Data4BPM, Part 2: BPEL4Data: Binding WS-BPEL to Business Entity Definition Language (BEDL)

Prabir Nandi (prabir@us.ibm.com)
Researcher
IBM

Florian Pinel (pinel@us.ibm.com)
Senior Software Engineer
IBM

Dieter Koenig (dieterkoenig@de.ibm.com)
Senior Technical Staff Member
IBM

Simon Moser (smoser@de.ibm.com)
WebSphere BPM Architect
IBM

Richard Hull (hull@us.ibm.com)
Research Manager
IBM

28 April 2011

Part 1 of this series discussed business entities and the use of Business Entity Definition Language (BEDL) to specify business entities. We also introduced BPEL4Data as a concrete instantiation to consume business entities in WS-BPEL processes. In Part 2, you'll learn about the BPEL4Data language elements and the architecture bringing together the BPEL family of languages (WS-BPEL, WS-HumanTask) with BEDL in execution scenarios.

View more content in this series

As discussed in Part 1, BEDL supports a set of capabilities that provide coverage for several types of access to a business entity (BE) runtime. A software component supporting BEDL provides an interface that enables processes in external components to invoke the following capabilities:

- Request metadata about BE types: This enables a process to query the BEDL component about the schemas of the BE types that it supports, as well as runtime metadata, such as the current set of readable/writeable attributes.

- Request to access data from a business entity instance: This enables a process to perform attribute creates (and appends), reads, updates, and deletes (and removes). If the requested access violates the access policies, then an `authorizationFailure` fault is thrown.
- Request to query BE instances: This enables a process to obtain the references to all BE instances of a given type that satisfy a query.
- Request to execute a transition of a business entity instance from one state to another: This enables a process to change the state of a business entity. This includes creating a BE instance, which moves the instance from the initial state to another state. If the state change request violates the lifecycle of the BE, an `invalidState` fault is thrown. If an access policy concerning guards is violated, an `inconsistentData` fault is thrown.
- Request to lock or unlock parts of a business entity instance: This is used in connection with concurrency control for BE instances, and is described in the Transaction management section. If the BE instance is not available (for example, because hard locks are being used and another process already has a conflicting lock), then a `lockFailure` fault is thrown.
- Subscribe to data or state changes of a class of business entity instances: This enables a process to subscribe to notifications about data and state changes in BE instances. Since the access policies can restrict read access to attributes based on role and current state, the subscription request may be partially or fully restricted. In the former case, a warning may be returned, and in the latter case an `authorizationFailure fault` is thrown. Receipt of such notifications may trigger new instances of a process to be initiated, terminate running process instances, or launch compensation activities. The process may use the WS-BPEL construct pick to wait for the notifications associated with a subscription.

By enabling BPEL processes to access the BEs in a BE runtime, BPEL4Data provides mechanisms to enable all of these types of access.

Design considerations for BPEL4Data

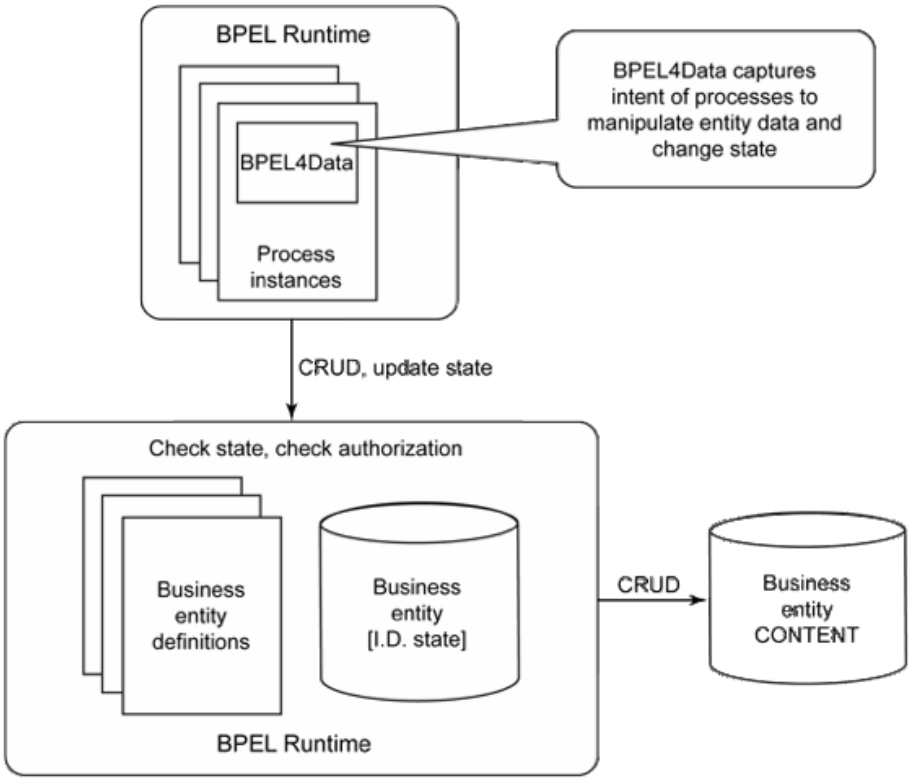
A variety of factors have gone into the design choices made for BPEL4Data. The overarching goal is to provide a specification that fits well into the existing family of relevant standards, including primarily BPEL and BPEL4People. The considerations at a domain-specific level include the following:

1. BPEL4Data will enable a BPEL process to take advantage of business entities that are supported by distinct BE runtimes. For a given BPEL process, each BE type must be hosted by a single BE runtime. (A single BE runtime may host multiple BE types.) The mapping of BE types to BE runtimes will be specified using a separate mechanism (a name resolution, which is not described in this article).
2. The BE (or BEDL) runtime is a component that manages the storage, persistence, and access to the instances of one or more BE types. It is thus analogous to a database management system (DBMS), but augmented with novel features related to business entity management. The BEDL runtime will maintain a record of each business entity instance ID, and typically also the state of that instance. The ID will be used to correlate the BE instance with its content, which can be stored in a separate data store (Business Entity CONTENT in Figure 1. Although the state data store is shown as logically separate from the content store,

the two can be physically a single store. However, the BEDL runtime will be the sole owner of the BE state.

