IBM

Business processes in a Web services world

A quick overview of BPEL4WS

Level: Introductory

Frank Leymann (ley1@de.ibm.com), IBM Distinguished Engineer, IBM Software Group
Dieter Roller (rol@de.ibm.com), IBM Senior Technical Staff Member, IBM Software Group

01 Aug 2002

BPEL4WS allows the definition of both business processes that make use of Web services and business processes that externalize their functionality as Web services. This short paper introduces the basic language elements of BPEL4WS based on a simple example. The concepts underlying the language are briefly explained: establishing bilateral partnerships, correlating messages and processes, defining the order of the activities of a business process, and handling exceptions via long-running transactions. We'll also look at the resulting programming model, and the usage of BPEL4WS in pure B2B scenarios.

Introduction

Web Services are self-contained, modular business process applications that are based on the industry standard technologies of WSDL (to describe), UDDI (to advertise and syndicate), and SOAP (to communicate). (see the Resources section below for links). They enable users to connect different components even across organizational boundaries in a platform- and language-independent manner.

However, none of these standards allow defining the business semantics of Web services. Thus, today's Web services are isolated and opaque. Breaking isolation means connecting Web services and specifying how collections of Web services are jointly used to realize more complex functionality -- typically a business process.

A *business process* specifies the potential execution order of operations from a collection of Web services, the data shared between these Web services, which partners are involved and how they are involved in the business process, joint exception handling for collections of Web services, and other issues involving how multiple services and organizations participate. This especially allows specifying long-running transactions between Web services, increasing consistency and reliability for Web services applications. Breaking the opaqueness of Web services means specifying constraints on how the operations of a collection of Web services and their joint behavior can be used -- it turns out that this too is very similar to specifying business processes.

Business Process Execution Language For Web Services (BPEL4WS or BPEL, for short; see Resources for a link) allows specifying business processes and how they relate to Web services. This includes specifying how a business process makes use of Web services to achieve its goal, as well as specifying Web services that are provided by a business process. Business processes specified in BPEL are fully executable and portable between BPEL-conformant environments. A BPEL business process interoperates with the Web services of its partners, whether or not these Web services are implemented based on BPEL. Finally, BPEL supports the specification of business protocols between partners and views on complex internal business processes.

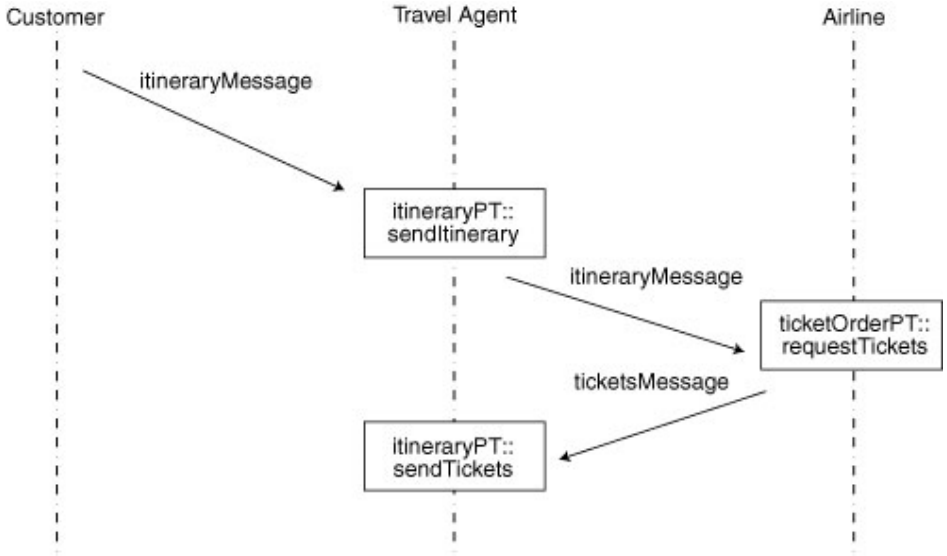
BPEL combines IBM's Web Services Flow Language and Microsoft's XLANG specifications (see Resources for links to both), superceeding both these specifications. This paper sketches the major concepts of BPEL.

