Draft UML 1.4 Profile for Automated Business Processes with a mapping to BPEL 1.0

Version 1.1
June 9th 2003

Jim Amsden, Tracy Gardner, Catherine Griffin, Sridhar Iyengar, IBM

Contact: Tracy Gardner <tgardner@uk.ibm.com>

Abstract

This document introduces a Unified Modeling Language (UML) profile for automated business processes with a mapping to the Business Process Execution Language for Web Services (BPEL4WS or BPEL). The concepts supported by the profile are based on those supported by BPEL. This approach enables automated business processes to be modeled using UML tools such as Rational Rose and Rational XDE and then translated into BPEL documents that can be executed on a BPEL runtime.

The profile described in this document is made available as a draft for comment and should be considered as a work in progress.

Before using this information, be sure to read the general information under "Notices" on page 87.
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>2</td>
</tr>
<tr>
<td>1 MODEL-DRIVEN BUSINESS INTEGRATION</td>
<td>6</td>
</tr>
<tr>
<td>2 AUTOMATED BUSINESS PROCESSES</td>
<td>7</td>
</tr>
<tr>
<td>2.1 BPEL 1.0 as a platform for executing automated business processes</td>
<td>7</td>
</tr>
<tr>
<td>3 UML PROFILE</td>
<td>8</td>
</tr>
<tr>
<td>4 DEFINING A BUSINESS PROCESS</td>
<td>8</td>
</tr>
<tr>
<td>4.1 An initial example – The purchase order process</td>
<td>8</td>
</tr>
<tr>
<td>5 DEPENDENCY MANAGEMENT</td>
<td>15</td>
</tr>
<tr>
<td>5.1 Concepts</td>
<td>15</td>
</tr>
<tr>
<td>5.2 Notation</td>
<td>16</td>
</tr>
<tr>
<td>5.3 Mapping to XSD/WSDL/BPEL</td>
<td>16</td>
</tr>
<tr>
<td>6 EXTERNAL PACKAGES</td>
<td>18</td>
</tr>
<tr>
<td>6.1 Concepts</td>
<td>18</td>
</tr>
<tr>
<td>6.2 Notation</td>
<td>19</td>
</tr>
<tr>
<td>6.3 Mapping to XSD/WSDL/BPEL</td>
<td>19</td>
</tr>
<tr>
<td>7 DATA TYPES AND INTERFACES</td>
<td>19</td>
</tr>
<tr>
<td>7.1 Concepts</td>
<td>19</td>
</tr>
<tr>
<td>7.2 Notation</td>
<td>22</td>
</tr>
<tr>
<td>7.3 Tool tips</td>
<td>22</td>
</tr>
<tr>
<td>7.4 Mapping to WSDL and XSD</td>
<td>23</td>
</tr>
<tr>
<td>8 PROPERTIES AND CORRELATIONS</td>
<td>24</td>
</tr>
<tr>
<td>8.1 Concepts</td>
<td>24</td>
</tr>
<tr>
<td>8.2 Notation</td>
<td>25</td>
</tr>
<tr>
<td>8.3 Tool tips</td>
<td>27</td>
</tr>
<tr>
<td>8.4 Mapping to WSDL or BPEL</td>
<td>28</td>
</tr>
<tr>
<td>9 PROTOCOLS</td>
<td>29</td>
</tr>
<tr>
<td>9.1 Concepts</td>
<td>29</td>
</tr>
<tr>
<td>9.2 Notation</td>
<td>30</td>
</tr>
<tr>
<td>9.3 Tool tips</td>
<td>31</td>
</tr>
</tbody>
</table>
Changes from version 1.0

- Container renamed to variable for forwards compatibility with BPEL 1.1 and to align with BPWS4J 2.0.
- XSD mapping to attributes and elements clarified.
- Rules for mapping decision nodes simplified (decision nodes always map to BPEL switches).
- Support added for inheritance between data types.
- Artifacts corresponding to external packages must be visible to the mapping tool (e.g. via the Java classpath).
- Rose-specific fault specification updated: if a parameter name or parameter type name ends with Fault then the parameter is treated as a fault.
- Loan Approval example updated in line with updates to mapping definition.
1 Model-driven business integration

Graphical modeling is a popular way of drawing designs, showing business information, communicating views and perspectives on IT systems, and defining executable things such as programs, flows, simulations, and rules. UML is a widely-used industry standard for graphical modeling, but there are many other representations and tools.

Figure 1 shows a distinction between models of different kinds.

**High-level models**

A high-level model does not represent the details of implementation artifacts. A high-level model shows only those aspects that are needed for another purpose, such as representing design, architecture, or requirements. It can show items that are not implemented or that can be defined in detail later; it can include representations of legacy systems, where only the interfaces need be specified.

**High-fidelity models**

A high-fidelity model is capable of representing the details of implementation artifacts, so that someone using a drawing tool does not need to see the textual representation of the model. (A person working with the model can see text, such as names and annotations, that is included in the drawing.)
WebSphere Studio provides tools for such high-fidelity models for runtime products such as WebSphere MQ Workflow and WebSphere MQ Integrator.

**Implementation artifacts**

*Implementation artifacts* are sometimes seen as kinds of models. Implementation artifacts in XML can often be viewed with convenient tools, but they are textual, not graphical.

In model-driven business integration, UML is used to exploit its status as a standard modeling language and to ease the transformation to and from high-level models. UML is a well-known popular standard; it is frequently used to express higher-level models, and it is capable of carrying the detail of a high-fidelity model. It is desirable for UML models to be able to express all the details of implementation artifacts, but the goal is to allow modelers the flexibility to express less than complete detail without prescribing how much detail is needed. Refinement is then a natural activity of adding further detail, while still conforming to an industry standard, rather than having to switch to a quite different modeling style. High-fidelity models contain sufficient detail be mapped directly to implementation artifacts.

One aspect of business integration is the development of automated business processes. The Business Process Execution Language for Web Services (BPEL4WS) provides an XML-based notation and runtime semantics for business processes that communication through web services.

This document specifies how to create high-fidelity UML models that describe business processes and which can be mapped to BPEL4WS.

## 2 Automated business processes

The term *automated business processes* is used to refer to loosely-coupled interacting software components that implement potentially long-running business-relevant tasks. Examples of automated business processes include purchase order, travel booking, and auction processes.

### 2.1 BPEL 1.0 as a platform for executing automated business processes

The Business Process Execution Language (BPEL) provides an XML notation for describing automated business processes and provides semantics for executing business processes.

From the BPEL 1.0 specification:

"Business processes can be described in two ways. *Executable* business processes model 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. The process descriptions for business protocols are called *abstract processes*. BPEL4WS is meant to be used to model the behavior of both executable and abstract processes.
BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the Web services interaction model and enables it to support business transactions. BPEL4WS defines an interoperable integration model that should facilitate the expansion of automated process integration in both the intracorporate and the business-to-business spaces."

The current version of this profile is concerned with the modeling of executable business processes.

BPEL specification is available at the following location:

2.2 BPEL 1.1

A BPEL 1.1 specification has recently been made available. The extensions supported in BPEL 1.1 will be supported in a future version of this profile.

3 UML profile

A UML profile is both a subset and extension of UML. It allows UML to be customized for modeling in a particular context. Customization is achieved by defining the following items:

- The subset of model elements to be used in the profile
- Stereotypes, tagged values and constraints to be used with specific model elements

The UML profile for automated business processes described in this document is a profile of UML 1.4.

Tools vary in the support they provide for profiles. Most tools support stereotypes, some support tagged values. Few tools support constraints and subsetting; that is, they do not enforce constraints and they do not prevent the modeler from using elements that are not in the profile.

The profile described in this document uses a subset of the UML modeling elements, and introduces some stereotypes to customize them. It does not use tagged values. Constraints are described informally, where appropriate. In addition, some tips are provided on how to use existing CASE tools, specifically Rational XDE and Rose, for creating models using the profile.

4 Defining a business process

4.1 An initial example – The purchase order process

This section introduces the UML profile through an initial example that defines a simple purchase order process. This example is taken from the BPEL 1.0 specification.
“On receiving the purchase order from a customer, the process initiates three tasks in parallel: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed in parallel, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer.”

The following diagram (a UML activity graph) provides an overview of the process. The icons in the lower left-hand corner of each activity indicate that the details of the activity have been hidden; this detail is introduced later in this section.

In summary, the diagram shows that the process has four ports: customer, invoiceProvider, shippingProvider and schedulingProvider, which are represented by the rectangular partitions. Activities that involve a message send or receive operation through a port appear in the corresponding partition. The arrows indicate the order in which the process performs the activities.

Figure 2: Purchase order process overview

1. The purchase order process begins by receiving a purchase order request from a customer (through the customer port).
2. The initiatePriceCalculation, initialiseShippingRequest and requestProductionScheduling activities begin executing, triggering further activities as they complete. Note that the requestShipping activity requires that both the initialiseShippingRequest and sendShippingPrice activities have taken place before it begins.
3. The returnInvoice activity returns a response back to the customer.
Figure 2 is only one of the diagrams required to model this process completely. Detail is elaborated using further diagrams supported by the profile.

The process participates in a protocol through each of its ports. A protocol links a pair of complementary roles and shows how these roles interact. Each role optionally provides a set of interfaces, which are required by the complementary role.

Data types are represented in the profile by UML classes. The class diagram in Figure 3 shows the classes that are used as operation parameter types in interfaces that are provided or required by roles in the purchase order process protocols. In this case, most of the classes needed are already defined in an existing package, PurchaseDataTypes. The stereotype <<external>> applied to the PurchaseDataTypes package indicates that package is defined outside of the current model and should not have artifacts generated in a mapping to a target platform (such as BPEL). A new package, Purchase, is introduced. This has an import dependency on PurchaseDataTypes so that the existing types can be reused\(^1\). One further class, PO, is introduced. This has purchaseOrder and customerInfo parts, where the PurchaseOrder and CustomerInfo are classes defined in the PurchaseDataTypes package.

![Diagram](image)

**Figure 3:** Message classes for use in the purchase order process.

Figure 4 introduces the interfaces that are provided or required in the purchase order protocols. These interfaces are also placed in the Purchase package.

---

\(^1\) Use of an <<import>> dependency means that names from the imported package can be used without qualification.
Figure 4: Interfaces uses by the purchase order process

Figure 5 introduces the protocols in which the purchase order will play roles and interact with partners playing the complementary roles. The roles played by the purchase order process appear on the left. Each protocol is placed within a subpackage of the Purchase package, the roles are placed within the corresponding protocol package. The Purchase package contains the classes and interfaces used by the protocols.

In the Purchase and Scheduling protocols only one role provides an interface; the other role describes a requester of the service. The Invoice service introduces an InvoiceService role that provides the ComputePrice interface and requires the user of that interface to provide an InvoiceCallback interface for an asynchronous response. The Shipping protocol also describes a two-way interaction.