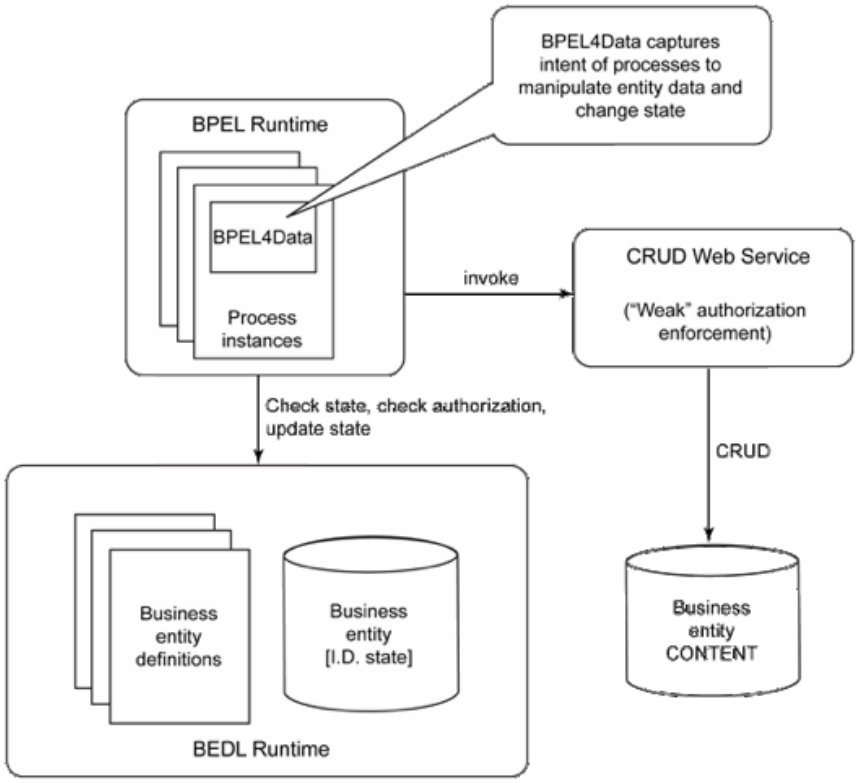
3. We envision two primary modes of interaction of BPEL processes with BE – the *controlling mode* and the *consulting mode*. In the controlling mode, the BE runtime will have sole ownership and control access to the BE content, as shown in Figure 1. All access to the BE content by the BPEL processes is channeled through the BE runtime.

Figure 1. Architecture for a BEDL runtime in controlling mode



The consulting mode is used to obtain some of the benefits of the business entity paradigm in legacy environments where the data corresponding to business entities is managed by pre-existing applications. Figure 2 shows the BPEL and BEDL runtime architecture for the consulting mode. BPEL will invoke external legacy web services to store and retrieve BE content after consulting with the BE runtime to ensure all authorization checks have passed. We also refer to this as the “good faith” mode, because this is done in good faith that the legacy web service is compliant with the BPEL4Data annotations. However, there is no way of checking and enforcing this.

Figure 2. Architecture for a BEDL runtime in consulting mode



4. The semantic differences between controlling and consulting modes are represented by different syntax, although the BPEL4Data specification strives for similar syntax wherever possible. The largest single difference is that for the controlling mode, BPEL4Data uses a new BPEL activity type, <b4d:entityAccess>, whose instances will contain one or more targeted BE access elements (for example, <b4d:read>, <b4d:update>). In consulting mode, the targeted BE access elements are inserted into existing BPEL message activities (<invoke>, <receive>, and so on) that access the legacy applications that hold the authoritative copies of the BEs. A BPEL process accessing BEs must indicate for each BE type whether it is to be used in controlling or consulting mode. (While in principle this can be inferred from the accesses made to the BE type, it helps with diagnosing syntax errors.)
5. If a BPEL process is managing a BE type in controlling mode, the BE runtime will completely control the storage and evolution of the BE instances of that type as they move through the business operations. Requested reads or updates that would violate the CRUDE constraints are blocked (and faults are thrown). In contrast, if a BPEL process is managing a BE type in consulting mode, the authoritative (or “golden”) copy of the data associated with BE instances of the BE type will be held by a pre-existing application, or possibly a collection of pre-existing applications. In this case, the BE runtime will maintain a shadow copy of some aspects of the BE instances of that type. The shadow will include at least the finite state machine of each BE instance, as well as correlation information that will allow linkage between a BE instance identifier used by the BE runtime and one or more identifiers for data objects in the pre-existing applications that hold data relevant to that BE instance. The shadow copy of BE

instances will also include some of the attribute values associated with the BE instances that are needed to evaluate and enforce CRUDE constraints.
When using BPEL4Data CRUDE commands to manipulate BE instances in consulting mode, it may be desirable to enforce the CRUDE access restrictions. This is supported based on the status and values of the shadow copy. As described below, this enforcement takes the form of a fault being thrown in connection with the BPEL message activity element that contains a targeted BE access command. The BPEL process may block the surrounding activity from occurring or let it occur anyway.
In some cases, the data about a BE instance in a pre-existing application may become out of synchronization with the information stored in the shadow copy. We do not address mechanisms for managing such cases in this article.

6. BE instances are a form of persistent data, and multiple processes may access them simultaneously. This means that there could be transactional consistency issues if two processes are accessing the same BE instance at the same time, and at least one of them is attempting to make an update. As a result, BPEL4Data includes mechanisms to either monitor or control concurrent access to BE instances, referred to as optimistic and pessimistic concurrency management, respectively.

Business entity management in controlling mode

This section describes the various elements of the BPEL4Data extension to manage and access BEs in controlling mode. The complete BPEL4Data schema is provided for download.

The Courier Shipment scenario shown in Figure 1 of Part 1 will be used both here and in Business entity management in consulting mode to illustrate the various aspects of BPEL4Data. This scenario focuses on the BE type used to manage shipments, viewed from end-to-end from the initial receipt or pick-up by the courier, through shipping and billing, and delivery to the recipient. Additional BE types for managing shipment deliveries might include those for truck trips and plane trips. The scenario also describes how a customer loyalty program could be supported with a second cluster of BE types that might focus on memberships in the loyalty program, promotional offers made by the loyalty program, and possibly others. As illustrated in Figure 2 of Part 1, it would be natural to have two business processes, one focused on managing shipment deliveries and the other focused on the loyalty program. When a shipment is first received from a customer, the managing BPEL program might reach into the loyalty program BE types, to see if either the sender or receiver are participants in the loyalty program, or to offer the sender a discount or other benefit. While the discussion in this article is fairly self-contained, it is recommended that the reader become familiar with the Courier Shipment scenario described in Part 1.

