JBoss jBPM BPEL User Guide

Orchestration made practical

Version: 1.1.Beta2

Table of Contents

1 Overview of BPEL

BPEL stands for *Business Process Execution Language*. Originally authored by a small charter of vendors in the software industry, it is currently under standardization at OASIS (1).

BPEL is an XML language for describing long-running interactions of web services centrally orchestrated by a coordinator, and exposing them as new web services. The coordinator makes decisions by interpreting «business processes».

BPEL has quickly become the dominant specification to standardize integration logic and process automation between Web services. It has bypassed a number of alternative specifications such as BPML and WSCI, and became a real candidate to drive the existing web service infrastructure into process-centric applications based on a service-oriented architecture.

1.1. The service-oriented architecture

Recent products and strategies announced by IBM, BEA, Microsoft and Oracle are targeted at the **service-oriented architecture** (2). SOA helps enterprises modify their infrastructure «on the flight», according to changes in the environment. Under this model, new applications are developed by wiring together software components, or services, that expose reusable business functions (3).

Leveraging more from existing assets is a real business challenge right now. The idea of enabling legacy applications to be deployed as (web) services and orchestrated across platforms is a key part of why BPEL is becoming a cornerstone of SOA (4).

1.2. Industry support

- IBM and BEA worked together in a proposal named BPELJ, with the purpose of extending BPEL to integrate it more closely with the J2EE platform (5)
- BEA will provide full support for BPELJ in the release of WebLogic Integration following version 8.1 (6). In addition, it has submitted a specification request (JSR 207) of an annotated Java syntax and APIs for developing business processes in Java and deploying them in J2EE containers (7)
- IBM already offers that support as part of WebSphere Business Integration Server 5.1 (8)
- Microsoft BizTalk 2004 includes BPEL import and export capabilities (9)
- Oracle acquired Collaxa, a company that focused on implementing the standard since 2002 (10)

1.3. OASIS Technical Comittee

Despite its acceptance, it is recognized that BPEL remains an emerging technology. Challenges await those interested in near-term deployment. Version 1.1, dated May 5, 2003, contains plenty of gaps and ambiguities that should come as no surprise, considering these factors:

- The separate roots of the specification (12). Microsoft initiated the adoption of Pi-Calculus with XLANG, whereas IBM revisited the use of Petri Nets with the Web Services Flow Language, WSFL. BPEL descends from both languages.
- The immaturity of the overall Web services «stack»

The OASIS WS-BPEL technical committee is working diligently to overcome these deficiencies. It released the first draft of version 2.0 in July 30, 2004 and continues to publish drafts periodically as it resolves issues. Version 2.0 is receiving contributions from a much broader community and becoming the standard everyone wants it to be.

1.4. Components and dependencies

Table 1.1. BPEL components

Component	**Dependency**
Partner relationships	WSDL 1.1 port types
Data representation	XML Schema 1.0
Data manipulation	XPath 1.0
Message exchange	WSDL 1.1 messages
Flow-control structures	-
Scoping: • Compensation («undo») • Error handling • Event processing	-

1.5. References

1. OASIS Web Services Business Process Execution Language TC [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel]

2. Service-Oriented Architecture [http://inews.webopedia.com/TERM/S/Service_Oriented_Architecture.html]

3. IBM Refocuses WebSphere on Service-Oriented Architectures [http://www.rednova.com/news/stories/3/2004/04/30/story103.html]

4. IBM Unveils First SOA Products, Services [http://www.internetnews.com/ent-news/article.php/3343231]

5. BEA, IBM Propose Java/BPEL 'Marriage' [http://itmanagement.earthweb.com/entdev/article.php/3337731]

6. BEA - BPEL & BPELJ [http://dev2dev.bea.com/technologies/bpel/index.jsp]

7. JSR 207: Process Definition for Java [http://www.jcp.org/en/jsr/detail?id=207]

8. WebSphere Business Integration Server Foundation [http://www-306.ibm.com/software/integration/wbisf/features/]

9. MSDN - Importing BPEL4WS [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sdk/htm/ebiz_prog_orch_bfpe.asp]

10. Oracle BPEL Process Manager [http://www.oracle.com/technology/products/ias/bpel/index.html]

11. Active Endpoints - Siebel Alliance [http://www.active-endpoints.com/siebel/]

12. A Convergence Path toward a Standard BPM Stack [http://www.bpmi.org/downloads/BPML-BPEL4WS.pdf]

2

Introduction to jBPM BPEL

jBPM is a platform for graph-based execution languages. Its design and pluggable architecture makes it possible to support different languages that can be shown as a graph and represent some sort of execution. The goal of this project is to fully implement the BPEL specification by leveraging the jBPM foundation.

jPDL is jBPM's core workflow language. It was designed to fully fledge the capabilities of the jBPM API. BPEL is an emerging standard for assembling a set of discrete services into an end-to-end process flow. Even when there is actually an overlap in their functionality, they are targeted at different audiences. Let us take a look at their similarities and differences. If you are a jBPM connoisseur and still wonder what BPEL is, this might help you get started.

2.1. Flow control

jPDL specifies the execution flow of a process in terms of a directed graph of nodes. It includes a set of node types that intend to cover most routing scenarios. Furthermore, its flexibility allows including custom routing logic when facing an eccentric process scenario. In contrast, BPEL has a fixed set of structured activities represented by XML elements. They are nested together to model a particular execution path. Among them we can find control structures present in most programming languages like `sequence`, `while`, and `switch`.

A more advanced structured activity worth mentioning is `flow`. It describes parallel paths of execution. Most importantly, it can declare links, which are control dependencies between its enclosed activities. Links allow for modelling directed graph flows, OR / AND join conditions for the execution of activities and even make it possible to detect dead paths of execution.

Below, a simplified version of the jBAY auction process coded in BPEL:

```
<process name="auction" targetNamespace="http://www.jBAY.com"
      xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
      ...
      <sequence>
      <!-- Seller registers a sale offering and bids from the clients are received.
       The highest bid is taken -->
       ...
      <flow>

         <sequence name="shippingSequence">
           <invoke name="sendItem" operation="shipItem" partnerLink="shipper"/>
           <receive name="receiveItem" operation="confirmDelivery" partnerLink="shipper"/>
          </sequence>

         <sequence name="billingSequence">
           <receive name="receiveMoney" operation="notifyDeposit" partnerLink="bank"/>
           <invoke name="sendMoney" operation="depositMoney" partnerLink="bank"/>
         </sequence>

       </flow>
      </sequence>

    </process>
```

The `receive` and `invoke` elements nested in the structured elements of the jBAY process are basic activities. They perform the actual work. The `receiveMoney` and `receiveItem` activities act as *wait states*. In jPDL, any node that interrupts the execution path is a wait state, whereas in BPEL this behavior is given by the `receive`, `wait` and `pick` activities. In general, the execution of a process is only interrupted to wait for either an alarm to go off or a message to arrive.

2.2. Data Handling

jPDL's variable context is based on POJOs. Process data can be manipulated by inserting BeanShell code in a script node or invoking the methods of class `ContextInstance` inside action handlers. BPEL's variable context is made up by XML constructs. You can manipulate data within the assign activity using XPath expressions and in the `receive` and `invoke` activities, where message content is received either by an external call or by the return value of a service invocation.

2.3. Interaction with the process

Clients of a jPDL definition are expected to start or resume process instances through the jBPM API. Methods such as `ProcessDefinition`. `createProcessInstance` and `Token`. `signal` allow client code to interact directly with an executing process.

BPEL takes a different approach. Instead of defining its own APIs, it accommodates custom web service interfaces with which clients interact. These interfaces describe meaningful business operations and hide the fact that clients are actually "talking" to an orchestrator. In the next figure, the participants of the jBAY process interact with the auction endpoint. The orchestrator remains opaque.

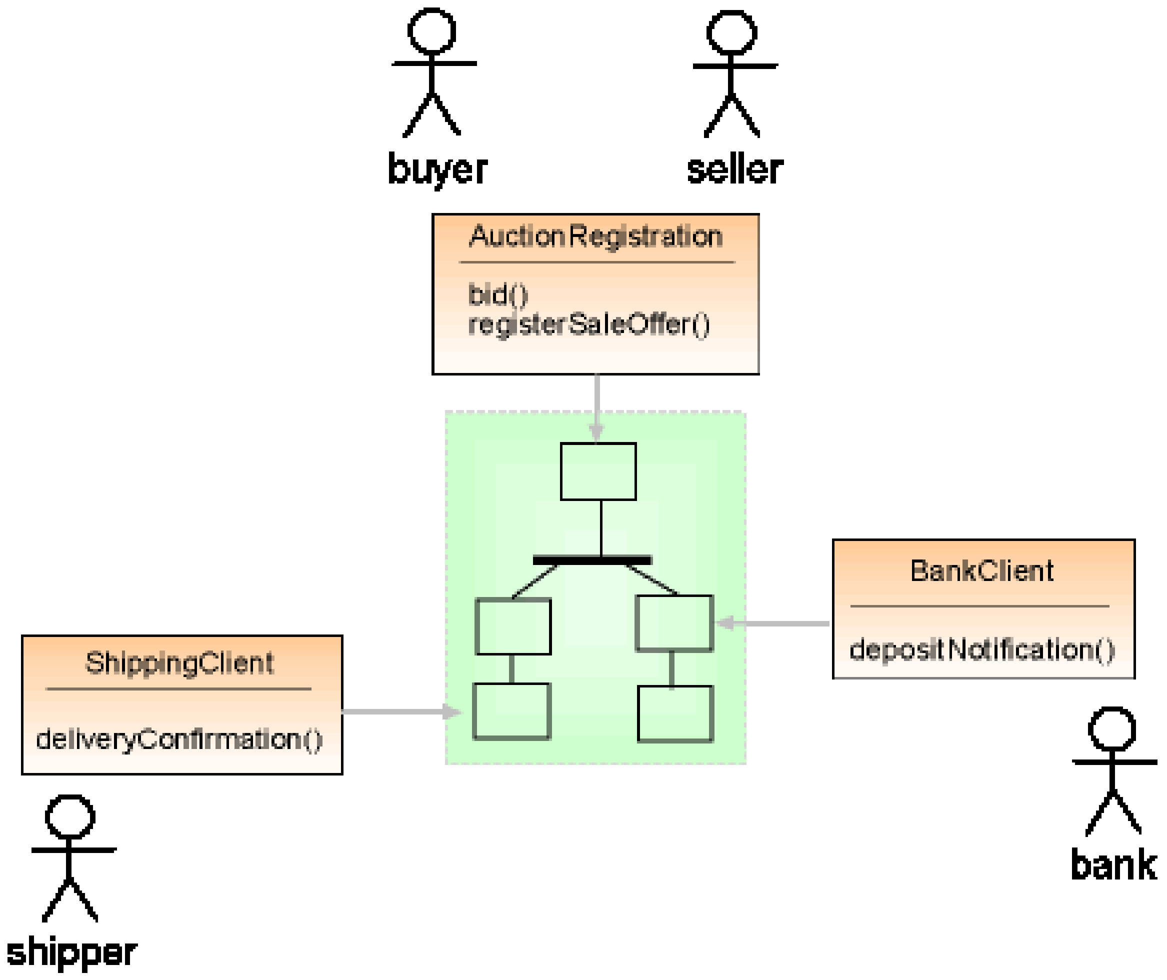


Figure 2.1. Participants of the auction process

The service operations are connected to the process definition through inbound message activities. They mark *entry points* inside a process definition. When a client invokes their corresponding operation they respond in one of two ways:

1. behave as start states and trigger a new process instance
2. resume a process instance previously suspended

2.4. Invoking services

jPDL processes use action handlers to invoke external services. Calls to Java component can be coded in an action handler and executed later inside a process. In BPEL, this behavior is achieved using the `invoke` activity. While it is only capable of calling web services right now, there are plans to use Apache WSIF [http://ws.apache.org/wsif/] to call any Java component described in a WSDL document

2.5. Which one is for me?

Some business processes involve frequent interactions with heterogeneous systems or partners. In these cases interoperability is essential, and XML is the obvious data transfer format. If you run into a similar scenario, we encourage you to choose BPEL. Manipulating XML documents with Java or any other non-XML language is tedious, verbose and error-prone. BPEL alleviates much of this pain through the following features:

- message exchange with efficiency (no Java/XML binding) and type safety (automatic format checking)
- comfortable message content manipulation (XPath 1.0 expressions)
- asynchronous message reception
- encapsulation of the underlying web services machinery

If this is not your case and you are mostly coordinating Java components, use jPDL. You'll find it easier to use and you will be able to model practically any imaginable scenario by leveraging jBPM's workflow-rich features.

3 Getting started

3.1. Getting JBoss jBPM BPEL

The releases of the jBPM BPEL extension can be found in the jBPM download page [http://www.jboss.com/products/jbpm/downloads]. The package JBoss jBPM BPEL contains the jBPM BPEL software (sources and binaries) plus the thirdparty libraries it depends on.

The BPEL extension is based on jBPM 3. Downloading the jBPM distribution separately is strongly recommended but strictly optional, as the BPEL package already contains the required jBPM binaries.

Once you download the package, unzip it to a suitable location in your machine. It should all unzip into a single directory named `jbpm-bpel-<version>`. Make sure you don't use a directory which has any spaces in the path (such as the `Program Files` directory on Windows) as this may cause problems.

Alternatively, you can get the software from CVS with the following parameters. Once you have access, look for module `jbpm.bpel`.

Table 3.1. CVS parameters

Parameter	Value
Connection type	pserver
User	anonymous
Password	(when prompted for a password, just press enter)
Host	anoncvs.forge.jboss.com
Port	2401 (the default)
Repository path	/cvsroot/jbpm

3.2. Getting JBoss Application Server

The jBPM BPEL software builds on J2EE 1.4 APIs and a number of open source projects. While it is not tied to JBoss AS, the guide provides deployment instructions for JBoss AS only. Even if you use a different application server in production, we still encourage you to try this software in development along with JBoss AS. If you like it and decide to take it to production, we can help you through the forum [http://tinyurl.com/kjru7] or our support services [http://www.jboss.com/services/index].