A first look

Let's take a look at how BPEL works by examining a simple process scenario of an application for a travel agency to accept an itinerary from a customer, purchase the tickets from the airline, and hand deliver them to the customer.

Listing 1 contains a simplified example that shows the basic elements of BPEL. Figure 1 shows a flow diagram of the activities in this business process. A travel agent specifies a business process called `ticketOrder` (line 1). The purpose of this

simplistic business process is to allow the agent to receive from a customer an itinerary (lines 20 to 23), to pass on this itinerary to an airline requesting the corresponding tickets (lines 26 to 29), and finally to receive these tickets from the airline (lines 33 to 36). To keep the example simple, we've assumed that the tickets will be picked up by the customer in person.



The set of partners the agent's process interacts with are defined in lines 2 to 10: lines 3 to 5 introduce the partner `customer`, and lines 6 to 9 introduce the partner `airline`. A partner definition involves specifying the Web services mutually used by the partner or process, respectively (see the next section, "Partners", for more details).

The messages that are persisted by the process are called `containers` (lines 11 to 14). Containers are WSDL messages that are received from or sent to partners (see the section entitled "Containers, Properties, And Correlation"for more details). For example, the process stores an `itineraryMessage` as an `itinerary` container. The `itineraryMessage` is received from the `customer` (line 20) when the customer uses the `sendItinerary` operation of the agent's itinerary port (lines 21 and 22). This message is stored into the `itinerary` container (line 23) once received. The process then passes on the `itinerary` message to the `airline` (line 26) by using the `requestTicket` operation of the `ticketOrder` port (lines 27 and 28); this message is a copy of the `itinerary container` (line 29).

The usage of an operation in a business process is called an *activity* (see the section entitled "Activities" for more details). To define the order in which the activities have to be performed, the `ticketOrder` process structures its activities as a `flow` (line 15). A `flow` is a directed graph with the activities as nodes and so-called `links` as edges connecting the activities. The `links` required to define the flow between the `ticketOrder` process's activities is specified in lines 16 to 19. And an activity specifies the `links` it is a source or target of. For example, the `receive` activity of line 20 is the `source` of the `order-to-airline link` (line 24). And this `link` has the `invoke` activity of line 26 as `target` (line 30).

Listing 1. A simple BPEL example

```
 1 <process name="ticketOrder">

 2   <partners>
 3      <partner name="customer"
 4               serviceLinkType="agentLink"
 5               myRole="agentService"/>
 6      <partner name="airline"
 7               serviceLinkType="buyerLink"
 8               myRole="ticketRequester"
 9               partnerRole="ticketService"/>
10   </partners>

11   <containers>
12     <container name="itinerary" messageType="itineraryMessage"/>
13     <container name="tickets" messageType="ticketsMessage"/>
14   </containers>

15   <flow>
16      <links>
```

```
17         <link name="order-to-airline"/>
18         <link name="airline-to-agent"/>
19      </links>

20      <receive partner="customer"
21               portType="itineraryPT"
22               operation="sendItinerary"
23               container="itinerary"
24            <source linkName"order-to-airline"/>
25      </receive>

26      <invoke  partner="airline"
27               portType="ticketOrderPT"
28               operation="requestTickets"
29               inputContainer="itinerary"
30            <target linkName"order-to-airline"/>
31            <source linkName"airline-to-agent"/>
32      </invoke>

33      <receive  partner="airline"
34               portType="itineraryPT"
35               operation="sendTickets"
36               container="tickets"
37            <target linkName"airline-to-agent"/>
38      </receive>
39   </flow>

40 </process>
```

Partners

Business processes involving Web services often interact with different partners. Partners are connected to a process in a bilateral manner called a *service link type*. A service link type specifies two collections of Web services that are mutually provided and required by the two connected partners. These collections of Web services are referred to as roles. Listing 2 contains an example of a service link type definition.

