Conversational state management in **Web Service Technologies**

Homework for

Seminars in Software Engineering

Author:

Claudio Di Ciccio

(dc.claudio@gmail.com)

Table of Contents

Introduction.. 4
The service..4
The work..4
Model-driven design..4
Service state and behavior management..5
Conversational Web Services.. 6
The goal.. 6
Design..8
Conversational Web Services.. 8
Web Services in J2EE v.5.. 8
WS-Addressing..9
WS-Addressing in JBossWS.. 10
The Message Handlers Stack..10
Conversational Web Services in JBoss WS..12
Resuming ideas..13
Services as Processes..15
Finite State Machines.. 15
jBPM / jPDL..16
From TSs to jPDL documents.. 17
HennessyMilner / XPath..22

Web Service / jBPM interaction.. 25

An example: WSter.. 26

Introduction

The service

The work we are presenting here is related to Web Service technologies. Web Service is, namely, a Service deployed on the Web.

> The service is a software artifact characterized by its behavior , the potential evolutions resulting from its interaction with some external systems, such as a client service [ASCVS].

Given this description, we are focusing on the behavioral aspect of services, while we use to consider Web Services as collections of remote procedures to call, widely spread around the network, based on XML communications.

The work

This project focuses on dynamic self-evolving services, concentrating the work on three main aspects:

1. the execution of processes behind the service;

2. the interaction with clients, self-developing step after step;

3. the separation between application logic and process-state logic.

(1.) and (2.) are often used as pre-conditions of many related works, so we may want to suggest a way of realizing services that do actually behave like exposed before. On the other hand, we may want to use a technique that clearly divide the business logic from the flow-control. A very simple way of addressing (1.) and (2.) should have been adding an optional state -related parameter in every signature, so that the service could check it out at every call and verify whether preconditions were respected or not: nothing formally incorrect, but a big mess in practice! Here is explained the reason for focusing on the third point, as well.

Model-driven design

> A conversation is a sequence of message exchanges that can occur between a client and a service as part of the invocation of a Web Service [MDWSD]

Commonly, Web Services are a collection of request-response operations: that is,

they do not keep any state information. As a consequence, they can be considered completely defined by the communication protocol, reported into WSDL files. WSDL is an acronym that stands for Web Service Definition Language , in fact, even though it mainly specifies just the types of messages exchanged between clients and servers at any operation call in a RPC fashion. We have seen that it is not enough, if we consider services as told before.

We may want to apply a model-driven design [MDWSD], a top-down design technique starting from the process behavior, and not from the operations interface: in other words we focus on actions, not on messages.

For example, suppose that only after a login (a web method to call) we can access another functionality (another web method to call): WSDL does not provide any formal construct to express this kind of precondition!

Why is it so important for services to specify (and, of course, respect) their own given behavior? Because, once their service schema ([MDWSD]) is defined, they can be part of a Services Community: being part of a Services Community means that it is possible for an Orchestrator to automatically synthesize a new Composite Service, built by properly calling Component Services (see [ASCBOBD] for further details). We said properly calling them: if a behavior was not defined, that adverb would have no sense but putting right values in the right places! We know that structured programs are quite more complex, then we want more.

The problem of Composition Existence (and Composition Synthesis) is EXPTIME-complete not only in case of fully-controllable component services, without faults, and centralized controllers (see [ASCBOBD] and [ASCVS] for a demonstration), but also for partially-controllable ones in a distributed environment ([ASGBMDB]) and in presence of failures ([BCPF]). Most important: composition can be done in EXPTIME, in the size of the available services; we know that usually services are not so complex to require huge defining schemata: hence, this cost is not impossible to afford.

Service state and behavior management

Given the service schema, we may need to be supported by an automatic state manager, in order to delegate it the control over processes' flow. It has to act like a referee for addressing

- concurrency,
    - (multiple clients can interact with the service at the same time)
- evolution,
    - (every service instance must follow a predetermined life-cycle)
- correctness

- (every transition from a state to another, for every instance, must respect the service schema)

Thus, we may say that the Web Service operations' business logic is separated from this kind of control check. On the other hand, the process manager does not mind neither the message exchange, nor the application variables to update! This design completely respects the point (3.) explained in the introduction.