For instructions on getting JBoss AS, see Downloading and Installing JBoss [http://tinyurl.com/kl9wl].

Note

Please make sure you select a JBoss AS version and server profile compatible with J2EE 1.4. Specifically, jBPM BPEL has been tested with versions 4.0.3SP1 and 4.0.4.GA in the `default` profile.

3.3. Getting Apache Ant

Ant scripts dramatically simplify the development and deployment of BPEL proceses. The tutorial chapter assumes you have installed Ant and are able to execute targets. Furthermore, you need Ant to build the jBPM BPEL service archive.

For directions on getting Ant, see Installing Ant [http://ant.apache.org/manual/install.html].

Note

In order to run `junit` tasks, Ant requires the JUnit library. You will find this library in the `lib/junit` directory of the jBPM BPEL distribution. Remember to copy it to Ant's `lib` directory.

3.4. Getting the Eclipse BPEL designer

The Eclipse community has begun a BPEL project [http://www.eclipse.org/bpel/]. The goal of this project is "to add comprehensive support to Eclipse for the definition, authoring, editing, deploying, testing and debugging of WS-BPEL 2.0 processes".

The JBoss jBPM team has been in touch with the comitters of this project. Expect tighter integration between the Eclipse Designer and jBPM BPEL shortly after key pieces of functionality such as Runtime Framework and Debug become available.

You do not have to wait, tough. You can get started with the Eclipse BPEL designer and jBPM BPEL **today**! Please visit our Wiki [http://wiki.jboss.org/wiki/Wiki.jsp?page=JbpmBpelDesigner] knowledge base for a Quick Start guide.

3.5. Service setup

The jBPM BPEL service archive (`.sar`) contains the libraries and configuration files required to execute BPEL processes. The following sections describe how to configure, package and deploy the jBPM BPEL service to JBoss AS.

3.5.1. Service configuration

The configuration files reside in `src/resources/jbpm-bpel.sar`. These files provide settings to Hibernate and to jBPM itself.

3.5.1.1. Hibernate configuration

Starting from version 1.0 alpha 3, the BPEL extension incorporates the jBPM persistence model, described in the chapter on persistence [http://docs.jboss.com/jbpm/v3/userguide/persistence.html] of the jBPM user guide. jBPM uses Hibernate as its object-relational mapper. The next table describes the Hibernate configuration files.

Table 3.2. Hibernate configuration files

File	Notes
`hibernate.properties`	Contains database connection properties and miscellaneus values. Since the service runs inside the application server, you should configure Hibernate to obtain connections from a data source registered in JNDI. Refer to the Configuration [http://tinyurl.com/e4mp3] chapter of the Hibernate manual for instructions.
`ehcache.xml`	Supplies parameters for the default second-level cache provider. The chapter on Performance [http://tinyurl.com/pv6w9] in the Hibernate manual describes the use of caches to improve performance.

Apart from the above files, jBPM BPEL service includes a data source deployment descriptor: `jbpm-bpel-ds.xml`. In addition to the preconfigured data source, this file also describes a `HypersonicDatabase` MBean. Upon deployment, this MBean creates a database in the same VM the server is running. This database is written to disk when the server shuts down, but no other process has access to it.

Administrators often prefer to separate data sources from services and applications. The data source described in the previous paragraph is intended for evaluation and development purposes. It should not be regarded as a production arrangement.

3.5.1.2. jBPM settings

The file `jbpm.cfg.xml` contains the jBPM configuration. The following table summarizes the relevant information items in the context of BPEL.

Table 3.3. Relevant jBPM information items for BPEL

Setting	Notes
`<jbpm-context>`	Configures the jBPM context with a set of services. The persistence and relation services are mandatory; other services may be disabled at will.

Setting	Notes
`<string name="resource.hibernate.cfg.xml">`	References an alternate XML resource listing the Hibernate mapping files.
`<string name="resource.hibernate.properties">`	References an alternate properties resource containing the JDBC connection parameters.

3.5.2. Packaging

Once you have set the configuration, building the service archive is easy. Copy or rename file `ant.example.properties` in your jBPM BPEL base directory to `ant.properties`. Edit the values therein to match the directory where you installed JBoss AS and your preferred server configuration.

```
# JBoss AS installation directory
jboss.home=/jboss/home
# JBoss server configuration
jboss.server=default

# jBPM library versions
jbpm.version=3.1.2
jbpm.bpel.version=1.1.Beta2
```

To build the service archive, locate `build.xml` in the base directory and run the target that corresponds to your JBoss AS version.

Table 3.4. Ant targets for building the service archive

Target	Version
`build.service.402`	4.0.2
`build.service.403`	4.0.3 SP1
`build.service.404`	4.0.4 GA

If all goes well you should see messages much like the following:

```
build.service.404:
      [jar] Building jar: .../jbpm-bpel.sar
```

JBoss AS 4.0.4 and later versions ship with the new web services stack, *JBossWS*. Like its predecessor in earlier JBoss AS versions, this stack implements the Web Services for J2EE specification, version 1.1. However, some idiosyncrasies of the Axis-based stack result in slight incompatibilities with the new implementation. Specifically, a number of SAAJ methods do not behave as expected. JBossWS exhibits results that match the reference implementation.

Starting from version 1.1.Beta2, jBPM BPEL has been revised to work with both stacks. The workarounds used to maintain compatibility with the old stack might fail when parsing or formatting certain SOAP envelopes. If you encounter a SOAP-related problem, please try upgrading to JBoss AS 4.0.4 or later to determine whether the old stack is causing the problem. Should the issue remain, let us know via the forum [http://tinyurl.com/kjru7].

3.5.3. Deployment

Before deploying the service, start JBoss in the server configuration of your choice (see Server Configurations [http://tinyurl.com/rjqkb]). Afterwards, run the `deploy-service` target of `build.xml` in the jBPM BPEL installation directory.

The following log entries confirm the deployment of the jBPM BPEL data source, service and web application, respectively.

```
00:39:45,390 INFO  [ConnectionFactoryBindingService] Bound ConnectionManager 'jboss.jca:name=jbpmBpelDS,s
00:39:45,406 INFO  [JbpmConfiguration] using jbpm configuration resource 'jbpm.cfg.xml'
00:39:45,437 INFO  [TomcatDeployer] deploy, ctxPath=/jbpm-bpel, warUrl=.../tmp/deploy/tmp30743jbpm-bpel-e
```

JBoss is now ready to deploy process definitions and BPEL-powered web services.

4 Tutorial

The best way to get acquainted with BPEL is to see it in action. For this reason, we included some examples that will help you get started quickly. At this point, you should have the jBPM BPEL service running inside JBoss. If you do not, please read the Getting started chapter first.

In this chapter we will guide you through the example setup procedure. The next two chapters contain the actual examples.

4.1. Getting the Java Web Services Development Pack

jBPM BPEL relies on the Web Services for J2EE (WSEE) model for exposing the functionality a BPEL process delivers. Some of the server artifacts required for a WSEE deployment can be automatically generated from WSDL documents.

Starting from JBoss AS 4.0.4, the web services implementation features a toolkit called `wstools`. If you are using that version of JBoss AS, no further action is required in this section.

For earlier JBoss AS versions, JBoss recommended the `wscompile` tool from the Java Web Services Development Pack. Please refer to the Java WSDP [http://java.sun.com/webservices/jwsdp/index.jsp] site for download and installation instructions.

4.2. Defining the environment

The resources related to each sample process reside in a separate subdirectory under `doc/examples`. Each subdirectory contains an Ant script to assist in carrying out process deployment tasks.

Note that the individual scripts import a template located in the `common` subdirectory. If you organize your resources in the same way as the examples, you can take advantage of this template in your own project.

Each runtime environment is likely to be different from all others. For this reason, the build template needs to be made aware of your particular environment. Copy or rename file `ant.example.properties` in the `doc/examples` subdirectory to `ant.properties`. Edit the values there to match your jBPM BPEL and JWSDP installation directories. The new values are shared among all examples.

```
# jBPM BPEL installation directory
jbpm.bpel.home=/jbpm/bpel/home

# JWSDP installation directory
# optional for JBoss 4.0.4+
# required for JBoss 4.0.3-
!jwsdp.home=/jwsdp/home
```

Controlling multiple processes at once becomes as easy as controlling a single process with the build file located in `doc/examples`. This script leverages the `subant` task for running build targets on a collection of projects.

4.3. Creating the database schema

Starting with version 1.0 alpha 4, a web application deployed along the jBPM BPEL service executes administrative requests such as creating the database schema and deploying a process definition.

Before you deploy process definitions to the database, the jBPM schema must exist in the database. The schema creation can be manual or automatic, at your choice.

Automatic schema creation is convenient for evaluation and development purposes. This is why it is the default setting. To disable it, remove the `hibernate.hbm2ddl.auto` property from `hibernate.properties` in `src/resources/jbpm-bpel.sar` and redeploy the jBPM BPEL service.

When you turn off automatic schema creation, you need a way to create it manually. The global process deployment script provides some targets for posting administrative schema requests to the jBPM BPEL web application: `create-schema` and `drop-schema`.

Hibernate will emit a few log messages to the server console when the schema gets created either way. The output below corresponds to manual creation.

```
22:45:00,796 INFO  [SchemaExport] Running hbm2ddl schema export
22:45:00,812 INFO  [SchemaExport] exporting generated schema to database
22:45:00,828 INFO  [DatasourceConnectionProvider] Using datasource: java:/jbpmBpelDS
22:45:00,921 INFO  [SchemaExport] schema export complete
```

4.4. Replacing the default implementation of the XML APIs

There are known issues with the implementations of the XML APIs in Java SE 1.4.2 and 1.5.0. The solution is to employ the Endorsed Standards Override Mechanism for replacing these implementations with more recent versions of Apache Xalan and Apache Xerces.

You will find adequate replacement libraries in the `lib/endorsed` subdirectory of JBoss AS. Please follow the instructions for deploying endorsed standards classes in the Java SE Guide to Features [http://java.sun.com/j2se/1.4.2/docs/guide/standards/].

5

Hello World Example

In this example, we will develop a very simple BPEL process that receives a message carrying the name of a person, composes a greeting phrase containing the name, and then replies with a message carrying the greeting. The resources related to this example come with in the download package. Look in `doc/examples/hello` .

The following picture represents the Hello World process.

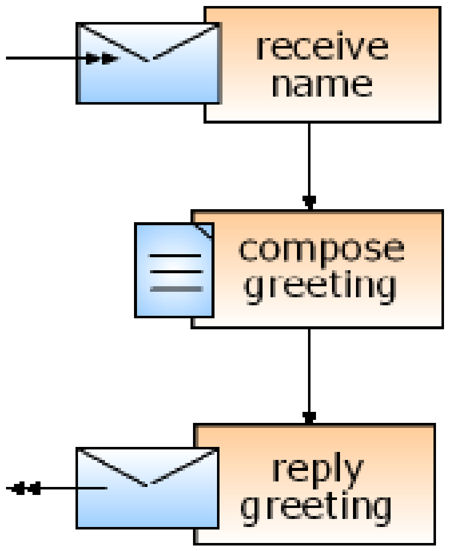


Figure 5.1. Graphical representation of the Hello World process

5.1. Define the BPEL process

5.1.1. Create the BPEL document

The first step is creating the process definition document. The description ahead assumes familiarity with the BPEL concepts, but even if you are new to BPEL you will find it easy to follow. If you get interested in BPEL and decide to take it to a real project, you should read the specification in its entirety. The OASIS BPEL Technical Committee web site [http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel] provides the specification document and a number of useful resources.

Let's call our process document `hello.bpel`. We create a partner link to establish a relationship with the client of the process. We indicate that the process will play the *Greeter* role. Next we create two variables to hold the incoming and outgoing messages. Finally, we create a sequence of three activities that receives a request message from a client, prepares a response message and sends it back.

```
<process name="HelloWorld" targetNamespace="http://jbpm.org/examples/hello"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="http://jbpm.org/examples/hello"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <!-- establishes the relationship with the caller agent -->
    <partnerLink name="caller" partnerLinkType="tns:Greeter-Caller"
      myRole="Greeter" />
  </partnerLinks>

  <variables>
    <!-- holds the incoming message -->
    <variable name="request" messageType="tns:nameMessage" />
    <!-- holds the outgoing message -->
    <variable name="response" messageType="tns:greetingMessage" />
  </variables>

  <sequence>

    <!-- receive the name of a person -->
    <receive operation="sayHello" partnerLink="caller" portType="tns:Greeter"
      variable="request" createInstance="yes" />

    <!-- compose a greeting phrase -->
    <assign>
      <copy>
        <from expression="concat('Hello, ',
            bpel:getVariableData('request', 'name'), '!')" />
        <to variable="response" part="greeting" />
      </copy>
    </assign>

    <!-- reply with the greeting -->
    <reply operation="sayHello" partnerLink="caller" portType="tns:Greeter"
      variable="response" />
  </sequence>

</process>
```

Note

The `partnerLinkType` and `messageType` attributes refer to external WSDL definitions. We will deal with them in the next section.

5.1.2. Create/obtain the WSDL interface documents

WSDL documents describe the interface of the process that will be presented to the outside world. To promote clarity and reuse, the WSDL specification [http://www.w3.org/TR/wsdl#_style] recommends separating the different elements of a service definition into independent documents according to their level of abstraction. The proposed levels are data type definitions, abstract definitions, and specific service bindings.