Listing 2. Service link type definition

```
1 <serviceLinkType name="buyerLink">
2   <role name="ticketRequester">
3     <portType name="itineraryPT"/>
4   </role>
5   <role name="ticketService">
6     <portType name="ticketOrderPT"/>
7   </role>
8 </serviceLinkType>
```

The service link type `buyerLink` consists of two roles. The role `ticketRequester` (lines 2 to 4) provides a port of type `itineraryPT` (line 3), and the role `ticketService` (lines 5 to 7) provides a port of type `ticketOrderPT` (line 6). The port types themselves are defined elsewhere.

When defining a partner within a business process, a reference to the service link type underlying the corresponding bilateral relation between the process and the partner is made (lines 4 and 7 in Listing 1). For example, the `airline` partner in Listing 1 refers to the `buyerLink` defined in this section. A partner definition further specifies which role of the underlying service link type the process itself accepts (`myRole`) and which role has to be accepted by the partner (`partnerRole`). Accepting a role comes with the obligation to provide the corresponding Web services -- that is, to provide an implementation of the port types of the role. The Web services that are expected by the process from the partner are referenced by the `partnerRole` attribute (e.g., line 9 in Listing 1) and the Web services provided by the process and that the partner can rely on and use are referred to by the `myRole` attribute (e.g., lines 8 in Listing 1).

Via the `myRole` construct a process defines the set of Web services that represent themselves to the outside world. And the

`partnerRole` construct allows for the specification of the dependencies of a business process on Web services provided by the outside -- that is, the Web services the business process require and will use.

Containers, properties, and correlation

Business processes specified via BPEL prescribe the exchange of messages between Web services. These messages are WSDL messages of operations of the port types involved in the roles of the service links established between the process and its partners. Some of the messages exchanged may be included in the so-called *business context* of the business process; this context is a collection of WSDL messages called *containers*, which represent data that is important for the correct execution of the business process, e.g. for routing decisions to be made or for constructing messages to be sent.

For example, line 23 of Listing 1 specifies that the message received from the `customer` via the `sendItinerary` operation of the process's `itineraryPT` port has to be copied to the `itinerary` container. And line 29 specifies that the message sent to the `airline`'s `ticketOrderPT` port as input of the `requestTickets` operation stems from the `itinerary` container.

Often, the business context must be stored persistently to avoid loss of the context, thus ensuring the correct execution of a business process even in case of planned or unplanned system outages. Because the likelihood of such outages increases with the lifetime of a business process, and business processes typically last for long time periods, it is a good practice to make the context persistent.

When messages are exchanged between business partners, they typically carry some data that are used to correlate a message with the appropriate business process. For example, the `ticketsMessage` may carry an `orderNumber` that is used by the travel agent and the airline to identify the purchase of tickets for the submitted itinerary of the customer. Such correlation data is referred to as a *property* in BPEL. Very often the same property is used within different messages as data to be used for correlation. For this purpose, BPEL supports the definition of properties as separate entities. The following defines the `orderNumber` as a property:

```
9 <property name="orderNumber" type="xsd:int"/>
```

Because a property is used by different messages to define correlation data, a mechanism is needed that allows pointing to a dedicated field of the message that represents this property. In BPEL, this mechanism is called *aliasing*. Listing 3 shows how the `orderNumber` property (line 10) is defined to be the `orderID` field of the `orderInfo` part of the `ticketsMessage`.

Listing 3. Aliasing

```
10 <propertyAlias propertyName="orderNumber"
11                messageType="ticketsMessage"
12                part="orderInfo"
13                query="/orderID"/>
```

Activities

Activities are the actions that are being carried out within a business process. In our initial example, you've already seen some of the activities that can be used within a business process.

