Supporting the implicit lifecycle model requires that the process container (the service manager unit in particular) decide, on receiving an application message, whether the message needs to be routed to an existing process instance or a new instance 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 is created by cloning a new instance of the in-memory runtime process model which will be used to hold instance specific execution state.

Message routing

The routing of incoming messages is the process by which correlation values are extracted from incoming messages and those values are used to identify the target process instance. The BPEL4WS 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 a message receive activities of a specific Web service operation. A set of possible error conditions arise from the flexibility of the BPEL4WS correlation mechanisms, most of which are not documented in the BPEL4WS specification, including: 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, 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 [7]; see also [8]).

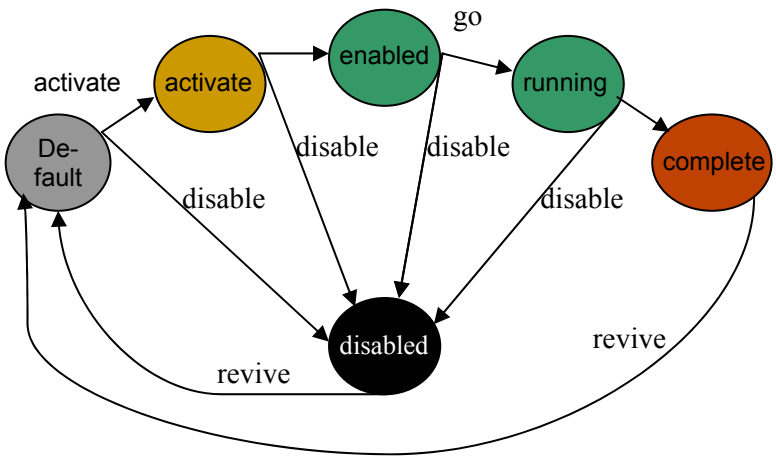
An Event Driven Process Interpreter

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 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 BPEL4WS 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.

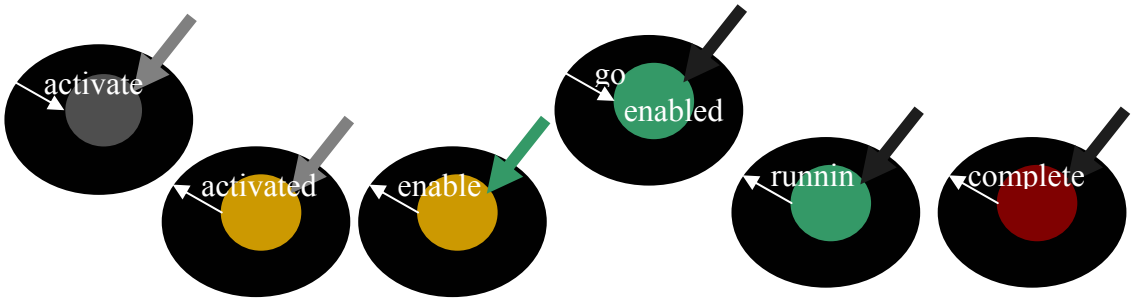
Activity Lifecycle

The event driven model described above relies on a well defined model for activity lifecycles. Activities follow a simple activate-enable-run-complete cycle which can be terminated 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 all incoming links is known and the join condition is true. On the other hand, the existence of control loops in BPEL4WS results in the need to "revive" activities according to the loop controlling conditions. The BPWS4J activity lifecycle is represented in Figure 2.



Events and Navigation

Once an instance is created, it runs and listens 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 include the outgoing requests for invocation and incoming results, requests for and results of assignment values which are sent out to an Xpath module, outgoing replies, and termination of notification status. 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 listend from and fired to the enclosed activities to control them as shown in Figure 2.



An activity starts running in BPWS4J once a combination of two things happens: An activity receives control from its enclosing activity, and all its incoming links have fired such that the join condition is true. This enables the first step for BPWS4J to natively encode BPEL4WS's combination of graph and structured process modeling. Activities

related through links in BPEL4WS 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

BPEL4WS 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 all its incoming links is known. In BPEL4WS, 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, all activities within it are 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 joinFailures. The resulting behavior is noted to be equivalent in the BPEL4WS 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 joinFailures and their suppression in BPEL4WS, see [10].

Conclusions and Future Work

The development of the BPWS4J engine has provided extremely valuable insight into the deployment and runtime operation management of BPEL4WS processes. The great flexibility provided by BPEL4WS is an important challenge to developers as well as 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 models offered in BPWS4J to incorporate dynamic discovery or partner information.

The sophistication and flexibility of BPEL4WS correlation as the basic mechanism for message routing and process instance identification is fully supported in BPWS4J, as well as the full implicit lifecycle model which is rooted on correlation sets as well.

Implicit in the adoption of the WS-Addressing [9] specification as the mechanism to encode partner references in BPEl4WS is new series of challenges that can profoundly affect how message routing takes service occurs in BPE4WS engines. Our experience implementing BPWS4J suggest the desirability of introducing a set of best practices or process patterns to enable a fully BPEL4WS compatible but simplified. 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 invocatios 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.

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. W. Nagy, R. Khalaf et al.BPWS4J, available at http://www.alphaworks.ibm.com/tech/bpws4j
6. F. Curbera et al. “Exception handling in the BPEL4WS language”, in Proceedings of the Business Process Management International Conference, BPM 2003, Eindhoven, The Netherlands, Springer LCNS 2678, 2003.
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. F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, **Exception Handling in the BPEL4WS Language**, Proc. of the Int'l Conf. on Business Process Management, Eindhoven, the Netherlands, Springer, LNCS 2678, pp 276-290, June 2003