A service interface document describes a specific type of service. It contains the `types`, `import`, `message` and `portType` elements; it can reference other abstract definitions documents using `import` elements. A service implementation document contains the description of a service that implements a service interface. It contains the `import`, `binding` and `service` elements. At least one of the `import` elements references the WSDL interface document.

The process definition is dependent on data type definitions and abstract definitions. The BPEL runtime is responsible of supplying the specific bindings for web services produced by a BPEL process. The specific bindings for partner services can be typically obtained at deployment or runtime.

We use only one WSDL interface document. Let's name it `hello.wsdl`. We create two mesages that respectively carry the name and greeting. Next we create a port type that describes the interface that the process presents to its callers. It exposes a single operation `sayHello`, which takes the name message as input and returns the greeting message as output.

Once the service interface is defined, we create a partner link type to characterize the relationship between greeter and caller. We define the roles played by each service and specify the interfaces (i.e. port types) they expose to each other. Because our greeter process does not call the client back, only one role appears. No responsibilities are placed on the caller.

```
<definitions targetNamespace="http://jbpm.org/examples/hello"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://jbpm.org/examples/hello"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:plt="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">

  <!-- characterizes the relationship between the greeter and its caller -->
  <plt:partnerLinkType name="Greeter-Caller">
    <plt:role name="Greeter">
      <plt:portType name="tns:Greeter" />
    </plt:role>
    <!-- the Caller does not provide services to the Greeter,
      this is why we omit the "Caller" role -->
  </plt:partnerLinkType>

  <!-- carries the name of a person -->
  <message name="nameMessage">
    <part name="name" type="xsd:string" />
  </message>

  <!-- carries the greeting -->
  <message name="greetingMessage">
    <part name="greeting" type="xsd:string" />
  </message>

  <!-- describes the interface presented to callers -->
  <portType name="Greeter">
    <operation name="sayHello">
      <input message="tns:nameMessage" />
      <output message="tns:greetingMessage" />
    </operation>
  </portType>

</definitions>
```

Notice the values of `name` attributes in child elements of `definitions` match the names used earlier in the process definition.

5.1.3. Deploy the process definition

jBPM provides a mechanism to package all files related to a process definition into a process archive, and then deploy the archive to a database. The BPEL extension uses this mechanism to make the definition available to the port components and the BPEL application (we will talk about it in Java mapping artifacts).

The central file in the process archive is the definition descriptor, `bpel-definition.xml`. It specifies the location of the process document within the package. The descriptor also indicates the location of WSDL interface files, either relative to the package root or at some absolute URL.

```
<bpelDefinition location="hello.bpel" xmlns="http://jbpm.org/bpel">

  <!-- makes WSDL interface elements available to the process -->
  <imports>
    <wsdl location="hello.wsdl" />
  </imports>

</bpelDefinition>
```

To deploy the process definition to the jBPM database, call:

```
ant deploy-definition
```

This will build a file named `hello-process.zip` and submit it to the jBPM service. When successful, the messages below appear in the server console:

```
23:55:21,750 INFO  [[/jbpm-bpel]] processDeployServlet: deploying process definition: file=file:/.../hell
23:55:22,078 INFO  [BpelReader] read wsdl definitions: hello.wsdl
23:55:22,171 INFO  [BpelReader] read bpel process: hello.bpel
23:55:22,250 INFO  [[/jbpm-bpel]] processDeployServlet: deployed process definition: HelloWorld
```

Note

If this is the first process you deploy after starting the jBPM service, you will see a flurry of logs from Hibernate before the last log shown above. This is normal as long as no ERROR entries appear. Hibernate initializes on demand.

5.2. Build the WSEE port components

In WSEE, a port component defines the server view of a web service. It services the operation requests defined by a WSDL `portType`. In this step, we will create one port component for each partner link that defines a process role; that is, `partnerLink` elements having a `myRole` attribute.

5.2.1. WSDL implementation documents

WSDL implementation documents for the process can be automatically generated from the process definition and related WSDL interface documents. The BPEL extension includes a tool just for that purpose, `servicegen`. This program reads a process archive.

To execute the tool, call:

```
ant generate-service
```

Three WSDL files will appear:

- `hello.wsdl` is an equivalent of the interface document we created in Create/obtain the WSDL interface documents. Because the location we specified in Deploy the process definition is relative, the tool writes this file so that its definitions are available to the port component.

- `binding1.wsdl` contains the SOAP binding for the `Greeter` port type we saw in Create/obtain the WSDL interface documents. Note that the target namespace of this document is the same as that of the port type: `http://jbpm.org/examples/hello`. The subsequent listing corresponds to the generated binding document, edited for clarity.

  Note

  If the process implemented other port types belonging to namespace `http://jbpm.org/examples/hello` they would appear in the above document, too. Bindings for port types in other namespaces would be placed in a separate `binding<n>.wsdl` file.

```
<definitions targetNamespace="http://jbpm.org/examples/hello"
  xmlns:tns="http://jbpm.org/examples/hello"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- makes WSDL interface elements available to binding elements -->
  <import namespace="http://jbpm.org/examples/hello" location="hello.wsdl"/>

  <!-- provides SOAP 1.1 protocol details for the Greeter interface -->
  <binding name="GreeterBinding" type="tns:Greeter">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="http://jbpm.org/examples/sayHello"/>
      <input>
        <soap:body use="literal" namespace="http://jbpm.org/examples/hello"/>
      </input>
      <output>
        <soap:body use="literal" namespace="http://jbpm.org/examples/hello"/>
      </output>
    </operation>
  </binding>
</definitions>
```

- `service.wsdl` contains a service element in the same target namespace as the process. The `GreeterPort` subelement implements the `Greeter` port type using its SOAP binding. The generated service document is reproduced next.

  Note

  The tool generates one port for each partner link having a `myRole` attribute.

```
<definitions targetNamespace="http://jbpm.org/examples/hello" xmlns:tns="http://jbpm.org/examples/hello"

  <import namespace="http://jbpm.org/examples/hello" location="binding1.wsdl"/>

  <!-- groups all endpoints served by the process -->
  <service name="HelloWorldService">

    <!-- supplies access information for the Greeter interface -->
    <port name="GreeterPort" binding="tns:GreeterBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URI"/>
    </port>

  </service>

</definitions>
```

Notice that the actual access information is left unspecified, as WSEE will replace this entry with the definitive location during deployment.

5.2.2. Java mapping artifacts

The Java mapping artifacts required for a WSEE deployment can be automatically generated from the WSDL implementation documents. Depending on your JBoss AS version, the build file calls either `wstools` or `wscompile`. Both tools read a (distinct) configuration file which points to a WSDL file, among other settings.

To execute the generation tool, call:

```
ant generate-artifacts
```

The subsequent snippet shows the input to `wstools`. Here, `package-namespace` elements associate Java packages with XML namespaces occurring in the WSDL file.

```
<configuration xmlns="http://www.jboss.org/jbossws-tools">
  <global>
    <package-namespace package="org.jbpm.bpel.tutorial.hello"
      namespace="http://jbpm.org/examples/hello" />
  </global>
  <wsdl-java file="wsdl/service.wsdl">
    <mapping file="jaxrpc-mapping.xml" />
  </wsdl-java>
</configuration>
```

Tip

The JBossWS user guide [http://tinyurl.com/r2jru] gives a full coverage of the available options.

The configuration for `wscompile` is quite similar. The `packageName` attribute associates the target namespace of the referenced WSDL definitions with the Java package `org.jbpm.bpel.tutorial.hello`.

```
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="src/main/resources/WEB-INF/wsdl/service.wsdl"
    packageName="org.jbpm.bpel.tutorial.hello" />
</configuration>
```

Tip

Refer to the JWSDP tools documentation [http://tinyurl.com/joocc] for a complete description of this file.

The generated Java sources contain the service endpoint interface `Greeter` and the service interface `HelloWorld-Service`. The generation tool also writes a document that describes how the WSDL interface definitions map to the produced Java types: `jaxrpc-mapping.xml`.

We will not analyze the Java mapping artifacts as jBPM BPEL is agnostic to them. Keep in mind that variables in a BPEL process are defined in terms of XML types and WSDL messages. jBPM BPEL extracts XML content from SOAP messages and places it in the process variables directly.

Nevertheless, the Java mapping artifacts still must be present for the JSR-109 deployment to be valid. Note that the supplied service implementation bean has empty methods only. The BPEL process specifies the behavior instead.

5.2.3. Port components as servlets

In WSEE, Java service endpoints are deployed as servlets in a web application. Hence, a `web.xml` deployment descriptor must be supplied.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">

  <!-- Greeter Service Endpoint -->
  <servlet>
    <servlet-name>greeterServlet</servlet-name>
    <servlet-class>org.jboss.test.ws.samples.wsbpel.hello.Greeter_Impl</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>greeterServlet</servlet-name>
    <url-pattern>/greeter</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

Note

We refer to the service implementation bean in the `servlet-class` element, even though it is not a servlet at all. This is a WSEE feature.

5.2.4. Partner integration

In order for the process to take requests, inbound message activities (`receive` , `onMessage` branch of `pick` and `onMessage` event) must be enabled for reception. jBPM BPEL provides a servlet for this purpose.

An additional `servlet` element in `web.xml` sets up the partner integration servlet. We indicate the container to load this servlet during startup, so that inbound activities are ready before requests begin arriving.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  ...
  <!-- jBPM BPEL Partner Integration -->
  <servlet>
    <servlet-name>integrationServlet</servlet-name>
    <servlet-class>org.jbpm.bpel.integration.jms.IntegrationServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>integrationServlet</servlet-name>
    <url-pattern>/integration</url-pattern>
  </servlet-mapping>

</web-app>
```

The partner relationship manager reads a special descriptor to set itself up: `bpel-application.xml` in `WEB-INF/classes`. It specifies the name and (optional) version of the process definition being enacted.

```
<bpelApplication name="HelloWorld" xmlns="http://jbpm.org/bpel" />
```

5.2.5. Web services deployment descriptor

The `webservices.xml` deployment descriptor lists the endpoints to be deployed in a servlet container.

In the JAX-RPC specification, handlers define a means for an application to access the raw SOAP message of a request or response. Class `org.jbpm.bpel.integration.server.SoapHandler` is a handler that lets jBPM BPEL manipulate SOAP messages sent to the enclosing port component. In the web services descriptor presented subsequently, the `GreeterSoapHandler` injects BPEL functionality to the `GreeterPort` component.

```
<webservices version="1.1" xmlns="http://java.sun.com/xml/ns/j2ee">

  <webservice-description>

    <!-- descriptive name for the service -->
    <webservice-description-name>Hello World</webservice-description-name>
    <!-- WSDL service file -->
    <wsdl-file>WEB-INF/wsdl/service.wsdl</wsdl-file>
    <!-- Java<->XML mapping file -->
    <jaxrpc-mapping-file>WEB-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>

    <port-component>

      <!-- logical name for the port (unique within the module) -->
      <port-component-name>GreeterPort</port-component-name>
      <!-- WSDL port element (in service.wsdl) -->
      <wsdl-port xmlns:portNS="http://jbpm.org/examples/hello">
        portNS:GreeterPort
      </wsdl-port>
      <!-- service endpoint interface class -->
      <service-endpoint-interface>
        org.jboss.test.ws.samples.wsbpel.hello.Greeter
      </service-endpoint-interface>
      <!-- associated servlet (in web.xml) -->
      <service-impl-bean>
        <servlet-link>greeterServlet</servlet-link>
      </service-impl-bean>

      <handler>

        <!-- logical name for the handler (unique within the module) -->
        <handler-name>GreeterHandler</handler-name>
        <!-- handler class (in jbpm-bpel.jar) -->
        <handler-class>
          org.jbpm.bpel.integration.server.SoapHandler
        </handler-class>

        <init-param>
          <description>
            name of the partner link served by this port
          </description>
          <param-name>partnerLinkHandle</param-name>
          <param-value>caller</param-value>
        </init-param>

      </handler>

    </port-component>

  </webservice-description>

</webservices>
```

5.2.6. Web application deployment

To deploy the web application to the app server, call:

```
ant deploy-web
```

This will build a web archive named `hello.war` and copy it to the `deploy` directory of your JBoss AS configuration. The structure of this module follows the rules in section 5.4 of the WSEE specification [http://jcp.org/en/jsr/detail?id=921].

On the server you should see messages similar to these:

```
23:55:35,171 INFO  [TomcatDeployer] deploy, ctxPath=/hello, warUrl=.../tmp/deploy/tmp44831hello-exp.war/
23:55:35,781 INFO  [[/hello]] integrationServlet: enabled message reception for process: HelloWorld
23:55:35,796 INFO  [WSDLFilePublisher] WSDL published to: file:/.../data/wsdl/hello.war/service.wsdl
23:55:35,812 INFO  [ServiceEndpointManager] WebService started: http://.../hello/greeter
```

At this point, your BPEL process is fully accessible to external clients through its web service interfaces. The remaining sections focus on testing the process using a J2EE application client.

5.3. Build the WSEE application client

Since we already generated the Java mapping artifacts, the WSEE client programming model is the most comfortable choice for testing purposes. In this step, we will create the application client.

5.3.1. Application client deployment descriptor

J2EE application clients use the `application-client.xml` deployment descriptor.

Clients must have access to the WSDL definitions as well as the Java mapping document. We already produced these files in the Build the WSEE port components section, so we just reference them from the descriptor.

There is one caveat to the WSDL reference. Application clients assume the WSDL document describes a deployed web service. Therefore, the port elements must point to actual locations. Recall the service document generated in WSDL implementation documents leaves port locations unspecified.

The provided build script solves the problem by taking the WSDL file that JBoss AS publishes after deploying the web application to the `data/wsdl` subdirectory of the server configuration.