Conversational Web Services

Web Services are stateless by default: this is quite obvious if we consider that their communication is based on stateless protocols (SOAP over HTTP, or SMTP...). Then, no information about the caller identity, neither about the client's interaction history is considered.

> A single web service may communicate with multiple clients at the same time, and it may communicate with each client multiple times.
>
> In order for the web service to track data for the client during asynchronous communication, it must have a way to remember which data belongs to which client and to keep track of where each client is in the process of operations. [DCWSBea]

Hence, the services we are going to describe, design, and implement so far, are based upon the so called Conversational Web Services .

The goal

Our purpose, in conclusion, is twofold:

1. Keep the state-related information in (stateful) Web Services
   - through Conversational Web Services
2. Integrate the Web Service with an automatic behavioral manager, besides the service implementation
   - through an automatic process manager

We will describe a framework in order to make this all real, that has to respect some requirements, such as:

- respect standards (for portability reasons)
  - this way, we will able to port our future products, based on this framework, on a large amount of existent Application Servers
- use already developed technologies
  - so that we can use them in real-world applications

The framework we are going to describe is based upon Java (Java Enterprise Edition v.5), and Java-related software infrastructures from the JBoss family (JBoss AS as the application server, JBoss WS as the Web Service plugin for JBoss AS, jBPM as the process instances container-manager).

Thus, we assume in the further discussion that the reader knows some J2EE basics that we are not going to describe, since the arguments we are now going to treat are high-level. Of course, the reader would have some knowledge of Web Services in general, for the same reason.

Design

Conversational Web Services

Web Services in J2EE v.5

Once given we are going to use Java-related technologies and software infrastructures, it is mandatory for us to see in a short summary how Web Services work on J2EE.

Let us, say, first of all, that any POJO (Plain Old Java Object), Stateless Session Bean (watch out: we said Stateless Session Bean), or Servlet, can be a Web Service Endpoint, given that it respects a defined interface. That abstract interface will contain the rules the server will expose for clients to interact, at any call, with it; this means that the abstract interface can be independent from the concrete service implementation. In other words, WSDL files are based upon the interface, not upon the bean behind it.

Usually, the software architect/programmer can define the protocols only through classes and interfaces definitions. On the other hand, only the container (Application Server) decides the policy of pooling instances, and that is the same strategy already seen for EJBs. So, users are unaware of which instance is actually serving the requests.

As told before, there is no mention about the business protocol[1].

The EJB-expert reader could ask: if almost any kind of Bean can stay behind a WS, why don't we use Stateful Session Beans to implement it? The Java compiler, actually, does not warn of any possible error. The point is that, instead, the container does: it's not admitted, as now, to use Stateful Session Beans.

The same reader could ask, then: why don't we associate Stateless Session Beans to Stateful Session Beans injecting them as inner properties? Given you can not decide which instance actually serves you, injecting Stateful Beans inside Stateless ones does not work, either: once more, there will be no compiler warning, and this time neither the container will object anything, but, typically, the container will overflow the memory stack after a few calls, as new Stateful objects will be created at any WS-call!

Hoping to have convinced even the most suspicious reader, we can say that actual technologies are not enough for use to make Web Services stateful.

1We call business protocol the specification of which messages exchange sequences are supported by the service ([MDWSD]), for example, expressed in terms of constraints on the order that service operations would be invoked in

WS-Addressing

WS-Addressing is a W3C standard, based on SOAP, that

> defines a family of message addressing properties that convey end-to-end message characteristics including references for source and destination endpoints and message identity that allows uniform addressing of messages independent of the underlying transport [WSAW3C]

Essentially, it defines some extra headers to put into the SOAP envelope, so that clients can have a pointer to the service endpoint, and identify the caller (see Illustration 1): it suits perfectly to our goal!