BE type declarations

A BPEL process that refers to business entity instances must declare the BE types that it uses, as shown in Listing 1. We assume here that the `courierShipmentInfo` and `shippingPromotionInfo` data types have been declared in a separate XML schema.

Listing 1. Courier Shipment and Shipping Promotion type declarations

```
<b4d:BEtypeDeclaration>
  <b4d:BEtype name="courierShipment" mode="controlling"
    structure="inf:courierShipmentInfo" />
  <b4d:BEtype name="shippingPromotion" mode="controlling"
    structure="inf:shippingPromotionInfo" />
</b4d:BEtypeDeclaration>
```

When working with BE types in controlling mode, a central construct of BPEL4Data is the `<b4d:entityAccess>` activity type, which can hold multiple interactions between the BPEL process and the BE runtimes. One reason for introducing this activity type is to enable simple groupings of multiple requests against the BE runtimes. As we'll discuss in more detail later, it is desirable to associate atomicity properties with the `<b4d:entityAccess>` activity type; these properties can be used to ensure that either all or none of the contained requests are performed. A second reason is to enable a parallel between the usage of BPEL4Data in both the controlling and consulting modes. In the consulting mode, a single interaction with a legacy application might have the effect of manipulating a single (virtual) BE instance in multiple ways, or of accessing or manipulating multiple (virtual) BE instances. This grouping of BE instance accesses can be mimicked in the controlling mode by using a single `<b4d:entityAccess>` activity.

Requesting BE creation

Listing 2 illustrates a `<b4d:entityAccess>` activity that calls for the creation of a new BE instance. This `<b4d:entityAccess>` activity might be used after a customer has requested a package shipment over a web site, but has not supplied all the needed information. The request states specifically that the BE runtime supporting the `courierShipment` type should create a new `courierShipment` instance, populate it with some data, and place it into the state "Draft". The output of this operation is a newly created BE instance, including a unique, printable identifier (in this example, the unique shipment ID).

Listing 2. Courier Shipment instance creation

```
<variables>
  <variable name="projectedCourierShipmentData" element="inf:courierShipment" />
  <!--  holds a "projected" instance of courierShipment -->
</variables>
<b4d:entityAccess>
  <b4d:create businessEntity="courierShipment" inputData="projectedCourierShipmentData"
    outputData="projectedCourierShipmentData" targetState="Draft" />
</b4d:entityAccess>
```

The `inputData` attribute points to a variable that is structured according to the XSD of the information structure of the `courierShipment` BE type. Typically, the value of `projectedCourierShipmentData` is a projected data value, as shown in Listing 3:

Listing 3. Courier Shipment creation input data

```
<inf:courierShipment>
  <senderInfo>
    <name>John Doe</name>
  </senderInfo>
  <recipientInfo>
    <name>Mary Smith</name>
  </recipientInfo>
</inf:courierShipment>
```

Requesting BE reads, updates and lifecycle transitions

Listing 4 illustrate several BPEL4Data features. In a single `<b4d:entityAccess>` activity there are three nested elements, which access two distinct BE instances from two BE types.

Listing 4. Courier Shipment instance manipulation

```
<variables>
  <variable name="shippingPromotionRef" element="inf:shippingPromotionReference" />
  <!-- holds a printable reference to a Shipping Promotion BE instance -->
  <variable name="shippingPromotionDataSet" element="inf:listOfShippingPromotionXPath" />
  <!-- holds a list of XPath expressions evaluated on inf:shippingPromotionInfo -->
  <variable name="projectedShippingPromotionData"
        element="inf:projectedShippingPromotionInfo" />
  <!-- holds a projected instance of shippingPromotionInfo -->
</variables>
<b4d:entityAccess>
  <b4d:update businessEntity="courierShipment"
    businessEntityID="courierShipmentRef" dataSet="courierShipmentDataSet"
    inputData="projectedCourierShipmentData" outputData="projectedCourierShipmentData" />
  <b4d:execute businessEntity="courierShipment"
    businessEntityID="courierShipmentRef" targetState="Ready" />
  <b4d:read businessEntity="shippingPromotion"
    businessEntityID="shippingPromotionRef" dataSet="shippingPromotionDataSet"
    outputData="projectedShippingPromotionData" />
</b4d:entityAccess>
```

The first nested element in Listing 5 is a `<b4d:update>`. When passing data between a BPEL runtime and a BE runtime, the BPEL4Data extension uses a generic approach allowing both partial and complete CRUD operations on the business entity. The `dataSet` attribute holds a list of XPath expressions, which are evaluated on the XSD of the information structure of the `courierShipment` BE type (`inf:courierShipmentInfo`). The `inputData` attribute holds a data value that holds information only in nodes identified by the XPath expressions in the `dataSet` input argument. These values are intended to fill in or overwrite the subtree in the current BE instance below the nodes identified by the XPath expression.

The second element, `<b4d:execute>`, executes the lifecycle state machine transition from state `Draft` to state `Ready`, which in this case indicates that the shipment is ready to ship.

Finally, the nested `<b4d:read>` element is used for individual reads. The `<b4d:read>` has `dataSet` and `outputData` attributes that are used in the same manner as in the `<b4d:update>` element. Note that variable `shippingPromotionDataSet` must be initialized before `<b4d:read>` is invoked, whereas the variable `projectedShippingPromotionData` will be populated when the `<b4d:read>` is executed.

According to the intended semantics, the three nested elements of `<b4d:entityAccess>` are considered an atomic unit - all should succeed or all should fail. Success or failure is guided by the CRUDE access restrictions associated with the BE type `courierShipment`. Note that if the `Courier Shipment` and `Promotion` BE types are maintained by distinct BE runtimes, those runtimes may need to support a distributed transaction scheme in order to preserve atomicity of the `<b4d:entityAccess>` activity.

Requesting bulk BE reads

Bulk reads can be done by specifying a query against the family of currently active BE instances of a BE type. Continuing with our example, suppose that a new `courierShipment` instance has moved to the `Ready` state, and no promotion was included in it. As a courtesy to premium customers, there might be a business policy to search for promotions that are applicable. This can be achieved using the `<b4d:entityAccess>` activity shown in Listing 5. In Listing 5, `shippingPromotionQuery` queries the `Shipping Promotion` BE type, and returns a projection of those having the format described by `shippingPromotionDataSet` (defined using a list of XPath expressions analogous to the `dataSet` attribute of `<b4d:read>`). The output is placed into the variable of attribute `outputData`, whose format will be a list of projected `Shipping Promotion` BE instances.