```
<application-client version="1.4" xmlns="http://java.sun.com/xml/ns/j2ee">

  <display-name>Hello World Service Client</display-name>

  <service-ref>

    <!-- JNDI name of service interface in client environment context -->
    <service-ref-name>service/Hello</service-ref-name>
    <!-- service interface -->
    <service-interface>
      org.jbpm.bpel.tutorial.hello.HelloWorldService
    </service-interface>
    <!-- published WSDL document -->
    <wsdl-file>META-INF/wsdl/service.wsdl</wsdl-file>
    <!-- Java<->XML mapping file -->
    <jaxrpc-mapping-file>META-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>

    <port-component-ref>
      <!-- service endpoint interface -->
      <service-endpoint-interface>
        org.jbpm.bpel.tutorial.hello.Greeter
      </service-endpoint-interface>
    </port-component-ref>

  </service-ref>

</application-client>
```

5.3.2. Environment context

In order to provide an environment context for the application client, we must allocate a name for it in the global JNDI context. This is done in `jboss-client.xml` .Be aware that this descriptor is specific to JBoss AS.

```
<jboss-client>
  <!-- JNDI name of client environment context -->
  <jndi-name>jbpmbpel-client</jndi-name>
</jboss-client>
```

5.4. Test the process

Once the process starts, we need to make sure it works as we expect. In this step, we will create a JUnit test case named `HelloTest`.

5.4.1. Remote web service access

Deploying the application client from Build the WSEE application client causes the app server to bind an instance of the service interface in the client environment context,using the logical name from the `service-ref` element. In our example, the logical name is `service/Hello`.

The test setup code looks up the service instance. This object is a factory that a client uses to get a service endpoint proxy.

```
private HelloWorldService service;

protected void setUp() throws Exception {
  InitialContext iniCtx = new InitialContext();
  /*
   * "service/Hello" is the JNDI name of the service interface instance
   * relative to the client environment context. This name matches the
   * <service-ref-name> in application-client.xml
   */
  service = (HelloWorldService) iniCtx.lookup("java:comp/env/service/Hello");
}
```

The test method uses the SEI proxy like a local java object.

```
public void testSayHello_proxy() throws Exception {
  // obtain dynamic proxy for web service port
  Greeter proxy = service.getGreeterPort();
  // use proxy as local java object
  String greeting = proxy.sayHello("Popeye");
  // check proper greeting
  assertEquals("Hello, Popeye!", greeting);
}
```

5.4.2. Client JNDI properties

The properties of the initial JNDI context are supplied in a separate `jndi.properties` file. The property `j2ee.clientName` indicates the JNDI name of the client environment context relative to the global context. The value `jbpmbpel-client` matches the `<jndi-name>` in `jboss-client.xml`.

```
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming.client
j2ee.clientName=jbpmbpel-client
```

5.4.3. Test execution

To execute the JUnit test, call:

```
ant run-test
```

If all goes well you should see the output below:

```
run-test:
    [junit] Running org.jbpm.bpel.tutorial.hello.HelloTest
    ...
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 1.532 sec
```

Other log entries will appear, including a printout of the SOAP messages exchanged with the server.

6 ATM Example

In this tutorial, we will develop a process that manages the interaction between an automated teller machine and the information system of a bank. The process drives ATMs in performing the operations listed below.

1. Connect to the server
2. Log a customer on
3. Query the state of the session
4. Obtain the acount balance
5. Withdraw and deposit funds
6. Log the customer off
7. Disconnect from the server

Not all operations are available at the same time. Most require another operation to complete for becoming available.

Four different modules participate in this orchestration. The picture below shows the relationships between modules plus the deployment configuration.

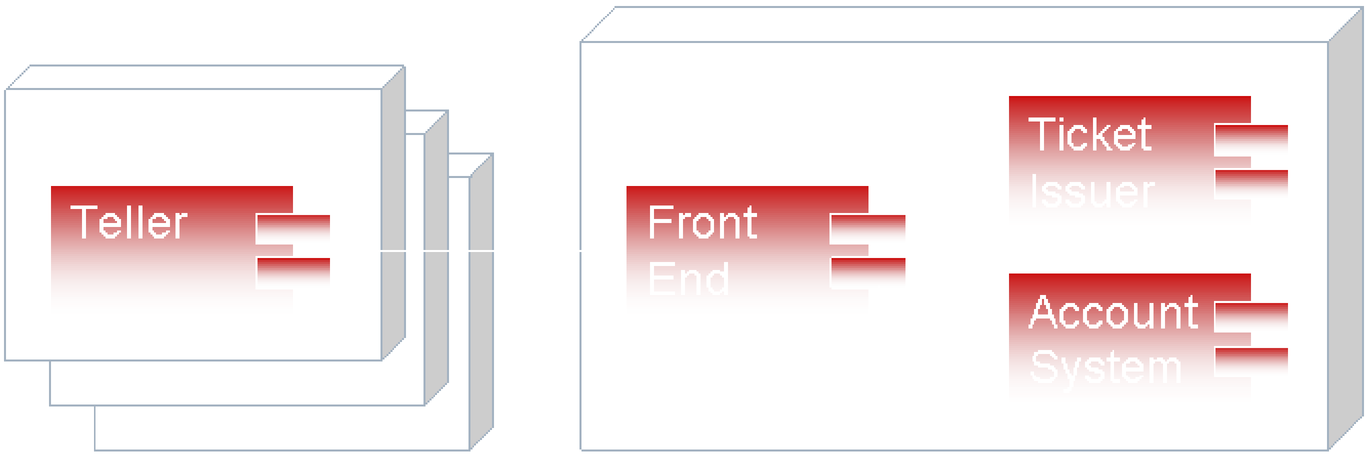


Figure 6.1. Top level graphical representation of the ATM process

On startup, the teller machine connects to the front end service. Inside the bank, the front end contacts the ticket issuer module to generate a number that uniquely identifies the teller. Subsequent exchanges with the bank indicate the ticket number.

When an account holder comes and authenticates himself/herself, the teller asks the front end to initiate a customer session. The front end resorts to the account system for checking access rights.

Once access is granted, the account holder looks at the account balance, deposits/withdraws funds or terminates the session. Because a given customer is not supposed to use multiple ATM at the same time, these exchanges carry the customer credentials instead of the ticket.

The front end contacts the account system as required to ensure the balance is accurate. Even tough the account system allows negative balances for the sake of other credit operations, ATMs do not dispense cash on credit. The front end must ensure enough funds exist and reject withdrawings that would result in a negative balance.

6.1. Define the BPEL process

6.1.1. Create the BPEL document

We will start with the process-level definitions. The `atm` partner link represents the relationship between a teller machine and the process. The process assumes the *FrontEnd* role; the ATM has no explicit role. The `ticket` definition links the process to the ticket issuer service. The ticket issuer plays the *TicketIssuer* role, while the process assumes no functions. Account system operations are available to the process through the `account` partner link. Again, no responsibility is placed on the process.

The variables `connectReq`, `ticketReq` and `ticketMsg` hold messages exchanged with partners. In turn, `connected` and `logged` are status flags. The `atm` correlation set distinguishes ATMs from each other through the ticket number property.

```
<process name="AtmFrontEnd" targetNamespace="urn:samples:atm"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="urn:samples:atm" xmlns:atm="urn:samples:atm" xmlns:typ="urn:samples:atm:types"
  xmlns:tic="urn:samples:ticket" xmlns:acc="urn:samples:account"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/ws/2003/03/business-process/
   http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <!-- relationship with the ATM -->
    <partnerLink name="atm" partnerLinkType="tns:Atm-Front" myRole="FrontEnd" />
    <!-- relationship with the ticket issuer -->
    <partnerLink name="ticket" partnerLinkType="tns:Front-Ticket" partnerRole="TicketIssuer" />
    <!-- relationship with the account system -->
    <partnerLink name="account" partnerLinkType="tns:Front-Account" partnerRole="AccountSystem" />
  </partnerLinks>

  <variables>
    <!-- ATM connection request -->
    <variable name="connectReq" messageType="atm:connectRequest" />
    <!-- ticket creation request -->
    <variable name="ticketReq" messageType="tic:ticketRequest" />
    <!-- ticket number wrapper -->
    <variable name="ticketMsg" messageType="tic:ticketMessage" />
    <!-- ATM connection flag -->
    <variable name="connected" type="xsd:boolean" />
    <!-- customer session flag -->
    <variable name="logged" type="xsd:boolean" />
  </variables>

  <correlationSets>
    <!-- conversation with a connected ATM -->
    <correlationSet name="atmInteraction" properties="tns:ticketId" />
  </correlationSets>
```

Let's move on to the control flow. The next figure is the bird eye view of the ATM front end process.

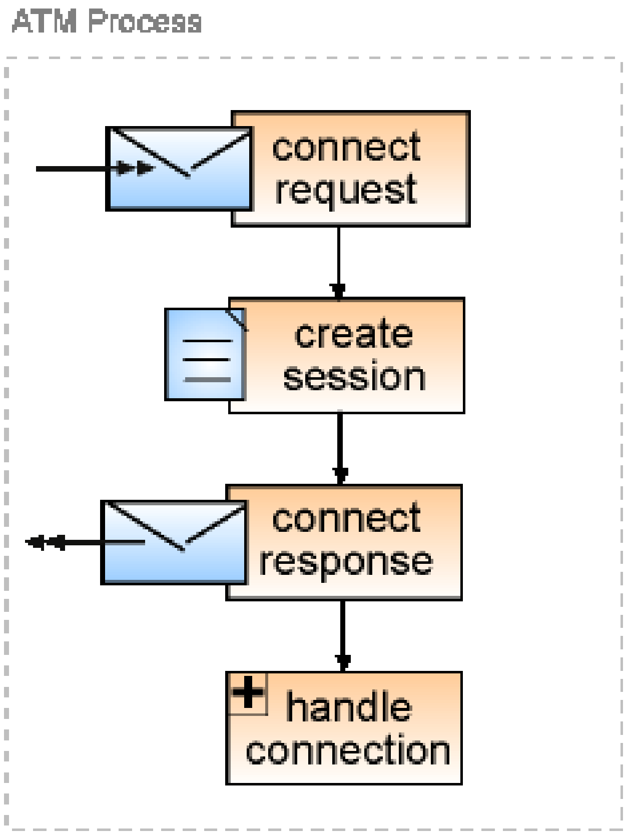


Figure 6.2. ATM main sequence

We define a main sequence for handling the lifecycle of an ATM connection. It consists of these activities: receive a connection request, invoke the ticket issuer service, initialize the status flags, send the ticket number back to the teller and handle the new connection.

```
<sequence name="mainSequence">

    <!-- receive a connection request -->
    <receive operation="connect" partnerLink="atm" portType="atm:FrontEnd" variable="connectReq"
      createInstance="yes" />

    <!-- generate a ticket number -->
    <invoke operation="createTicket" partnerLink="ticket" portType="tic:TicketIssuer"
      inputVariable="ticketReq" outputVariable="ticketMsg">
      <correlations>
        <correlation set="atmInteraction" pattern="in" initiate="yes" />
      </correlations>
    </invoke>
    <!-- initialize the status flags -->
    <assign name="initConnection">
      <copy>
        <from expression="true()" />
        <to variable="connected" />
      </copy>
      <copy>
        <from expression="false()" />
        <to variable="logged" />
      </copy>
    </assign>

    <!-- send the ticket number back to the ATM -->
    <reply operation="connect" partnerLink="atm" portType="atm:FrontEnd" variable="ticketMsg">
      <correlations>
        <correlation set="atmInteraction" />
      </correlations>
    </reply>

    <!-- handle the ATM connection -->
    <scope name="connectionUnit">
      ...
    </scope>

  </sequence>

</process>
```

The `scope` element at the end of the sequence encapsulates the connection handling logic. This activity allows specifying a nested execution flow and local definitions of variables, correlation sets and fault/event handling behavior. The following diagram shows the control flow of the `connectionUnit` :

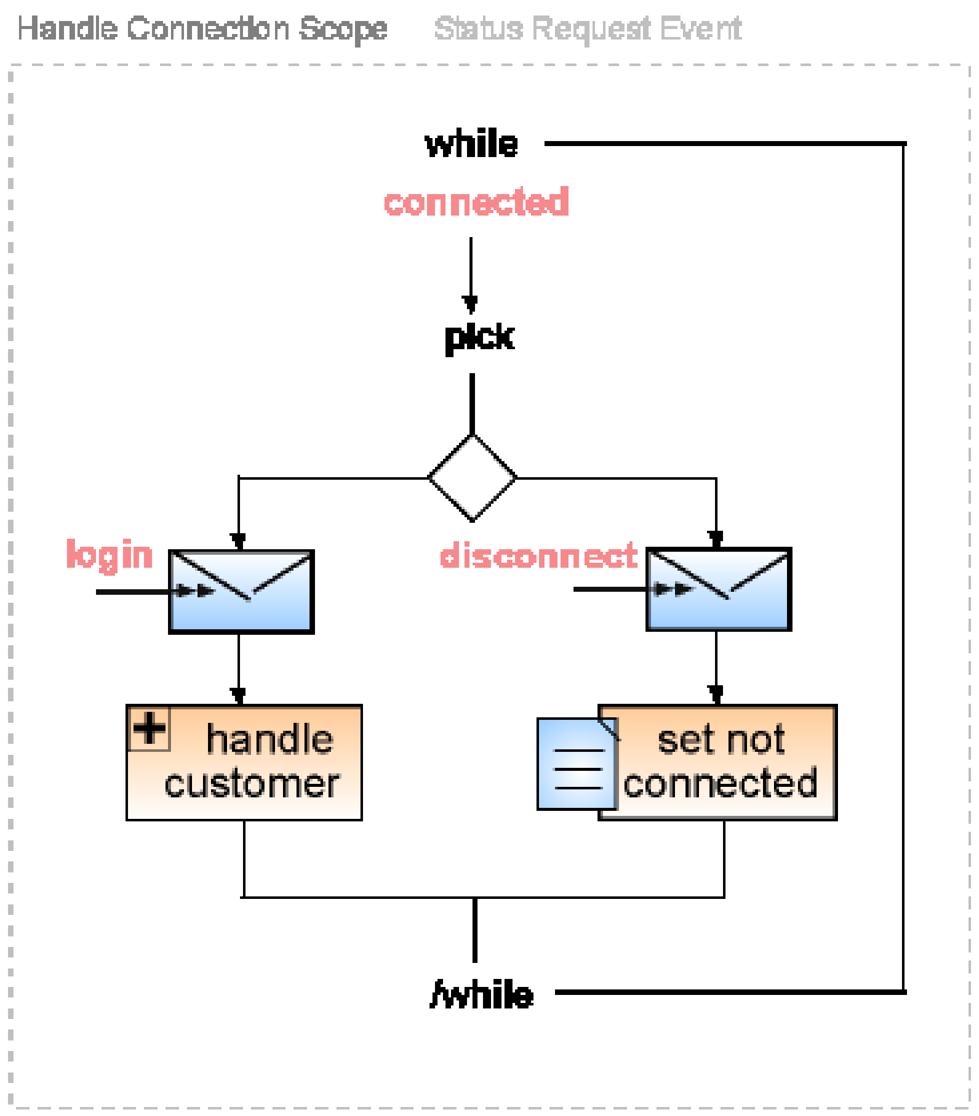


Figure 6.3. ATM connection handling logic

The local variables `logOnReq` and `statusRsp` are placeholders for message exchanges.

The connection handling logic consists of listening for requests from the ATM and processing them one at a time. This is an iterative behavior. The `connectionLoop` activity causes the front end to keep taking requests as long as the `connected` flag stays on.

At this point, the front end accepts one of two requests: initiate a customer session or terminate the connection. The `connectionMenu` performs the activity associated with the first request to arrive.