Example 1-1. Use of message addressing properties in a SOAP 1.2 message.

```
(01) <S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
                 xmlns:wsa="http://www.w3.org/2005/08/addressing">
(02)   <S:Header>
(03)    <wsa:MessageID>http://example.com/6B29FC40-CA47-1067-B31D-00DD010662DA</wsa:MessageID>
(04)    <wsa:ReplyTo>
(05)      <wsa:Address>http://example.com/business/client1</wsa:Address>
(06)    </wsa:ReplyTo>
(07)    <wsa:To>http://example.com/fabrikam/Purchasing</wsa:To>
(08)    <wsa:Action>http://example.com/fabrikam/SubmitPO</wsa:Action>
(09)   </S:Header>
(10)   <S:Body>
(11)     ...
(12)   </S:Body>
(13) </S:Envelope>
```

Lines (02) to (09) represent the header of the SOAP message where the mechanisms defined in the specification are used. The body is represented by lines (10) to (12).

Lines (03) to (08) contain the message addressing header blocks. Specifically, line (02) specifies the identifier for this message and lines (04) to (06) specify the endpoint to which replies to this message should be sent as an endpoint reference. Line (07) specifies the address URI of the ultimate receiver of this message. Line (08) specifies an action URI identifying expected semantics.

Illustration 1: Use of message addressing properties in SOAP 1.2 message

> A reference may contain a number of individual parameters that are associated with the endpoint to facilitate a particular interaction (...) Reference parameters are provided by the issuer of the endpoint reference and are assumed to be opaque to other users of an endpoint reference. [WSAW3C]

Since we have to identify the client, and WS-Addressing headers specification admit the `replyTo` node to have reference parameters in, we can insert the client (conversational) id as a `replyTo`'s reference parameter: every call will be identified by this unique id.

The client will take care of saving it, and sending it properly at any call: this technique is similar to the SESSION-header for HTTP protocols: like the SESSION-header for HTTP protocols, it can support the maintenance of client-server conversation state, over a stateless protocol.

Thus, we make use of a kind of piggy-backing on the payload, delegating the

conversation issue to the Transport layer as we would have expected, in theory.

WS-Addressing in JBossWS

JBossWS has not a native framework to actively support WS-Addressing. However, its libraries provide two classes that may help us:

1. `org.jboss.ws.extensions.addressing.jaxws.WSAddressingClientHandler`
2. `org.jboss.ws.extensions.addressing.jaxws.WSAddressingServerHandler`

These classes implement the interface `javax.xml.ws.handler.Handler`, that specify those modules that are able to manage SOAP Messages; the previous two classes are designed to add the SOAP headers all the tags requested to conform WS-Addressing. Unfortunately, they do not fill those fields, so you have to do it by hand. That is not a useless effort, since we had to add some extra-headers in order to insert optional reference parameters (that is, the conversational id!).

Implementing our custom (so called) Message Handlers is not enough to make them work: remember that we are altering a Transport Layer behavior, so we have not a direct access to that context! We can impose the underlying platform to use them, when receiving or sending messages, by associating to the Web Service implementation a custom Handler Chain (see Illustration 2) where they are listed.