Listing 5. Courier Shipment bulk read

```
<variables>
  <variable name="shippingPromotionQuery" element="inf:shippingPromotionXQuery" />
  <!-- holds a query against the Shipping Promotion BE type -->
  <variable name="shippingPromotionListData"
        element="inf:projectedShippingPromotionInfoList" />
  <!-- holds a list of projected instances of Shipping Promotion BE type,
            in this case structured according to shippingPromotionDataSet -->
</variables>
<b4d:entityAccess>
  <b4d:query businessEntity="shippingPromotion" query="shippingPromotionQuery"
    dataSet="shippingPromotionDataSet" outputData="projectedShippingPromotionListData"
    queryLanguage="http://www.w3.org/XML/Query" />
</b4d:entityAccess>
```

In general, `<b4d:query>` elements may use an arbitrary query language for bulk queries. It is assumed that the queries are passed through the BPEL4Data down to the BE runtime. The query language to be used by a particular BPEL process must be indicated in the `<b4d:query>` element. In the example above, the query is assumed to be expressed using XQuery, which would be typical. The query should be designed so that the answer is a projection of a selection of instances from the BE type. The structure of the output is based on the list of XPath queries in the `dataSet` attribute, which must be initialized before the `<b4d:query>` is invoked. The `outputData` attribute is used to hold the actual list of projected BE instances returned by the query.

Transaction management

Multiple BPEL processes can access the same BE instance, which raises the issue of ensuring that simultaneous access does not lead to undesirable outcomes. To that end, the BE runtime provides concurrency control at the BE instance level, similar to the locking mechanisms described in the Java™Persistence 2.0 specification. Upon entering a `<b4d:entityAccess>` activity, the system will seek to obtain locks on the affected BE instances that the user is entitled to access. Depending on the implementation, various concurrency control modes are available. In a

pessimistic mode, while the implementation of multiple `<b4d:entityAccess>` activities might be interleaved, they must be serializable; that is, they must have an effect equivalent to the effect of a serial application of the multiple activities. This can be achieved using classical database management techniques, such as two-phase locking. In an optimistic mode, the implementation of multiple `<b4d:entityAccess>` activities might again be interleaved, but in this case a warning will be issued if it appears that a clash between two or more such activities has occurred. As in the JPA specification, unlocking always occurs implicitly whenever the transaction is ended.

The selected concurrency mode is specified directly on `<b4d:read>` or `<b4d:query>` using the `lockMode` attribute. In some cases, additional explicit locking may be used, as shown in Listing 6. Note the attribute `lockMode` used to indicate the locking style.

Listing 6. Courier Shipment instance lock

```
<variables>
  <variable name="courierShipmentRef" element="inf:courierShipmentReference" />
  <!-- holds a printable reference to a Courier Shipment BE instance -->
</variables>
<b4d:entityAccess>
  <b4d:lock businessEntity="courierShipment" businessEntityID="courierShipmentRef"
    lockMode="optimistic" />
</b4d:entityAccess>
```

Events and subscriptions

Until now we have focused on manipulations of BEs that are driven from the BPEL processes – where those processes are "pulling" information from the BE instances or "pushing" information or transition requests to them. BE types can also push information to the BPEL processes through the event mechanism. The standard BPEL activities `<receive>`, `<pick>`, and `<eventHandlers>` are used to react to these events.

A BEDL specification may include one or more event declarations. Each event declaration must contain a name that is unique to the corresponding BE type. A BE type may have one or more event declarations associated with it, which focus on the conditions under which an event of a given type should be generated. An event declaration can be based on various types of changes to a BE instance, including changes in state, updates to attributes, and changes in the value of a predicate. They may be combined using "and" and "or". When such events occur, a message is sent to all subscribed processes with information about the BE instance and the changes made. In the formal semantics, the specified combination of changes must happen as the result of a single `<b4d:entityAccess>` activity invoked by some executing BPEL process.

The subscription definition itself is out of the scope of BPEL4Data. Existing mechanisms, as described by the WS-Eventing specification, can be used here, including subscribe/unsubscribe requests. The actual format of the event notification message is left open in this specification. For simplicity in this article, we just assume that a one-way web service call is performed from the BE runtime to the WS-Eventing runtime. The payload of the call is the BE data as defined in the information model of the BE.

A typical interaction sequence between the BE runtime, a WS-Eventing runtime, and the BPEL runtime, might look as follows:

1. A BE definition containing event declarations is deployed to the BE runtime.
2. A BPEL process containing an event handler for consuming an event notification is deployed to the BPEL runtime.
3. The BPEL process subscribes to a BE event by sending a WS-Eventing subscription request message to a WS-Eventing runtime.
4. A BE instance emits an event causing the BE runtime to publish a notification to the WS-Eventing runtime.
5. The WS-Eventing runtime propagates the notification to the subscribed consumers, including the BPEL process from step 2.
6. The event handler in the BPEL process consumes the notification message.

Let's map the above sequence to a concrete example based on the `Courier Shipment` BE type, as shown in Listing 7. First, we add an event declaration to the `Courier Shipment` BE definition. The BE runtime holding the BE type will watch for cases of a `Courier Shipment` instance where the sender is a premium customer, and either the instance has just moved into the `Transit` state or the `Shipment Info` attribute (which holds a list of each step in the shipment process, including loading onto a truck or plane and delivery from a truck or plane) is updated. In this example, the event is named `transitEvent`.

Listing 7. Courier Shipment transit event

```
<be:businessEntity name="courierShipment">
  ...
  <be:event name="transitEvent">
    <be:triggerAnd>
      <be:triggerCondition condition="shipmentSenderPremium" />
      <be:triggerOr>
        <be:triggerStateEntry enteredState="Transit" />
        <be:triggerValueChange changedAttribute="courierShipmentInfo" />
      </be:triggerOr>
    </be:triggerAnd>
  </be:event>
  ...
</be:businessEntity>
```

A BPEL process sends a WS-Eventing subscription request, as shown in Listing 8.

