Chapter 13

Process Specifications

No service is an island. The key point about service-oriented computing is that it involves extended, loosely coupled activities among two or more autonomous business partners. Such activities can be thought of as (business) processes that engage several services in a manner that brings about the desired (business) outcome. The previous chapter described the underpinnings of processes from the perspective of transactions. This chapter covers process specifications, discussing their modeling and enactment, as well as key emerging standards.

13.1 Processes

A process is an activity. Generally a process would be a composite activity and be geared to serve some purpose. Depending on the specific process, its tasks could be some combination of services that correspond to queries, transactions, applications, and administrative activities. These services may be distributed within or across enterprises and would be coordinated by constraining control and data among them. The services may themselves be composite, i.e., implemented as processes. The following discussion emphasizes business processes consisting of services, but the concepts developed could apply equally well to scientific computing and other settings. Examples of settings where processes apply include *intraenterprise* environments (i.e., within an enterprise), such as production scheduling and inventory control, and *interenterprise* environments (i.e., across enterprises), such as supply-chain management and purchase negotiation. Clearly, intraenterprise and interenterprise processes need to correlate with each other, because intraenterprise activities are needed to support interenterprise interactions.

Processes present a number of technical challenges. First, we must be able to model a process, incorporating correctness of executions with respect to the model, and respecting the constraints of the underlying services and their resources. The normal executions of a process are often easy, since they can be as simple as a partial order of the activities in the process. By contrast, the exception conditions can be difficult to model and handle. More importantly, because interesting business processes are often long-running, their mutual interactions are nonatomic, leading to the prospect that the information they take as input may be subject to revision and thereby causing their own results to be invalidated. Exceptions and revisions are the main sources of complication in the modeling of a process.

Second, we must be able to interface a process to underlying functionalities. In the case of database systems, these would include a suitable model of transactions that incorporates constraints on the concurrency control and recovery mechanisms of a DBMS. A transaction model provides the necessary abstractions and shields process models from the implementational details of DBMSs.

Because processes are used in a number of places in an enterprise to support its internal functioning as well as its interactions with its business partners, processes can end up being modeled in several different ways, typically based upon process representations that are proprietary to the software vendors involved. For example, if production scheduling software employs a different modeling formalism than purchase order processing software, then the enterprise's participation in a supply chain may be adversely affected. However, interoperation among processes, while clearly an important need in practical settings, is nearly impossible without some kind of translator among process models. The challenges of heterogeneity that Section 5.1 on page 74 discussed in the context of information sharing apply equally to process model interaction.

Before we get into the details, it is worth describing the main perspectives we can have on processes and the distinctions between them.

Orchestration. This takes the view of a process as a program or a partial order of operations

that need to be executed. This view is logically centralized in that it views a process from the perspective of one "orchestrating" engine. It is as if the process specification is being executed under the control of or on behalf of a specific party. Orchestration corresponds best to the workflow representations discussed in Section 13.3 and to process languages such as BPEL4WS. Representations such as OWL-S (introduced in Section 15.5.2 on page 331) enable the right orchestrations to be produced, given the requirements for a desired process and the functionalities of the available services.

Choreography. This takes the view of a process as being a set of message exchanges between participants. The message exchanges are constrained to occur in various sequences and may be required to be grouped into various transactions. Choreography corresponds best to languages such as WSCL and WSCI.

Collaboration. This takes the view of a process as a collaboration among business partners. The business partners not only send messages to one another, but also enter into business relationships such as contracts and obligations. They generate flexible message exchanges depending on the evolving circumstances and their local policies, e.g., to handle business exceptions. Collaboration is emerging as a serious approach for carrying out large-scale business processes.

13.2 Describing Dynamics with UML

UML provides graphical constructs that can be used to describe (1) actions and activities, and (2) their temporal precedences and flows of control. The allowable control constructs are

- *Sequence*, which is a transition from one activity to the next in time.
- *Branch*, which is a decision point among alternative flows of control.
- *Merge*, where two or more alternative flows of control rejoin.
- *Fork*, which is a splitting of a flow of control into two or more concurrent and independent flows of control.
- *Join*, which is a synchronization of two or more concurrently executing flows of control into one flow.

These control constructs are a sufficient set for describing an arbitrary process or workflow. As such, they can also describe a composite Web service. A particular process is shown on an activity diagram, an example of which is shown in Figure 13.1 on the following page.

A UML sequence diagram is used to show the interactions among concurrently existing objects or concurrently executing threads and process instances. It focuses on the time ordering of the messages between such entities. Figure 12.5 on page 244 is an example of a sequence diagram.

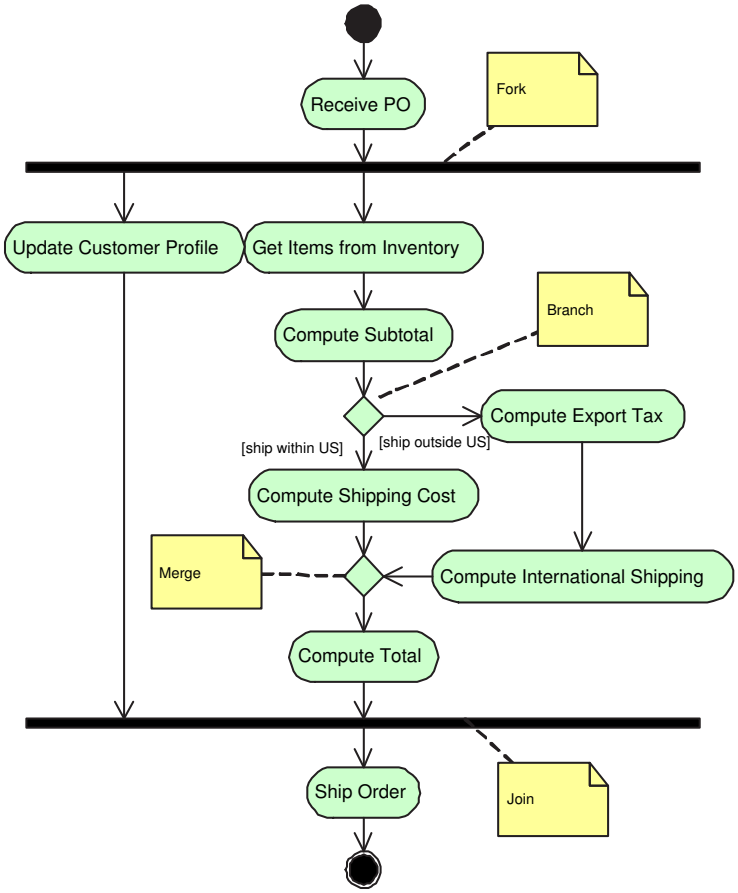

Figure 13.1: An example UML activity diagram showing the allowable control constructs that can be used to describe workflows and processes

13.3 Workflows

A *workflow* is an activity that addresses some business need by carrying out specified control and data flows among subactivities that involve information resources and possibly humans. A classic example of a workflow is loan processing: when you apply for a loan, you fill out a form, a clerk reviews it for completeness, an auditor verifies the information, and a supervisor invokes an external credit agency or uses a credit risk assessment tool. Each person in the loan process receives information concerning your application, modifies or adds to it, and forwards the results.

There is a fine line between processes and workflows. Some research seems to treat these as isomorphic. A lot of the research on processes is based on previous research on workflow. For example, one of the inputs to the current leading standard for processes, BPEL4WS (discussed in Section 13.4.1 on page 267), is the Web Services Flow Language (WSFL), which was closely based on IBM's Flowmark workflow product.

For the purposes of this book, workflows are a narrower concept than processes. Processes may be realized through workflows, but possibly through other means as well, e.g., business protocols or conversations among agents, which are introduced in later chapters. The above definition of workflows emphasizes the control and data flow among subactivities that are the essence of workflows. These flows are necessary to realize the desired processes. Ultimately, no matter how you specify a process, control and data flows will occur when it is enacted. The key point is that in workflow technology, such control and data flows are directly specified from a logically central perspective. This modeling assumption accounts for both the strengths and the weaknesses of workflow technology. Service providers can manage workflows that are used to implement the given service. An implementation based on workflow techniques can help manage potential exception conditions better than a traditional application, which would hide the necessary reasoning. However, workflows too have their limitations as discussed below.

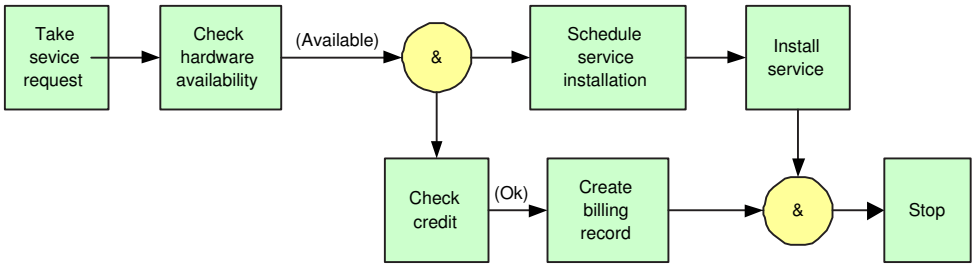

Figure 13.2: A workflow for processing a telecommunications service order

Figure 13.2 illustrates an example of a workflow that is executed when you order a service from a telecommunications provider. You initiate the order by interacting with a sales representative from the provider, who fills out a form on your behalf. The sales representative

checks with a provisioning database to determine whether the necessary hardware is in place. If it is, you receive an estimate of when the service will be ready for your use. A local service installer is dispatched to install your service, while the telecommunications provider checks your credit history.

13.3.1 Exceptions

If all goes well, the installer successfully installs the service, the auditors find your credit history acceptable, the billing department is notified to begin charging you, and the workflow concludes successfully. However, things do not always go that smoothly. For example, in checking whether you already have an account, the telecommunications provider might discover that you have an unpaid and overdue balance—or that someone else previously at the same address has an unpaid balance. Such discoveries would raise a red flag.