An important action in a business process is to simply wait for a message to be received from a partner. This kind of action is specified via a `<receive>` activity. It specifies the partner from which the message is to be received, as well as the port and operation provided by the process used by the partner to pass the message (lines 14 to 16 in Listing 4 below).

A more powerful mechanism is the usage of a `<pick>` activity. This kind of activity specifies a whole set of messages that

can be received from the same or different partners. Whenever one of the specified messages is received, the <pick> activity is completed, and processing of the business process continues. Additionally, one may specify that processing should continue if no message is received in a given time.

The start activities of a business process must be <receive> or <pick> activities. Flagging them with createInstance="yes" (lines 18 and 23 in Listing 4) indicates that an instance of the specified business process should be created if none exists already. Listing 4 illustrates this behavior using a business process that needs to accept the requests from two different partners. The sequence in which these messages arrive is unclear.

Listing 4. Start activities of a business process

```
14 <receive partner="hotel",
15          portType="roomPT",
16          operation="sendBooking",
17          container="stayInfo"
18          createInstance="yes"/>
19
20 <receive partner="rentalCar",
21         portType="carPT",
22         operation="sendBooking",
23         container="rentalInfo"
24         createInstance="yes"/>
```

Regardless which message arrives first, a process instance is created. After the initial message, the business process waits for the second message. For example, if the first message is received from a hotel partner, a process instance is created and then the business process waits for the message to arrive from a rentalCar partner.

Taking this approach, there is no need to have explicit life cycle commands, such as a command to create a process instance. Having no explicit life cycle commands makes life very easy for the requestors of Web services that represent business processes: There is no need to know whether a process instance has already been created or not. As a result, requestors can interact with Web services representing business processes as with any other Web service.

The initial example in Listing 1 does not send a response back to the customer; however in most practical cases a response must be returned. As illustrated in Listing 5, the <reply> activity is used to specify a synchronous response to the request corresponding to a <receive> activity.

Listing 5. Specifying a response

```
25 <receive partner="customer",
26          portType="itineraryPT",
27          operation="sendItinerary",
28          container="itinerary"
29          createInstance="yes"/>
30
31 <reply   partner="customer",
32          portType="travelPT",
33          operation="sendTickets",
34          container="tickets"/>
```

In Listing 5, the process provides an in-out operation: The input message of this operation is consumed by the <receive> activity, and the output message of this operation is produced via the <reply> activity.

If the response to the original request is to be sent asynchronously, the response is delivered via the invocation of a Web service provided by the requester. Consequently, the <invoke> activity is used within the process that produces the asynchronous response. The original requester will use a <receive> activity to consume the response delivered by the <invoke> activity.

Furthermore, the <invoke> activity can be used within a process to synchronously invoke an in-out operation of a Web service provided by a partner.

All activities discussed so far (except <pick>) are called *simple activities*, indicating that they have no structure and are not

allowed to enclose other activities. Other simple activities are: <wait, which indicates that a business process should wait for a specified time period or until a specified point in time has been reached; <empty>, which has no action associated and serves as a means to specify that nothing should be done or to synchronize parallel processing within the process; <terminate>, which indicates that a business process should be terminated immediately; <throw>, which signals the occurrence of an error; <assign>, which copies fields from containers into other containers; and <compensate>, which undoes the effects of already completed activities (see the section below entitled "Fault Handling").

The initial example in Listing 1 showed the use of <flow, one of the two most important structured activities. It allows defining sets of activities (including other flow activities) that are wired together via links, providing for the potential parallel execution of parts of the flow. Each link may be associated with a transition condition, which is a Boolean expression using values in the different containers of the process. When the business process is carried out, a particular link is followed when the associated transition condition evaluates to true.

The other important structured activity is <scope>, which allows building groups of activities (called *scopes*) and assigning to this group of activities certain characteristics. One of them is the notion of a fault handler that is automatically called when an error occurs within the scope; another is the notion of a compensation handler that can undo the changes made within the scope (see "Fault Handling").