Listing 8. Event subscription request

```
<soap:Envelope
  xmlnsoap:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlnsoap:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlnsoap:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing"
  xmlns:eh="http://example.com/courierShipmentEventHandler">
  <soap:Header>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe
    </wsa:Action>
  </soap:Header>
  <soap:Body>
    <wse:Subscribe>
      <wse:Delivery>
        <wse:NotifyTo>
          <wsa:Address>
            http://example.com/courierShipmentEventHandler
                <!-- BPEL process WS address  -->
          </wsa:Address>
```

```
          <wsa:ReferenceProperties>
            <eh:MySubscription>42</eh:MySubscription>
                <!-- BPEL process instance ID -->
          </wsa:ReferenceProperties>
        </wse:NotifyTo>
      </wse:Delivery>
    </wse:Subscribe>
  </soap:Body>
```

A BPEL event handler, as shown in Listing 9, can consume zero or more events of type `PremiumCustomerShipmentAlert`. The variable `courierShipmentData` will hold the payload of the event sent from the BE runtime. Its type must match the information model of the BE.

Listing 9. BPEL event handler

```
<eventHandlers>
  <onEvent partnerLink="courierShipmentEventPL" operation="courierShipmentEvent"
    variable="courierShipmentData" >
    <scope>
      <variables>
        <variable name="courierShipmentData" element="inf:courierShipmentInfo" />
      </variables>
      ...
    </scope>
  </onEvent>
</eventHandlers>
```

A notification message may look like the following:

Listing 10. Transit event notification

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:eh="http://example.com/courierShipmentEventHandler">
  <soap:Header>
    <wsa:Action>http://example.com/courierShipment/transitEvent</wsa:Action>
    <wsa:To>http://example.com/courierShipmentEventHandler</wsa:To>
    <eh:MySubscription>42</eh:MySubscription> <!-- BPEL process instance ID -->
  </soap:Header>
  <soap:Body>
    <cs:courierShipment>
      ...
    </cs:courierShipment>
  </soap:Body>
</soap:Envelope>
```

Business entity management in consulting mode

This section explains the usage of the BPEL4Data extension in consulting mode, using the same Courier Shipment scenario as in the previous section. BE type declarations and subscriptions are performed the same way as in controlling mode and will not be discussed here. The CRUDE operations, on the other hand, require a web service invocation, and the BPEL4Data operations are embedded inside a BPEL invoke.

Specifying the contents of the BE instance shadows

The list of attributes to include in the BE instance shadows must be specified in the `<b4d:BEtype>` elements as shown in Listing 11. Note that the primary key attributes are included implicitly.

Listing 11. Courier Shipment shadow definition

```
<b4d:BEtypeDeclaration>
  <b4d:BEtype name="courierShipment" mode="controlling"
    structure="inf:courierShipmentInfo"
    shadowAttributes=”@inf:customerType @inf:deliveryTime”>
  </b4d:BEtype>
</b4d:BEtypeDeclaration>
```

Requesting BE instance creation, update or transition

If partner links have been defined for the CRUDE web services, as shown in Listing 12, `<b4d:create>` and `<b4d:update>` can be used inside an `<invoke>` using the syntax described earlier, as shown in Listings 12 and 13. The shadow copies of the BE instances will be created and updated by the runtime accordingly.

Listing 12. Courier Shipment instance creation in consulting mode

```
<partnerLinks>
  <partnerLink name=”CourierShipmentPartner”
       partnerLinkType=”ns:CourierShipmentPartnerLink”/>
</partnerLinks>
<variables>
  <variable name="projectedCourierShipmentData" element="inf:courierShipment" />
  <!--  holds a "projected" instance of courierShipment -->
</variables>
<invoke name="InvokeCreateCourierShipment" operation="create"
  partnerLink="CourierShipmentPartner" portType="ns:CourierShipmentPT">
  <b4d:create businessEntity="courierShipment" inputData="projectedCourierShipmentData"
    outputData="projectedCourierShipmentData" targetState="Draft" />
</invoke>
```

Listing 13. Courier Shipment instance update in consulting mode

```
<variables>
  <variable name="shippingPromotionRef" element="inf:shippingPromotionReference" />
  <!-- holds a printable reference to a Shipping Promotion BE instance -->
  <variable name="shippingPromotionDataSet" element="inf:listOfShippingPromotionXPath" />
  <!-- holds a list of XPath expressions evaluated on inf:shippingPromotionInfo -->
  <variable name="projectedShippingPromotionData"
            element="inf:projectedShippingPromotionInfo" />
  <!-- holds a projected instance of shippingPromotionInfo that has data at least
            in nodes corresponding to the data set -->
</variables>
<invoke name="InvokeUpdateCourierShipment" operation="update"
    partnerLink="CourierShipmentPartner"
    portType="ns:CourierShipmentPT">
    <b4d:update businessEntity="courierShipment"
      businessEntityID="courierShipmentRef"
      dataSet="courierShipmentDataSet"
      inputData="projectedCourierShipmentData"
      outputData="projectedCourierShipmentData"/>
</invoke>
```

Requesting BE instance reads or bulk reads

Reading BE instances is performed in the same fashion as creations and updates, but the data that was read will be handled in the output mapping of the web service, so the `outputData` attribute of `<b4d:read>` or `<b4d:query>` is irrelevant.

CDR: Need to introduce this code section.

Listing 14. Courier Shipment instance and bulk reads

```
<variables>
  <variable name="shippingPromotionRef" element="inf:shippingPromotionReference" />
  <!-- holds a printable reference to a Shipping Promotion BE instance -->
  <variable name="shippingPromotionDataSet" element="inf:listOfShippingPromotionXPath" />
  <!-- holds a list of XPath expressions evaluated on inf:shippingPromotionInfo -->
</variables>
<invoke name="InvokeReadCourierShipment" operation="read"
  partnerLink="CourierShipmentPartner" portType="ns:CourierShipmentPT">
  <b4d:read businessEntity="shippingPromotion"
    businessEntityID="shippingPromotionRef" dataSet="shippingPromotionDataSet" />
</invoke>