The Message Handlers Stack

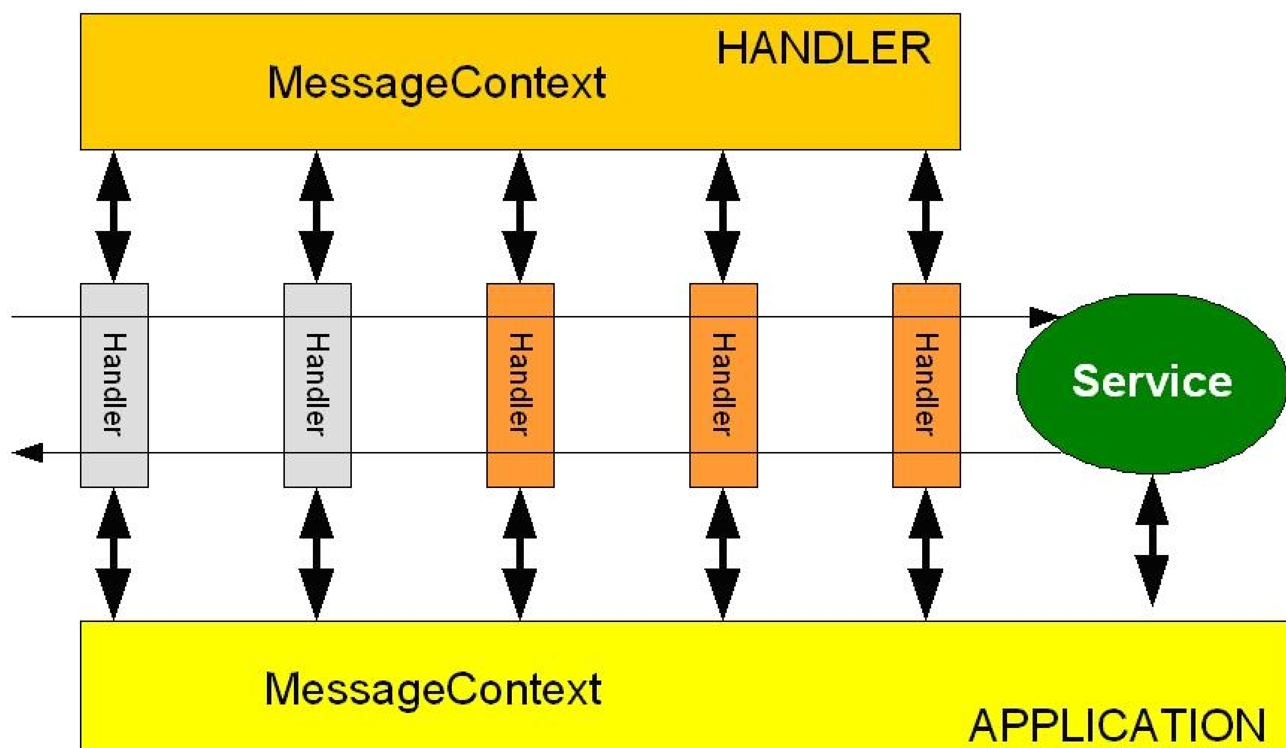

Illustration 2: Message Handlers Stack

The link among Transport and Application Layer is created by Message Handlers. Client-side and server-side handlers are organized into an ordered list known as Handler Chain . The handlers within a handler chain are invoked each time a message is sent or received, in the order that the programmer explicitly declared.

In order to associate a given Handler Chain, server-side, we write an XML file like the one in Illustration 3.

```
1<handler-chains xmlns="http://java.sun.com/xml/ns/javaee"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee javaee_web_services_1_2.xsd">
4
5    <handler-chain>
6        <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
7        <handler>
8            <handler-name>WS-Addressing Server Handler</handler-name>
9            <handler-class>
10               org.jboss.ws.extensions.addressing.jaxws.WSAddressingServerHandler
11           </handler-class>
12       </handler>
13       <handler>
14           <handler-name>Conversational Server Handler</handler-name>
15           <handler-class>
16               it.uniroma1.dis.sis.wster.service.handler.ServerHandler
17           </handler-class>
18       </handler>
19   </handler-chain>
20
21</handler-chains>
```

Illustration 3: Handler Chain XML descriptor

As you can see, we can put our custom class into the list of handlers, and thus declare we want the Message Context (that is, the context related to the lower layer, where SOAP Envelopes are directly accessed) be processed. Once the file is written, the way to link it to the Web Service is to declare it through the @HandlerChain annotation (see Illustration 4)

```
39 @WebService(
40         name = "WSterService",
41         targetNamespace = "http://www.dis.uniroma1.it/sis/convws/wster",
42         serviceName = "WSter",
43         wsdlLocation = "WEB-INF/wsdl/WSter.wsdl",
44         endpointInterface = "it.uniroma1.dis.sis.wster.service.WSterService"
45 )
46 @EndpointConfig(configName = "Standard WSAddressing Endpoint")
47 @HandlerChain(file = "WEB-INF/jaxws-handlers.xml")
48 @SOAPBinding(style = SOAPBinding.Style.RPC)
49 public class WSterServiceEndpoint implements WSterService {
```

Illustration 4: Code snippet: a conversational Web Service declaration

The Message Handler acting into the server is required to access and read WS-Addressing elements inside the headers, among the whom the conversation id lies as

a reference parameter, while handling inbound messages, and write them all back, inside outbound messages (see the `it.uniroma1.dis.sis.wster.service.handler.ServerHandler` class into the attached code for further information).