Perhaps the service installer for your area calls in sick, requiring a revision in the installation schedule. Or the installer might discover that the available hardware is unusable and must be replaced. Each of these situations can lead to modified behavior, as illustrated in Figure 13.3. Such modifications might lead to an additional change in schedule or possibly even cause you to cancel the order altogether because you do not want to wait indefinitely.

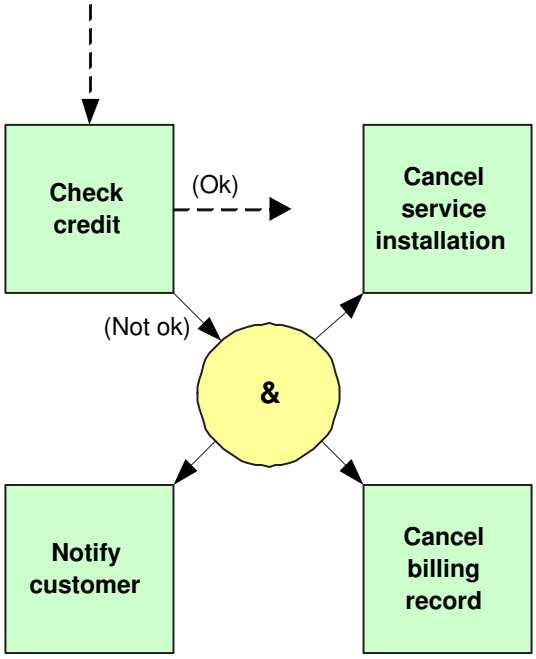

Figure 13.3: Exceptions—unexpected occurrences that interrupt and possibly alter a workflow—can arise during workflow execution

These occurrences are instances of exceptions that can arise during process execution. The number of possible exceptions is extremely large; their scope and the great variety of possible contexts make it practically impossible to specify all exceptions statically and in advance. Unfortunately, the only sure thing about exceptions is that they are far from exceptional. As a consequence, most natural processes are inherently incomplete.

Exceptions are not just alternative flows of control; indeed, the two are conceptually distinct. Attempting to include all exceptions is not only futile, but also would clutter the workflow so much as to render it incomprehensible. For the same reasons that programming languages such as Java treat exceptions separately, it is preferable to think of exceptions as parasitic on the main workflow. Of course, if some exceptions occur often enough to become almost routine, they will be incorporated as explicit alternatives within the workflow, as illustrated in Figure 13.4.

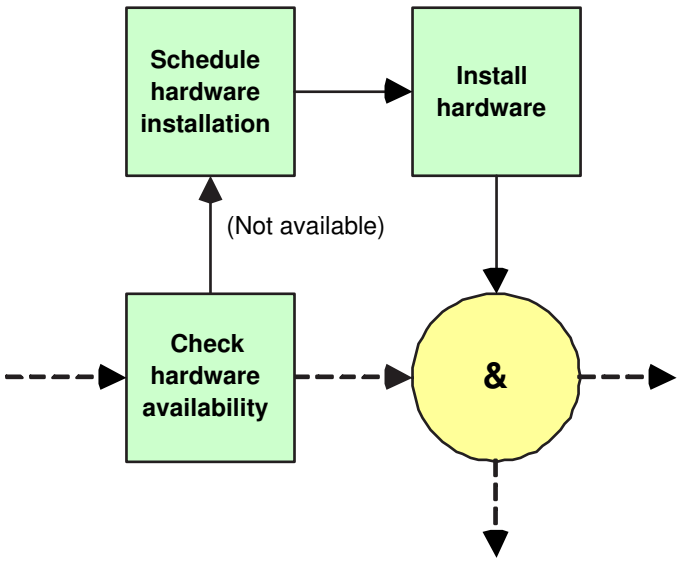

Figure 13.4: An exception that occurs often enough to be considered routine can be incorporated into the workflow as an alternative flow of control

In many cases, multiple workflows can arise and interact with each other. For example, in a telecommunications setting, a channel assignment workflow must wait until enough channels have been created by another workflow. Workflow interactions necessarily occur when business partners collaborate. By design, these interactions are intended to be useful, although some might be pernicious in that one workflow could cause the failure of another. The challenge is to identify the potential interactions and to control them appropriately.

13.3.2 Workflow Interoperability

Workflows interact by sharing data or functionality. An interaction can occur (1) directly, (2) via message passing, (3) through a gateway that translates protocols, or (4) by mutual use of a common repository. For each of these means of interaction, there are three primitive patterns for the interoperability—chained, nested, and synchronized—as depicted in Figure 13.5.

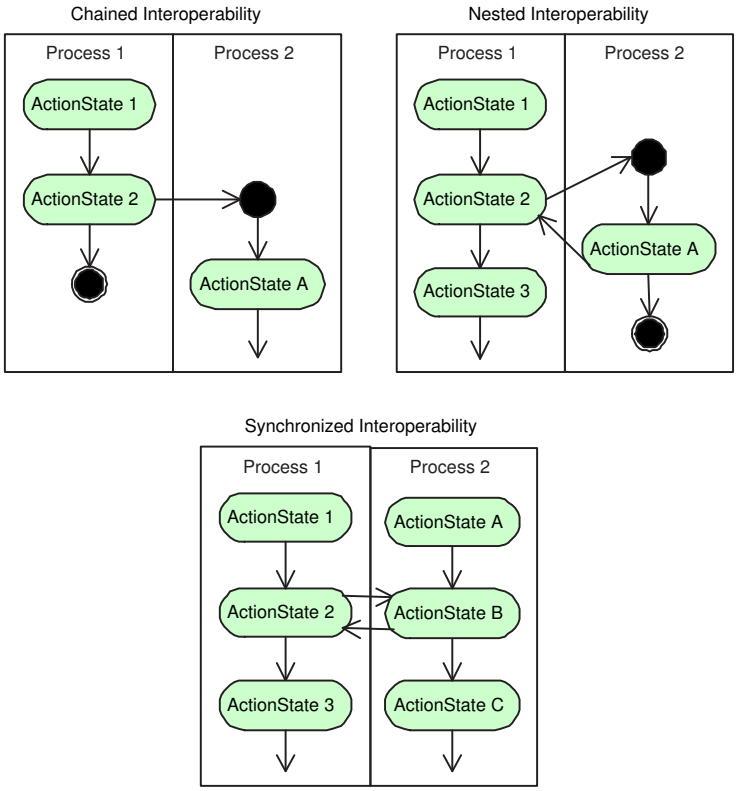

Figure 13.5: Three primitive patterns for workflow interoperability

In a *chained pattern*, one process triggers the creation and enactment of another. The triggering process either terminates at this point or continues independently and concurrently with the second. In a *nested pattern*, a triggering process creates and enacts the other, and then waits for the other to return results and terminate. The triggering process can also execute concurrently with the other process, and receive its results at a later step [yoon Jung et al., 2004]. In a *synchronized pattern*, two concurrently executing processes synchronize at a specified point in their respective executions. Only after both reach that point do they continue independently.

13.3.3 A Metamodel for Workflow

The following terms and meanings are defined and advocated by the Workflow Management Coalition (WfMC):

- A *process*, typically a business process, is a collection of tasks organized into a graph. This reflects the workflow view of processes.
- A *task* is an atomic work item.
- A *service* implements a task and may be implemented.
- An *actor* is a human or machine that performs a task by fulfilling a service.
- A *role* abstracts a set of tasks.
- A *workflow* is an instance of a process that binds and consumes resources in fulfilling the tasks of a process.

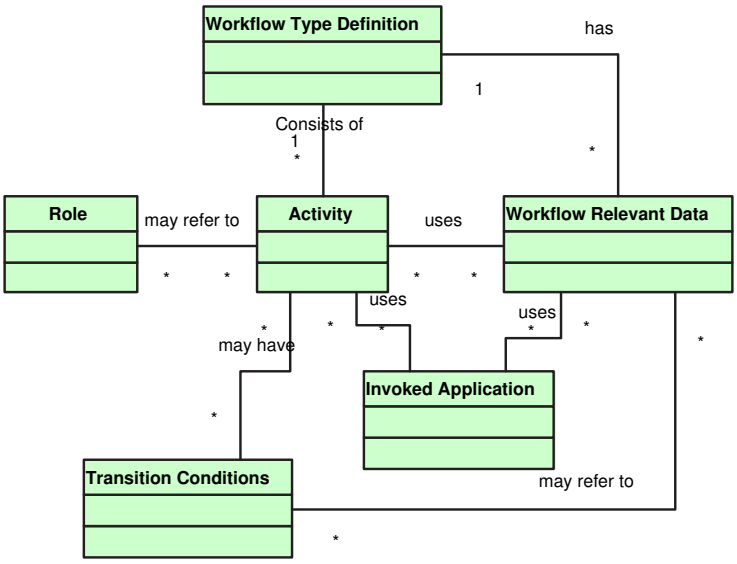

Figure 13.6: A basic metamodel for the definition of a workflow and its terminology [WfMC]

A metamodel that shows the relationship among some of these terms is depicted in Figure 13.6. Figure 13.7 shows the states and state transitions that can occur during the execution of a typical workflow.