Figure 5: Protocols and roles

At this point, all the definitions that the purchase order process depends on have been provided. Although these definitions have been introduced within the purchase order example they can be reused independently of the purchase order process. If
partners of the process are also modeled as automated business processes, then they reuse these definitions. Other processes that provide some or all of the same roles as the purchase order process can also reuse the definitions.

This example only needs to show how the purchase order process fulfills its roles (through its ports). It does not need to show how the complementary roles are fulfilled by the partners of the process.

Figure 6 introduces a class called PurchaseOrder, stereotyped as <<process>>, that represents the purchase order process. The process has four ports corresponding to the four partners with which it interacts (according to the protocols introduced in Figure 5).

Each port is a connection point for a partner with which the process interacts. The interfaces that are provided or required through that port are defined by the role that the process plays in the protocol. A port is represented by an association (stereotyped <<port>>) to the role played by the process.\(^2\) Recall that the protocols define the interfaces that must be provided and are required by each role.

The roles are qualified by the names of the protocol because the protocol packages are not imported directly, the Purchase package containing the protocol packages is imported by the PurchaseOrderProcess package. (The protocol packages could have been imported directly by the PurchaseOrderProcess package, but the qualification by protocol provides useful additional information to the reader of the diagram.)

The PurchaseOrder process provides the PurchaseOrder interface through its customer port, and requires the Scheduling interface through its schedulingProvider port.

The invoiceProvider port supports two-way communication: the PurchaseOrder process requires the ComputePrice interface and provides the InvoiceCallback interface, as specified by the InvoiceRequester role of the Invoice protocol. The shippingProvider port is also two-way.

The state of the process is shown as attributes of the PurchaseOrderProcess class, whose types were introduced in the Purchase package or imported from the PurchaseDataTypes package.

---

\(^2\) UML 2 provides a semantics and graphical notation for ports. Because this is not available in UML 1.x, ports are represented using the constructs that are available.
Figure 6: PurchaseOrder process ports and state

The activity graph in Figure 7 shows the behavior of the purchase order process. It is the same as Figure 2, except that more detail has been revealed. The activity graph is attached to the PurchaseOrder process class.
A partition (swimlane) is introduced corresponding to each port. Activities which represent an interaction through a port (either incoming or outgoing) are placed in the appropriate partition with control flow arrows indicating the sequencing of activities.

Each activity has a descriptive name and an entry action detailing the work performed by the activity. Typically the actions would be hidden on an activity graph diagram. Figure 7 expands a number of activities to show the detail.

The receivePO activity is stereotyped with <<receive>> to indicate that it corresponds to the receipt of a message through a port, in this case the customer port. The action expression, sendPurchaseOrder(PO), indicates the operation that is invoked, through the customer port, on the purchase order process and the attribute (the name of which appears as a parameter to the operation) into which the incoming message is placed. The sendPurchaseOrder operation signature is defined in the PurchaseOrder interface in Figure 4, which the PurchaseOrder process provides through its customer port.

The returnInvoice activity is stereotyped with <<reply>> to indicate that it returns a result to the invoker of an earlier corresponding <<receive>> activity. Both activities refer to the same operation, sendPurchaseOrder. The receive activity specifies the attribute into which the incoming message is placed. The reply activity specifies the attribute from which to take the reply message. The expression

sendPurchaseOrder() := invoice

indicates that the value of the invoice attribute will be returned as a response to the sendPurchaseOrder operation (which must have previously been invoked).

The requestShipping and requestScheduling activities are stereotyped as <<invoke>> to indicate that they invoke an operation through a port. The requestScheduling activity does not specify an attribute into which to place a response and is therefore one-way (asynchronous). The
requestProductionScheduling(PO) expression indicates that the requestProductionScheduling operation is invoked (through the schedulingProvider port) with the value of the PO attribute. The requestShipping activity is two-way (synchronous), the expression

```
shippingInfo := requestShipping(shippingRequest)
```

indicates that the requestShipping operation is invoked (via the shippingProvider port) with the value of the shippingRequest attribute as a parameter; the synchronous response to this invocation is placed in the shippingInfo attribute.

The initialiseShippingRequest activity does not involve interaction with a partner and is therefore not placed in a partition. It performs a simple copy of data from one variable to another. The := operator is used for assignment.

As in BPEL, data passing in this example is performed through variables scoped to the process. Those variables are represented as attributes in the class that represents the process. For example, the receivePO activity receives a value (that the customer port) into the PO variable which is then read by the requestScheduling activity and used as an input to the requestScheduling operation that it invokes (through the schedulingProvider port).

## 5 Dependency management

### 5.1 Concepts

Models are typically structured into packages to support namespace management and provide appropriate units for reuse. Figure 8 shows the package structure for a LoanApproval example. Dependencies between packages are represented using import dependencies.

Figure 8: Example of package dependency
5.2 Notation

Packages are used to group elements and provide a namespace. In UML 1.x, dependencies between packages are referred to as permissions. Package dependency is indicated with a UML <<import>> permission arrow as shown in Figure 9. Use of an import permission allows the importing package to refer to imported elements without namespace qualification. For example, if a class X is defined in Package1 then Package2 can refer to class X directly, there is no need to use Package1::X.

The profile also supports the use of UML <<access>> permissions that allow elements from the accessed packaged to be used within the accessing package, but require the use of qualified names such as Package1::X rather than X.

![Figure 9: Notation for package and package dependency](image)

5.3 Mapping to XSD/WSDL/BPEL

The package hierarchy of a model maps to an XML namespace hierarchy. The name of the model itself (as the top level package) also contributes to the namespace. The namespace prefix used in generated XML files is specified during model mapping. The namespace structure is also mirrored in the folder structure into which files are generated.

Packages containing elements that map to XSD/WSDL artifacts also map to files of the corresponding type. New files are not created for protocol packages, their elements are placed within the WSDL file corresponding to the parent package.

Each BPEL process maps to a BPEL definition in a BPEL file. Typically there is one BPEL process per package. Multiple BPEL processes can be placed in a single package, in which case multiple BPEL files are generated, but this is not recommended.

A package dependency maps to an XML namespace import.

Figure 10 shows the package structure of the purchase order example that was introduced in Section 4.1.

In this example the Purchase package leads to the generation of a Purchase.wsdl file. The PurchaseOrderProcess package contributes to the namespace of the PurchaseOrder.bpel file (generated due to the <<process>> class that it contains).
Note that the <<external>> stereotype indicates that no artifacts should be generated from the PurchaseDataTypes package.

Figure 10: Package structure of the Purchase Order example

Figure 11 shows an extract from the WSDL file generated from the Purchase package. The target namespace prefix http://acme.org is specified during mapping. The name of the model (PurchaseOrder) and the name of the package (Purchase) provide the remainder of the namespace.

```xml
<definitions targetNamespace="http://acme.org/PurchaseOrder/Purchase"
    xmlns:PurchaseTypes="http://manufacturing.org/xsd/purchase" ... />

<import namespace="http://manufacturing.org/xsd/purchase"
    location="http://manufacturing.org/xsd/purchase.xsd"/>

<message name="POMessage">
    <part name="customerInfo" type="PurchaseTypes:CustomerInfo"/>
    <part name="purchaseOrder" type="PurchaseTypes:PurchaseOrder"/>
</message>
```

Figure 11: WSDL resulting from import of PurchaseTypes package into Purchase package

The PurchaseOrderProcess package contains a <<process>> class so a PurchaseOrderProcess.bpel file is generated. The imports of the PurchaseOrderProcess package are mapped to namespace imports in BPEL. BPEL processes specify only the namespaces of their imports; the locations are provided at deployment time. BPWS4J requests this information through its administrative interface at deployment time.

3 Platform-specific deployment artifacts could be generated during the mapping.
6 External packages

6.1 Concepts

A model containing one or more automated business processes can depend on elements that are defined outside of that model; such elements are referred to as external. For example, the purchase order example in section 2.1 uses a set of data types that are defined outside the purchase order model. Interfaces and protocols can also be defined externally and reused in the definition of a process.

External packages contain elements that are defined in another model or elements that are defined directly as platform-specific artifacts (such as Web services or BPEL documents). External packages are not mapped to platform-specific artifacts.

This profile does not currently provide a reverse mapping to enable existing artifacts to be automatically imported into a model for reuse. Elements that are reused can be modeled explicitly and placed in a package stereotyped with <<external>>. Figure 13 shows the use of the externally-defined PurchaseDataTypes package within the purchase order model.
6.2 Notation

The stereotype <<external>> is used to indicate that the elements in the PurchaseDataTypes package are defined outside of the current model and consequently should not be mapped to platform-specific artifacts.

Figure 14: Notation for external packages

6.3 Mapping to XSD/WSDL/BPEL

External packages do not lead to the generation of artifacts. It is assumed that folders and files with names corresponding to those of the external packages will be accessible to the mapping tool. For example, the external artifacts are accessible within the classpath in a Java environment.

In Figure 15 the PurchaseDataTypes external package must have corresponding XSD artifacts in a PurchaseDataTypes.xsd file which is visible to the mapping tool.

Figure 15: External package example.

7 Data types and interfaces

7.1 Concepts

The automated business process domain builds on the concepts and infrastructure provided by the Web services domain. A core concept for automated business processes is peer-to-peer communication between Web Services with both the process and the partners being described as Web Services. The Web Services concepts required for this profile are introduced in this document. It is anticipated
that this profile will be aligned with a more general Web Services profile in the future.

A port type in Web services terminology defines a set of related operations. Port types are represented by interfaces. Operations take parameters that are typed by message types. A message type has a number of parts, each of which is of a specified data type. Data types are represented by classes stereotyped as <<data>>; message types are represented by classes stereotyped as <<messageContent>>. This profile does not require message types to be introduced where they have only a single part; a data type can be used directly.

A data type should only be introduced where the grouping of attributes makes sense beyond its usage in a particular operation. Use messageContent where the grouping makes sense only in the context of an operation (or related set of operations).

Figure 16 shows a BuyerSeller package which contains two interfaces (Buyer and Seller) and three message types (BuyerInfo, Negotiation and SellerInfo). The optional dependency arrows indicate the data types that are used by each interface (these dependency arrows are not used by the mapping). In this example, message types are introduced to indicate that the groupings of attributes are used only for the purposes of packaging up operation inputs⁴.

The attributes of a <<data>> class can have basic types or user-defined types. All of the attributes in the data types introduced in Figure 16 have basic types. The PO data type shown in Figure 16 has two parts which both have user-defined data types. Attributes of a <<data>> class are mapped to XSD attributes, containment relationships are mapped to XSD elements.

---

⁴ It is necessary to distinguish between the same grouping of parameter types occurring by accident and those groupings that represent uses of the same message type. It is important that this distinction can be made, because the mapping is to a messaging paradigm. A future version of this profile could enable users to specify operations with multiple parameters and to generate the corresponding message types.
Figure 16: Interfaces and data types

Figure 17: Parts with user defined types
7.2 Notation

Figure 18 shows the notation for interfaces, data types, and message types.