We need to use a custom Handler Chain on the client, such that the conversational id and all the others WS-Addressing elements are filled in requests, too (see `it.uniroma1.dis.sis.conversational.client.handler.ConversationClientHandler` for further details). Client-side, we cannot use an XML descriptor such as the previous, we have to write it down into the Java code (see Illustration 5).

```
((ConfigProvider)port)
.setConfigName("Standard WSAddressing Client");

List<Handler> customHandlerChain = new ArrayList<Handler>();
customHandlerChain.add(new ConversationClientHandler(
        WSterService.IDQN,
        WSterService.URI,
        WSterService.ACTION,
        this.conversationId));
customHandlerChain.add(new WSAddressingClientHandler());
```

Illustration 5: Code snippet: a conversational Web Service client

`it.uniroma1.dis.sis.wster.service.handler.ServerHandler` and `it.uniroma1.dis.sis.conversational.client.handler.ConversationClientHandler` classes have been encoded following the test code released by JBoss authors: you can find them at [WSAT].

Conversational Web Services in JBoss WS

Just doing what explained until now, everything would work out right. But, due to portability reasons, it is better to complete some additional steps (see [WSAJBoss]).

First of all, we have to explicitly declare the JBoss WS platform to consider the server and the client WS-Addressing compliant. Server-side, we have to write what you can read at line 46 of Illustration 4's code snippet (that is, through the @EndpointConfig annotation beware it is not a standard J2EE annotation). Client-side, we need to write down the first two lines of code in Illustration 5. Even though they are JBoss specific configurations, they are useful to be declared because future versions of JBoss WS will natively support WS-Addressing (so, you will not have to fill the standard WS-Addressing headers by hand).

Then, we have to alter the standard WSDL file, automatically associated to the Web Service once the WAR package is deployed on the Application Server, by adding the UsingAddressing tag into the port node, as shown in Illustration 6.

```
177         <operation name="searchByTitle">
178             <soap:operation soapAction="" />
179             <input>
180                 <soap:body
181                     namespace="http://www.dis.uniroma1.it/sis/conws/wster"
182                     use="literal" />
183             </input>
184             <output>
185                 <soap:body
186                     namespace="http://www.dis.uniroma1.it/sis/conws/wster"
187                     use="literal" />
188             </output>
189         </operation>
190     </binding>
191     <service name="WSter">
192         <port binding="tns:WSterServiceBinding"
193             name="WSterServicePort">
194             <soap:address
195                 location="http://127.0.0.1:8080/ConversationalService" />
196             <UsingAddressing
197                 xmlns="http://www.w3.org/2006/05/addressing/wsdl" />
198         </port>
199     </service>
200</definitions>
```

Illustration 6: Code snippet: WS-Addressing WSDL file

Resuming ideas

In order to implement a conversational Web Service framework we have to

- Implement the WS-Addressing framework
  - Implement Message Handlers, both for client and server
    - able to insert WS-Addressing tags into the SOAP headers
  - Put those Message Handlers into the Handlers Stack
  - Declare the WS-Addressing standard compliance into the WSDL file
- Insert the conversational id as a parameter of WS-Addressing headers
  - updating properly the message handling methods into the previously defined Message Handler classes

A simplified schema of what happens once those steps are done, and a request is sent, is drawn in Illustration 7 - we say it is simplified because the number of involved actors is bigger than the modules represented there, but it can be useful to keep the ideas clear in mind.