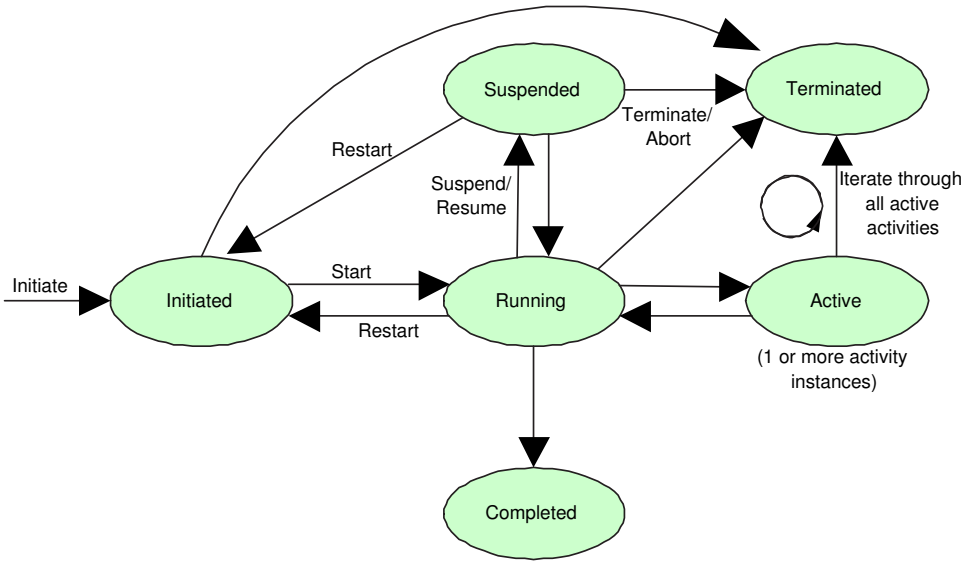

Figure 13.7: A state-transition diagram for an activity or a workflow [WfMC]

13.3.4 Interoperation

A workflow represents the interoperation of possibly several applications and databases. This interoperation can be achieved by building an appropriate workflow from scratch. However, standards activities, such as those being led by the WfMC, attempt to define a reference model for workflow management. The reference model describes how workflow engines ought to be connected to applications, databases, development tools, runtime tools, and each other. Figure 13.8 shows the WfMC reference model, depicting the major components and their interfaces within a workflow architecture. Agents can contribute to achieving interoperation among the different resources while satisfying the resources' local constraints.

Another, more profound, kind of interoperation occurs among different workflows. A workflow represents a meaningful unit of processing that affects a number of people and information resources. Clearly, multiple units must interact with each other, because some people participate in more than one, and the units inevitably share resources. Workflow designers must understand, model, and manage these interactions properly. If they do not, all manner of chaos may ensue—and indeed often does. For example, one workflow of the above communications provider might be underway to upgrade wiring with a view to discarding the old wiring, while another workflow might treat the old wiring as freely available and be actively assigning new telephone circuits to it.

This requires an ability to communicate and negotiate. Such coordination benefits from standards that enable workflows modeled and managed by tools from different vendors to be related. One standard is the Simple Workflow Access Protocol (SWAP) announced by WfMC

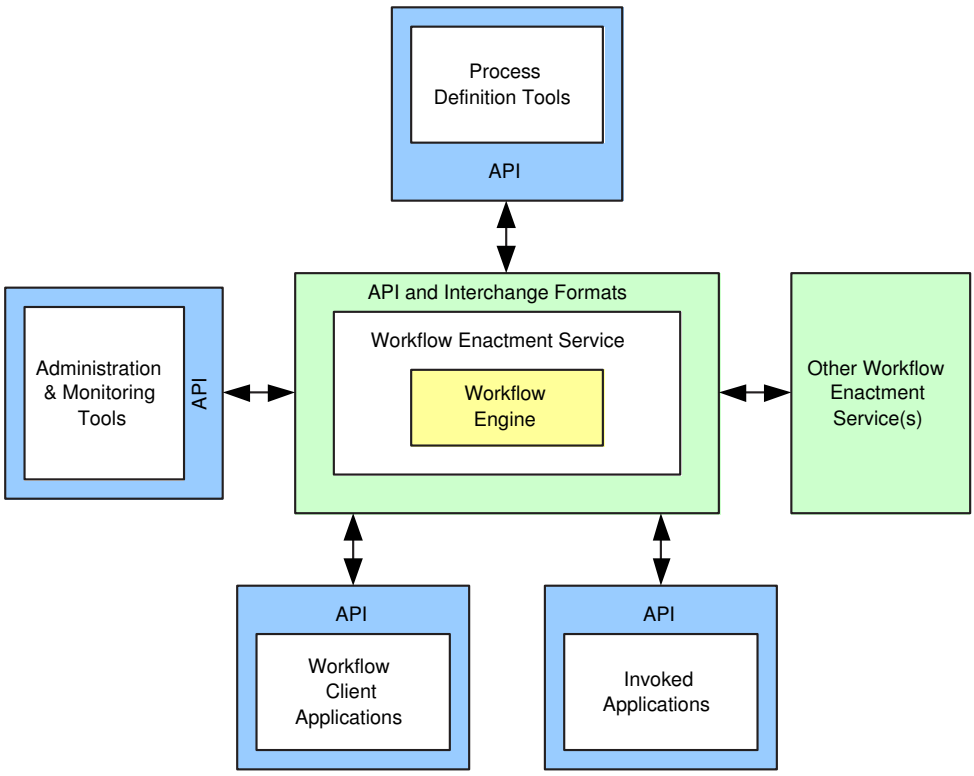

Figure 13.8: A reference model for workflow management systems, showing their major components and interfaces [WfMC]

and the IETF. SWAP governs both the control and monitoring of workflows. Control means instantiating the workflow, starting it, stopping it, being informed of exceptions, being informed of completions, and obtaining the results. Monitoring means checking on the current status of the workflow and obtaining its history. SWAP's protocol for basic interaction is

- The client invokes createProcessInstance command on the workflow server.
- The server returns the URI of the workflow instance.
- The client sends its own URI to the instance.
- When it is done, the workflow instance invokes the completed command on the client.

Other commands can be invoked by both the client and server during execution to provide status, exception, and result information. The resultant protocol is lightweight and, although it has largely been superseded by business process protocols such as BPEL4WS, it is representative of the capabilities needed to describe, control, and monitor a workflow.

13.3.5 State of the Art

There are many workflow tools available in the marketplace—at least 100, and by some counts as many as 250. Each tool provides some type of modeling mechanism coupled with an execution framework. In general, the metamodels underlying most workflow tools are based on a variant of activity networks, which show different activities as nodes and use links to represent various temporal and exception dependencies among the nodes. Figures 13.2 to 13.4 reflect this general idea.

System analysts design workflows on the basis of their understanding of the given organization and the abstractions the chosen workflow tool supports. Once designed, the workflow can be executed automatically by the tool, typically resulting in improved efficiency. For example, when workflows involve human workers, the workers can be automatically informed of the tasks they should be performing and given the resources they need to complete the tasks, thereby reducing idle time.

13.3.6 Challenges Facing Workflow Technology

Workflow technology is not universally acclaimed, and many CIOs are not convinced of its capabilities and benefits. One problem is that current workflow technology is often too rigid. Because workflows are constructed prior to use and are enforced by some central authority, this rigidity is inevitable. However, the lack of freedom accorded to human participants causes workflow management systems to appear unfriendly. As a result, workflows are often ignored or circumvented.

This rigidity also causes productivity losses by making it harder to accommodate the flexible, *ad hoc* reasoning that is the strong suit of human intelligence. This need for flexibility is most apparent when an exception occurs and rigid workflow management tools behave

incorrectly. In our earlier example, if the credit bureau is unresponsive, a poorly designed workflow might just wait indefinitely, whereas a flexible one would let a human make a decision based on available information.

Another challenge is that system requirements are rarely static. A workflow's design context might not remain applicable in every detail over the workflow's lifetime. Dynamic requirements can necessitate arbitrary extensions not recorded in the workflow model itself. Suppose our telecommunications provider makes a special offer at the start of an academic year whereby it waives credit-history checks of full-time students. Would this change require the workflow to be redesigned and reinstalled?

In the future, much as they enable databases to interoperate today, agents (discussed in Chapter 15) will enable Internet-wide workflow processes to be coordinated and executed flexibly.

13.4 Business Process Languages

Business processes are distinguished by being possibly long-running, involving multiple autonomous participants, and having correctness and completion guarantees. Such guarantees might have contractual and even legal implications.

Business processes can be described in two ways. *Executable business processes* model the actual behavior of a participant in a business interaction. *Business protocols*, in contrast, use process descriptions that specify the mutually visible message exchange behavior of each of the parties involved in the protocol, without revealing their internal behavior. That is, the descriptions specify interfaces. The process descriptions for business protocols are called abstract processes and cannot be executed.

13.4.1 BPEL4WS

The Business Process Execution Language for Web Services (BPEL4WS) can serve as both an implementation language for executable processes and a description language for nonexecutable business protocols. It defines a model and a grammar for describing how multiple Web service interactions among the process's participants, termed *partners*, are coordinated to achieve a business goal, as well as the state and the logic necessary for the coordination to occur. Interactions with each partner occur through lower-level Web service interfaces, as might be defined in WSDL. BPEL4WS can define mechanisms for dealing with exceptions and processing faults, including how individual or composite process components are to be compensated when exceptions and faults occur or when a partner requests an abort. Figure 13.9 shows the metamodel for BPEL4WS.

A BPEL4WS document uses XML to describe the following aspects of a business process:

- *partners:* a list of the Web services invoked as part of the process.

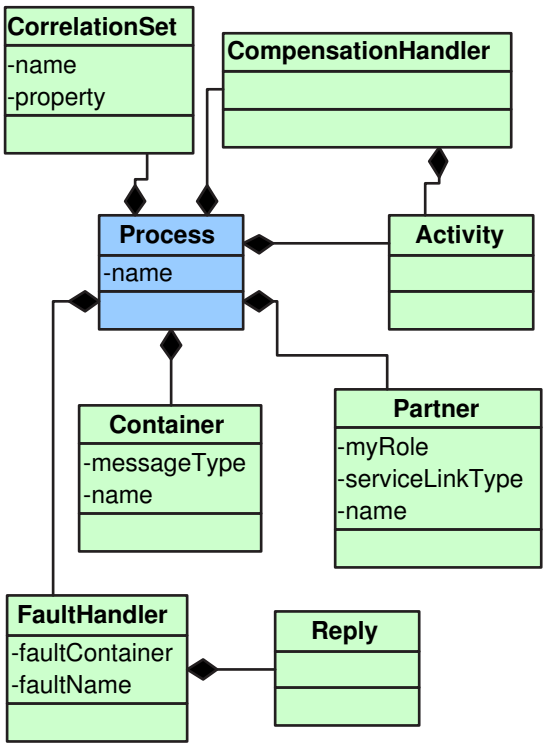

Figure 13.9: The BPEL4WS metamodel, specifying that a process consists of an activity, a number of partners and containers with specific correlation sets, fault handlers, and compensation handlers

- *containers:* the data containers used by the process, providing their definitions in terms of WSDL message types. Containers are needed to store state data and process history based on messages exchanged among the component processes.
- *variables:* the variables that are used and flow through the process.
- *faultHandlers:* the exception handling routines.
- *compensationHandler:* compensation actions to take when a transaction rollback occurs.
- *eventHandlers:* routines for handling external (asynchronous) events.
- *correlationSets:* precedences and correlations among Web service invocations that cannot be expressed as part of the main process logic.
- *main process logic*: a series of nested control flow structures that combine primitive activities into more complex algorithms. The control structures include
  - *sequence*, for serial execution.
  - *while*, to implement a loop.
  - *switch*, for multiway branching.
  - *pick*, for choosing among alternative paths based on an external event.
  - *flow*, for parallel execution. Within activities executing in parallel, execution order constraints are indicated by using service links.
- The control structures relate the following atomic actions:
  - invoke, invoking a specific Web service.
  - receive, a server waiting to receive a message from a client, which would invoke the server's service.
  - reply, generating the response to an invocation request.
  - wait, waiting either for a deadline or some amount of time.
  - assign, assigning a value, which might have come from a received message, to a variable.
  - throw, indicating that something went wrong.
  - terminate, terminating an entire service instance.
  - empty, doing nothing.