![Diagram showing notation for interfaces, data types, and message types.]

Inheritance between interfaces is not supported in this version of the automated business process profile. Single inheritance between data types is supported via a mapping to XSD extensions.

Exceptions are represented by out parameters stereotyped as «fault>>. Note that the stereotype is not visible on the diagram in Rational XDE (see “Tool tips” for information about how to specify faults using Rational Rose). This enables you to use the Web services or BPEL approach to fault handling, in which faults are named – the name of the parameter provides the name of the fault.

7.3 Tool tips

7.3.1 Rational XDE

---

5 Note that inheritance is not supported in the current version of the WSDL standard, so support at the modeling level would need to be mapped to a single layer in a platform-specific model. This is not yet supported.

6 When using the Java profile for Rational Rose or XDE, you can specify exception types that can be thrown from an operation. This profile is intended to be used with basic UML models that do not have this feature. Additionally, Java-style exceptions are specified by type only, but Web services and BPEL faults can be named.
An operation signature can be expanded on a class diagram, but the <<fault>> stereotype is not visible. It can be accessed through the properties view.

7.3.2 Rational Rose

Rational Rose does not support stereotyping of parameters with <<fault>>, and the Unisys XMI exporter does not distinguish between in and out parameters.

A naming convention is used to indicate parameters that are faults. If the name of a parameter, or the name of its type ends with 'Fault' then the parameter is treated as a fault.

For example, a parameter named processingFault with type Response is treated as a fault and so is a parameter named error with type LoanApprovalFault.

7.4 Mapping to WSDL and XSD

Each package containing interfaces is mapped to a separate WSDL document. Each interface within the package is mapped to a WSDL port type in the corresponding WSDL document.

Any class that has the stereotype <<data>> is mapped to an XSD type. Any <<messageContent>> class is mapped to a WSDL message type.

WSDL does not permit data types defined in XSD to be used directly as parameter types for operations. A message type containing multiple parts (each of which is of an XSD type) must be introduced.

This profile does not require that a new class is introduced to represent a message type containing a single part – the class representing the part type can be used directly. If a <<data>> class is used directly as the type of a process attribute, then a corresponding WSDL message type is introduced in the WSDL file for the package where the <<data>> class is used. (Repeated use of the same <<data>> class within a model does not lead to multiple message types.)

Additional classes can be introduced, if required; for example, to distinguish between different uses of the same type that should be mapped to different WSDL message types.

Figure 19 and Figure 20 show the definition of an interface and message type and the corresponding WSDL elements. Note that the [out] parameter represents a fault (the <<fault>> stereotype is not visible on the diagram).

```
<wsdl:message name="Approval">
  <wsdl:messageContent> Approval
    + accept : String
  </wsdl:messageContent>
</wsdl:message>

<wsdl:interface> LoanApproval
  + approve ( [in] creditInfo : CreditInformation , [out] loanProcessFault : LoanRequestError ) : Approval
</wsdl:interface>
```

Figure 19: Interface and message type example
8 Properties and correlations

8.1 Concepts

With loosely-coupled communicating business processes, explicit object references are not used to identify process instances. Instead, values contained within an incoming request are used to identify a process instance, or create a new instance if there is no existing instance matching those values. This is referred to as a correlation. A set of values used to identify an instance in a correlated exchange is referred to as a correlation set.

The messages within a correlated exchange must carry the values of the corresponding correlation set. However, the values can appear in different locations within the message. For example, instances of a Marketplace process providing a mediated negotiation can be identified by a correlation set containing a negotiation identifier. A message from a buyer can place this information in a negotiationItem attribute, but the seller might place it in the ‘item’ attribute of a message. For this reason, correlation sets are defined in terms of properties. Properties provide an abstraction of a piece of information that is contained in many messages but may appear in a different element of each message. Properties can be used to correlate incoming messages with process instances.

A property has a name and a type. For each message containing the property, a property alias specifies how to locate the property in the message.

Any message that is sent or received as part of a correlated exchange must contain correlation information.

Figure 21 shows the definition of a negotiatedItem property of type String. Two property aliases are defined, which specify how to extract the negotiatedItem from a BuyerInfo and SellerInfo message. The computations performed by these aliases are very simple in this case: using XPath syntax, /item and /inventoryItem (the alias expressions are not visible on the diagram but are present in the definitions of the operations).
Figure 21: Negotiated item property definition.

A process can specify one or more correlation sets by which its instances can be identified. Figure 22 shows a correlation set containing one property (the negotiatedItem property introduced in Figure 21) that is used by the Marketplace process. The Negotiation class stereotyped <<correlation>> represents the correlation set, and the attribute stereotyped <<correlation>> represents a usage of that correlation set by the process.

Figure 22: Correlation example

Figure 23 shows a receive activity within a correlated exchange that initializes the negotiatedItem correlation set.

Figure 23: Correlation

8.2 Notation

Property definitions are introduced in a class stereotyped with <<properties>>. Each property is specified as an attribute with a name and type corresponding to that of the property. Property aliases are modeled as operations with the same name as the property and a return type corresponding to the type of the property. A property alias operation has a single input parameter of the type of message from which it extracts the property. An XPath expression is associated with the operation,
specifying how the property is computed from the message. The exact way in which the XPath expression is associated with the operation depends on the tool being used (see "Tool tips").

The MyProperties class in Figure 24 is a <<properties>> class that introduces two properties: property1 and property2. Two aliases are defined for each property. For example, there is an alias for extracting property1 from messages of type MessageType1. The name of the class, MyProperties, is not used in the mapping, the class is simply used as a grouping mechanism for properties and property aliases.

```
+ property1 : String
+ property2 : Integer
+ property1 ( [in] message1 : MessageType1 ) : String
+ property1 ( [in] message2 : MessageType2 ) : String
+ property2 ( [in] message2 : MessageType2 ) : String
+ property2 ( [in] message3 : MessageType3 ) : String
```

Figure 24: Notation for properties and property aliases

Property aliases can be defined in a package other than the one in which the property is defined. This is useful if a particular solution requires additional aliases for an existing property. In this case the aliases are placed in a subclass of the class that contains the property definitions.

A correlation set is defined by a class stereotyped by <<correlation>> containing attributes with names and types matching those of properties defined within its namespace. A process specifies that it uses a correlation set through an attribute with the type of the correlation set. The stereotype <<correlation>> can also be applied (redundantly) to the attribute to distinguish it from other attributes. Figure 25 shows the definition and use of a correlation set with two properties.

```
+ «correlation» correlationSet1 : CorrelationSet1
```

Figure 25: Notation for definition and use of a correlation set

In the simplest case, all messages coming into a process and going out of a process carry the same correlation set which is initialized at the start of the process. In this case you only need to associate a correlation set with the process as shown in Figure 25.

In other cases correlation sets need to be specified for individual invoke, receive and reply activities. This is achieved by placing a correlation expression in an entry action of a correlated activity as shown in Figure 26. This is in addition to the entry action containing the receive expression for the activity. Only interaction activities (receive, replay, and invoke) can contain correlation expressions.
**8.3 Tool tips**

**8.3.1 Rational XDE**

In Rational XDE, `defaultExpression` for the return type of the operation is used to hold an XPath expression. That XPath expression describes how the property alias represented by the operation is computed.

---

7 BPEL allows multiple concurrent activities to support initialization of a particular correlation set, the first one to occur initializes the correlation set and the other activities are then followers.
8.3.2 Rational Rose

In Rational Rose, the *semantics* element of an operation used to hold an XPath expression. That XPath expression describes how the property alias represented by the operation is computed.

8.4 Mapping to WSDL or BPEL

Each attribute within a `<<properties>>` class maps to a property definition within the WSDL file corresponding to the package where it is defined. BPEL properties are defined using WSDL extension elements and are therefore placed in a WSDL file rather than a BPEL file.

Each operation within a `<<properties>>` class maps to a property alias definition within the corresponding WSDL file.

The `<<properties>>` class in Figure 21 maps to the BPEL (WSDL extensions) shown in Figure 27. Note that the queries are not visible on the diagram, but do appear in the model.

```
<bpws:property name="negotiatedItem" type="xsd:string"/>

<bpws:propertyAlias propertyName="tns:negotiatedItem"
    messageType="tns:SellerInfo"
    part="inventoryItem"
    query="/inventoryItem"/>

<bpws:propertyAlias propertyName="tns:negotiatedItem"
    messageType="tns:BuyerInfo"
    part="item"
    query="/item"/>
```

Figure 27: BPEL property and property aliases

Each `<<correlation>>` attribute on a process maps to the definition of a correlation set in the BPEL file for that process, the details of the correlation set come from the `<<correlation>>` class that types the attribute.

The correlation set defined in Figure 22 maps to the BPEL in Figure 28.

```
<correlationSets>
  <correlationSet name="negotiationIdentifier"
      properties="tns:negotiatedItem"/>
</correlationSets>
```

Figure 28: BPEL for a correlation set

Correlation expressions on activities map to correlation expressions in the corresponding activities.

The correlated activity shown in Figure 23 maps to the BPEL in Figure 29.
9 Protocols

9.1 Concepts

The relationship between a business process and partner can need messages to flow in either or both directions.

A protocol links together two complementary roles, such as Bidder and AuctionHouse and specifies the interfaces that are provided or needed by each role.

A process has a number of ports, with each port corresponding to a role in a protocol. The role determines the interfaces that are required and provided through that port.

Figure 30 show four protocols. Purchase and Scheduling are one-way protocols. A process playing one of the roles in these protocols either sends messages to a partner playing the complementary role, or receives messages from the partner. Invoice and Shipping are two-way protocols in which both parties both send and receive messages.

Figure 30: Protocols and Roles

In addition to showing the structuring of the roles, it is also valuable to show the valid sequences of message exchanges between roles. This can be shown using one or more interaction (sequence) diagrams (collaboration diagrams can also be used). The profile currently does not require such sequence diagrams, but they can provide
valuable additional documentation for a human user of the profile, and could be used for validation purposes.\footnote{An interaction diagram could be used to generate abstract processes for both roles. An executable process that is playing one of the roles would then need to conform to the corresponding abstract process. An interaction diagram can also provide a starting point for a BPEL process that would need to have activities corresponding to the valid message sends and receives for its role.}

Figure 31 shows an interaction diagram for the Invoice protocol. In this case there is only one valid message exchange sequence so a single interaction diagram is sufficient to capture all valid sequences.

The lifelines represent instances of the roles defined for the protocol.

\begin{center}
\includegraphics[width=\textwidth]{interaction_diagram.png}
\end{center}

\textbf{Figure 31: Interaction diagram for the Invoice protocol}

\section*{9.2 Notation}

A protocol is modeled as a package stereotyped as \texttt{<<protocol>>}. A protocol contains a pair of classes stereotyped as \texttt{<<role>>} as shown in Figure 32. The name of the package is the name of the protocol.