Other structured activities are: <sequence>, which causes the enclosed activities to be carried out in the order in which they are listed; <switch>, which selects one path out of many paths using selection criteria that references values in containers; and <while, which causes the enclosed activities to be carried out as long as the condition associated with the while-activity evaluates to true.

Fault handling

BPEL processes interact with WSDL ports, and such ports may send fault messages back to the process. Furthermore, a process itself might detect erroneous situations that result in internal faults. BPEL provides mechanisms that allow attempts to recover from such faulty situations.

Central to these mechanisms are so-called *fault handlers* that can catch and deal with faults. The fault handler in Listing 6 below defines the set of faults it attempts to handle via a corresponding set of <catch> elements (line 36). Within such an element any kind of activity (simple or structured) may be nested. This activity will be performed when the corresponding fault occurs. In Listing 6, the fault handler catches a noSeatsAvailable fault returned by an airline partner. When this fault occurs, a corresponding rejection message is sent to the customer via the nested <invoke> activity (lines 37 to 40).

Listing 6. A fault handler

```
35 <faultHandlers>
36    <catch  faultName="noSeatsAvailable">
37       <invoke  partner="customer"
38                portType="travelPT"
39                operation="sendRejection"
40                inputContainer="rejection"/>
41    </catch>
42 </faultHandlers>
```

Typically, reactions to exceptional situations such as faults depend on the actual progress the process has already made or the state the process is in. In BPEL, this can be specified via so-called *scopes*. A scope is a structured activity (see the section above entitled "Activities") that allows grouping of activities. In addition, it allows the definition of a common execution context for its corresponding collection of activities, e.g. fault handlers that catch faults that occur while performing one of the activities included within the scope.

When a fault occurs within a scope, the regular processing within the scope is interrupted and the signaled fault is passed to the catching fault handler. The activity nested within this fault handler tries to correct the situation such that regular processing can continue outside the scope or alternate ways to complete the process can be taken.

All of this might require undoing actions that have already been completed within the scope. For example, if the tickets

required for a trip are not available, reservations for hotel rooms or rental cars that have already been completed must be canceled. The actions required to undo already completed activities are called *compensation handlers*. A fault handler of a scope may make use of compensation handlers to undo actions performed within that scope. If the exceptional situation cannot be corrected, the fault handler will rethrow the fault or signal the occurrence of another fault, which will be finally caught by a fault handler of another enclosing scope.

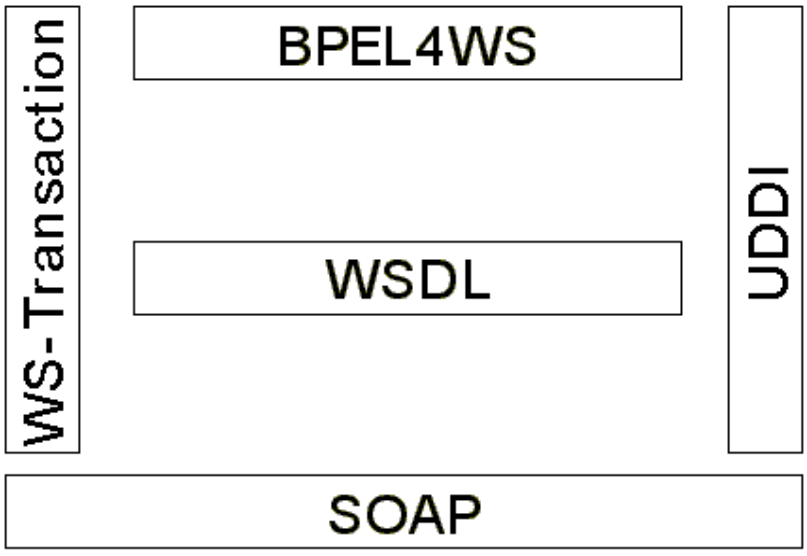
Thus, BPEL allows via its scope mechanism the definition of sets of activities that can be collectively undone in erroneous situations. Such a set of activities is some sort of unit of work, some sort of transaction. Activities that are performed within a scope either all complete or are all compensated. In contrast to this, the well-known traditional transactions (like database transactions) are implemented based on locks, allocating resources to a particular transaction for the duration of the transaction. This assumes that transactions are short-lived units of work and that locks can thus be released quickly. Because BPEL scopes are typically long-running, locking resources doesn't work in practice; one has to use compensation actions instead. This allows releasing locks once an enclosed activity completes, but one has to run compensation logic to undo already completed actions. The resulting units of work or transactions are referred to as *long running transactions*.