<invoke name="InvokeQueryCourierShipments" operation="query"
  partnerLink="CourierShipmentPartner" portType="ns:CourierShipmentPT">
  <b4d:query businessEntity="shippingPromotion" query="shippingPromotionQuery"
    dataSet="shippingPromotionDataSet" queryLanguage="http://www.w3.org/XML/Query" />
</invoke>
```

Incorporating human tasks

The BPEL4Data annotations make the specification and execution of human tasks much richer. In particular, you can now support BE interactions within the task execution, a potentially long-running interaction between the human client and the BPEL runtime. The interaction is quite ad hoc in nature, and may comprise multiple arbitrary updates/reads of the BE data before the task is marked as completed. Today this interaction is handled without the knowledge of the BPEL runtime by using custom ad hoc frameworks. With the introduction of BPEL4Data, this interaction is now supported in a first-class way, without sacrificing the ad hoc nature of the interaction.

Figure 3. Handling human tasks

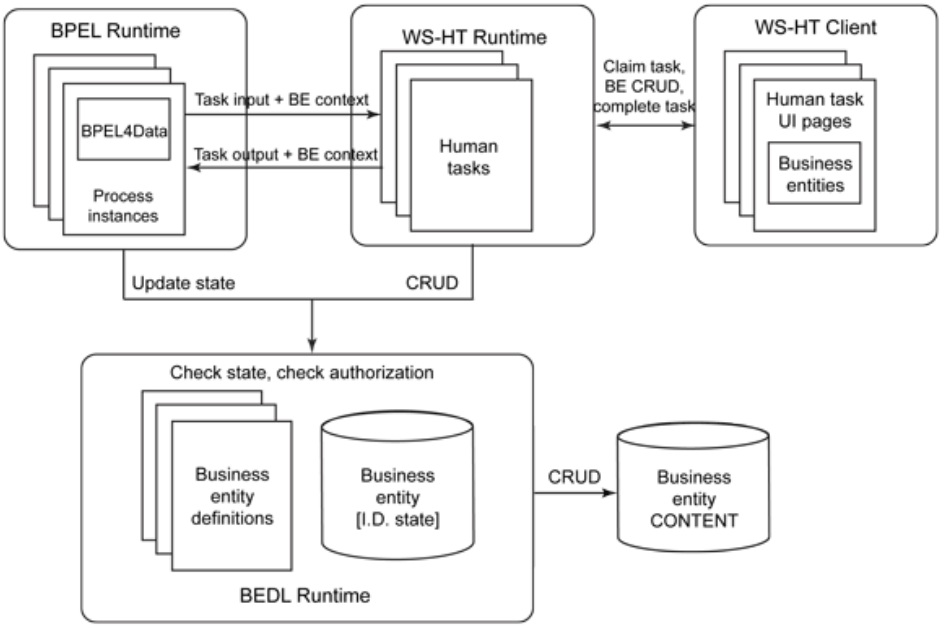
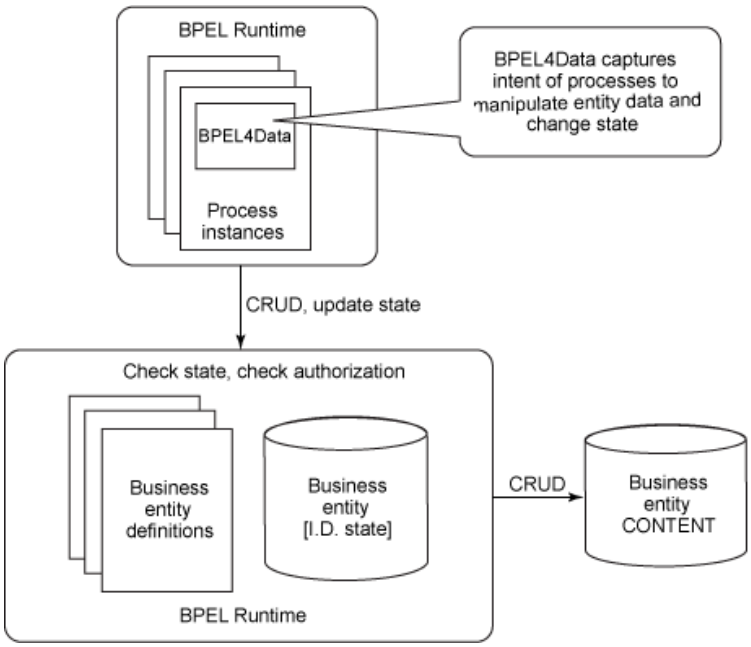


Figure 3 shows the human task execution architecture. Besides the BPEL and BEDL runtimes introduced earlier, we now also have the WS-HT tuntime (executing the WS-HumanTask specification) and the WS-HT Client (the set of UI pages associated with the human task, through

which the end user interacts). With the introduction of BEDL, the interaction sequence can be logically broken into the following four phases:

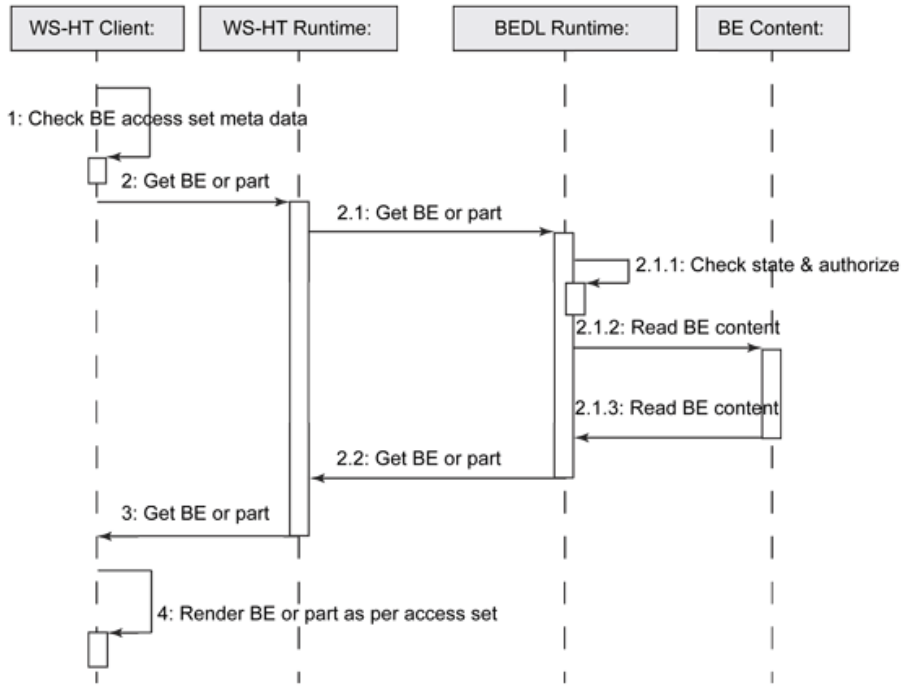
1. **Claim task**: During BPEL execution, a human task activity is encountered (Figure 4). The WS-HT runtime instantiates the human task (1 and 2 below). The task shows up in the work list of WS-HT client. The user logs in (with a certain role) and claims the task (3, 4 and 5). The claim is passed back onto the WS-HT runtime and the WS-HT runtime marks the state of the task as claimed (5.1). If the activity has a BPEL4Data annotation, the proper BE context is part of the task input and passed back to the WS-HT client (6).

Figure 4. Claim task sequence



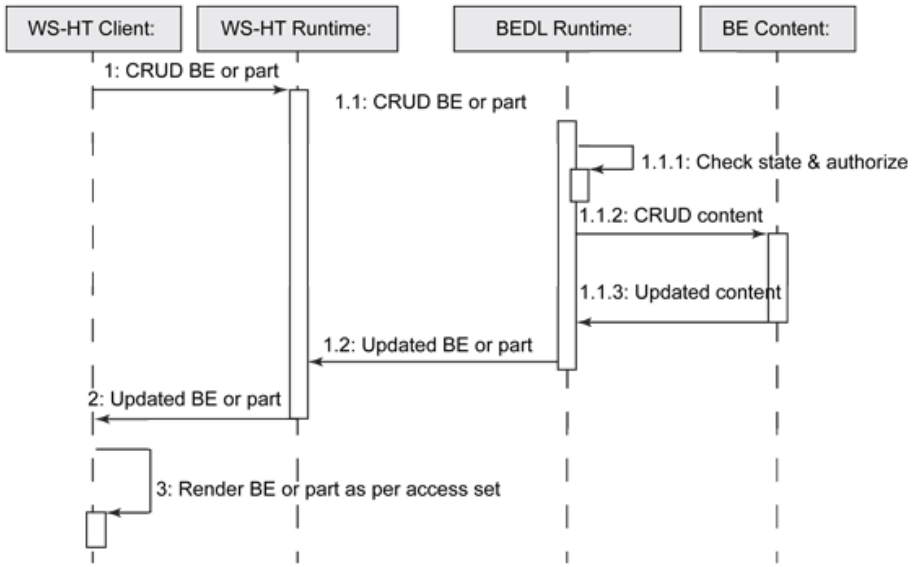
2. **Get and render BE content**: In this phase, the WS-HT client obtains any BE content required by the current task and renders it on the appropriate UI page (Figure 5). Based on the BE context obtained as part of the claimed task input, the WS-HT client can access the appropriate BE access policy metadata applicable in the current context (1). The WS-HT client then requests BE content (or some parts of it) through the WS-HT runtime to the BEDL runtime (2 and 2.1). The BEDL runtime checks the current state of the BE and, after authorization (2.1.1), retrieves the BE content. The WS-HT client renders the BE content (2.1.2 - 4). Note that some tasks do not involve any BE content in their input. This is the case when a new BE is being created for example.

Figure 5. Get and render BE content sequence



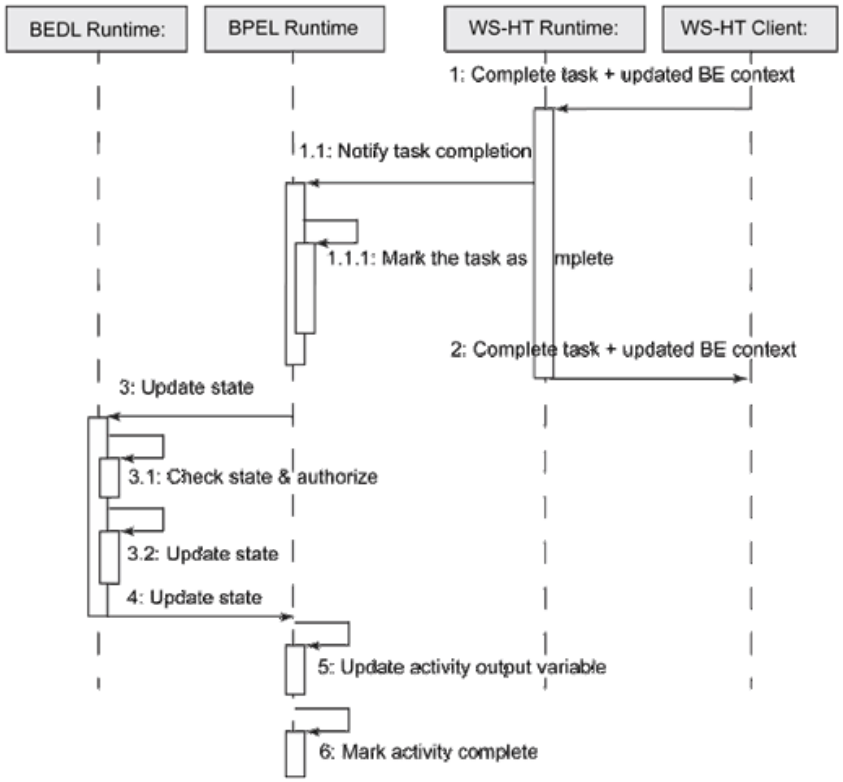
3. **Ad hoc CRUD on BE**: Once the BE content (if it exists) is rendered on the client, the client can have an extended interaction manipulating the BE content as needed to perform the task (Figure 6). Essentially, the interaction is identical to the retrieval of BE content in the previous phase. The WS-HT client can perform ad hoc CRUD on the BE content (1). The requests are passed on the BEDL runtime (1.1), where necessary state checks and authorization checks are performed (1.1.1) and, if successful, the proper CRUD APIs are invoked to get/set BE content (1.1.2 and 1.1.3). If one of the checks fails, then no CRUD operation is performed on the BE and an appropriate fault is returned to the WS-HT client.