In modeling a business protocol as an abstract process, BPEL4WS describes just public aspects of the protocol. For example, in a supply-chain protocol, BPEL4WS would describe the roles of a buyer and a seller as abstract processes, with their relationship modeled as a

service link. Abstract processes are restricted to manipulation of values contained in message properties, and use nondeterministic values to reflect the results of hidden private behavior.

In modeling an executable business process, BPEL4WS does not have to completely define a partner's individual implementation, but it does define a portable execution format for business processes. Such processes execute and interact with their partners in a consistent way regardless of the supporting platform or the programming model used by a particular implementation.

The result of using BPEL4WS to model an executable business process is a new Web service composed out of existing services. The interface of the composite service is a collection of WSDL portTypes, just like any other Web service. Figure 13.10 illustrates this external view of a BPEL4WS process.

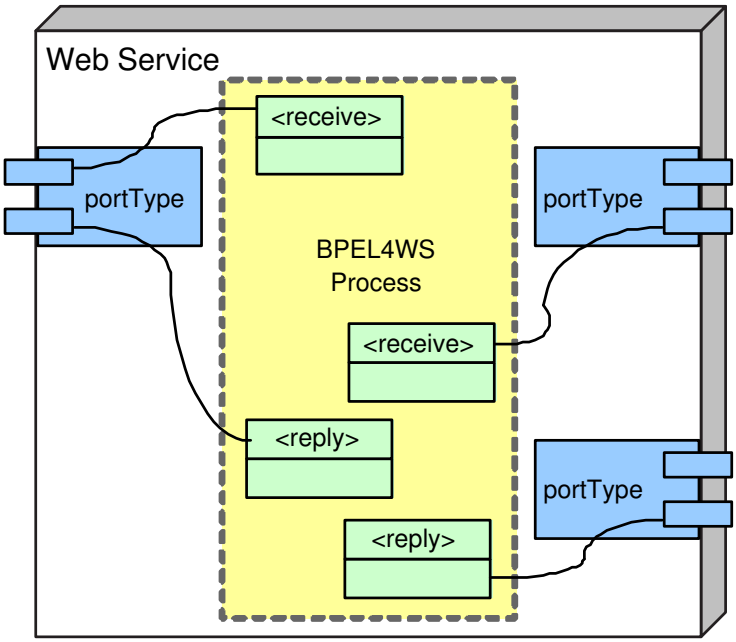

Figure 13.10: A BPEL4WS process is a composite Web service with an interface that is a collection of WSDL portTypes, just like any other Web service

13.4.1.1 Transaction Flow

BPEL4WS provides a compensation protocol that is a variant of earlier work on Sagas and open nested transactions (see Section 11.5 on page 224). It enables flexible control of roll-backs and reversals through application-specific definitions for fault handling and compensation, resulting in Long-Running (Business) Transactions (LRTs).

An LRT can be undone by reversing individual operations, using business rules that typically depend on the application. Scope elements delineate the parts of a behavior that are allowed to be reversible by a compensation handler. Scopes can be nested to an arbitrary depth.

An LRT occurs within a single business process instance, and there is no distributed coordination among the partners regarding an agreed-upon outcome. Achieving distributed agreement is outside the scope of BPEL4WS, to be solved by using protocols described in WS-Transaction (see Section 12.4 on page 249). In essence, WS-Transaction uses WS-Coordination to extend BPEL4WS to provide a context for transactional agreements between services. Different agreements may be described in an attempt to achieve consistent, desirable behavior while respecting service autonomy.

13.4.1.2 Implementing a BPEL4WS Web Service

A BPEL4WS process is an algorithm expressed as a flow chart, where each step is an activity. Information is passed between activities through data containers and the use of ⟨assign⟩ statements. For example, a customer's address would be copied from a purchase order to a shipping request by the following:

```
<assign>
  <copy>
    <from container="PO" part="customerAddress"/>
    <to container="shippingRequest" part="customerInfo"/>
  </copy>
</assign>
```

A service link is used to define the relationship between two partners and the role that each partner plays. For example, a service link between a buyer and seller might be

```
<serviceLinkType name="BuySellLink"
 xmlns="http://schemas.xmlsoap.org/ws/2002/07/service-link/">
  <role name="Buyer">
    <portType name="BuyerPortType"/>
  </role>
  <role name="Seller">
    <portType name="SellerPortType"/>
  </role>
</serviceLinkType>
```

The following is a complete example of an executable BPEL4WS process for the implementation of a stock quoting service:

```
<!ENTITY BPEL
  "http://schemas.xmlsoap.org/ws/2002/07/business-process"
<process name="simple"
```

```
targetNamespace="urn:simple:stockQuoteService"
xmlns:tns="urn:simple:stockQuoteService"
xmlns:sqp="http://tempuri.org/services/stockquote"
xmlns=&BPEL;/>

<containers>
  <container name="request"
             messageType="tns:request"/>
  <container name="response"
             messageType="tns:response"/>
  <container name="invocationRequest"
             messageType="sqp:GetQInput"/>
  <container name="invocationResponse"
             messageType="sqp:GetQOutput"/>
</containers>

<partners>
  <partner name="caller"
           serviceLinkType="tns:StockQuoteSLT"/>
  <partner name="provider"
           serviceLinkType="tns:StockQuoteSLT"/>
</partners>

<sequence name="sequence">
  <receive name="receive" partner="caller"
           portType="tns:StockQuotePT"
           operation="wantQuote" container="request"
           createInstance="yes"/>
  <assign>
    <copy>
        <from container="request" part="symbol"/>
        <to container="invocationRequest" part="symbol"/>
    </copy>
  </assign>
  <invoke name="invoke" partner="provider"
          portType="sqp:StockQuotePT"
          operation="getQuote"
          inputContainer="invocationRequest"
          outputContainer="invocationResponse"/>
  <assign>
    <copy>
        <from container="invocationResponse" part="quote"/>
        <to container="response" part="quote"/>
    </copy>
  </assign>
  <reply name="reply" partner="caller"
```

```
            portType="tns:StockQuotePT"
            operation="wantQuote" container="response"/>
  </sequence>
</process>
```

This process is a simple five-step sequence that begins when a request for a quote is received from the caller. The request is copied to an invocation container, the getQuote operation is invoked with the parameters of the request, the result is copied to a result container, and a reply is returned to the requester.

13.4.1.3 UML to BPEL4WS Translation

The Unified Modeling Language (UML) is a popular representation and methodology for characterizing software and information processes, so we consider its use here for describing business processes. BPEL4WS processes are stateful and can have instances, so the appropriate UML construct for modeling them is a class with stereotype ≪Process≫ and whose attributes are the state variables of the process. The behavior of the class is described using an activity diagram. Other aspects of a mapping from UML to BPEL4WS are shown in Table 13.1.

Table 13.1: UML to BPEL4WS Mappings

UML Construct	**BPEL4WS Concept**
≪*process*≫ *class*	BPEL process definition
Activity graph on a ≪process≫ class	BPEL activity hierarchy
≪*process*≫ *class attributes*	BPEL variables
Hierarchical structure	BPEL sequence and flow activities
Control flow	BPEL sequence and flow activities
≪*receive*≫ activities	BPEL activities
≪*reply*≫ activities	BPEL activities
≪*invoke*≫ activities	BPEL activities

13.4.2 BPML

The Business Process Modeling Language (BPML) and BPEL4WS share similar roots in Web services (SOAP, WSDL, and UDDI), take advantage of the same XML technologies (XPath and XML Schema), and are designed to leverage other specifications (WS-Security and WS-Transaction). Beyond these areas of commonality, BPML supports the modeling of real-world business processes through its support for advanced semantics, such as nested

processes and complex compensated transactions. BPML builds on the foundation of WSCI for expressing public interfaces and choreographies.

13.4.3 ebXML

The Electronic Business Extensible Markup Language (ebXML) has been established by the United Nations CEFACT (Centre for Trade Facilitation and Electronic Business) and the OASIS (Organization for the Advancement of Structured Information Standards) group to provide specifications for defining standard business processes and trading agreements among different organizations. It also specifies the business messages that are exchanged as part of a business process. The objective is for ebXML to be a global standard for governmental and commercial organizations of all sizes to find business partners and interact with them.

Suppose you are the owner of a disk-drive manufacturing company that sells its disk drives to the computer industry, and you decide that your company should receive purchase orders electronically. To implement this as an ebXML business process, you could follow the typical three-step procedure described in Figure 13.11.