Long running transactions in BPEL are centered on scopes, and scopes can be nested. There is an agreement protocol between a scope and its parent scope to determine the outcome of the long-running transaction represented by a scope. The corresponding protocol has been described in WS-Transaction. While BPEL long-running transactions currently assume that a scope and all its nested scopes are contained within a single process and are hosted by a single BPEL engine, the agreement protocol in WS-Transaction does not assume this. Thus, a future extension of BPEL may support long-running transactions that are distributed across processes and even across BPEL engines.

WS-Transaction also specifies protocols for coordinating distributed atomic transactions. A future extension of BPEL may support distributed atomic transactions consisting of activities of a single process or even of different processes.

The general relationship between the standards involved is summarized in Figure 2. BPEL is a layer on top of WSDL -- that is, it uses WSDL to specify actions that should take place in a business process, and to describe the Web services provided by a business process. WS-Transaction specifies a protocol for the long running transaction model defined in BPEL as well as atomic transactions between regular Web services.

Figure 2. Relationship among standards discussed



Process-based applications

Applications created with BPEL are called *process-based applications* (see Resources for a related link).This kind of application structure splits an application into two strictly separated layers: the top layer, the business process, is written in BPEL and represents the flow logic of the application, whereas the bottom layer, the Web services, represents the function logic of the application.

This structure has several advantages over more conventional approaches. For one thing, the underlying business process as well as the invoked Web services can be changed without any impact on the other Web services within the application or on the Web services that the business process represents. In addition, the application can be developed and tested in two separate stages: the business process is developed and tested separately from the development and test of the individual Web services. This approach provides for great flexibility in changing the application.

Applications written in BPEL have another major advantage over conventional approaches, as they allow tailoring the ready application to the needs and circumstances of a particular environment without touching the application itself. This is achieved by separating the definition of the partners that a business process deals with from the characteristics of the actually involved partners. Within BPEL, one specifies only the port types and operations the different partners are expected to provide.

When such a business process is being carried out, the information about the actual ports or Web services that a concrete partner chosen provides need to be available. The information about the Web services or ports is collectively subsumed in BPEL under the notion of a *service reference*. Concrete mechanisms for providing service references for the different partners within BPEL have been deliberately left out of the specification (aside from a few exceptions). One of the exceptions deals with the situation that a requestor provides the provider with its own service reference so that the provider can respond back to the requestor.

The typical approach for providing service references is to provide this information when the business process is installed (or *deployed*) in the form of a deployment descriptor. Assigning a service reference to a partner comes in many flavors. In the simplest approach, a partner would be assigned a service reference containing fixed information. When the business process is being carried out, this fixed service reference is used to invoke the Web service. In the most complex case, the deployment information could just point to some mechanism; when the business process is being carried out, this mechanism would determine the appropriate service reference, and possibly invoke the selected Web service right away. This mechanism could, for example, go to UDDI, get all the detail information about potential service providers, and then based on that information select the most appropriate service provider.

Applications created based on BPEL are portable between environments supporting BPEL and Web services: The BPEL processes can be executed by any BPEL engine, and during their execution a BPEL engine will interact with the Web services that are discovered based on the deployment information.

Business protocols

BPEL isn't just for specifying executable processes; you can also use it for specifying *business protocols*. A business protocol specifies the potential sequencing of messages exchanged by one particular partner with its other partners to achieve a business goal. In other words, a business protocol defines the ordering in which a particular partner sends messages to and expects messages from its partners based on actual business context. An example of a business protocol is the RosettaNet PIPs (see Resources for a link).