\footnote{The diagram was drawn using Rational XDE, which uses the half arrowhead for asynchronous invocation. In Rational Rose, you can also use the full open arrowhead, which also means asynchronous.}
Each role has a “realizes” dependency to the interfaces it provides and a one-way dependency to the interfaces it requires.

![Diagram of Protocol notation]

Figure 32: Protocol notation

Figure 32 shows a protocol named Protocol1. Role1 provides Interface1 and requires Interface2, and Role2 provides Interface2 and requires Interface1.

In one-directional protocols only one role provides an interface, which the other role requires.

A role can provide multiple interfaces in which case the corresponding role requires all of the interfaces. In the current version of the profile it is not possible to distinguish between operations with the same signature that occur in different interfaces provided by the same role.

### 9.3 Tool tips

#### 9.3.1 Rational XDE

Role classes do not introduce any attributes or operations so they can be shown without additional compartments. Select a role class in a diagram, then click Format-> Compartments-> None from the main menu.

To introduce an optional interaction diagram, navigate to the appropriate protocol package, then chose Add Diagram-> Sequence: Instance. To introduce lifelines that represent instances of roles, drag the role classes on to the diagram.

To adjust the length of lifelines, change the Height value in the properties menu for that lifeline.

### 9.4 Style guidelines

Class diagrams that show protocols can be displayed inside the box representing the package that corresponds to the protocol (if the tool being used supports this). When defining a group of related protocols, it can be convenient to place them in a common diagram, which should be placed in the package that contains the related protocol packages.

### 9.5 Mapping to BPEL

Each protocol package in the UML profile maps to a BPEL service link type. The roles in the protocol become roles in the service link type with interfaces provided, becoming port types within the roles.
Protocols with a one-way dependency are mapped to service link types with only one role.

The protocol shown in Figure 33 maps to the BPEL extract (actually a WSDL extension, because service link types are placed in a WSDL document) shown in Figure 34.

Figure 33: Shipping protocol

```xml
<slnk:serviceLinkType name="Shipping">
  <slnk:role name="ShippingService">
    <slnk:portType name="Purchase:Shipping"/>
  </slnk:role>
  <slnk:role name="ShippingRequester">
    <slnk:portType name="Purchase:ShippingCallback"/>
  </slnk:role>
</slnk:serviceLinkType>
```

Figure 34: BPEL (actually WSDL extension) for shipping service link type

Interaction and collaborations diagrams do not contribute to the mapping.

10 Process, state, and ports

10.1 Concepts

An automated business process describes the execution of a multi-party business-relevant task such as an auction or a purchase order. The process is stateful and interacts with a number of partners through defined protocols.

A process can be involved in many protocols with different partners, and it can be involved in the same protocol with multiple partners. When modeling an individual process the connections to partners are not modeled, rather the connections are made indirectly through ports. Each port is an interaction point of the process and specifies a role in a protocol that is supported by the process. The role defines the set of interfaces that are provided or required through that port.

UML 2 [] provides direct support for a port-connector model and corresponding notation as shown in Figure 35.
The UML 2 notation is not available to this UML 1.x profile so an alternative is provided. Figure 36 shows a PurchaseOrder process with four ports. The process has state, recorded using five attributes.

Like in BPEL, a port can be connected to zero or one partners at any time. Support for broadcast ports is not supported in the current version of this profile.10

10 Broadcast ports could be modeled in UML; for example, by using the multiplicity of the port association. This aspect of the profile would need to be mapped to another middleware component since BPEL does not support the notation of broadcast.

10.2 Notation
A process is modeled as a class stereotyped as `<process>` . The attributes of the class correspond to the state of the process. Each port is represented by an association, stereotyped `<port>`, to the role offered by that port.

Figure 37 shows a process named Process1 which maintains its state in attribute1 and attribute2. Process1 has two ports, port1 and port2, indicating that the process participates in ProtocolX (in Role1) and ProtocolY (in Role2).

The package name is the name of the protocol (for example, ProtocolX::Role1). If the protocol package is directly imported by the process package, you can omit the package name in the name of roles played by the process.

By convention, ports have public visibility, but this is not required for the mapping.

**10.3 Presentation options**

When showing ports, the roles can be expanded to show the interfaces that are required and provided through each port. These are defined as part of the corresponding protocols, but it can be convenient to view all relationships on a single diagram. Figure 38 shows four ports with expanded roles. The customer port provides the PurchaseOrder interface and the schedulingProvider port requires the Scheduling interface. Both the invoiceProvider and shippingProvider ports both provide and require an interface.
The examples above follow the convention of using attributes to represent variables and associations to represent ports. In fact, attributes or associations can be used in either case. An attribute representing a port must be stereotyped as <<port>>, and its type must be a class stereotyped as <<role>>. Note that when using attributes to represent ports it is not possible to show associations to provided and required interfaces as shown in Figure 38.

**10.4 Tool tips**

**10.4.1 Rational XDE**

To add required and provided interfaces to a diagram, when using a role that has been defined as part of a protocol, right-click on a <<role>> class in a class diagram then click Add Related Shapes.

**10.5 Mapping to BPEL**

A class stereotyped as <<process>> when using this UML profile will be mapped to a BPEL process definition with the following attributes:
<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstractProcess</td>
<td>no</td>
<td>This profile supports the modeling of executable processes only.</td>
</tr>
<tr>
<td>variableAccessSerializable</td>
<td>no</td>
<td>This concept is not yet supported by this profile.</td>
</tr>
<tr>
<td>enableInstanceCompensation</td>
<td>no</td>
<td>Instance compensation after completion of the process requires platform specific support which is not yet available.</td>
</tr>
<tr>
<td>SuppressJoinFailure</td>
<td>yes</td>
<td>Join failures occur when the status of all incoming links to an activity is negative (using the default join condition). This is not considered to be an error condition in models developed using this profile.</td>
</tr>
</tbody>
</table>

The target namespace prefix of the process is provided during mapping. The remainder of the namespace is taken from the name of the model, the package structure and the name of the process class.

The attributes of the process map to BPEL variables and the ports (associations to roles) map to BPEL partner declarations. In the UML profile there is no need to specify the role played by the partner since this is always the complementary role that is defined in the same protocol as the role played by the process. The BPEL partnerRole comes from the name of the complementary role in the protocol.

Figure 39 and Figure 40 show a LoanApproval process with its corresponding BPEL.

![LoanApproval process](image-url)

**Figure 39: LoanApproval process**
Each `<data>` class is mapped to an XSD type. Any `<data>` class that is used as the type of an attribute in a process is additionally mapped to a WSDL message type. The WSDL message type is placed in the same WSDL file as the corresponding `<data>` class. If the package is `<external>` then the WSDL message type is placed in a WSDL file containing WSDL elements scoped to the model (the namespace and location of which will be configurable during mapping).

Note that `<messageContent>` classes used as the type of an attribute have WSDL messages generated in the WSDL file corresponding to the package in which the `<messageContent>` class is defined. The corresponding message types are generated regardless of whether the types are actually used to type process attributes.

The RiskAssessmentDetails class used as the type of the riskAssessment variable is a `<data>` class. The WSDL in Figure 41 is generated into the WSDL file for the LoanAssessment package where RiskAssessmentDetails is defined.

```xml
<wsdl:message name="RiskAssessmentDetails">
  ...
</wsdl:message>
```
11 Data handling

11.1 Concepts

Within a business process, you need to be able to manipulate and update the state of the process. This is achieved through having an expression language.

Expressions are used in the following places within this profile:

- Boolean expressions are used in transition guards
- Time expressions are used as triggers in onAlarm events and in wait activities
- General expressions are used in assignment activities

Both UML and BPEL permit any scripting language to be used to write such expressions. BPEL uses XPATH as the default scripting language and provides conventions for accessing variable data and BPEL-provided utility functions from XPATH expressions. This version of the profile supports XPATH as the default language for expressions and provides conventions for accessing attributes. This provides a straightforward mapping to BPEL.

In future, other expression languages can be supported. In particular, if a standard surface language for the UML activity and action semantics is agreed upon then we would expect to support that.

11.2 Notation and mapping to XPATH or BPEL

11.2.1 Accessing variables and properties

Within expressions you need to be able to refer to elements of the state of a process. You refer to state elements through XPATH expressions rooted to the process.

Each expression begins with the name of an attribute of the process. Navigation to a part of the message, or a property of the message is achieved by appending a forward slash followed by the part or property name. Further navigation into the internal structure of a part can be specified by appending additional XPATH elements to the expression.

For example, the expression

\[
\text{shippingRequest/customerInfo}
\]

in a purchase order process, refers to the customerInfo attribute of the shippingRequest attribute of the purchase order process.

11.2.1.1 Mapping to XPATH/BPEL
The BPEL specification defines the following functions for use within XPATH expressions:

**bpws:getVariableData ('variableName', 'partName', 'locationPath'?)**

This function extracts values from variables. The first argument names the source variable for the data, the second names the part to select from that variable, and the third optional argument, when present, provides an absolute location path (with '/' meaning the root of the document fragment representing the entire part) to identify the root of a subtree within the document fragment representing the part. The return value of this function is a node set containing the single node representing either an entire part (if the third argument is absent) or the result of the selection based on the locationPath. If the given locationPath selects a node set of a size other than one, then the standard fault **bpws:selectionFailure MUST be thrown by a compliant implementation.**

**bpws:getVariableProperty ('variableName', 'propertyName')**

This function extracts global property values from variables. The first argument names the source variable for the data and the second is the qualified name (QName) of the global property to select from that variable ... If the given property does not appear in any of the parts of the variable's message type, then the standard fault **bpws:selectionFailure MUST be thrown by a compliant implementation.** The return value of this function is a node set containing the single node representing the property. If the given property definition selects a node set of a size other than one, then the standard fault **bpws:selectionFailure MUST be thrown by a compliant implementation.**

Data access expressions are mapped to uses of these functions. The first element of a data access expression is always an attribute, which maps to the first argument in either a getVariableData or a getVariableProperty call.

The second element in a data access expression is used to determine whether the expression is to be mapped to a use of getVariableData or getVariableProperty. This depends on whether the element corresponds to a part or a property of the attribute.

If the second element is a part then there can be more elements that map to a location path. (The location path must be an absolute path from the part, so it also includes the part name as the first element.)

If a message has a part and a property alias for a property with the same name as that part, then the property alias must refer to that part. Such as expression is mapped to a getVariableProperty call unless, it has elements following the second element which map to a location path, in which case a getVariableData call be be generated.

Examples:

<table>
<thead>
<tr>
<th>request/amount where: request is an attribute of type Request which has an</th>
<th>bpws:getVariableData('request','amount')</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
attribute named amount

stockResult/level
where:
stockResult is an attribute of type StockResult for which a property alias is defined for the level property

bpws:getVariableProperty(stockResult, level)

customer/address/name/last
where:
customer is an attribute and the remainder of the expression navigates its internal structure
The expression extracts the value of the 'last' attribute.

bpws:getVariableData('customer', 'address', '/address/name/last')

### 11.2.2 General expressions

As in the BPEL specification: "These are expressions that conform to the XPath 1.0 Expr production where the evaluation results in any XPath value type (string, number, or Boolean).