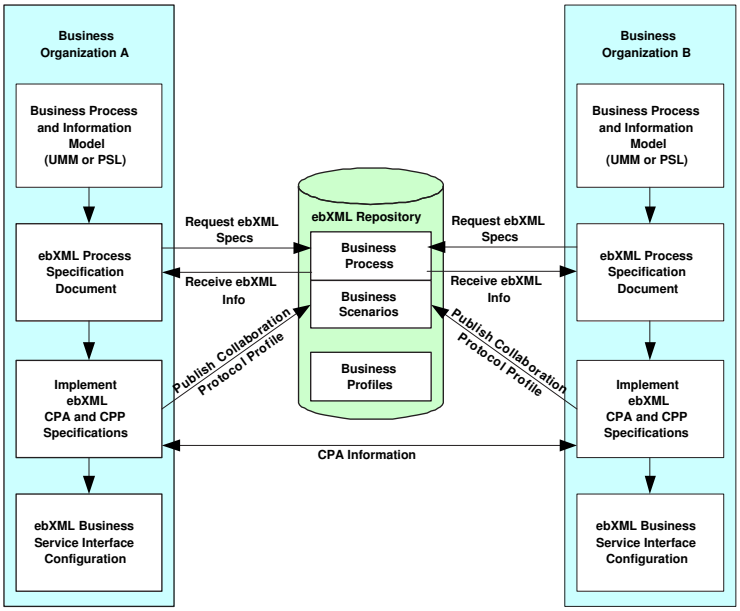

Figure 13.11: The design of an ebXML system typically follows the steps shown here, from modeling a business process to constructing the CPP and CPA specifications

According to this procedure, the recommended way for you to design an ebXML process is to first construct a model of one of your business processes, using a process modeling language. For example, you might use the UN/CEFACT Modeling Methodology (UMM), which

is a qualified UML notation for business processes, or you might use the Process Specification Language (PSL). Based on your process model and using the ebXML Business Process Specification Schema (BPSS), you would then extract and format the set of elements necessary to configure an ebXML runtime system that will be able to execute the required set of ebXML business transactions. The result is an ebXML Process-Specification Document, which might be a RosettaNet Partner Interface Process (PIP) as introduced in Section 13.4.4 on page 284. The following example describes a transaction whereby a customer (buyer) issues a request for a purchase order (PO) and your company, the seller, confirms the purchase order.

Listing 13.1 is the Process-Specification Document corresponding to a well-known RosettaNet PIP for purchase orders and acknowledgments. RosettaNet has given this PIP an identifier 3A4, hence the use of that string in the document.

Listing 13.1: An example ebXML Business Process Specification Schema document

```
<ProcessSpecification
  xmlns="http://www.ebxml.org/BusinessProcess"
  name="PIP3A4RequestPurchaseOrder">

<!-- The request document and its XML Schema -->
 <BusinessDocument name="PO␣Request"
   nameID="Pip3A4PORequest"
   specificationLocation="PurchaseOrderRequest.xsd"/>

<!-- The confirmation document and its XML Schema -->
 <BusinessDocument name="PO␣Confirmation"
   nameID="Pip3A4POConfirmation"
   specificationLocation="PurchaseOrderConfirmation.xsd"/>

<!-- This process specification has one business -->
<!-- transaction consisting of a requesting and -->
<!-- a responding business activity-->
 <BusinessTransaction name="Request␣PO"
  nameID="RequestPO_BT">
   <RequestingBusinessActivity
    name="PO␣Request␣Action"
    nameID="PORequestAction"
    isAuthorizationRequired="true"
    isNonRepudiationRequired="true"
    timeToAcknowledgeReceipt="PT2H">
    <DocumentEnvelope
      businessDocument="PO␣Request"
      businessDocumentIDRef="Pip3A4PurchaseOrderRequest"/>
   </RequestingBusinessActivity>
   <RespondingBusinessActivity
    name="PO␣Confirmation␣Action"
    nameID="POConfirmationAction"
```

```
    isAuthorizationRequired="true"
    isNonRepudiationRequired="true"
    timeToAcknowledgeReceipt="PT2H">
    <DocumentEnvelope
     businessDocument="PO␣Confirmation"
     businessDocumentIDRef="Pip3A4PurchaseOrderConfirmation"/>
  </RespondingBusinessActivity>
 </BusinessTransaction>

<!-- The binary collaboration asserts that the buyer is -->
<!-- the initiator of the above business transaction and -->
<!-- the seller is the responder, and the process begins -->
<!-- in the Request PO state -->
 <BinaryCollaboration name="Request␣PO"
   nameID="RequestPO_BC">
  <InitiatingRole name="Buyer" nameID="BuyerId"/>
  <RespondingRole name="Seller" nameID="SellerId"/>
  <Start toBusinessState="Request␣PO"/>
  <BusinessTransactionActivity name="Request␣PO"
    nameID="RequestPO_BTA"
    businessTransaction="Request␣PO"
    businessTransactionIDRef="RequestPO_BT"
    fromAuthorizedRole="Buyer"
    fromAuthorizedRoleIDRef="BuyerId"
    toAuthorizedRole="Seller"
    toAuthorizedRoleIDRef="SellerId"
    timeToPerform="PT1D"/>
 </BinaryCollaboration>
</ProcessSpecification>
```

This example specifies two business documents—a purchase order request and a purchase order confirmation—whose format has been defined by RosettaNet, one business transaction named "Request PO" that your company will support, and a choreography for this transaction into a binary collaboration with buyer and seller roles and associated parameters, such as timeToPerform. The various times are encoded as strings such as "PT1D" and "PT2H"—these use the XML Schema duration syntax adopted from the ISO 8601 standard and mean periods of time equal to one day and two hours, respectively.

You and your customers use this ebXML Process-Specification Document to form ebXML Collaboration-Protocol Profiles (one for you (as the seller) and one for each customer (as a buyer)). Finally, you would construct a Collaboration-Protocol Agreement, which links you with any one of your customers. Listing 13.2 is an example of the CPP for one of your customers, named PCInc and identified by its DUNS number.

Listing 13.2: An example of an ebXML Collaboration Protocol Profile

```
<cp:CollaborationProtocolProfile
```

```
  xmlns:cp="http://www.ebxml.org/specs/cpp-cpa-v2_0.xsd">
  <cp:PartyInfo cp:partyName="PCInc"
    cp:defaultMshChannelId="asyncChannelA1">
    <cp:PartyId
      cp:type="urn:ebxml-cppa:partyid-type:duns">
        123456789
    </cp:PartyId>
    <cp:PartyRef
      xlink:href="http://PCInc.com/about.html"/>
    <cp:CollaborationRole cp:id="BuyerId">
      <cp:ProcessSpecification cp:version="2.0"
        cp:name="PIP3A4RequestPurchaseOrder"
        xlink:type="simple"
        xlink:href=
         "http://www.rosettanet.org/processes/3A4.xml"/>
     <cp:Role cp:name="Buyer"
        xlink:href=
         "http://www.rosettanet.org/processes/3A4.xml#Buyer"/>
      <cp:ServiceBinding>
        <cp:Service>
          bpid:icann:rosettanet.org:3A4v2.0
        </cp:Service>
        <cp:CanSend>
          <cp:ThisPartyActionBinding cp:id="PCInc_ABID1"
            cp:action="PO Request Action"
            cp:packageId="PCInc_RequestPackage">
            <cp:ChannelId>asyncChannelA1</cp:ChannelId>
            <cp:BusinessTransactionCharacteristics
              cp:isNonRepudiationRequired="true"
              cp:isSecureTransportRequired="true"
              cp:isAuthorizationRequired="true"
              cp:timeToAcknowledgeReceipt="PT2H"
              cp:timeToPerform="PT1D"/>
          </cp:ThisPartyActionBinding>
        </cp:CanSend>
      </cp:ServiceBinding>
    </cp:CollaborationRole>
  </cp:PartyInfo>
</cp:CollaborationProtocolProfile>
```

Listing 13.3 provides a feel for the additional practically important, but conceptually trivial details that must be worked out in a collaboration protocol profile to make it effective. These details involve message delivery, transport protocol (the following is a wordy way of specifying HTTPS), reliable messaging, and security, including nonrepudiation.

Listing 13.3: Additional details to include within the partyInfo element of Listing 13.2 to

make it complete

```
<cp:DeliveryChannel cp:channelId="asyncChannelA1"
  cp:transportId="transportA2"
  cp:docExchangeId="docExchangeA1">
  <cp:MessagingCharacteristics
    cp:syncReplyMode="none"
    cp:ackRequested="always"
    cp:ackSignatureRequested="always"
    cp:duplicateElimination="always"/>
</cp:DeliveryChannel>
<cp:Transport cp:transportId="transportA2">
  <cp:TransportSender>
    <cp:TransportProtocol cp:version="1.1">
      HTTP
    </cp:TransportProtocol>
    <cp:TransportClientSecurity>
      <cp:TransportSecurityProtocol cp:version="3.0">
        SSL
      </cp:TransportSecurityProtocol>
   </cp:TransportClientSecurity>
  </cp:TransportSender>
</cp:Transport>
<cp:DocExchange cp:docExchangeId="docExchangeA1">
  <cp:ebXMLSenderBinding cp:version="2.0">
    <cp:ReliableMessaging>
      <cp:Retries>3</cp:Retries>
      <cp:RetryInterval>PT2H</cp:RetryInterval>
      <cp:MessageOrderSemantics>
        Guaranteed
      </cp:MessageOrderSemantics>
   </cp:ReliableMessaging>
   <cp:SenderNonRepudiation>
     <cp:NonRepudiationProtocol>
        http://www.w3.org/2000/09/xmldsig#
     </cp:NonRepudiationProtocol>
     <cp:HashFunction>
        http://www.w3.org/2000/09/xmldsig#sha1
     </cp:HashFunction>
     <cp:SignatureAlgorithm>
        http://www.w3.org/2000/09/xmldsig#dsa-sha1
     </cp:SignatureAlgorithm>
   </cp:SenderNonRepudiation>
   <cp:SenderDigitalEnvelope>
     <cp:DigitalEnvelopeProtocol cp:version="2.0">
       S/MIME
     </cp:DigitalEnvelopeProtocol>
```

```
        <cp:EncryptionAlgorithm>
          DES–CBC
        </cp:EncryptionAlgorithm>
      </cp:SenderDigitalEnvelope>
    </cp:ebXMLSenderBinding>
  </cp:DocExchange>
```

The CPP specifies the buyer role for this customer in a RosettaNet PIP, with a *service binding element* specifying the customer's ability to send a purchase order request. A *delivery channel element* defines characteristics of the business transaction and the messaging. The *transport element* defines the buyer's network communication capabilities. Together, the CPA and CPP agreements serve as configuration files (e.g., messaging headers) for ebXML Business Service Interface software.