Typically, the messages exchanged result from performing activities within internal business processes. Thus, a business protocol may be perceived as a view on a private business process; internal details like access to back-end systems, the complete structure of the messages making up the context, complex data manipulation steps, business rules determining branch selection, and so on are omitted from such a view.

In BPEL, the language for specifying business protocols is a subset of the language used for specifying executable processes. This enables you to specify an internal executable process together with its views within the same language. It supports an outside-in proceeding, starting with a view and extending it into an internal process, as well as an inside-out proceeding starting with an internal process projecting it onto its views.

In general, a business protocol (or view) is not executable. For example, the messages making up the context may be a simple projection of the real internal context messages; it may not be completely specified how messages are constructed that are sent to a partner, and branching conditions may not be defined precisely in terms of the data making up the visible context of the business protocol. This results from the fact that a business protocol hides internal details and complexity deliberately.

Because a business protocol may be neither executable nor deterministic but still expressed as a process, BPEL refers to it as an *abstract process*. It abstracts away complex details of an internal executable process. In this sense, abstract processes may be perceived as simple or easy-to-comprehend processes. And while an abstract process is not guaranteed to be executable,

abstract processes can be easily specified in such a manner that they are in fact executable! This allows beginning with simple variants of a process and refining them iteratively into the final complex business process.

Finally, an abstract process may be used to easily specify constraints on the usage patterns of a collection of port types. The port types to be constrained are the port types provided by the abstract process, and the operations that are to be constrained are used within activities of the abstract process.

Conclusion

BPEL supports the specification of a broad spectrum of business processes, from fully executable complex business processes over more simple business protocols to usage constraints of Web services. It provides a long-running transaction model that allows increasing consistency and reliability of Web services applications. Correlation mechanisms are supported that allow identifying stateful instances of business processes based on business properties. Partners and Web services can be dynamically bound based on service references.

Resources

- Check out the W3C's WSDL page.
- You can get more information on UDDI at UDDI.org.
- The W3C also has information on SOAP.
- Take a look at the BPEL4WS specification.
- IBM has more information on WSFL (in PDF format).
- Find out more about XLANG.
- For more on process-based applications, see "Workflow-Based Applications," F. Leymann and D. Roller (*IBM Systems Journal,* 1997).
- You can find out more about RosettaNet's PIP at the company's Web site.

About the authors

Frank Leymann is an IBM Distinguished Engineer working out of IBM Software Group. He is also a member of the IBM Academy of Technology, and a professor of computer science at the University of Stuttgart in Germany. He is the chief architect of IBM's flow technology, and a member of the WebSphere Platform Architecture Board that sets the overall technical direction of IBM's middleware. In addition, he is very active in Web services standardization, advanced technology, and productization.

In the past, Frank worked on database systems, database tools, and transaction processing. He has published many papers in various journals and conference proceedings, filed a multitude of patents, and is the coauthor of textbooks on repositories and on workflow systems. He served as a member of program committees and organization committees for many international conferences, and is editor of the journal of the DBMS SIG of the German computer society (GI). Contact Frank at ley1@de.ibm.com.

Dieter Roller is an IBM Senior Technical Staff Member and a member of the IBM Academy of Technology. He joined IBM in 1974 as a junior programmer and has held several technical and management positions in his IBM career. His current focus is on the architecture and design of MQSeries Workflow. He contributes to all facets of the development and enterprise-wide

deployment of workflow-based applications and is deeply involved in customer projects from this area. In addition, he is working in the area of Web services, their usage for application construction, and related middleware aspects.

Dieter has published papers in various journals and conference proceedings, mainly on workflow technology, filed many patents in this area as well as related areas, such as database and transaction technology, and has given talks at conferences and professional society meetings. He is the coauthor of a textbook about workflow systems. Contact him at rol@de.ibm.com.