```
<scope name="connectionUnit">

  <variables>
    <!-- customer log on request -->
    <variable name="logOnReq" messageType="atm:logOnRequest" />
    <!-- connection status response -->
    <variable name="statusRsp" messageType="atm:statusResponse" />
  </variables>

  <!-- process ATM requests, one at a time -->
  <while name="connectionLoop" condition="bpel:getVariableData('connected')">

    <!-- listen for either disconnect or log on request -->
    <pick name="connectionMenu">

      <onMessage operation="disconnect" .../>

      <onMessage operation="logOn" .../>

    </pick>

  </while>

</scope>
```

- *logOn*: the `customerUnit` scope encapsulates the customer session logic.

```
<onMessage operation="logOn" partnerLink="atm" portType="atm:FrontEnd"
  variable="logOnReq">

  <correlations>
    <correlation set="atmInteraction" />
  </correlations>

  <!-- handle the customer session -->
  <scope name="customerUnit">
    ...
  </scope>

</onMessage>
```

- *disconnect*: the `setDisconnected` assign turns off the `connected` flag, causing the `connectionLoop` to break shortly afterwards.

```
<onMessage operation="disconnect" partnerLink="atm" portType="atm:FrontEnd"
  variable="ticketMsg">

  <correlations>
    <correlation set="atmInteraction" />
  </correlations>

  <!-- turn off connected flag for breaking the connection loop -->
  <assign name="setDisconnected">
    <copy>
      <from expression="false()" />
      <to variable="connected" />
    </copy>
  </assign>

</onMessage>
```

To spice up the example the scope defines an event for handling status requests on par with the primary activity. The `status` event lets the ATM query the connection status anytime, as long as the scope is active:

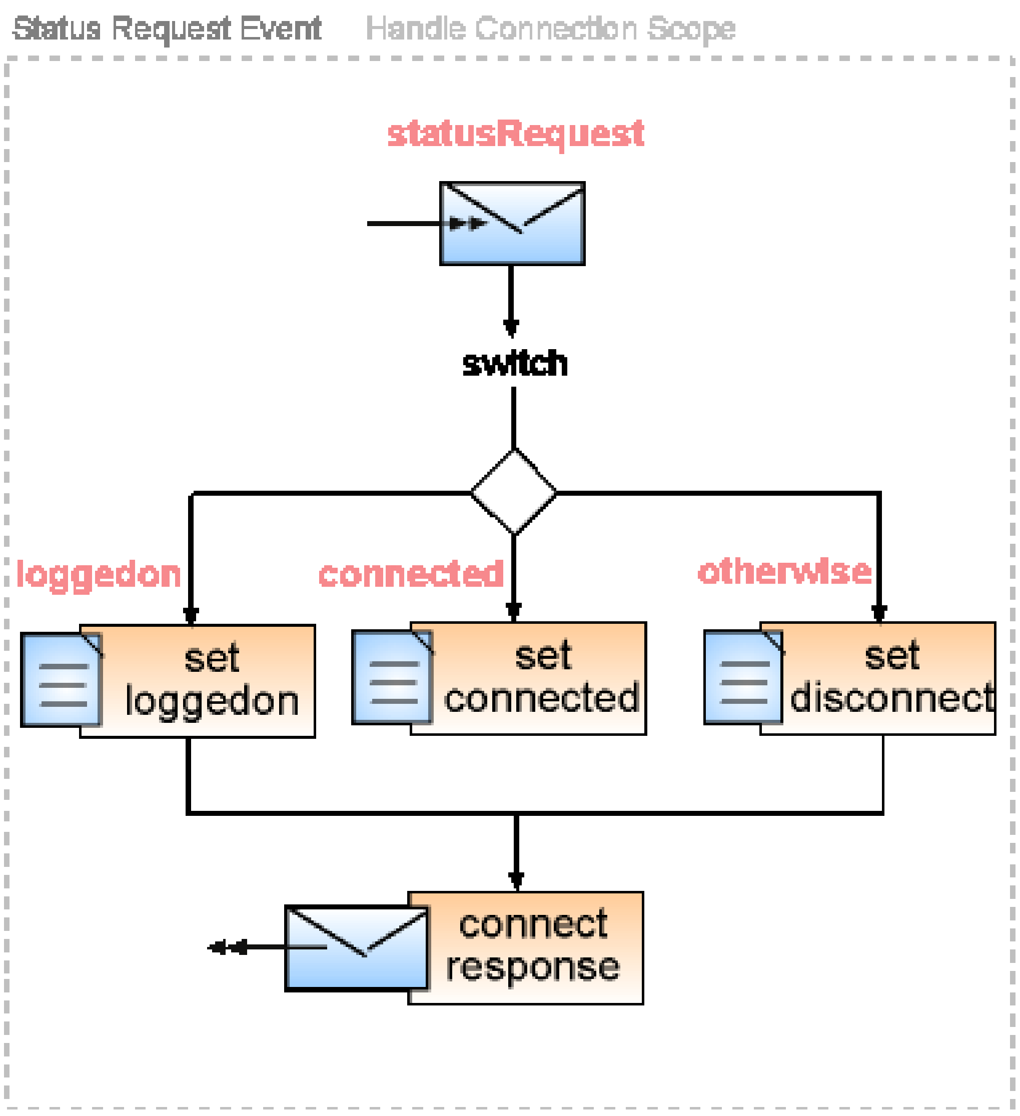


Figure 6.4. ATM connection status event

The following snippet shows the scope's event handling code:

```
<scope name="connectionUnit">
  ...
  <eventHandlers>

    <!-- listen for connection status requests -->
    <onMessage operation="status" partnerLink="atm" portType="atm:FrontEnd"
      variable="ticketMsg">

      <correlations>
        <correlation set="atmInteraction" />
      </correlations>

      <!-- report the connection status -->
      <sequence name="statusSequence">

        <!-- set a status string depending on the flag values -->
        <switch name="statusSwitch">
```

```
          <case condition="bpel:getVariableData('logged')">

            <assign name="setStatusLogged">
              <copy>
                <from expression="'logged'" />
                <to variable="statusRsp" part="status" />
              </copy>
            </assign>

          </case>

          <case condition="bpel:getVariableData('connected')">

            <assign name="setStatusConnected">
              <copy>
                <from expression="'connected'" />
                <to variable="statusRsp" part="status" />
              </copy>
            </assign>

          </case>

          <otherwise>

            <assign name="setStatusDisconnected">
              <copy>
                <from expression="'disconnected'" />
                <to variable="statusRsp" part="status" />
              </copy>
            </assign>

          </otherwise>

        </switch>
        <!-- send the status back to the ATM -->
        <reply operation="status" partnerLink="atm" portType="atm:FrontEnd"
          variable="statusRsp" />

      </sequence>

    </onMessage>

  </eventHandlers>
  ...
</scope>
```

The `customerUnit` scope lies at the core of the ATM front end process. It encapsulates the logic to serve account holder requests. The following picture summarizes its flow control:

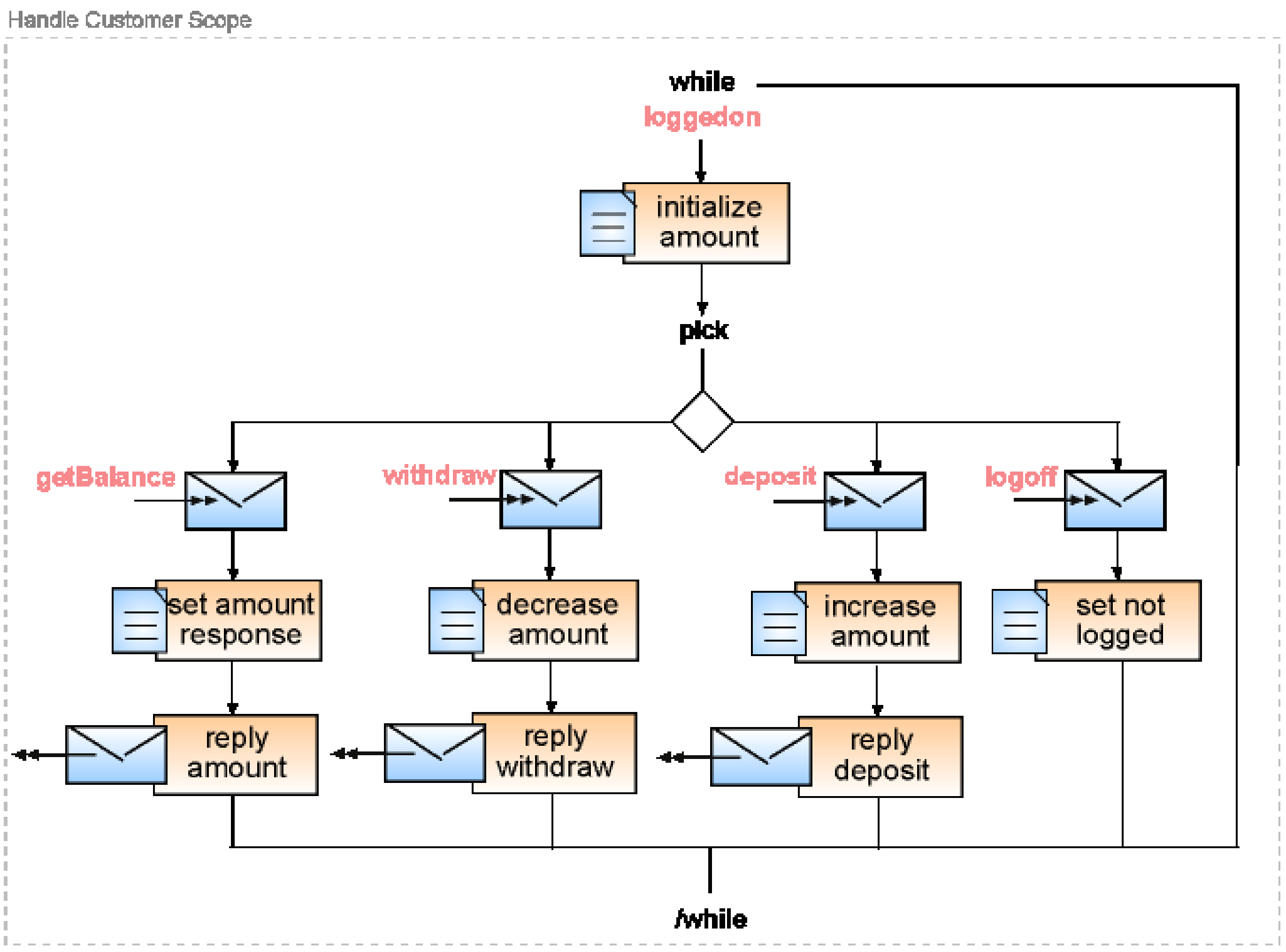


Figure 6.5. ATM customer session handling logic

The scope declares a number of local variables for incoming and outgoing messages. Apart from them, one variable of simple type, `newBalance`, stores the result of a computation that determines the remaining amount after withdrawing.

One correlation set, `customerInteraction`, distinguishes logged account holders from each other through the customer name property. One feature of correlation sets opens a potential pitfall. In order to ensure consistency constraints, correlation sets are immutable. However, the ATM most likely will serve a different customer at each iteration. For this reason, the `customerInteraction` declaration appears inside the loop rather than outside. In this way, the set can assume different values in every new session.

Customer session handling works as follows. The front end must verify the customer holds an active account. Verification is outside the responsibilities of the process; it is a function of the bank account system. Therefore, the front end invokes the account system to check the customer access privilege. If the system grants access, the front end replies the log on request with an acknowledgement and turns on the `logged` flag. Conversely, when the system denies access, the front end sends a `unauthorizedAccess` back to the ATM. It leaves the `logged` flag turned off so that the customer session ends immediately.

Note that the aforementioned fault appears in the WSDL definition of the operation. If it did not appear, jBPM

BPEL would report an error at deployment time.

```
<portType name="FrontEnd">
  ...
  <operation name="logOn">
    <input message="tns:logOnRequest" />
    <output message="tns:logOnResponse" />
    <fault name="unauthorizedAccess" message="tns:unauthorizedAccess" />
  </operation>
  ...
</portType>
```

After completing the `logOn` operation either way, the process enters a loop that processes customer requests one at a time. The next section will describe the logic inside that `customerLoop`.