To summarize, the vocabulary used for an ebXML specification consists of the following three parts:

1. A Process-Specification Document describing the activities of the parties in an ebXML interaction.

2. A Collaboration Protocol Profile (CPP), which describes an organization's profile, i.e., which business processes it supports, its roles in those processes, the messages exchanged, and the transport mechanism for the messages.

3. A Collaborative Partner Agreement (CPA), which is an intersection of two CPP's, representing a technical agreement between two or more partners, and potentially negotiated as shown in Figure 13.12. It may have legal binding.

13.4.3.1 Implementing ebXML

ebXML is just a set of specifications, and an enterprise may build and deploy its own ebXML compliant application. In addition, ebXML compliant applications and components are commercially available as shrink-wrapped products. ebXML can also be implemented by a *Business Service Interface* (BSI), a wrapper that enables a noncompliant party to properly participate in an ebXML exchange. As shown in Figure 13.13, a Business Service Interface can interface with a legacy system, is aware of its own Collaborative Protocol Profile, and handles transactions based on all of the current agreements (CPAs).

Listing 13.4: The general form of an ebXML Collaboration Protocol Agreement

```
<CollaborationProtocolAgreement
  xmlns="http://www.ebxml.org/namespaces/tradePartner"
  xmlns:cp="http://www.ebxml.org/specs/cpp-cpa-v2_0.xsd"
  xmlns:bpm="http://www.ebxml.org/namespaces/businessProcess"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  cp:cpaid="OurMutualCPA" cp:version="2.0">
```

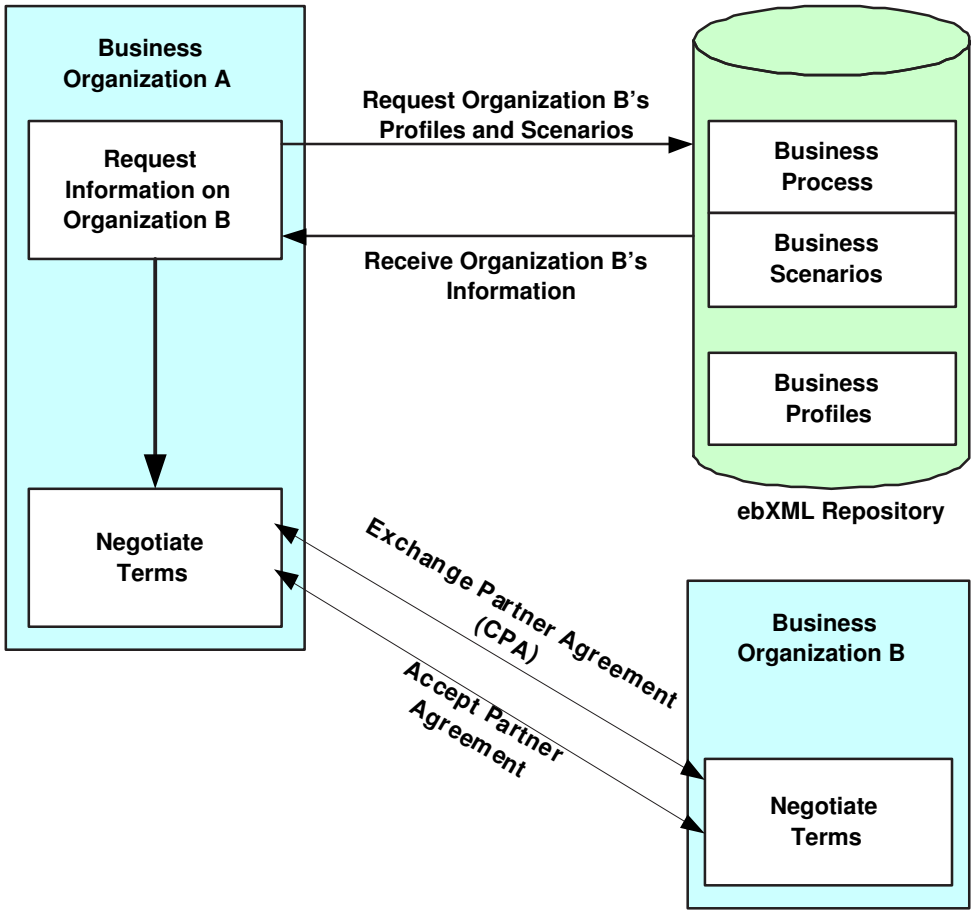

Figure 13.12: Discover partner information and negotiate

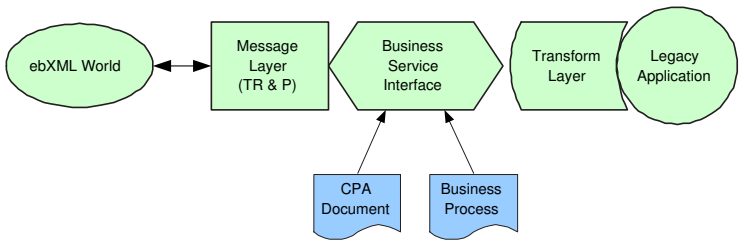

Figure 13.13: Business Service Interface

```
  <cp:Status value="proposed"/>
  <cp:Start>2004-04-07T08:30:00</cp:Start>
  <cp:End>2006-04-07T23:59:59</cp:End>
  <!--ConversationConstraints MAY appear 0 or 1 time-->
  <cp:ConversationConstraints cp:invocationLimit="100"
    concurrentConversations="4"/>
  <cp:PartyInfo> <!--my information as in CPP -->
        ...
  </cp:PartyInfo>
  <cp:PartyInfo> <!--your information as in CPP -->
        ...
  </cp:PartyInfo>
  <cp:Packaging id="N20"> <!--one or more-->
        ...
  </cp:Packaging>
  <!--ds:signature MAY appear 0 or more times-->
  <ds:Signature>any combination of text and elements
  </ds:Signature>
  <!-- cp:Comment may appear 0 or more times-->
  <cp:Comment xml:lang="en-gb">any text</cp:Comment>
</CollaborationProtocolAgreement>
```

The PartyInfo element consists of the following child elements:

- One or more REQUIRED PartyId elements that provide a logical identifier for the organization (Party).
- A REQUIRED PartyRef element that provides a pointer to more information about the Party.
- One or more REQUIRED CollaborationRole elements that identify the roles that this Party can play in the context of a Process Specification.
- One or more REQUIRED Certificate elements that identify the certificates used by this Party in security functions.
- One or more REQUIRED DeliveryChannel elements that define the characteristics of each delivery channel that the Party can use to receive Messages. It includes both the transport level (e.g., HTTP) and the messaging protocol (e.g., ebXML Message Service).
- One or more REQUIRED Transport elements that define the characteristics of the transport protocol(s) that the Party can support to receive Messages.
- One or more REQUIRED DocExchange elements that define the message-exchange characteristics, such as the messaging protocol, that the Party can support.

Listing 13.5: The PartyInfo field for an ebXML Collaboration Protocol Agreement

```
<PartyInfo>
    <PartyId type="..."> <!--one or more-->
        ...
    </PartyId>
    <PartyRef xlink:type="...", xlink:href="..."/>
    <CollaborationRole>  <!--one or more-->
        ...
    </CollaborationRole>
    <Certificate> <!--one or more-->
        ...
    </Certificate>
    <DeliveryChannel> <!--one or more-->
        ...
    </DeliveryChannel>
    <Transport> <!--one or more-->
        ...
    </Transport>
    <DocExchange> <!--one or more-->
        ...
    </DocExchange>
</PartyInfo>
```

Listing 13.6: The CollaborationRole field for an ebXML Collaboration Protocol Agreement

```
<CollaborationRole id="N11">
  <ProcessSpecification name="BuySell" version="1.0">
     ...
  </ProcessSpecification>
  <Role name="buyer" xlink:href="..."/>
  <CertificateRef certId="N03"/>
  <!--primary binding with preferred DeliveryChannel-->
  <ServiceBinding name="aProc"
    channelId="N02" packageId="N06">
    <!--override default DeliveryChannel-->
    <Override action="OrderAck"
      channelId="N05" packageId="N09"
      xlink:type="simple" xlink:href="..."/>
  </ServiceBinding>
  <!-- the first alternate binding -->
  <ServiceBinding channelId="N04" packageId="N06">
    <Override action="OrderAck"
      channelId="N05" packageId="N09"
      xlink:type="simple" xlink:href="..."/>
  </ServiceBinding>
</CollaborationRole>
```

Based on the above CPA and CPP documents, the following would be an example of a message header for sending a Purchase Order Request document from a buyer to a seller.

Listing 13.7: An example SOAP message header for sending a Purchase Order Request document

```
<SOAP:Envelope
  xmlns:SOAP="http://schema.xmlsoap.org/soap/envelope/">
  <SOAP:Header
    xmlns:eb="http//www.ebxml.org/msg-header-2_0.xsd">
    <eb:MessageHeader id="123" eb:version="2.0"
      SOAP:mustUnderstand="1">
      <eb:From><eb:PartyId>123456</eb:PartyId></eb:From>
      <eb:To>
        <eb:PartyId eb:type="someType">987654</eb:PartyId>
        <eb:Role>
          http://rosettanet.org/processes/3A4.xml#seller
        </eb:Role>
      </eb:To>
      <eb:CPAId>uri:companyA-and-companyB-cpa</eb:CPAId>
      <eb:ConversationId>987654321</eb:ConversationId>
      <eb:Service eb:type="anyURI">
        bpid:icann:rosettanet.org:3A4v2.0
      </eb:Service>
      <eb:Action>Purchase Order Request Action</eb:Action>
      <eb:MessageData>
        <eb:MessageId>UUID-2</eb:MessageId>
        <eb:Timestamp>2000-07-25T12:19:05</eb:Timestamp>
        <eb:RefToMessageId>UUID-1</eb:RefToMessageId>
      </eb:MessageData>
      <eb:DuplicateElimination/>
    </eb:MessageHeader>
  </SOAP:Header>
  <SOAP:Body
    xmlns:eb="http//www.ebxml.org/msg-header-2_0.xsd">
    <eb:Manifest eb:version="2.0">
                  ...
    </eb:Manifest>
  </SOAP:Body>
</SOAP:Envelope>
```

All things considered, ebXML's BPSS enables us to express collaboration protocols and agreements about protocols in a nice manner. This facilitates interoperation in cross-enterprise settings. However, despite its overall appealing structure, BPSS is quite limited in what it can express. In particular, it suffers from two main limitations:

Expressiveness. BPSS is limited to simple request-response protocols. Business interactions can be quite complex and, as we saw in the context of extended transactions, can be long-lived. It is important to understand the contents of the states of an ongoing interaction, so that exceptions can be accommodated in a manner that respects the participants' local constraints as well as the context in which they are doing business.