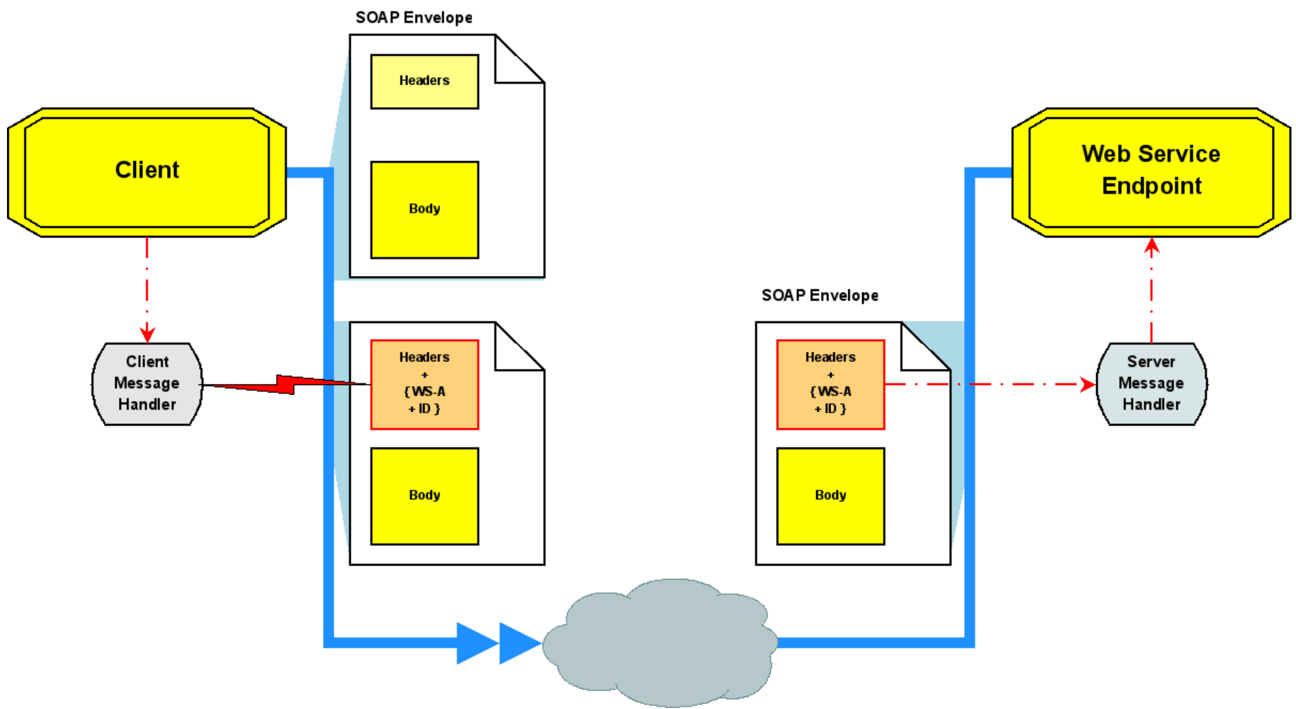

Illustration 7: A simplified Conversational Web Service request

Services as Processes

Finite State Machines

FSMs (Finite State Machines) are the construct that we use to describe services' behavior (in the so called *Roman approach* [ASCVS]). We shortly recall here that a deterministic FSM, used as a TS, standing for Transition System, is defined as follows:

$$TS=\langle \Sigma, S, s_0, \delta, F \rangle$$

Where Σ is the finite alphabet of actions, S is the finite set of states, s_0 is the initial state, in S, δ the transition function, F the set of finite states, in S.

It is quite common for Web Services to have transitions that do not directly depend from clients' invocations. For example, the server, not the client, decides whether user-names and passwords provided are correct for the user to login, so that he/she can access protected functionalities (that is, make the service evolve to states that, otherwise, would be not accessible). Thus, we have the so called devilish non-determinism. Consequently, we have to consider this new tuple

$$TS_N=\langle \Sigma, S, s_0, \delta_N, F \rangle$$

where every symbol maintains the previous meaning, except

$$\delta_N \subseteq S \times \Sigma \times S$$

that is a relation, not a function anymore (the same action can make the service proceed through different states).