Expressions with operators are restricted as follows:

- All numeric values including arbitrary constants are permitted with the equality or relational operators (<, <=, =, !=, >=, >).
- Values of integral (short, int, long, unsignedShort, and so on) type including constants are permitted in numeric expressions, provided that only integer arithmetic is performed. In practice, this means that division is disallowed. It is difficult to enforce this restriction in XPath 1.0 because XPath 1.0 lacks integral support for types. The restriction should be taken as a statement of intent that will be enforced in the future when expression languages with more refined type systems become available.
- Only equality operators (=, ! =) are permitted when used with values of string type including constants."

### 11.2.2.1 MAPPING TO XPATH OR BPEL

General expressions are mapped to XPATH expressions, with the contained data access expressions being mapped as described above.

Examples:

| responses/expected – responses/received | bpws:getVariableData('responses', 'expected') – bpws:getVariableData('responses', |
11.2.3 Boolean Expressions

As in the BPEL specification: “These are expressions that conform to the XPath 1.0 `Expr` production where the evaluation results in Boolean values.” []

Examples:

```
buyer/offerPrice >= seller/reservePrice
```

11.2.3.1 MAPPING TO BPEL

Boolean expressions are mapped as for general expressions.

Examples:

```
<table>
<thead>
<tr>
<th>request/amount &gt;= 10000</th>
<th>bpws:getVariableData('request', 'amount') &gt;= 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>riskAssessment/risk = 'low'</td>
<td>bpws:getVariableData('riskAssessment', 'risk') = 'low'</td>
</tr>
</tbody>
</table>
```

11.2.4 Time expressions

11.2.4.1 DEADLINE-VALUED EXPRESSIONS

As in the BPEL specification: “These are expressions that conform to the XPath 1.0 `Expr` production where the evaluation results in values that are of the XML Schema types `dateTime` or `date`.” []

11.2.4.2 DURATION-VALUED EXPRESSIONS

As in the BPEL specification: “These are expressions that conform to the XPath 1.0 `Expr` production where the evaluation results in values that are of the XML Schema type `duration`.” []

11.2.4.3 MAPPING TO BPEL

Time expressions are mapped directly to time expressions in BPEL. For an example, see Section 12.5.3 where the mapping of wait activities to BPEL is covered.

11.2.5 Assignment

Assignment statements are represented by using the : = operation between general expressions. This indicates that the value of the element on the right-hand side of the expression is copied into the element specified on the left-hand side.

For example, the assignment statement:
shippingRequest/customerInfo := PO/customerInfo

copies the value of the customerInfo attribute of the PO attribute into the customerInfo attribute of the shippingRequest attribute.

### 11.2.5.1 MAPPING TO BPEL

An assign activity maps to a BPEL assign activity with each entry action mapping to a copy element. The right-hand side of each assign statement provides the details of a from element and the left-hand side provides then details of the to element.

Figure 42 and Figure 43 show an `<assign>` activity and its corresponding BPEL.

```xml
<assign name="assign">
  <copy>
    <from expression="'yes'"/>
    <to variable="approvalInfo" part="accept"/>
  </copy>
</assign>
```

Figure 42: Example assign activity

Figure 43: BPEL assign activity

## 12 Basic activities

### 12.1 Invoking operations through ports

#### 12.1.1 Concepts

A process often invokes operations in its partners as part of its execution. This can be done synchronously or asynchronously:

- Synchronously. A fault can be thrown and a return value can be provided
- Asynchronously. The activity completes as soon as the request has been issued and there is no corresponding return or fault.

An asynchronous reply is modeled as a further one-way invocation from receiver to caller.
Figure 44 Illustration of invoke and receive/reply activities

Figure 44 (which is a repeat of Figure 7) contains a number of invoke activities. A synchronous invocation is exemplified by the requestShipping activity which invokes the requestShipping operation through the shippingProvider port. An asynchronous invocation is exemplified by the requestScheduling activity, which invokes the requestProductionScheduling operation through the schedulingProvider port.

12.1.2 Notation

An invocation of an operation on a partner is represented as an activity with stereotype <<invoke>> with an entry action indicating the operation to be invoked and the attribute containing the input message. For two-way messages, assignment notation is used to indicate the attribute that is updated with the reply message.

Communication with partners is through ports. So the partner on which the operation is invoked is specified by placing the activity in the partition corresponding to the port through which communication with the partner takes place. Note that, as in BPEL, a port can be connected to only one partner so an invoke activity causes an operation to be invoked on one partner.

Figure 45 shows two invoke activities. Activity1 is a one-way activity (it does not specify an attribute for a result) which invokes operation1 through port1 with attr1 as the input message. Activity1 completes as soon as the operation invocation request has been sent and does not wait for a reply. Activity2 is a two-way invoke which invokes operation2 through port2 with attr2 as the input message and waits for a synchronous reply which it places in attr3.
When using a hierarchical structure to denote control flow, it is not possible in Rational XDE or Rational Rose to place nested activities in partitions. In this case, use the alternate notation for partitions that is provided in UML 2: the name of the partition is placed in round brackets above, or in front of, the activity name as shown in Figure 46.

**12.1.3 Mapping to BPEL**

Invoke activities are mapped to BPEL invoke activities. The name of the UML activity is used for the name of the BPEL activity. The port name becomes the name of the partner. The operation name and optional input and output variables come from the invoke expression in the entry action of the activity. The port type corresponds to the name of the required interface on the port that contains the operation.

Figure 47 shows the BPEL generated from the invokeAssessor and invokeApprover activities in Figure 48. Because the BPEL activities are contained within a flow, the activities also contain information on the links in which they are the source or target.
Figure 47: Loan Approval example with invoke activities

```
<invoke inputVariable="request" name="invokeAssessor"
    operation="check" outputVariable="riskAssessment"
    partner="assessor" portType="LoanAssessor:RiskAssessment"
    suppressJoinFailure="no">
    <target linkName="receive1_to_invokeAssessor"/>
    <source linkName="invokeAssessor_to_invokeApprover"
        transitionCondition="bpws:getVariableData('riskAssessment','risk') != 'low'"/>
    <source linkName="invokeAssessor_to_assign"
        transitionCondition="bpws:getVariableData('riskAssessment','risk') = 'low'"/>
</invoke>

...

<invoke inputVariable="request" name="invokeApprover"
    operation="approve" outputVariable="approvalInfo"
    partner="approver" portType="LoanApprover:LoanApproval"
    suppressJoinFailure="no">
    <target linkName="receive1_to_invokeApprover"/>
    <target linkName="invokeAssessor_to_invokeApprover"/>
    <source linkName="invokeApprover_to_reply"/>
```
12.2 Providing operations through ports

12.2.1 Concepts

A process provides operations to its partners through receive activities. Synchronous replies are modeled using corresponding reply activities and the caller is assumed to block. Asynchronous requests do not have corresponding reply activities.

Examples of receive and reply activities are provided by Figure 44 on page 43. An asynchronous request is exemplified by receiveInvoice from InvoiceProvider, and a synchronous request/reply is illustrated by receivePO and returnInvoice from or to Customer.

12.2.2 Notation

Receive activities have the stereotype <<receive>> and an entry action, which indicates the operation call that is expected and the attribute into which the input message will be placed. The partner that the operation call is expected from is specified by placing the activity in the partition that corresponds to the port through which the partner communicates.

Figure 49 shows a receive/reply pair of activities (Activity1 and Activity4) modeling the receipt of a synchronous request and the corresponding reply, and the receipt of a one-way operation call (Activity3) which has no corresponding reply.

When it receives control, Activity1 waits for the receipt of an invocation of operation1 from port1. When Activity1 receives that invocation, the input message is placed in attr1 and Activity1 completes.

When control reaches Activity4, attr2 is returned as the result of operation1. As in BPEL, there can be only one request outstanding for a particular partner and operation.
Figure 49: Receive notation

In addition to returning a response to a successful invocation, it is also possible to return an exception response. The BPEL-style of returning an exception to a caller through a reply activity is used. As in a non-exception reply statement, the left-hand side identifies the operation that has resulted in the exception. The right-hand side of the reply statement identifies the name of the exception that has been generated and the attribute that holds any exception data. The right-hand side takes the form: `exceptionName(attributeName)`. The notation for returning an exception reply is shown in Figure 50.

Figure 50: Reply with exception notation

### 12.2.3 Mapping to BPEL
Receive and reply activities are mapped to BPEL receive and reply activities with the expressions of the entry actions providing the detail in a similar manner to the mapping for invoke activities.

The BPEL generated for the receive/reply pair in Figure 47 is shown in Figure 51.

```
<receive variable="request" createInstance="yes"
    name="receive1" operation="approve" partner="customer"
    portType="LoanApprover:LoanApproval" suppressJoinFailure="no">
    <source linkName="receive1_to_invokeApprover"
        transitionCondition="bpws:getVariableData('request','amount') &gt;= 10000"/>
    <source linkName="receive1_to_invokeAssessor"
        transitionCondition="bpws:getVariableData('request','amount') &lt; 10000"/>
</receive>

<reply variable="approvalInfo" name="reply" operation="approve"
    partner="customer" portType="LoanApprover:LoanApproval"
    suppressJoinFailure="no">
    <target linkName="assign_to_reply"/>
    <target linkName="invokeApprover_to_reply"/>
</reply>
```

Figure 52 BPEL generated for the receive/reply pair in Figure 46

For exception replies, the exception name and exception attribute are used to populate the faultName and faultVariable attributes of the corresponding BPEL reply activity.

### 12.3 Signaling exceptions

#### 12.3.1 Concepts

An exception can be implicitly thrown when an invoked operation returns an exception. It is also possible to explicitly throw an exception using a **throw activity**.

This document uses the Web services and BPEL convention of supporting named exceptions (faults in web services terminology) which may optionally specify exception data.

The activity shown in Figure 53 depicts throwing an OutOfStock exception on discovery that an item is out of stock.
12.3.2 Notation

Throw activities, identified by the <<throw>> stereotype, have an entry action indicating the exception to be thrown. Exceptions are always thrown by name. An attribute which contains associated exception data may optionally be specified as a parameter.

Activity4 in Figure 54 is a throw activity which throws an exception named Exception1, no exception data is specified (there is no need to introduce empty parameter brackets, as in Exception1(), though this is also valid). Activity7 in Figure 54 is a throw activity which throws an exception names Exception2 and has data specified as attr1, which must reference an attribute of the containing process.

12.3.3 Mapping to BPEL

A <<throw>> activity is mapped to a BPEL throw activity. The entry action expression provides the faultName and the faultVariable attribute (if any).