Figure 6. Sequence diagram



4. **Complete task**: After WS-HT Client indicates completion of the task, it is passed to the BPEL runtime through the WS-HT runtime 1 and 1.1 in Figure 7). The updated BE context is passed along the call to the BPEL runtime as part of the task output. Control returns to the BPEL runtime, which invokes a custom operation to mark the task complete (1.1.1). If a new state is indicated on the BPEL4Data annotation of the activity, the BPEL runtime invokes the update state on the BEDL runtime (3). The BEDL runtime performs the necessary conditional and authorization checks (3.1) and makes sure that the current state of the BE is still valid for the transition to the new state. Once the checks passes, the BE state is updated to the new state and the call returns to the BPEL Runtime (3.2 and 4). The BPEL runtime updates the activity's output variable and marks the activity as complete (5 and 6).

Figure 7. Sequence diagram

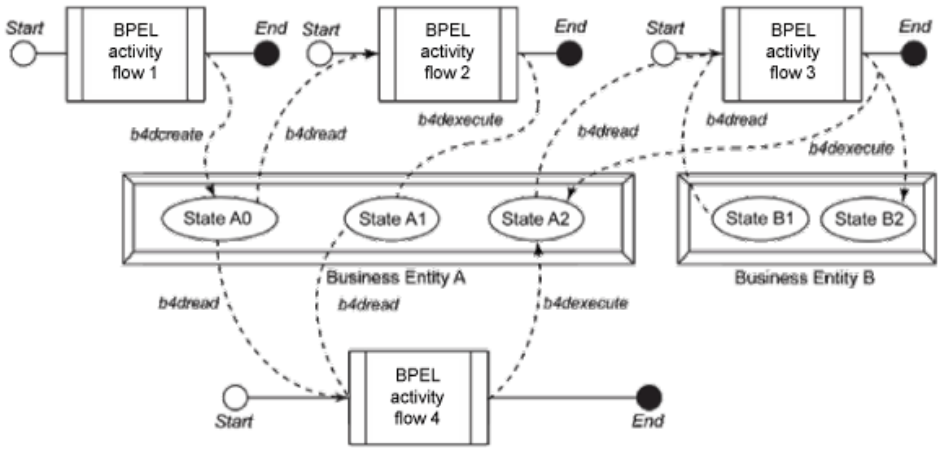


Best practice for designing BPEL processes with BEDL

Business entity lifecycle states capture business operation milestones. A process is likely to accurately reflect business intent and also be vastly simplified if it is refactored to a set of discrete flows of activities, each of which takes one or more business entities from one state to the next. In other words, when designing BPEL processes with BEDL, the best practice would be to have flow elements lie within the boundary of the BE state transition (that is, finish with the state change of a BE), and not have a BE state change activity happen in the middle of the flow. For example, this practice could lead to a process design where CRUD operations on a BE are grouped within BPEL flow activities, and the flow activity itself is annotated in order to trigger the BE state change, such that the flow activity acts as a control block not only for the control flow of the business process but also for the state transitions of the BE.

A sample ideal design is shown in Figure 8. It shows four BPEL activity flows (Flows 1 to 4) and two business entities (BEs A and B) and the BPEL4Data annotations linking them. The activity flows may or may not be part of the same BPEL process (see Additional BPEL process modeling considerations for a discussion of the advantages and disadvantages of both options). Each activity flow starts with a `<b4d:read>` (except for Flow 1, where the business entity is created). The `<b4d:read>` is constrained to the set of business entities available in a particular state.

Figure 4. BPEL-Business Entity Design Best Practice



For example, Flow 2 can only work with instances of BE A in the A0 state. Flow 3 on the other hand can work with either A0 or A1 states. Flow 4 needs both BEs A and B, but state A2 for BE A and state B1 for BE B. All activity flows can have multiple `<b4d:update>` activities and the flow ends with a `<b4d:execute>` (state change) for each BE read into the flow.

This guiding principle leads to better design of business processes, as well as increased reuse potential and maintainability of the BPEL processes. Of course there will be exceptions, in particular when dealing with multiple business entities in the same flow.

Additional BPEL process modeling considerations

Let's look at two extreme flavors of the best practice described above. The activity flows may be part of the same BPEL process, or each activity flow for a particular state transition constitutes a separate BPEL process.

Consolidating all activity flows in a single BPEL process has the advantage of providing a single point of control over all business logic related to a BE. Moreover, there may be common logic (for example, for fault handling or compensation) for multiple activity flows.