```
<scope name="customerUnit">

  <variables>
    <!-- customer name wrapper -->
    <variable name="customerMsg" messageType="acc:customerMessage" />
    <!-- access check response -->
    <variable name="accessMsg" messageType="acc:accessMessage" />
    <!-- customer log on response -->
    <variable name="logOnRsp" messageType="atm:logOnResponse" />
    <!-- account balance wrapper -->
    <variable name="balanceMsg" messageType="acc:balanceMessage" />
    <!-- balance change request -->
    <variable name="balanceChange" messageType="atm:balanceChange" />
    <!-- account system operation request -->
    <variable name="accountOperation" messageType="acc:accountOperation" />
    <!-- customer log on fault -->
    <variable name="unauthorizedAccess" messageType="atm:unauthorizedAccess" />
    <!-- withdraw fault -->
    <variable name="insufficientFunds" messageType="atm:insufficientFunds" />
    <!-- resulting balance after withdraw -->
    <variable name="newBalance" type="xsd:double" />
  </variables>

  <correlationSets>
    <!-- conversation with a logged customer -->
    <correlationSet name="customerInteraction" properties="tns:customerId" />
  </correlationSets>

  <sequence name="customerSequence">

    <!-- populate access check request -->
    <assign name="fillAccessCheck">
      <copy>
        <from variable="logOnReq" part="customerName" />
        <to variable="customerMsg" part="customerName" />
      </copy>
    </assign>

    <!-- check account access privilege -->
    <invoke operation="checkAccess" partnerLink="account" portType="acc:AccountSystem"
      inputVariable="customerMsg" outputVariable="accessMsg">
      <correlations>
        <correlation set="customerInteraction" pattern="out" initiate="yes" />
      </correlations>
    </invoke>

    <!-- decide outcome of customer session request -->
    <switch name="accessDecision">
```

```
      <case condition="bpel:getVariableData('accessMsg', 'granted')">

        <!-- accept customer session -->
        <sequence name="accessGrantedSequence">

          <!-- turn on logged flag for starting session loop  -->
          <assign name="setLoggedOn">
            <copy>
              <from expression="true()" />
              <to variable="logged" />
            </copy>
          </assign>

          <!-- send acknowledgement back to ATM -->
          <reply operation="logOn" partnerLink="atm" portType="atm:FrontEnd"
            variable="logOnRsp" />

        </sequence>

      </case>

      <otherwise>

        <!-- reject customer session -->
        <sequence name="accessDeniedSequence">

          <!-- populate the log on fault -->
          <assign name="fillAccessDenial">
            <copy>
              <from variable="logOnReq" part="customerName" />
              <to variable="unauthorizedAccess" part="detail"
                query="/typ:unauthorizedAccess/customerName" />
            </copy>
          </assign>

          <!-- send fault back to the ATM -->
          <reply operation="logOn" partnerLink="atm" portType="atm:FrontEnd"
            variable="unauthorizedAccess" faultName="atm:unauthorizedAccess" />

        </sequence>

      </otherwise>

    </switch>

    <!-- process customer requests, one at a time -->
    <while name="customerLoop" condition="bpel:getVariableData('logged')">
      ...
    </while>

  </sequence>

</scope>
```

Inside `customerLoop`, the process waits for one of four possible requests. These requests appear as `onMessage` child elements of the `customerMenu` activity.

```
<while name="customerLoop" condition="bpel:getVariableData('logged')">

  <pick name="customerMenu">

    <onMessage operation="logOff" .../>

    <onMessage operation="getBalance" .../>

    <onMessage operation="deposit" .../>

    <onMessage operation="withdraw" .../>

    <onAlarm for="'PT2M'" .../>

  </pick>

</while>
```

- *logOff*: the `setLoggedOff` assign turns off the `logged` flag to break the `customerLoop` and terminate the customer session.

```
<onMessage operation="logOff" partnerLink="atm" portType="atm:FrontEnd"
  variable="customerMsg">

  <correlations>
    <correlation set="customerInteraction" />
  </correlations>

  <!-- turn off logged flag for breaking the customer loop -->
  <assign name="setLoggedOff">
    <copy>
      <from expression="false()" />
      <to variable="logged" />
    </copy>
  </assign>

</onMessage>
```

- *getBalance*: the `balanceSequence` queries the account system for the current balance and hands that back to the ATM.

```
<onMessage operation="getBalance" partnerLink="atm" portType="atm:FrontEnd"
  variable="customerMsg">

  <correlations>
    <correlation set="customerInteraction" />
  </correlations>

  <sequence name="balanceSequence">

    <!-- get current account balance -->
    <invoke operation="queryBalance" partnerLink="account"
      portType="acc:AccountSystem" inputVariable="customerMsg"
      outputVariable="balanceMsg">
      <correlations>
        <correlation set="customerInteraction" pattern="out" />
      </correlations>
    </invoke>

    <!-- hand the balance back to the ATM -->
    <reply operation="getBalance" partnerLink="atm" portType="atm:FrontEnd"
      variable="balanceMsg" />

  </sequence>

</onMessage>
```

- *deposit*: the `depositSequence` posts the positive update to the account system. The front end process gets the new balance in return and makes it available to the ATM.

```
<onMessage operation="deposit" partnerLink="atm" portType="atm:FrontEnd"
  variable="balanceChange">

  <correlations>
    <correlation set="customerInteraction" />
  </correlations>

  <sequence name="depositSequence">

    <!-- populate balance update request -->
    <assign name="fillDepositUpdate">
      <copy>
        <from variable="balanceChange" part="customerName" />
        <to variable="accountOperation" part="body" query="/body/customerName" />
      </copy>
      <copy>
        <from variable="balanceChange" part="amount" />
        <to variable="accountOperation" part="body" query="/body/amount" />
      </copy>
    </assign>

    <!-- post positive balance update -->
    <invoke operation="updateBalance" partnerLink="account"
      portType="acc:AccountSystem" inputVariable="accountOperation"
      outputVariable="balanceMsg">
      <correlations>
        <correlation set="customerInteraction" pattern="out" />
      </correlations>
    </invoke>

    <!-- make new balance available to ATM -->
    <reply operation="deposit" partnerLink="atm" portType="atm:FrontEnd"
      variable="balanceMsg" />

  </sequence>

</onMessage>
```

- *withdraw*: the `withdrawSequence` first queries the account system for the current balance. Later, it computes the amount that would remain in the account after the negative update.

```
<onMessage operation="withdraw" partnerLink="atm" portType="atm:FrontEnd"
  variable="balanceChange">

  <correlations>
    <correlation set="customerInteraction" />
  </correlations>

  <sequence name="withdrawSequence">

    <!-- populate balance query request -->
    <assign name="fillWithdrawQuery">
      <copy>
        <from variable="balanceChange" part="customerName" />
        <to variable="customerMsg" part="customerName" />
      </copy>
    </assign>

    <!-- get current account balance -->
    <invoke operation="queryBalance" partnerLink="account"
      portType="acc:AccountSystem" inputVariable="customerMsg"
      outputVariable="balanceMsg">
      <correlations>
        <correlation set="customerInteraction" pattern="out" />
      </correlations>
    </invoke>

    <!-- compute amount that would remain in the account -->
    <assign name="decreaseBalance">
      <copy>
        <from
          expression="bpel:getVariableData('balanceMsg', 'balance') -
            bpel:getVariableData('balanceChange', 'amount')" />
        <to variable="newBalance" />
      </copy>
    </assign>

    <!-- decide outcome of withdraw request -->
    <switch name="balanceDecision">

      <case condition="bpel:getVariableData('newBalance') &gt;= 0.0" .../>

      <otherwise .../>

    </switch>

  </sequence>

</onMessage>
```

- If there are enough funds, the `positiveBalanceSequence` posts the negative update to the account system, gets the new balance and returns that to the ATM.

```
<case condition="bpel:getVariableData('newBalance') &gt;= 0.0">

  <!-- accept withdrawing -->
  <sequence name="positiveBalanceSequence">

    <!-- populate balance update request -->
    <assign name="fillWithdrawUpdate">
      <copy>
        <from variable="balanceChange" part="customerName" />
        <to variable="accountOperation" part="body"
          query="/body/customerName" />
      </copy>
      <copy>
        <from
          expression="-bpel:getVariableData('balanceChange', 'amount')" />
        <to variable="accountOperation" part="body" query="/body/amount" />
      </copy>
    </assign>

    <!-- post negative balance update -->
    <invoke operation="updateBalance" partnerLink="account"
      portType="acc:AccountSystem" inputVariable="accountOperation"
      outputVariable="balanceMsg">
      <correlations>
        <correlation set="customerInteraction" pattern="out" />
      </correlations>
    </invoke>

    <!-- return new balance to ATM -->
    <reply operation="withdraw" partnerLink="atm" portType="atm:FrontEnd"
      variable="balanceMsg" />

  </sequence>
```

```
</case>
```

- Otherwise, the `negativeBalanceSequence` rejects the withdraw by returning a fault to the ATM. The update is not posted.

```
<otherwise>

  <!-- reject withdrawing -->
  <sequence name="negativeBalanceSequence">

    <!-- populate withdraw fault -->
    <assign name="fillNoFunds">
      <copy>
        <from variable="balanceChange" part="customerName" />
        <to variable="insufficientFunds" part="detail"
          query="/typ:insufficientFunds/customerName" />
      </copy>
      <copy>
        <from variable="balanceMsg" part="balance" />
        <to variable="insufficientFunds" part="detail"
          query="/typ:insufficientFunds/amount" />
      </copy>
    </assign>

    <!-- return fault to ATM -->
    <reply operation="withdraw" partnerLink="atm" portType="atm:FrontEnd"
      variable="insufficientFunds" faultName="atm:insufficientFunds" />

  </sequence>

</otherwise>
```

The final `onAlarm` subelement terminates the customer session after two minutes if none of the above requests arrives within that interval.

```
<onAlarm for="'PT2M'">

  <!-- log off after 2 minutes of inactivity -->
  <assign name="setLoggedOff">
    <copy>
      <from expression="false()" />
      <to variable="logged" />
    </copy>
  </assign>

</onAlarm>
```

6.1.2. Create/obtain the WSDL interface documents

To better organize WSDL definitions, the process uses four interface documents for the ATM service.

The first document, `ticket.wsdl` contains the interface of the ticket issuer service. Supposedly, someone has already deployed this service somewhere and the WSDL definitions came from there.

```
<definitions targetNamespace="urn:samples:ticket" xmlns:tns="urn:samples:ticket"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <!-- ticket creation request -->
  <message name="ticketRequest" />

  <!-- ticket number wrapper -->
  <message name="ticketMessage">
    <part name="ticketNo" type="xsd:int" />
  </message>

  <!-- interface to ticket issuer service -->
  <portType name="TicketIssuer">

    <!-- generate a ticket number, distinct from previous calls -->
    <operation name="createTicket">
      <input message="tns:ticketRequest" />
      <output message="tns:ticketMessage" />
    </operation>

  </portType>

</definitions>
```

Another document, `account.wsdl` describes the published functions of the account system. One custom XML Schema definition, `AccountOperation`, introduces a data transfer type for account operations.

```
<definitions targetNamespace="urn:samples:account" xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="urn:samples:account" xmlns:typ="urn:samples:account"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>

    <schema targetNamespace="urn:samples:account" xmlns="http://www.w3.org/2001/XMLSchema">

      <!-- account data transfer type -->
      <complexType name="AccountOperation">
        <sequence>
          <element name="customerName" type="xsd:string" />
          <element name="amount" type="xsd:double" />
        </sequence>
      </complexType>

    </schema>

  </types>

  <!-- customer name wrapper -->
  <message name="customerMessage">
    <part name="customerName" type="xsd:string" />
  </message>

  <!-- access check response -->
  <message name="accessMessage">
    <part name="granted" type="xsd:boolean" />
  </message>

  <!-- account balance wrapper -->
  <message name="balanceMessage">
    <part name="balance" type="xsd:double" />
  </message>

  <!-- account operation request -->
  <message name="accountOperation">
    <part name="body" type="typ:AccountOperation" />
  </message>

  <!-- published account functions -->
  <portType name="AccountSystem">

    <!-- tell whether a customer has an active account -->
    <operation name="checkAccess">
      <input message="tns:customerMessage" />
      <output message="tns:accessMessage" />
    </operation>

    <!-- retrieve the balance of an account -->
    <operation name="queryBalance">
      <input message="tns:customerMessage" />
      <output message="tns:balanceMessage" />
    </operation>

    <!-- increase/decrease the balance of an account -->
    <operation name="updateBalance">
      <input message="tns:accountOperation" />
      <output message="tns:balanceMessage" />
    </operation>

  </portType>

</definitions>
```

The third document, `frontend.wsdl`, contains the interface the process presents to ATMs. Because it reuses a number of messages from the ticket issuer and the account system, it imports the WSDL documents that describe these services.

Some custom XML schema definitions appear in the `types` section. They define the elements that the front end interface uses to inform ATMs of business logic errors and the types that characterize those elements.

WSDL messages, in terms of the foregoing definitions and predefined schema types, define the exchange format between the ATM and the bank front end. Finally, the `FrontEnd` port type lists the bank functions available to ATMs.