Thus, we may have a Finite State Machine as the following (Illustration 8), for instance

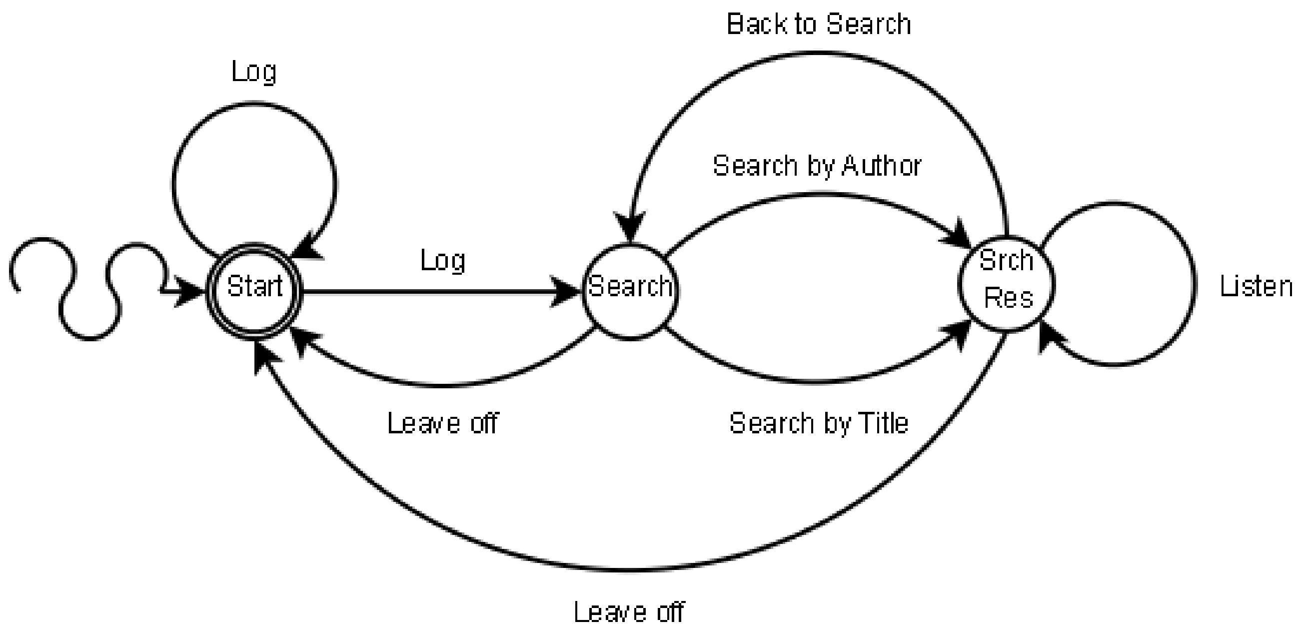

Illustration 8: A sample Finite State Machine representing a service

As we can see, we have a non-deterministic action (Log) that represent the fact that it is not decided a priori whether the user is giving the right credentials in order to log in. The Start state is the final state, too: remember, anyway, that a final state is not a state where the process evolution must actually   it can. Once the user logs in, he/she may search for a song, by title, or by author. Then, he can listen one of the result songs, or come back to the Search state. At any moment, he can leave off (for example, with a Logout).

Remember that this is supposed to be the service schema; if it will serve a website, this does not represent its page-flow   since the latter completely independent from the former.

jBPM / jPDL

jBPM (JBoss Business Process Manager) is a framework for advanced processes management: it is designed to be a real workflow-management tool. Hence, we are going to use it, but in a subset of its facilities: for example, it could manage automatic sending of emails: clearly, we are not interested in such a functionality, at all. On the other hand, its engine can store, update and monitor the evolution of processes, addressing problems like process instances' identity and concurrent accesses: this is why we chose it as the service/process manager, mainly. More over, it can be integrated with ESB tools such as JBoss ESB, and integrate BPEL descriptors. Of course, a reason for choosing this product has been the complete integration to JBoss AS: in fact, jBPM authors offer a full free JBoss AS distribution with JBoss WS Native plugin and jBPM extension, free to download.