Figure 55 shows the BPEL corresponding to the <<throw>> activity in Figure 53. The OutOfStock fault name is defined in an imported namespace which is referred to with the prefix FLT.

```
<throw name="OutOfStockError"
    faultName="FLT:OutOfStock"
    faultVariable="outOfStockDetails"/>
```

Figure 55: BPEL for throw activity

12.4 Terminating the process instance

12.4.1 Concepts
A process terminates normally when all final activities (those with no outgoing control links) have completed.

A process can be explicitly terminated by introducing a control link to a final state. When a final state is reached, all activities are terminated.

For example, the receipt of a ‘cancel’ message could terminate an insurance request process during the cool off period through a control link to a Cancelled terminate activity, as shown in Figure 56.

**Figure 56: Explicit terminate**

### 12.4.2 Notation

A terminate activity is shown by a ‘bullseye’ (☉). Figure 57 shows a final state that terminates the process on completion of Activity2.
**12.4.3 Mapping to BPEL**

A final state in UML is mapped to a BPEL terminate activity. The name of the final state (if any) maps to the name of the terminate activity. Figure 58 shows the BPEL for the Cancelled final state in Figure 56.

```xml
<terminate name="Cancelled"/>
```

Figure 58: BPEL terminate example

**12.5 Waiting**

**12.5.1 Concepts**

A business process may need to wait until a particular deadline is reached before proceeding. This is modeled using a wait activity. Figure 59 shows a wait activity that waits until a specific time and on Christmas Eve 2002, when the waitUntilChristmas activity completes, the seasonalGreeting activity runs.
12.5.2 Notation

This is achieved using an activity stereotyped as «wait» that has a single entry action containing a time expression (as defined in section 11.2.4 on page 41) prefixed by either for or until to indicate a duration or deadline expression respectively.

12.5.3 Mapping to BPEL

A «wait» activity is mapped to a BPEL wait activity. The entry action expression provides the time expression and whether it is placed into a for or an until attribute within the wait activity.

The BPEL for the waitUntilChristmas «wait» activity in Figure 59 is shown in Figure 61.

<wait until="2002-12-24T18:00+01:00"/>
13 Activity coordination

13.1 Basic control flow

13.1.1 Concepts

Control dependencies between activities are modeled using control links. Processes support concurrent flows of control. An activity is executed when all of its incoming control links are enabled. When an activity completes, all of its outgoing control links are enabled.

Figure 62 repeats the purchase order example of section 4.1, which shows the use of control links to sequence activities.

Optionally, control links may have guards. A control link with a guard is only enabled if the guard (a Boolean expression) evaluates to true when its source activity completes.

Hierarchical structure can also be used to model control flow. When a parent activity is started, any child activities without incoming control links are started. When all child activities that do not have outgoing control links have completed then the parent activity completes.

13.1.2 Notation

A control link is shown by an arrow connecting two activity nodes.
An activity node may have zero, one, or more incoming and outgoing control links. An activity node may be nested inside another activity node. Control links may cross activity boundaries; that is, a nested activity can have an incoming or outgoing control link to a node outside the node in which it is nested. Loops in control structure are not permitted. The structured `<loop>` construct introduced in Section 13.3 should be used.

Figure 63 demonstrates the notation for control flow with nesting.

![Figure 63 Notation for control links and nesting](image)

### 13.1.3 Mapping to BPEL

Mapping complex control structure to the structured constructs of BPEL is not straightforward if you want to obtain BPEL that is close to ‘hand-written’ BPEL as possible.

Where there is a set of activities (either at the top level or within a composite activity) that are executed one after another, one at a time, then a BPEL sequence activity is generated to contain those activities.

For a more complex control structure a BPEL flow can be used. All activities could be placed into a BPEL flow with the appropriate links. This would be semantically correct, but a BPEL programmer would be more likely to structure the flow using sequences.

Therefore the following heuristic is used to structure the flow. If all activities with a common direct parent that involve interaction with the same partner can be placed in a sequence, the activities are placed in the sequence with any non-interaction activities (assign, wait, and so on) that have control links only to other elements within the sequence. (Note that it is not possible to have multiple concurrent activities interacting with the same partner.) Any elements that are not placed within a sequence become top-level activities within the flow.

The example in Figure 62 maps to the BPEL shown in Figure 64 (detail omitted).
The UML modeler also has the option of using hierarchical nesting when appropriate, however flows and sequences are not distinguished in UML. Each level in the hierarchy is recursively given the same treatment as described above.

### 13.2 Decision and merge

#### 13.2.1 Concepts

Decision and merge nodes can be introduced to fine-tune control of concurrent behavior.

Decisions enable the behavior of a process to be dependent on its current state. When control reaches a decision node, the guards on each of its outgoing control links are evaluated, when a guard evaluates to true that control link is enabled and there is no further evaluation of guards.

At most one control link can have the guard ‘otherwise’ indicating that that control link is enabled if the guard for all the other control links evaluate to false.

A merge node allows any one of a number of incoming control links to cause an outgoing control link to be enabled. A merge node can be used to merge the alternative flows of control created by a decision node to indicate that behavior after the merge is common in all cases.
Figure 65 shows the use of a decision node and a merge node to provide different request handling behavior depending on whether the customer making the request has gold status or not.

![Decision and Merge Node Diagram]

**Figure 65: Example of decision and merge nodes**

### 13.2.2 Notation

The notation is illustrated by Figure 66. Decision and merge nodes are both shown as diamonds (there is one of each in the diagram).

Decision nodes are choices, where each outgoing control link is guarded by an expression. The guard is shown within square brackets in a label on the control link. Optionally one (and only one) outgoing control can be guarded by the expression ‘otherwise’.

Merge nodes have a series of incoming control links and one outgoing control link. Note that if you draw a ‘diamond’ node with multiple incoming and outgoing links then this means a merge followed by a condition. That is, the node will be evaluated when any of the incoming links is activated and only one of the outgoing links is activated depending on the guards on outgoing links.
In UML a decision node can have only one incoming link and a merge can have only one outgoing link. If a diamond is drawn with multiple incoming and outgoing links, then it is interpreted as a merge node followed by a decision node.

This means that implicit fork and join for control links is not available for decision and merge nodes. To perform a join before a decision node, or a fork after a merge node, explicit join or fork nodes must be introduced.

The Marketplace process shown in Figure 67 demonstrates the use of a join (solid bar) before a decision and a fork (also represented by a solid bar) after a merge.

Figure 67: Use of explicit fork and join nodes
13.2.3 Style guidelines

Decision node behavior can be achieved through placing guards on multiple outgoing transitions from an activity.

However, you are recommended to introduce a decision node for clarity whenever a mutually-exclusive choice is made (as it is in this case). When using guarded transitions, there can be multiple transitions that evaluate to true and they would all be activated.

13.2.4 Mapping to BPEL

A decision node and corresponding merge (the end of the process is deemed to be an implicit merge) is mapped to a BPEL switch statement.

Figure 65 shows a well-nested decision and merge node and maps to the BPEL switch activity outlined in Figure 68.

```
<switch>
  <case condition="bpws:getVariableData(‘customerInfo’, ‘status’) = ‘gold’">
    <!--- fast track request --->
  </case>
  <otherwise>
    <!--- handle request --->
  </otherwise>
</switch>
```

Figure 68: BPEL switch activity

13.3 Looping

13.3.1 Concepts

A business process can repeat a particular activity while a condition evaluates to true. Rather than permitting unstructured loops in activity graphs this profile supports an explicit structured looping node. This conforms to BPEL and other workflow languages.

The loop activity provides support for while loops as in BPEL; if the guard condition is false on entry to the loop, then the loop body is never executed (contrast with a do-until loop, which always performs the loop body at least once).

Figure 69 shows a loop activity that repeatedly executes the DoSomething activity while the numberOfItems attribute of the orderDetails attribute is greater than 100.
13.3.2 Notation

A looping node is shown as an activity with the stereotype <<loop>>, which contains a decision node and an activity to be repeated, with a control link from the decision node to the activity. The guard on the control link provides the Boolean expression which determines whether the activity is executed each time round the loop.

13.3.3 Mapping to BPEL

A <<loop>> activity maps to a BPEL while activity. Figure 71 shows the BPEL corresponding to the loop activity in Figure 69.}

```xml
<while condition="bpws:getVariableData(orderDetails,numberOfItems) > 100">
  <scope name="DoSomething">
    ...
  </scope>
</while>
```

13.4 Pick

13.4.1 Concepts
It is often useful to group a set of receive activities and optionally a wait activity so that when one of them occurs, the remainder are disabled. For example, after sending an approval request to an approver, an order process could wait for either an accept message, a reject message, or a deadline to be reached – whichever happens first.

BPEL refers to this notion as pick activity, and this profile also uses the term pick.

Figure 72 shows the use of a pick within an order process. This pick might be placed within a loop to enable the order process to respond to a series of inputLineItem requests followed by an orderComplete request, with inactivity triggering timeout handling. When control reaches the pick node (a decision node stereotyped with <<pick>>) the ReceiveLineItem, ReceiveCompletion and OrderCompletionTimeout is enabled; when one of the activities takes place the others are disabled. ReceiveLineItem and ReceiveCompletion are both receive activities that are triggered by receipt of a request received through the buyer port. OrderCompletionTimeout is a wait activity which waits for three days and ten hours.

**Figure 72: Pick example**

### 13.4.2 Notation

Figure 73 shows the notation for a pick activity, which is shown as a decision node with the stereotype <<pick>>. The outgoing control links from the decision node must all connect to activity nodes stereotyped by <<receive>> or <<wait>>.
Figure 73: Pick notation

The profile also supports the alternative notation shown in Figure 74, which is more in line with UML 2, where interruptible regions can be used to represent pick behavior. However, UML 2 uses nesting and therefore does not permit graphical partitions to be used to represent partners (see Section 12.1.2 for the alternate partition notation that is used in this case).

Figure 74: Pick alternative notation

14 Error handling

14.1 Exception handling

14.1.1 Concepts

When an invoked operation throws an exception, or a throw activity explicitly throws an exception, normal execution within the containing scope is terminated.

An exception can be caught within the containing scope so that error recovery behavior can be performed.

Exception handling in this profile follows the BPEL fault handling semantics, whereby exceptions can be thrown and caught by name or type.

Figure 75 shows a catch activity at the top level of a process which catches any loanProcessFault that occurs in the process and places it in the error attribute of the process. The process responds to a synchronous invocation of its approve operation, so the recovery behavior is to return the exception that occurred. The notation for
returning an exceptional response is the same as that for returning a normal response, except that the right-hand-side of the expression contains a fault name and the name of the attribute containing the exception data\textsuperscript{11}.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{catch_activity.png}
\caption{Catch activity}
\end{figure}

\subsection{14.1.2 Notation}

One or more catch activities can be placed within a parent activity to catch exceptions that occur in any of its children. A catch activity is scoped by its parent and catches matching exceptions that occur in any child of its parent.