```
<definitions targetNamespace="urn:samples:atm" xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="urn:samples:atm" xmlns:typ="urn:samples:atm" xmlns:tic="urn:samples:ticket"
  xmlns:acc="urn:samples:account" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <import namespace="urn:samples:ticket" location="ticket.wsdl" />
  <import namespace="urn:samples:account" location="account.wsdl" />

  <types>

    <schema targetNamespace="urn:samples:atm" xmlns="http://www.w3.org/2001/XMLSchema">

      <complexType name="UnauthorizedAccess">
        <sequence>
          <element name="customerName" type="xsd:string" />
        </sequence>
      </complexType>

      <element name="unauthorizedAccess" type="typ:UnauthorizedAccess" />

      <complexType name="InsufficientFunds">
        <sequence>
          <element name="customerName" type="xsd:string" />
          <element name="amount" type="xsd:double" />
        </sequence>
      </complexType>

      <element name="insufficientFunds" type="typ:InsufficientFunds" />
    </schema>

  </types>

  <message name="connectRequest" />

  <message name="logOnRequest">
    <part name="ticketNo" type="xsd:int" />
    <part name="customerName" type="xsd:string" />
  </message>

  <message name="logOnResponse" />

  <message name="statusResponse">
    <part name="status" type="xsd:string" />
  </message>

  <message name="balanceChange">
    <part name="customerName" type="xsd:string" />
    <part name="amount" type="xsd:double" />
  </message>

  <message name="unauthorizedAccess">
    <part name="detail" element="typ:unauthorizedAccess" />
  </message>

  <message name="insufficientFunds">
    <part name="detail" element="typ:insufficientFunds" />
  </message>

  <!-- bank functions available to ATMs -->
  <portType name="FrontEnd">

    <!-- initiate bank connection -->
    <operation name="connect">
      <input message="tns:connectRequest" />
      <output message="tic:ticketMessage" />
    </operation>

    <!-- terminate bank connection -->
    <operation name="disconnect">
      <input message="tic:ticketMessage" />
    </operation>

    <!-- retrieve bank connection status -->
    <operation name="status">
      <input message="tic:ticketMessage" />
      <output message="tns:statusResponse" />
    </operation>

    <!-- initiate customer session -->
    <operation name="logOn">
      <input message="tns:logOnRequest" />
      <output message="tns:logOnResponse" />
      <fault name="unauthorizedAccess" message="tns:unauthorizedAccess" />
    </operation>

    <!-- terminate customer session -->
    <operation name="logOff">
      <input message="acc:customerMessage" />
    </operation>

    <!-- retrieve account balance -->
    <operation name="getBalance">
      <input message="acc:customerMessage" />
      <output message="acc:balanceMessage" />
    </operation>

    <!-- increase account balance -->
    <operation name="deposit">
      <input message="tns:balanceChange" />
      <output message="acc:balanceMessage" />
    </operation>

    <!-- decrease account balance -->
    <operation name="withdraw">
      <input message="tns:balanceChange" />
      <output message="acc:balanceMessage" />
      <fault name="insufficientFunds" message="tns:insufficientFunds" />
    </operation>

  </portType>

</definitions>
```

The last document, `atm.wsdl`, contains extensibility elements that glue together the BPEL process and the WSDL definitions. At the beginning, the document imports the previous three documents to reference their definitions. Later, it defines some properties for correlation purposes. `ticketId` distinguishes ticket numbers in messages exchanged within an ATM connection, while `customerId` represents customer names in messages exchanged during a customer session. The property aliases adjacent to these property definitions map these properties to key information items inside messages.

Partner link types characterize the relationship between ATMs and the process (`Atm-Front`), the process and the ticket issuer (`Front-Ticket`) as well as the process and the account system (`Front-Account`). They define the roles these services play and specify the interface they present to each other. The coordinator does not call back the ATM. The ticket issuer or the account system do not call back the coordinator either. Therefore, all partner link types have a single role.

```
<definitions targetNamespace="urn:samples:atm"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:samples:atm"
  xmlns:atm="urn:samples:atm" xmlns:tic="urn:samples:ticket"
  xmlns:acc="urn:samples:account"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:plt="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <import namespace="urn:samples:atm" location="interface/frontend.wsdl" />
  <import namespace="urn:samples:ticket" location="interface/ticket.wsdl" />
  <import namespace="urn:samples:account" location="interface/account.wsdl" />

  <!-- customer name property -->
  <bpel:property name="customerId" type="xsd:string" />

  <!-- location of costumerId inside messages -->
  <bpel:propertyAlias propertyName="tns:customerId"
    messageType="atm:logOnRequest" part="customerName" />
  <bpel:propertyAlias propertyName="tns:customerId"
    messageType="atm:balanceChange" part="customerName" />
  <bpel:propertyAlias propertyName="tns:customerId"
    messageType="acc:customerMessage" part="customerName" />
  <bpel:propertyAlias propertyName="tns:customerId"
    messageType="acc:accountOperation" part="body" query="/body/customerName" />

  <!-- ticket number property -->
  <bpel:property name="ticketId" type="xsd:int" />
  <!-- location of ticketId inside messages -->
  <bpel:propertyAlias propertyName="tns:ticketId"
    messageType="tic:ticketMessage" part="ticketNo" />
  <bpel:propertyAlias propertyName="tns:ticketId" messageType="atm:logOnRequest"
    part="ticketNo" />

  <!-- relationship between the ATM and the process -->
  <plt:partnerLinkType name="Atm-Front">
    <plt:role name="FrontEnd">
      <plt:portType name="atm:FrontEnd" />
    </plt:role>
  </plt:partnerLinkType>

  <!-- relationship between the process and the ticket issuer -->
  <plt:partnerLinkType name="Front-Ticket">
    <plt:role name="TicketIssuer">
      <plt:portType name="tic:TicketIssuer" />
    </plt:role>
  </plt:partnerLinkType>

  <!-- relationship between the process and the account system -->
  <plt:partnerLinkType name="Front-Account">
    <plt:role name="AccountSystem">
      <plt:portType name="acc:AccountSystem" />
    </plt:role>
  </plt:partnerLinkType>

</definitions>
```

6.1.3. Deploy the process definition

The `bpel-definition.xml` file points to the `atm.bpel` and `atm.wsdl` documents from the previous two sections. It is unnecessary to reference other WSDL documents, because `atm.wsdl` imports them.

```
<bpelDefinition location="atm.bpel" xmlns="http://jbpm.org/bpel">

  <!-- makes WSDL interface elements available to the process -->
  <imports>
    <wsdl location="atm.wsdl" />
  </imports>

</bpelDefinition>
```

To deploy the process definition to the jBPM database, call:

```
ant deploy-definition
```

The above target creates a file named `atm-process.zip` and submits it to the jBPM service. The server console should read:

```
14:30:22,437 INFO  [[/jbpm-bpel]] processDeployServlet: deploying process definition: file=file:/C:/.../a
14:30:22,968 INFO  [BpelReader] read wsdl definitions: atm.wsdl
14:30:23,281 INFO  [BpelReader] read bpel process: atm.bpel
14:30:29,984 INFO  [[/jbpm-bpel]] processDeployServlet: deployed process definition: AtmFrontEnd
```

6.2. Build the WSEE port components

6.2.1. WSDL implementation documents

Generate the WSDL implementation definitions with:

```
ant generate-service
```

The `servicegen` tool writes the following documents:

- `atm.wsdl`, `frontend.wsdl`, `ticket.wsdl` and `account.wsdl` are equivalents of the interface WSDL documents we saw in Create/obtain the WSDL interface documents
- `binding1.wsdl` contains the SOAP binding for the `FrontEnd` port type
- `service.wsdl` has the service that corresponds to the process. The `AtmFrontEndService` service contains a `FrontEndPort` associated to the `atm` partner link.

6.2.2. Java mapping artifacts

Generate the Java mapping artifacts with this command:

```
ant generate-artifacts
```

The configuration for `wstools` is:

```
<configuration xmlns="http://www.jboss.org/jbossws-tools">
  <global>
    <package-namespace package="org.jbpm.bpel.tutorial.atm"
      namespace="urn:samples:atm" />
  </global>
  <wsdl-java file="wsdl/service.wsdl">
    <mapping file="jaxrpc-mapping.xml" />
  </wsdl-java>
</configuration>
```

For `wscompile` we have:

```
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <wsdl location="output/resources/WEB-INF/wsdl/service.wsdl"
    packageName="org.jbpm.bpel.tutorial.atm" />
</configuration>
```

6.2.3. Port components as servlets

The subsequent listing presents the `web.xml` descriptor used for this example.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">

  <!-- ATM Front End -->
  <servlet>
    <servlet-name>frontEndServlet</servlet-name>
    <servlet-class>org.jbpm.bpel.tutorial.atm.FrontEnd_Impl</servlet-class>
  </servlet>
  <servlet-mapping>
```

```
    <servlet-name>frontEndServlet</servlet-name>
    <url-pattern>/frontEnd</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

6.2.4. Partner integration

In order for the process to take requests, add the partner integration servlet to your `web.xml`. Refer to the Partner integration section in the Hello World example for a full explanation.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  ...
  <!-- jBPM BPEL Partner Integration -->
  <servlet>
    <servlet-name>integrationServlet</servlet-name>
    <servlet-class>org.jbpm.bpel.integration.jms.IntegrationServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>integrationServlet</servlet-name>
    <url-pattern>/integration</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

6.2.5. Activity scheduler

Processes that include time-driven activities (`wait`, `onAlarm` branch of `pick` and `onAlarm` event) require a scheduler component. The scheduler examines the list of pending activities and executes them when they are due.

jBPM provides such a component in the form of a servlet. An extra `servlet` element in `web.xml` sets up the scheduler servlet. Similarly to the partner integration servlet, the scheduler servlet must be loaded at startup, so that activities that became due during downtime execute immediately.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee">
  ...
  <!-- jBPM Scheduler -->
  <servlet>
    <servlet-name>schedulerServlet</servlet-name>
    <servlet-class>org.jbpm.scheduler.impl.SchedulerServlet</servlet-class>
    <init-param>
      <param-name>interval</param-name>
      <param-value>120000</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>schedulerServlet</servlet-name>
    <url-pattern>/scheduler</url-pattern>
  </servlet-mapping>

</web-app>
```

Tip

The `interval` parameter is optional. It defines how often the scheduler checks for new pending activities. In general, you should set this value to the shortest duration in your process.

In our example, the only duration is the customer session expiration time: 2 minutes. This duration matches the above value: 120,000 milliseconds.

6.2.6. Web services deployment descriptor

The web services descriptor, `webservices.xml`, appears next. Notice the `FrontEndHandler`, which injects BPEL functionality to the `FrontEndPort` component.

```
<webservices version="1.1" xmlns="http://java.sun.com/xml/ns/j2ee">

  <webservice-description>

    <!-- descriptive name for the service -->
    <webservice-description-name>ATM Front End</webservice-description-name>
    <!-- WSDL implementation file -->
    <wsdl-file>WEB-INF/wsdl/service.wsdl</wsdl-file>
    <!-- Java<->XML mapping file -->
    <jaxrpc-mapping-file>WEB-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>

    <port-component>

      <!-- logical name for the port (unique within the module) -->
      <port-component-name>FrontEndPort</port-component-name>
      <!-- WSDL port element (in service.wsdl) -->
      <wsdl-port xmlns:portNS="urn:samples:atm">portNS:FrontEndPort</wsdl-port>
      <!-- service endpoint interface class -->
      <service-endpoint-interface>
        org.jbpm.bpel.tutorial.atm.FrontEnd
      </service-endpoint-interface>
      <!-- associated servlet (in web.xml) -->
      <service-impl-bean>
        <servlet-link>frontEndServlet</servlet-link>
      </service-impl-bean>

      <handler>

        <!-- logical name for the handler (unique within the module) -->
        <handler-name>FrontEndHandler</handler-name>
        <!-- handler class (in jbpm-bpel.jar) -->
        <handler-class>
          org.jbpm.bpel.integration.server.SoapHandler
        </handler-class>

        <init-param>
          <description>
            name of the partner link served by this port
          </description>
          <param-name>partnerLinkHandle</param-name>
          <param-value>atm</param-value>
        </init-param>
        <init-param>
          <description>time to wait for response messages (millis)</description>
          <param-name>responseTimeout</param-name>
          <param-value>5000</param-value>
        </init-param>
        <init-param>
          <description>time to expire one-way messages (millis)</description>
          <param-name>oneWayTimeout</param-name>
          <param-value>60000</param-value>
        </init-param>

      </handler>

    </port-component>

  </webservice-description>

</webservices>
```

Tip

The `responseTimeout` and `oneWayTimeout` parameters are optional.

When `responseTimeout` is present, clients of your process that invoke a request/response operation will get a SOAP fault message back when the process is unable to reply within the specified interval. The `fault-code` of said message is `soapenv:Server`.

The `oneWayTimeout` parameter enables jBPM BPEL to get rid of unattended one-way messages. This is useful in processes having picks or message events involving one-way operations. In cases where the expected message arrives too late, the server has no other way to detect it will never be received.

6.2.7. Web application deployment

To deploy the web application to the app server, call:

```
ant deploy-web
```

This builds a web archive named `atm.war` and copies it to the `deploy` directory of your JBoss AS configuration. On the server, you should see logs like:

```
17:01:48,281 INFO  [TomcatDeployer] deploy, ctxPath=/atm, warUrl=.../tmp/deploy/tmp60912atm-exp.war/
17:01:49,062 INFO  [[/atm]] integrationServlet: enabled message reception for process: AtmFrontEnd
17:01:49,156 INFO  [WSDLFilePublisher] WSDL published to: file:/.../data/wsdl/atm.war/service.wsdl
17:01:49,265 INFO  [ServiceEndpointManager] WebService started: http://.../atm/frontEnd
```

Back in Create/obtain the WSDL interface documents the description assumed the partner services were available somewhere. At this point, they must be actually up and running. The resources of each partner service reside in separate directories under `doc/examples`.