Semantics. BPSS lacks a formal semantics and, thus, it is not clear if specifications constructed by one party would have the same interpretations by another party. Such disagreements are common and expensive. The motivation behind the development of ontologies as discussed in the context of describing services also applies to describing processes involving services.

Both of these limitations are being addressed in emerging approaches, some of which are introduced later in this book.

13.4.4 RosettaNet

RosettaNet is a consortium of information technology, semiconductor manufacturing, and telecommunications companies working to create and implement open e-business process standards. These standards, in the form of business documents, such as purchase orders, and Partner Interface Processes (PIPs), such as purchasing, comprise a common e-business language that can align the interactions among supply-chain partners on a global basis. As exemplified in Figure 13.14, a PIP in RosettaNet defines the process of exchanging messages between two partners. Unfortunately, after a message departs a partner's computer system, it is impossible to find out whether it was received and correctly processed by the partner organization. All RosettaNet offers is a fixed time-out for the confirmation of each message. If the message is not confirmed during this time (e.g., 2 hours), the original partner (recognizing that the message has been sent already) resends the message to the same or another partner.

Companies need to monitor the processes and status of messages not only internally, but also externally. It means that they must know not only the status of the order from their own perspective (e.g., that the PO has been sent 1 hour ago), but also the status of the order from the perspective of their partners (e.g., the order has been scheduled for acceptance 10 minutes ago). Distributed information systems remain isolated entities and do not allow for this level of visibility within supply chains.

Some organizations are adopting proprietary business protocols that are not machine interpretable. For example, RosettaNet uses UML diagrams to describe PIPs and relationships among the messages exchanged as part of the PIPs. The meaning of the UML diagrams is informal and no direct machine interpretation is possible. Electronic Data Interchange (EDI) does not define any standard processes at all: industrial applications of EDI are based on best practices. However, there is no recognized way to describe the process of exchanging EDI messages.

We saw above how ebXML's BPSS could be used to define a RosettaNet PIP. An important distinction between RosettaNet PIPs and ebXML BPSS is that PIPs define specific

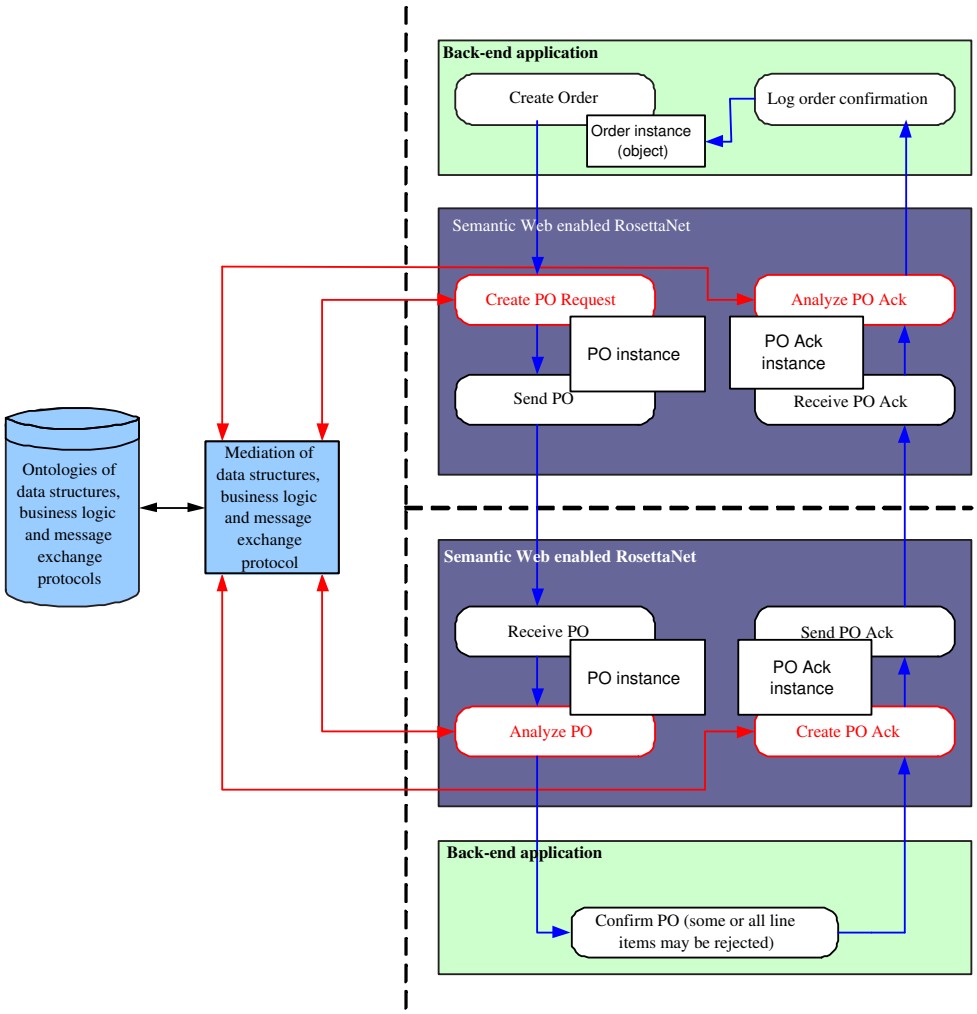

Figure 13.14: Creating a purchase order in accordance with a RosettaNet PIP

processes (like a purchase-order process), whereas BPSS is a *language* for defining processes. ebXML as such does not define processes and RosettaNet does not provide a process modeling or definition language.

13.5 The Process Specification Language

The Process Specification Language (PSL) is designed for describing or exchanging information among models of discrete processes, i.e., processes consisting of individually distinct events, tasks, or service invocations [Gruninger, 2003]. Examples of such processes are production scheduling, resource planning, workflows, and project management. PSL is not appropriate for continuous processes, whose behavior might be more appropriately described by differential equations.

PSL is intended to be a process representation that is common to all business and manufacturing applications, and powerful enough to represent the processes in any given application. This representation would facilitate interoperation by serving as an interlingua (i.e., a *lingua franca*) for process models. To achieve this, PSL has a formally defined semantics in the language of first-order logic and represented using the Knowledge Interchange Format (KIF). (KIF is now included in the proposed ISO standard called Common Logic.) The semantics consist of a set of KIF definitions that enable PSL statements about processes to be understood. For example, the KIF statement

```
(between ?task1 ?task2 ?task3)
```

is given semantics by the definition

```
(defrelation between (?a ?b ?c) ≡
  (and (before ?a ?b)(before ?b ?c)))
```

which defines *between* in terms of *before*. The semantics of *before* is provided by axioms, such as

```
(forall (?x) (not (before ?x ?x)))
```

which states that nothing can be before itself.

A first-order semantics for PSL has several advantages. First, we can specify and implement inference techniques that are sound and complete with respect to models of the theories, i.e., a theory is consistent if and only if there exists a model that satisfies the axioms of the theory. Second, a process ontology with a first-order axiomatization can be more easily integrated with other ontologies. Third, a first-order semantics allows a simple characterization of incomplete service specifications.

There are six basic things that are important to consider for an ontology of business and manufacturing processes, such as PSL:

- *Objects*, which are concepts in the world that have identity. An example is Mike's credit card.

- *ActivityOccurrences*, which are actions or events that have a temporal extent and involve specific objects. An example is checking Mike's credit rating beginning at 10:00 a.m. and ending at 11:00 a.m. on October 24.
- *TimePoints*, which are instances that separate discrete states. An example is the instant between Mike's account having a balance of $1,000 and a balance of $900.

Each of these can be typed, i.e.,

- The type of an Object is a Class.
- The type of an ActivityOccurrence is an Activity.
- The type of a TimePoint is Time.

Next, we can define relationships between some of the pairs of these six basic things. Ignoring Time and TimePoint for the moment, out of the ten possibilities for domain-independent relationships among the remaining four things, the following seven are meaningful:

1. *instanceOf* between Object and Class.
2. *subclass* between Class and Class.
3. *subclass* between Activity and Activity.
4. *occurrenceOf* between ActivityOccurrence and Activity.
5. *partOf* between Object and Object.
6. *subactivityOf* between ActivityOccurrence and ActivityOccurrence.
7. *participatesIn* between Object and ActivityOccurrence.

A relationship between Activity and Class is not meaningful, e.g., between all hammering *ActivityOccurrences* and all hammers. Similarly, relationships between Object and Activity and between Class and ActivityOccurrence are not meaningful. Note that the binary relations *subclass*, *partOf*, and *subactivityOf* are partial orders (transitive, antisymmetric, and reflexive) as described in Section 6.5.

Each of the seven basic relationships leads to a form of inference: classification and instantiation for things related by *instanceOf*, subsumption for things related by *subclass*, aggregation for things related by *occurrenceOf*, *subactivityOf*, and *partOf*, and association for things related by *participatesIn*.

When Time and TimePoint are included, there are 21 possible binary relationships. In addition to the seven above, the following are meaningful:

1. *existsAt* between Object and TimePoint.
2. *existingFor* between Object and Time.

3. *occursAt* between ActivityOccurrence and TimePoint.
4. *occurringFor* between ActivityOccurrence and Time.
5. *subset* between Time and Time.
6. *instanceOf* between TimePoint and Time.
7. *equality*, *lessThan*, and *greaterThan* between TimePoint and TimePoint.