Each catch activity has an entry action which specifies the kind of exception that it can handle and the attribute to which the exception should be copied when caught. The exception name ‘Any’ is used to indicate that all exceptions should be caught.

Example, to catch activities within a process:

- \texttt{attr1 := faultName1} – catches a \texttt{faultName1} exception and copies the exception into \texttt{attr1}
- \texttt{attr1} – where \texttt{attr1} is the name of an attribute of the process, catches an exception of a type matching that of \texttt{attr1} and copies the exception into \texttt{attr1}
- \texttt{faultName1} – where \texttt{faultName1} is not the name of an attribute of the process, catches a \texttt{faultName1} exception (any associated exception data is lost).
- \texttt{Any} – catches any exception not caught by a more specific catch activity.

Outgoing control links from a catch activity indicate the behavior to be performed on catching the exception. The recovery behavior can be nested within the catch activity if using hierarchical nesting to indicate control flow.

Figure 76 shows two catch activities that handle exceptions that occur within the scope of Activity1. Activity6 catches exceptions of type Exception1 and performs Activity7 followed by Activity8 as recovery behavior. Activity9 catches any other exception that can occur within the scope of Activity1 and performs Activity10 as recovery behavior.

\footnotesize{\textsuperscript{11} The approach of using a reply activity to return a fault is consistent with BPEL.}
When handling exceptions that occur as a result of an invoke it is necessary to specify a variable into which the exception object should be placed if that exception object needs to be accessed. The type of this attribute determines the exceptions that may be caught.

### 14.1.3 Style guidelines

Scopes without any incoming or outgoing control links can be introduced purely for the scoping of exception and compensation handlers (compensation is introduced in Section 14.2). Such activities can be omitted from activity diagrams that do not show exception handling. The handling of exceptions within a scope can then be shown on a separate diagram.

### 14.1.4 Mapping to BPEL

Catch activities at the top level map to corresponding catch activities within the BPEL faultHandlers element at the process scope.

Nested catch activities cause a BPEL scope to be introduced corresponding to the parent activity. Catch activities then map to catch activities within the faultHandlers element of that scope.

The catch activity in Figure 75 is at the top-level and therefore maps to the process-level fault handler shown in Figure 77.

```xml
<process>
  <faultHandlers>
    <catch faultVariable="error"
      faultName="LoanAssessor:loanProcessFault">
```

Figure 76: Catch notation
14.2 Compensation

14.2.1 Concepts

In some cases it can be necessary to reverse work that has already been performed (for example, due to a failure of a subsequent piece of work).

Compensation handler activities can be defined to reverse the work performed by a scope, if necessary.

Compensation handler activities are not executed when control reaches the parent activity. If the parent activity completes successfully then the compensation handler is installed. Compensation can be triggered by a compensate activity.

We follow the BPEL semantics for compensation and when it can be triggered. In particular, a compensate activity is only permitted in the following places:

- In a catch activity of the scope that immediately encloses the scope for which compensation is to be performed.
- In the compensation handler of the scope that immediately encloses the scope for which compensation is to be performed.

Compensation examples are currently under development and will be added when available.

14.2.2 Notation

A compensation activity has the stereotype <<compensationHandler>>. At most one compensation activity can be placed within an activity to compensate the work performed by the children of that activity.

A compensate activity has the stereotype <<compensate>>.

Figure 78 shows an activity X which contains an activity doX that is executed when control reaches X. On successful completion of X, the compensationHandler Activity3 is installed.
If an exception of type Exception1 is thrown within the scope of Activity2 (which includes activity Y) then catch activity Activity2 executes, causing Activity1 to compensate the scope X.

Compensating scope X causes the causes Activity3 to execute (if installed) which causes the reserveX activity to take place.

Activities to be performed on compensation can be nested within the compensationHandler activity (as shown) or a control link from the compensationHandler activity to the compensation activity may be used.

Figure 78: Compensation notation

**14.2.3 Mapping to BPEL**

To be added when available.
Appendix A - Marketplace Example

This example is based on a sample provided with the BPWS4J runtime for BPEL [].

The marketplace process mediates a negotiation between a buyer and a seller. When the marketplace process has an asking price from the seller and an offer from the buyer (which can happen in either order) it checks to see if the offer meets or exceeds the asking price. A response indicating the success or failure of the negotiation is returned to both the buyer and the seller.

In this case the marketplace process offers Buyer and Seller interfaces to its partners and does not use any interfaces provided by its partners. The Buyer interface introduces an operation for submitting an offer, and the Seller interface introduces an operation for submitting an asking price. A class, stereotyped <<messageContent>>, is introduced for each parameter type used by the operations in these interfaces. Note that messageContent rather than data is used here to indicate that the grouping of attributes is for the purposes of a message exchange rather than to introduce a new reusable data type.

The dependency arrows indicate which classes are used by which interfaces. Dependency arrows are optional and are introduced here only for clarity.

The interface and class definitions are grouped together in a BuyerSeller package.

Figure 79 – Interfaces
The Marketplace example uses the identifier of the negotiated item for correlation. Both the SellerInfo and BuyerInfo classes need to carry this correlation information. In the case of the BuyerInfo class the item attribute is used, in the case of the SellerInfo class the inventoryItem attribute is used.

Therefore, a negotiatedItem property, with aliases for both BuyerInfo and SellerInfo, is introduced. The property and aliases are placed in a Properties class within the BuyerSeller package as shown in Figure 80. The query expressions for extracting the properties from the messages do not appear on the class diagram since they are placed in the defaultExpression of the return parameter. For SellerInfo the expression is /inventoryItem, and for BuyerInfo it is /item.

```
+ negotiatedItem : String
+ negotiatedItem ([in] message : SellerInfo) : String
+ negotiatedItem ([in] message : BuyerInfo) : String
```

Figure 80: BuyerSeller properties

The Marketplace process participates in the two protocols shown in Figure 81.

```
+ buyerInfo : BuyerInfo
+ sellerInfo : SellerInfo
+ negotiationOutcome : Negotiation
+ negotiationIdentifier : NegotiationIdentifier
```

Figure 82: Marketplace process, ports, attributes and correlation

Finally, the behavior of the Marketplace business process is defined. This is done by attaching an activity graph to the Marketplace class. A partition corresponding to

```
+ negotiatedItem : String
```

Figure 80: BuyerSeller properties

The Marketplace process participates in the two protocols shown in Figure 81.

```
+ buyerInfo : BuyerInfo
+ sellerInfo : SellerInfo
+ negotiationOutcome : Negotiation
+ negotiationIdentifier : NegotiationIdentifier
```

Figure 82: Marketplace process, ports, attributes and correlation

Finally, the behavior of the Marketplace business process is defined. This is done by attaching an activity graph to the Marketplace class. A partition corresponding to
each port is defined. Many activities within an automated business process represent an interaction through a port. The port is indicated by placing the activity in the appropriate partition.

In the Marketplace example, either the buyer or the seller can initiate a process by sending a submit request referring to an item for which there is not an existing process instance. If the buyer sends the initiating request for a particular item then a subsequent message from the seller containing matching correlation data is received by the same process instance.

![Figure 83: Behavior of the Marketplace process](image)

### Appendix B – Loan approval example

This loan approval example is based on the loan approval sample provided with BPEL4J, which is in turn based on an example from the BPEL specification.

The loan approval process is initiated when a customer makes a loan request. Approval from an approver partner must be received if the loan is of high value, or if a risk assessment from an assessment does not determine the loan to be low risk. A synchronous reply indicating whether or not the approval has succeeded is returned to the customer.

The packaging structure of the loan approval example is shown in Figure 84. The LoanApprover and LoanAssessor packages contain the interfaces that are used in the example and the classes referred to by those interfaces. The LoanDefinitions package contains classes that are referred to in both the LoanApprover and LoanAssessor packages. The protocols through which the loan approval process communicates with its partners are defined in the LoanApprovalProtocols package. The loan approval process is defined in the LoanApprovalProcess package.
The classes introduced in the LoanDefinitions package are shown in Figure 85. These are stereotyped by <<messageContent>> to indicate that they are used to introduce a message type and should lead to the generation of WSDL message types and not XSD types.

![Diagram showing LoanDefinitions package and its components](image)

**Figure 84: Loan approval packing**

The LoanApproval package, shown in Figure 86, introduces an interface for requesting a loan approval and receiving a synchronous response.

```plaintext
<interface>
LoanApproval
+ approve ([in] creditInfo : CreditInformation, [out] loanProcessFault : LoanRequestError) : Approval
</interface>
```

**Figure 86: LoanApproval package**

Figure 87 shows the LoanAssessment package which introduces an interface for requesting a risk assessment (and receiving a synchronous response).
The LoanApprovalProtocols package, shown in Figure 88, introduces the collaborations in which the loan approval process participates. Each protocol is placed in a subpackage of LoanApprovalProtocols, the LoanApproval:: and RiskAssessment:: prefixes to the role names indicate the subpackage to which they belong.

Note that although the process has three partners, only two protocols are provided. This is because the process participates in the LoanApproval protocol twice, once as an Approver (to the customer) and once as an ApprovalRequester (to the approver).

The loan approval process is defined in the LoanApprovalProcess as shown in Figure 89. The loan approval process has four attributes for storing the state of the process, the types of these attributes are classes that are defined in the packages described earlier.

LoanApproval participates in collaborations with three partners and therefore has three ports: customer, assessor and approver.

The interfaces provided to, or required from, the customer are defined by the Approver role, in this case, the LoanApproval process provides the LoanApproval interface through the customer port.

The interfaces provided to, or required from, the approver are defined by the ApprovalRequester role, in this case the process requires the LoanApproval interface through the approver port.

The interfaces provided to/required from the assessor are defined by the AssessmentRequester role, in the case the process requires the RiskAssessment interface via the approver port.

There is no need for correlation in this example since there are no messages initiated by partners after the initial request that creates a new instance of the process.
The behavior of the process is shown in Figure 90. The process starts when an approve request is received from the customer. The amount and risk-level (obtained from the assessor, if necessary) of the loan determine whether an approval is required from the approver.

The merge node before the invokeApprover activity indicates that there are two paths that can lead to the execution of this activity (either the requested loan amount is over 10,000 or it is below that amount but not low risk). The merge node is required since the meaning of two incoming control links to the same activity is that both must be activated for the activity to execute.

The LoanApproval process invokes operations which may throw exceptions. Exception handling is not shown in Figure 90; instead we show error handling on a separate diagram.

Figure 91 shows a catch activity, handleLoanProcessFault, which is defined at the top-level of the process and therefore applies to any activity in the process. If a loanProcessFault is thrown then it is caught by the handleLoanProcessFault activity and placed into the error attribute of the process. It is then returned as an invalidRequest exception from the approve operation which was synchronously invoked by the customer.