To deploy the ticket issuer, change to the `ticket` dir and call:

```
ant redeploy
```

This will generate the Java mapping artifacts, build the `ticket.war` module and deploy it to JBoss AS. The console output is:

```
17:35:56,937 INFO  [TomcatDeployer] deploy, ctxPath=/ticket, warUrl=.../tmp/deploy/tmp60917ticket-exp.war
17:35:57,093 INFO  [WSDLFilePublisher] WSDL published to: file:/.../data/wsdl/ticket.war/ticket-impl.wsdl
17:35:57,093 INFO  [ServiceEndpointManager] WebService started: http://.../ticket/ticketIssuer
```

Call the same target in the `account` directory to deploy the accout system. The server emits these logs:

```
17:43:46,828 INFO  [TomcatDeployer] deploy, ctxPath=/account, warUrl=.../tmp/deploy/tmp60918account-exp.w
17:43:46,953 INFO  [WSDLFilePublisher] WSDL published to: file:/.../data/wsdl/account.war/account-impl.ws
17:43:46,984 INFO  [ServiceEndpointManager] WebService started: http://.../account/accountSystem
```

6.3. Build the WSEE application client

6.3.1. Application client deployment descriptor

Reference your WSDL definitions and Java mapping artifacts from the `application-client.xml` descriptor.

```
<application-client version="1.4" xmlns="http://java.sun.com/xml/ns/j2ee">

  <display-name>ATM Front End Client</display-name>

  <service-ref>

    <!-- JNDI name of service interface in client environment context -->
    <service-ref-name>service/ATM</service-ref-name>
    <!-- service interface class -->
    <service-interface>org.jbpm.bpel.tutorial.atm.AtmFrontEndService</service-interface>
    <!-- published WSDL document -->
    <wsdl-file>META-INF/wsdl/service.wsdl</wsdl-file>
    <!-- Java<->XML mapping file -->
    <jaxrpc-mapping-file>META-INF/jaxrpc-mapping.xml</jaxrpc-mapping-file>

    <port-component-ref>
      <!-- service endpoint interface class -->
      <service-endpoint-interface>org.jbpm.bpel.tutorial.atm.FrontEnd</service-endpoint-interface>
    </port-component-ref>

  </service-ref>

</application-client>
```

6.3.2. Environment context

Allocate a JNDI name for the client environment context in `jboss-client.xml` .

```
<jboss-client>
  <!-- JNDI name of client environment context -->
  <jndi-name>jbpmbpel-client</jndi-name>
</jboss-client>
```

Tip

The name above is the same we used in the JBoss client descriptor of the Hello World example.

You can share one JNDI name among multiple application clients to keep them organized and reduce the number of top-level entries in the global JNDI context. Just be careful to give different `service-ref-names` to each client in `application-client.xml`.

6.4. Test the process

Once our process is up and running, we need to make sure that it is working as expected. Here we create a JUnit test case called `AtmFrontEndTest` and exercise several scenarios.

6.4.1. Remote web service access

This is the setup code for establishing a connection with the ATM front end:

```
private FrontEnd frontEnd;

protected void setUp() throws Exception {
  InitialContext iniCtx = new InitialContext();
  /*
   * "service/ATM" is the JNDI name of the service interface instance
   * relative to the client environment context. This name matches the
   * <service-ref-name> in application-client.xml
   */
  AtmFrontEndService frontEndService =
    (AtmFrontEndService) iniCtx.lookup("java:comp/env/service/ATM");

  // obtain dynamic proxy for web service port
  frontEnd = frontEndService.getFrontEndPort();
}
```

The test scenarios are described next.

1. `testConnect`: establish a connection to the bank.

```
public void testConnect() throws RemoteException {
  // connect to bank
  int ticketNumber = frontEnd.connect();
  assertTrue(ticketNumber > 0);

  // check atm is connected
  String status = frontEnd.status(ticketNumber);
  assertEquals("connected", status);

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

2. `testLogOnAuthorized`: initiate a session as an authorized customer.

```
public void testLogOnAuthorized() throws RemoteException {
  // connect to bank
  int ticketNumber = frontEnd.connect();

  // begin customer session
  final String customerName = "grover";
  try {
    frontEnd.logOn(ticketNumber, customerName);
  }
  catch (UnauthorizedAccess e) {
    fail("log on of authorized customer should succeed");
  }

  // end customer session
  frontEnd.logOff(customerName);

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

3. `testLogOnUnauthorized`: initiate a session as an unauthorized customer.

```
public void testLogOnUnauthorized() throws RemoteException {
  // connect to bank
  int ticketNumber = frontEnd.connect();

  // begin customer session
  final String customerName = "misterx";
  try {
    frontEnd.logOn(ticketNumber, customerName);
    fail("log on of unauthorized customer should fail");
  }
  catch (UnauthorizedAccess e) {
    assertEquals(customerName, e.getCustomerName());
  }

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

4. `testDeposit`: deposit funds

```
public void testDeposit() throws RemoteException, UnauthorizedAccess {
  // connect to bank
  int ticketNumber = frontEnd.connect();

  // begin customer session
  final String customerName = "ernie";
  frontEnd.logOn(ticketNumber, customerName);

  // get current balance
  double previousBalance = frontEnd.getBalance(customerName);

  // deposit some funds
  double newBalance = frontEnd.deposit(customerName, 10);
  // check the new balance is correct
  assertEquals(previousBalance + 10, newBalance, 0);

  // end customer session
  frontEnd.logOff(customerName);

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

5. `testWithdrawUnderBalance`: withdraw funds not exceeding account balance.

```
public void testWithdrawUnderBalance() throws RemoteException,
    UnauthorizedAccess {
  // connect to bank
  int ticketNumber = frontEnd.connect();

  // begin customer session
  final String customerName = "ernie";
  frontEnd.logOn(ticketNumber, customerName);

  // get current balance
  double previousBalance = frontEnd.getBalance(customerName);

  // withdraw some funds
  try {
    double newBalance = frontEnd.withdraw(customerName, 10);
    // check new balance is correct
    assertEquals(previousBalance - 10, newBalance, 0);
  }
  catch (InsufficientFunds e) {
    fail("withdraw under balance should succeed");
  }

  // end customer session
  frontEnd.logOff(customerName);

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

6. `testWithdrawOverBalance`: withdraw funds exceeding account balance.

```
public void testWithdrawOverBalance() throws RemoteException,
    UnauthorizedAccess {
  // connect to bank
  int ticketNumber = frontEnd.connect();

  // begin customer session
  final String customerName = "bert";
  frontEnd.logOn(ticketNumber, customerName);

  // get current balance
  double previousBalance = frontEnd.getBalance(customerName);

  // try to withdraw an amount greater than current balance
  try {
    frontEnd.withdraw(customerName, previousBalance + 1);
    fail("withdraw over balance should fail");
  }
  catch (InsufficientFunds e) {
    assertEquals(customerName, e.getCustomerName());
    // check account balance has not changed
    assertEquals(previousBalance, e.getAmount(), 0);
  }

  // end customer session
  frontEnd.logOff(customerName);

  // disconnect from bank
  frontEnd.disconnect(ticketNumber);
}
```

6.4.2. Client JNDI properties

Because the `<jndi-name>` in `jboss-client.xml` is the same as in the Hello World example, the `jndi.properties` file is shared between the examples.

See the Client JNDI properties in the Hello World example for a listing of the properties.

6.4.3. Test execution

The following target executes the junit test case:

```
ant run-test
```

If everything goes well the target output should look like this:

```
run-test:
    [junit] Running org.jbpm.bpel.tutorial.atm.AtmFrontEndTest
    ...
    [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 6.109 sec
```

6.4.4. Interactive execution

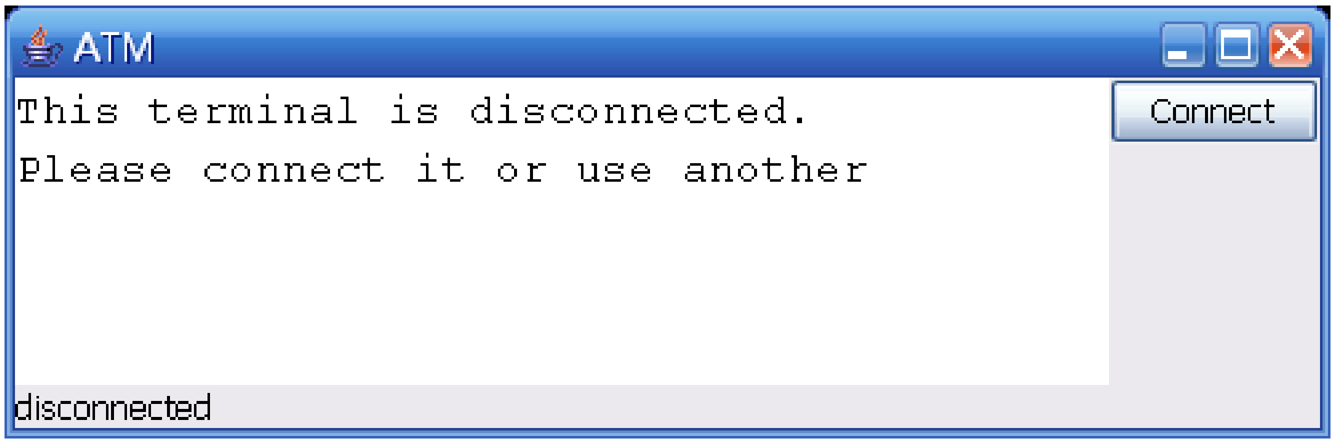
Last, but not least, the ATM example offers a rich client built from Swing components. This program resembles an actual teller machine. It has a double goal:

- Let you click your way through the process instead of writing unit tests to explore new scenarios.
- Assist you in demonstrating the BPEL technology to your customer, manager or colleague using an easy-to-follow interface.

To bring up the interactive terminal, call:

```
ant run-terminal
```

The frame bellow appears.

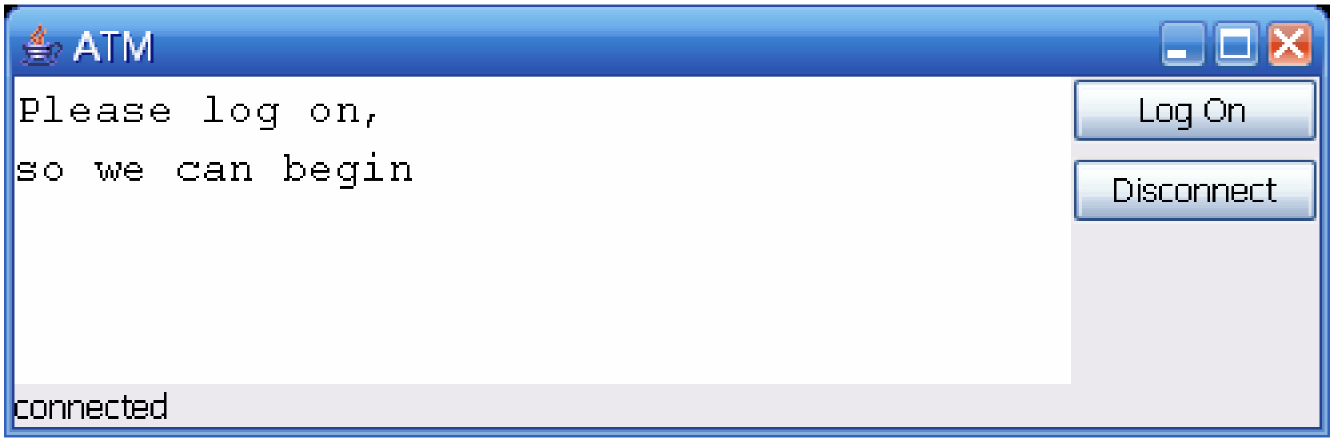


```
This terminal is disconnected.
Please connect it or use another
```

disconnected

Figure 6.6. Disconnected terminal

When you click *Connect*, the terminal establishes a connection with the server.

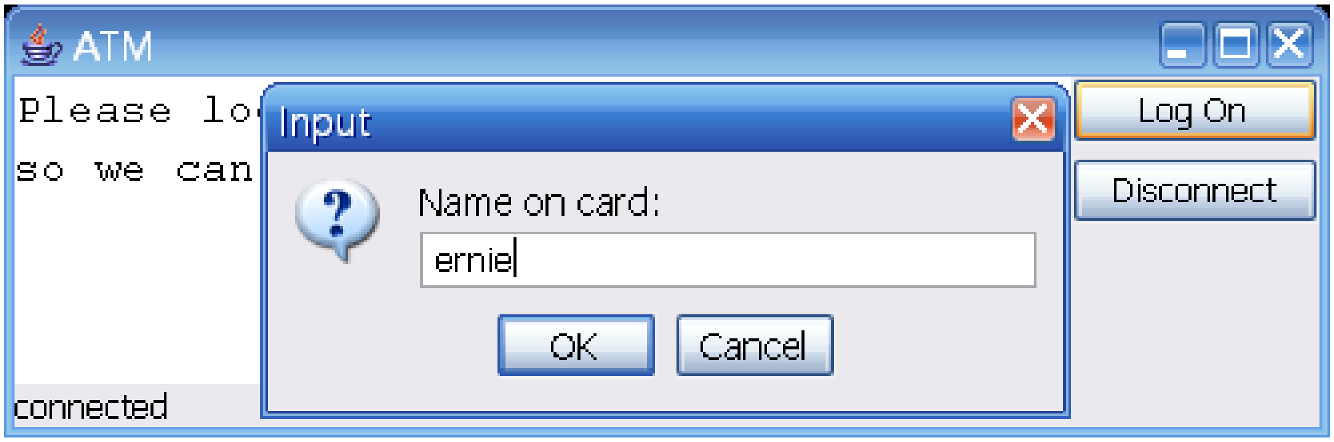


Please log on,
so we can begin

connected

Figure 6.7. Connected terminal

Clicking *Log On* starts a customer session. The program asks you for a customer name. Type in any name from the `accounts.xml` file in the `account` service.



Please lo
so we can

Name on card:
ernie

connected

Figure 6.8. Terminal starting a customer session

Enjoy!