If, on the other hand, each transition is modeled as a separate BPEL process, there will be more flexibility with respect to the maintenance of individual activity flows. A higher degree of reuse is typically achieved only when separate BPEL processes represent meaningful business services by themselves.

Conclusion

In Part 1 we discussed the concept of business entities as the means of representing a business view of data and how this contributes to a more holistic BPM architecture. We also discussed the Business Entity Definition Language (BEDL), an XML language to define a business entity. In Part 2, you learned about BPEL4Data, an extension to WS-BPEL to consume and work with business entities as part of the process flow. You also learned the runtime architecture demonstrating concretely the possible interaction between the WS-BPEL, WS-HumanTask and BEDL runtime containers.

Downloads

Description	Name	Size
BPEL4Data process example - consulting	CourierShipment_BPEL4Data_consulting.zip	6KB
BPEL4Data process example - controlling	CourierShipment_BPEL4Data_controlling.zip	5KB
BPEL4Data Schema Definition	BPEL4Data.xsd	7KB

Resources

- See Part 1 for a complete list of resources for more information.
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.
- IBM BPM Journal: Get the latest articles and columns on BPM solutions in this quarterly journal, also available in both Kindle and PDF versions.

About the authors

Prabir Nandi

Prabir Nandi is a Research Staff Member in the Business Services Department at the IBM T. J. Watson Research Center. He is an inventor of the business entity (BE) concept and has led its continued research and development for the last several years, including the now commercially available Business Entity Lifecycle Analysis (BELA) and Business Value Modeling (BVM) capabilities as part of IBM Global Business Services method and tools.

Florian Pinel

Florian Pinel joined the IBM T.J. Watson Research Center in 1999 and currently works in the Business Informatics Department as a Senior Software Engineer. He received a M.S. in Computer Science and Engineering from Ecole Centrale de Paris, France. He has been working on Business-Entity-Centric Process Management for several years, and has published several papers on the subject.

Dieter Koenig

Dieter Koenig is a Software Architect for IBM WebSphere BPM products. He is a member of several OASIS technical committees for the standardization of Web Services Business Process Execution Language (WS-BPEL) and Service Component Architecture (SCA) specifications. Dieter has published many articles and has given talks at conferences about Web services and workflow technology, and is co-author of two books about Web services.

Simon Moser

Simon Moser is a Software Engineer and Architect with the Business Process Solutions Group at IBM's Software Laboratory in Boeblingen, Germany. He has published many papers and given talks at international conferences, mainly in the area of Web service systems and business processes. He holds a M.Eng degree in Computer Science and Engineering from the Technical University of Ilmenau, Germany.

Richard Hull

Richard Hull is a Research Manager in the Business Informatics Research Department at IBM's T.J. Watson Research Center. He is widely recognized for his research contributions in the areas of Web services, business process management, and database theory. He became an ACM Fellow in 2007. His current research foci include bringing a declarative style to the business entity lifecycle approach for the modeling of business operations, and applying the business entity lifecycle approach to the challenges of service composition and interoperation.