Figure 91: Exception handling for loan approval process
Loan approval WSDL

WSDL for the LoanDefinitions package:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/"
    xmlns:LoanDefinitions="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:message name="CreditInformation">
        <wsdl:part name="firstName" type="xsd:string"/>
        <wsdl:part name="name" type="xsd:string"/>
        <wsdl:part name="amount" type="xsd:int"/>
    </wsdl:message>
    <wsdl:message name="LoanRequestError">
        <wsdl:part name="errorCode" type="xsd:int"/>
    </wsdl:message>
</wsdl:definitions>
```

WSDL for LoanApprover package:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprover/"
    xmlns:LoanApprover="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprover/"
    xmlns:LoanDefinitions="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:import
        location="http://localhost:8080/BPEL/LoanApproval/LoanDefinitions.wsdl"
        namespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/"/>
</wsdl:definitions>
```
WSDL for LoanApproval package:

```xml
<wsdl:definitions
    targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApproval/
    xmlns:LoanApproval="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApproval/
    xmlns:LoanDefinitions="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:message name="Approval">
        <wsdl:part name="accept" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="LoanApproval">
        <wsdl:operation name="approve">
            <wsdl:input message="LoanDefinitions:CreditInformation" name="creditInfo"/>
            <wsdl:output message="LoanApprover:Approval" name="loanProcessFault"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

WSDL for LoanAssessor package:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanAssessor/
    xmlns:LoanAssessor="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanAssessor/
    xmlns:LoanDefinitions="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:import
        location="http://localhost:8080/BPEL/LoanApproval/LoanDefinitions.wsdl"
        namespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/"/>
    <wsdl:message name="RiskAssessmentDetails">
        <wsdl:part name="risk" type="xsd:string"/>
    </wsdl:message>
    <wsdl:portType name="RiskAssessment">
        <wsdl:operation name="check">
            <wsdl:input message="LoanDefinitions:CreditInformation" name="creditInfo"/>
            <wsdl:output message="LoanAssessor:RiskAssessmentDetails"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```
WSDL, with BPEL extensions, for LoanApprovalProtocols package:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprovalProtocols/
    xmlns:LoanApprover="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprover/
    xmlns:LoanAssessor="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanAssessor/
    xmlns:slnk="http://schemas.xmlsoap.org/ws/2003/03/service-link"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <wsdl:import
        location="http://localhost:8080/BPEL/LoanApproval/LoanApprover.wsdl"
        namespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprover/
    </wsdl:import>

    <wsdl:import
        location="http://localhost:8080/BPEL/LoanApproval/LoanAssessor.wsdl"
        namespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanAssessor/
    </wsdl:import>

    <slnk:serviceLinkType name="LoanApprovalProtocols:LoanApproval">
        <slnk:role name="Approver">
            <slnk:portType name="LoanApprover:LoanApproval"/>
        </slnk:role>
    </slnk:serviceLinkType>

    <slnk:serviceLinkType name="LoanApprovalProtocols:RiskAssessment">
        <slnk:role name="Assessor">
    </slnk:serviceLinkType>

</wsdl:definitions>
```
Loan approval BPEL

BPEL for LoanApproval process:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process abstractProcess="no" enableInstanceCompensation="no"
     name="LoanApproval" suppressJoinFailure="yes"
     targetNamespace="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprovalProcess/LoanApproval/
     variableAccessSerializable="no"
     xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/
     xmlns:LoanApproval="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprovalProcess/LoanApproval/
     xmlns:LoanApprover="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanApprover/
     xmlns:LoanAssessor="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanAssessor/
     xmlns:LoanDefinitions="http://www.bpel-examples.ibm.com/BPEL/LoanApproval/LoanDefinitions/">
  <partners>
    <partner name="approver" partnerRole="Approver"
     serviceLinkType="LoanApprovalProtocols:LoanApproval"/>
    <partner name="assessor" partnerRole="Assessor"
     serviceLinkType="LoanApprovalProtocols:RiskAssessment"/>
    <partner myRole="Approver" name="customer"
     serviceLinkType="LoanApprovalProtocols:LoanApproval"/>
  </partners>
  <variables>
    <variable messageType="LoanDefinitions:CreditInformation"
     name="request"/>
    <variable messageType="LoanAssessor:RiskAssessmentDetails"
     name="riskAssessment"/>
  </variables>
</process>
```
<variable messageType="LoanApprover:Approval"
name="approvalInfo"/>
<variable messageType="LoanDefinitions:LoanRequestError"
name="error"/>
</variables>
<faultHandlers>
  <catch faultName="LoanApprover:loanProcessFault"
faultVariable="error">
    <reply faultName="invalidRequest" name="throwInvalidRequest"
      operation="approve" partner="customer"
      portType="LoanApprover:LoanApproval" variable="error"/>
  </catch>
</faultHandlers>
<flow>
  <links>
    <link name="receive1_to_Switch"/>
    <link name="invokeApprover_to_reply"/>
    <link name="assign_to_reply"/>
  </links>
</flow>
<flow>
  <links>
    <link name="Empty_to_invokeApprover"/>
    <link name="Empty1_to_invokeApprover"/>
  </links>
  <switch name="Switch">
    <case
case
      condition="bpws:getVariableData('request','amount')
      &gt;= 10000">
        <empty name="Empty">
          <source linkName="Empty_to_invokeApprover"/>
        </empty>
      </case>
      condition="bpws:getVariableData('request','amount')
      &lt; 10000">
        <sequence>
          <invoke inputVariable="request"
            name="invokeAssessor" operation="check"
<sequence>
  <case condition="bpws:getVariableData('riskAssessment','risk&apos;os;) != &apos;low&apos;">
    <empty name="Empty1">
      <source linkName="Empty1_to_invokeApprover"/>
    </empty>
  </case>
  <case condition="bpws:getVariableData('riskAssessment','risk&apos;os;) = &apos;low&apos;">
    <assign name="assign">
      <copy>
        <from expression="&apos;yes&apos;"/>
        <to part="accept" variable="approvalInfo"/>
      </copy>
      <source linkName="assign_to_reply"/>
    </assign>
  </case>
</sequence>
<target linkName="receive1_to_Switch"/>
<invoke inputVariable="request" name="invokeApprover" operation="approve" outputVariable="approvalInfo" partner="approver" portType="LoanApprover:LoanApproval">
  <source linkName="invokeApprover_to_reply"/>
</invoke>
</flow>
<receive createInstance="yes" name="receive1"/>
Quick reference

Dependency management

Package dependency

External packages

External package
## Interfaces and data types

<table>
<thead>
<tr>
<th>Interface</th>
<th>Interface1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Operation1 ([in] Message1, [out] Parameter2) : Data4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data types</th>
<th>Data1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Attribute1</td>
<td></td>
</tr>
<tr>
<td>+ Attribute2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message type</th>
<th>Message1</th>
</tr>
</thead>
<tbody>
<tr>
<td>- theData1</td>
<td>Data1</td>
</tr>
<tr>
<td>+ Attribute1</td>
<td></td>
</tr>
<tr>
<td>+ Attribute2</td>
<td></td>
</tr>
<tr>
<td>- theData2</td>
<td>Data2</td>
</tr>
<tr>
<td>+ Attribute1</td>
<td></td>
</tr>
<tr>
<td>+ Attribute2</td>
<td></td>
</tr>
</tbody>
</table>

## Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Protocol1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Role1</td>
<td></td>
</tr>
<tr>
<td>- theData1</td>
<td></td>
</tr>
<tr>
<td>+ Attribute1</td>
<td></td>
</tr>
<tr>
<td>+ Attribute2</td>
<td></td>
</tr>
</tbody>
</table>

## Process, state, and ports

<table>
<thead>
<tr>
<th>Process</th>
<th>Process1</th>
</tr>
</thead>
</table>
Data handling
### Basic activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Invoke</strong></td>
<td><img src="image" alt="Invoke Diagram" /></td>
</tr>
<tr>
<td><strong>Receive and reply</strong></td>
<td><img src="image" alt="Receive and reply Diagram" /></td>
</tr>
<tr>
<td><strong>Signaling exceptions</strong></td>
<td><img src="image" alt="Signaling exceptions Diagram" /></td>
</tr>
</tbody>
</table>
### Properties and correlation

<table>
<thead>
<tr>
<th>Property or property alias</th>
<th>«properties»</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MyProperties</td>
</tr>
<tr>
<td>+ property1 : String</td>
<td></td>
</tr>
<tr>
<td>+ property2 : Integer</td>
<td></td>
</tr>
<tr>
<td>+ property1( [in] message1 : MessageType1 ) : String</td>
<td></td>
</tr>
<tr>
<td>+ property1( [in] message2 : MessageType2 ) : String</td>
<td></td>
</tr>
<tr>
<td>+ property2( [in] message2 : MessageType2 ) : String</td>
<td></td>
</tr>
<tr>
<td>+ property2( [in] message3 : MessageType3 ) : String</td>
<td></td>
</tr>
</tbody>
</table>
| Correlation set | **CorrelationSet1**  
+ property1 : String  
+ property2 : Integer |
|-----------------|----------------------|
| Use of correlation set by process | **Process1**  
+ **correlation** correlationSet1 : CorrelationSet1 |
| Initializing correlation set | **receive**  
Activity1  
.Entry/correlation: initialize correlationSet1 |
| Use of correlation sets by activity | **receive**  
Activity2  
.Entry/correlation: correlationSet1, correlationSet2 |
| Use of correlation by invoke activity | **invoke**  
Activity3  
.Entry/out correlation: correlationSet1  
.Entry/in correlation: correlationSet1, initialize correlationSet2 |
Activity coordination

<table>
<thead>
<tr>
<th>Control links and nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity1 -&gt; Activity2</td>
</tr>
<tr>
<td>Activity2 -&gt; Activity3</td>
</tr>
<tr>
<td>Activity3 -&gt; Activity7</td>
</tr>
<tr>
<td>Activity3 -&gt; Activity1</td>
</tr>
<tr>
<td>Activity3 -&gt; Activity5</td>
</tr>
<tr>
<td>Activity3 -&gt; Activity4</td>
</tr>
<tr>
<td>Activity1 -&gt; Activity6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decision and merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity1</td>
</tr>
<tr>
<td>[expression1]</td>
</tr>
<tr>
<td>Activity2</td>
</tr>
<tr>
<td>[expression2]</td>
</tr>
<tr>
<td>Activity3</td>
</tr>
<tr>
<td>[otherwise]</td>
</tr>
<tr>
<td>Activity4</td>
</tr>
<tr>
<td>Activity5</td>
</tr>
</tbody>
</table>
Explicit fork and join

Activity1 → Activity2

Activity3 → Activity4

Activity5 → Activity6

Loop

«loop»

Activity6

[expression1]

Activity1

Pick

«pick>>

«receive»

Activity1

Activity5

Activity1

Activity2

Activity6

Activity2

Activity3

Activity7
Error handling

Catch and throw

Compensation
15 Notices

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk,
NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106,
Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.
If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks and service marks**

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

IBM

WebSphere

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.