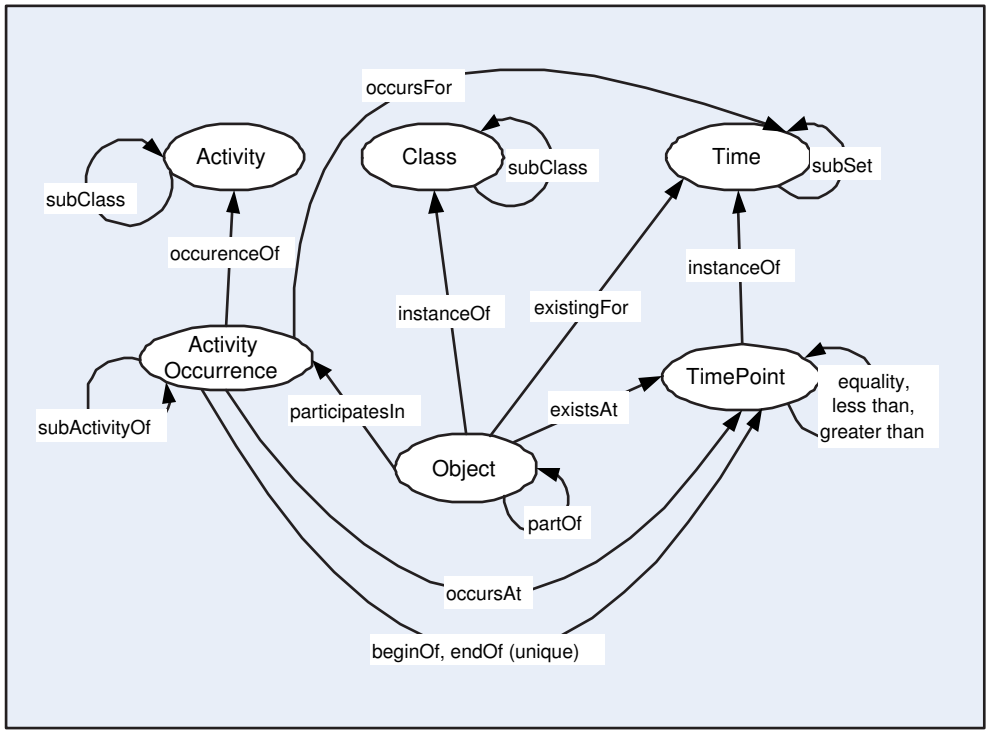

Figure 13.15: The key concepts and relationships of PSL.

Class and Time, Class and TimePoint, Activity and Time, and Activity and Timepoint do not participate in meaningful relationships. Finally, PSL provides the two functions *beginOf* and *endOf*, which return the TimePoints that define the temporal extent of an ActivityOccurrence.

In the PSL conceptual model, time is understood as discrete. This is based on the intuition that measurable time is essentially discrete, although mathematically approximated via a continuous set, such as the real numbers. A Time can then be represented by a linearly ordered set of integers. Hence, the correct relationship between TimePoint and Time is *instanceOf*, not *partOf*.

PSL consists of the core set of concepts listed above and several extensions. The PSL core includes axioms specifying the semantics of the core concepts. An example axiom is that everything is either an ActivityOccurrence, an Object, or a TimePoint, and that each of these are distinct.

There are five defined extensions: durations, activities and duration, temporal ordering relations, reasoning about state, and interval activities. Within these extensions are definitions for

Ordering. ActivityOccurrences can take place in sequences delimited by TimePoints.

Concurrency. ActivityOccurrences can take place at the same time, i.e., during the same time interval.

Resource. A Resource is an Object that is used or consumed during an ActivityOccurrence.

PSL also provides support for you to define your own extensions for specific domains or applications. More importantly, PSL can be used to define translations among different process ontologies. For example, we can specify that the *composedOf* property in OWL-S is equivalent to the *subactivity* relation in PSL by the following KIF statement:

```
(forall (?activity1 ?activity2)
  (iff (composedOf ?activity1 ?activity2)
    (subactivity ?activity2 ?activity1)))
```

For another example, a *CompositeProcess* in OWL-S (see Section 15.5.2 on page 331) is a PSL *activity* that is not *primitive*:

```
(forall (?activity)
  (iff (CompositeProcess ?activity)
    (and (activity ?activity)
      (not (primitive ?activity)))))
```

Via PSL, specifications for Web services in workflows, BPEL4WS (see Section 13.4.1 on page 267), OWL-S, and others can be given a sound and complete axiomatization, and interoperation among services specified in different formalisms can be facilitated.

13.6 Notes

In 2003, BPEL4WS was submitted to the OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee. The committee is working on refining the specification for the language.

Piccola is an experimental composition language that has been developed recently [Achermann and Nierstrasz, 2001]. It is based on the π-calculus, and represents an attempt to define a formal execution semantics for workflows composed from Web services.

Information on RosettaNet is available from http://www.rosettanet.org.

13.7 Exercises

13.1. Construct a UML activity diagram for a process with which you are familiar, such as withdrawing money from an ATM, paying for a purchase with a credit card, registering for classes, paying tuition and fees, or obtaining a student or visitor visa. Imagine that each is a two-party interaction: you and the other participant (bank, merchant, university registrar, university cashier, foreign consulate, respectively).

13.2. For the scenario described in Exercise 13.1, construct an equivalent BPEL4WS description.

13.3. Repeat Exercise 13.1 but model three parties for each case. For example, add in the ATM network, credit card company, university department, financial aid office, and host (who provides you with an invitation letter for a visa), respectively.

13.4. For the scenario described in Exercise 13.3, construct an equivalent BPEL4WS description.

13.5. A bank manager must be able to monitor a customer's account to make sure the customer has not transferred any money to terrorists. For this use-case, the detailed interactions between the manager and the system are

> Manager starts the monitoring system.
> Monitoring system asks for customer ID.
> Manager enters customer ID.
> Monitoring system retrieves customer information from account database.
> Monitoring system requests table of terrorist names from FBI database.
> FBI database sends table of terrorist names to monitoring system.
> Monitoring system compares customer information with table from FBI.
> Monitoring system notifies manager if there is a match.
> Manager halts the monitoring system.

Construct a UML activity diagram for this scenario.

13.6. For the scenario described in 13.5, construct an equivalent BPEL4WS description.

13.7. Consider the workflow for recording student registration in Figure 13.16. As shown, some of the tasks require operations on databases. Assume that each database management system implements the two-phase commit protocol for transactions. When student Bob registers, Task #2 is a check with the Graduate Coordinator to verify that he has completed the necessary prerequisites for the courses for which he is registering. Assume that Tasks #3, #4, and #5 succeed, but that Task #2 fails.

- As the system administrator, what operations would you have to perform in order to restore consistency to your system?

- How would you modify the workflow in order to prevent problems such as this from occurring in the future?

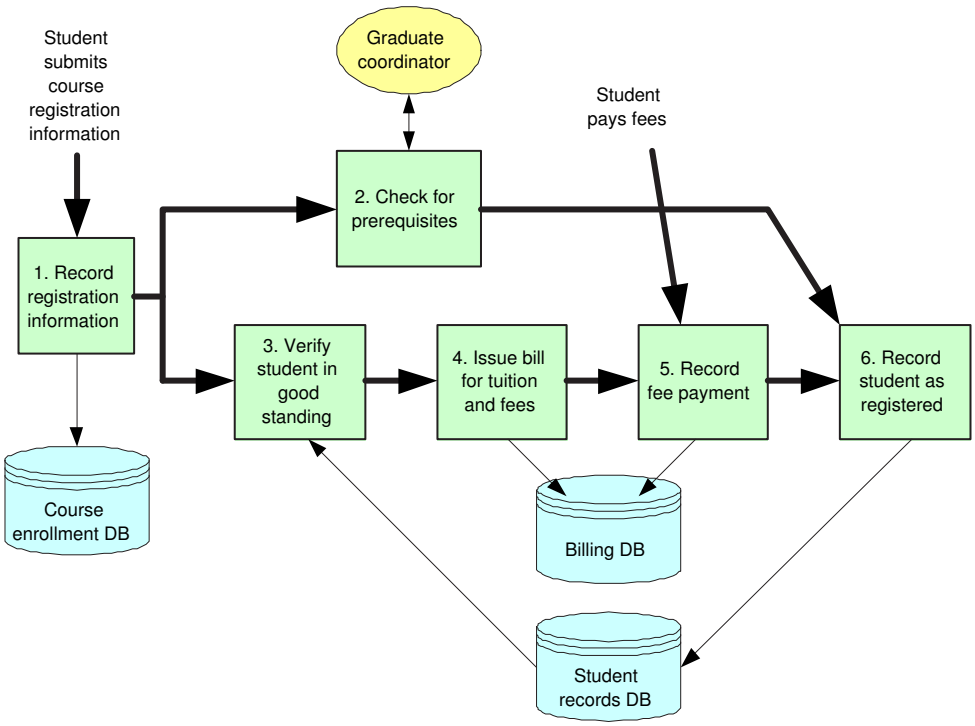

Figure 13.16: An example workflow for student registration

13.8. Develop an OWL representation of PSL's conceptual model as captured in Figure 13.15.

13.9. PSL: Two time intervals meet if the end time of the first equals the start time of the second. This can be represented by the following defrelation:

$$\begin{array}{l}(\text{defrelation meet } (?\text{task1 } ?\text{task2}) \equiv \\ \quad (\text{equal } (\text{endOf } ?\text{task1})(\text{beginOf } ?task2)))\end{array}$$

Assuming that the temporal predicate *before* has not already been defined in PSL, define *before* in terms of *meet*.

13.10. PSL has a foundation in the *situation calculus*, which has the primitives *situation*, *action*, and *fluent* (a fluent is a relation that might vary over time). It has been claimed that if a situation involves more than one process and if information about the exact

timings of the steps in the processes is unavailable, then a situation-calculus reasoner will fail. Discuss why this claim might or might not be true.