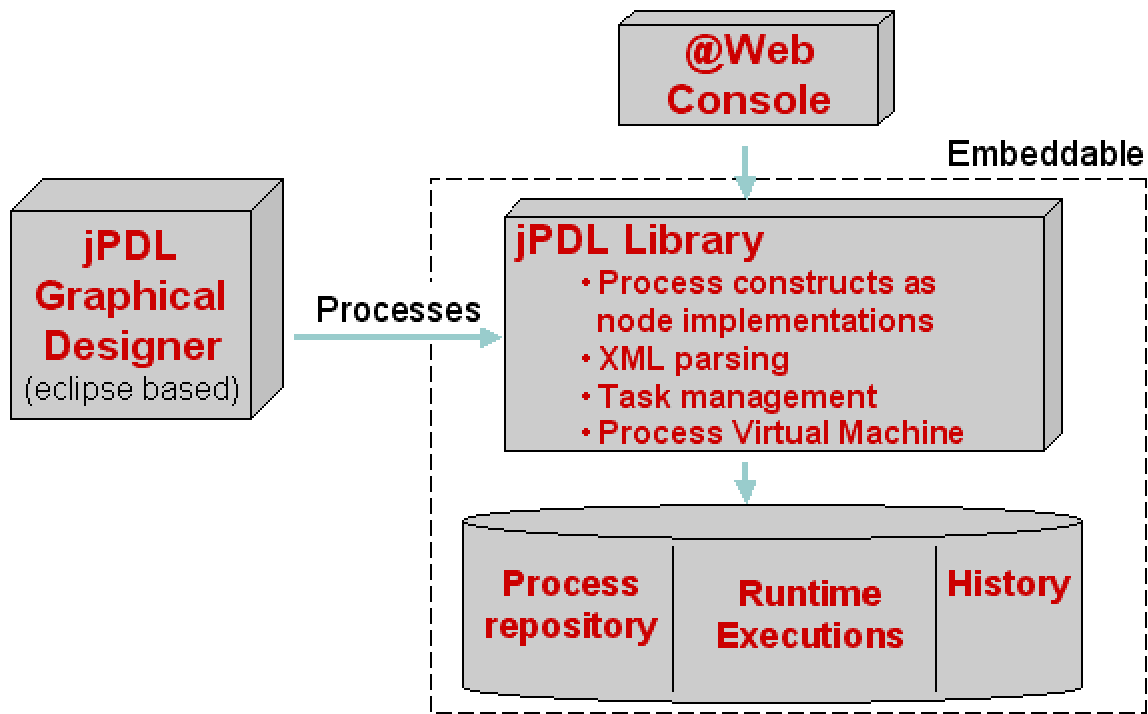

Illustration 9: jBPM

Natively, jBPM supports its own Process Definition Language (jPDL), XML-based, and Graph-Oriented (for further reading, see [JBPM]). The isomorphism of jPDL to Graphs of the XML-based syntax is a very important characteristic: in fact, XML in general is widely supported, structured but far looser than relational schemata. More over, many frameworks, languages, technologies for interacting, transforming, querying, updating XML data are already defined.

You can also make queries over contents and structures: for example, as we may use HennessyMilner formulas for asserting properties about FSMs, we can use XPath for expressing them about XML-based jPDL descriptors for more powerful languages, such as PDL or Modal Mu-Calculus you have to use XSLT, or XQuery.

We will see later some examples of translations from HennessyMilner to XPath equivalent formulas.

Last but not least, jPDL syntax is very close to WS-TSL (the specification language already seen in [PSP] during the course of Seminars in Software Engineering), so its documents will look quite familiar.

From TSs to jPDL documents

Mapping deterministic FSMs to jPDL (XML) documents is quite straightforward, as you can read in the following table:

s_0	`<start-state name="s0">`
$s \in S$	`<state name="s">`
$f \in F$	`<end-state name="f"/>`
$\delta(s,a)=s'$	`<state name="s">` `<transition to="s'" name="a">` `</state>`

But two conditions are imposed by jPDL, in this case (deterministic FSMs):

- start-states and end-states cannot be the coincident
- transitions cannot end to start-states, neither start from end-states

Thus, first of all, a FSM like the previous (see Illustration 8) has to be changed, as shown in Illustration 10

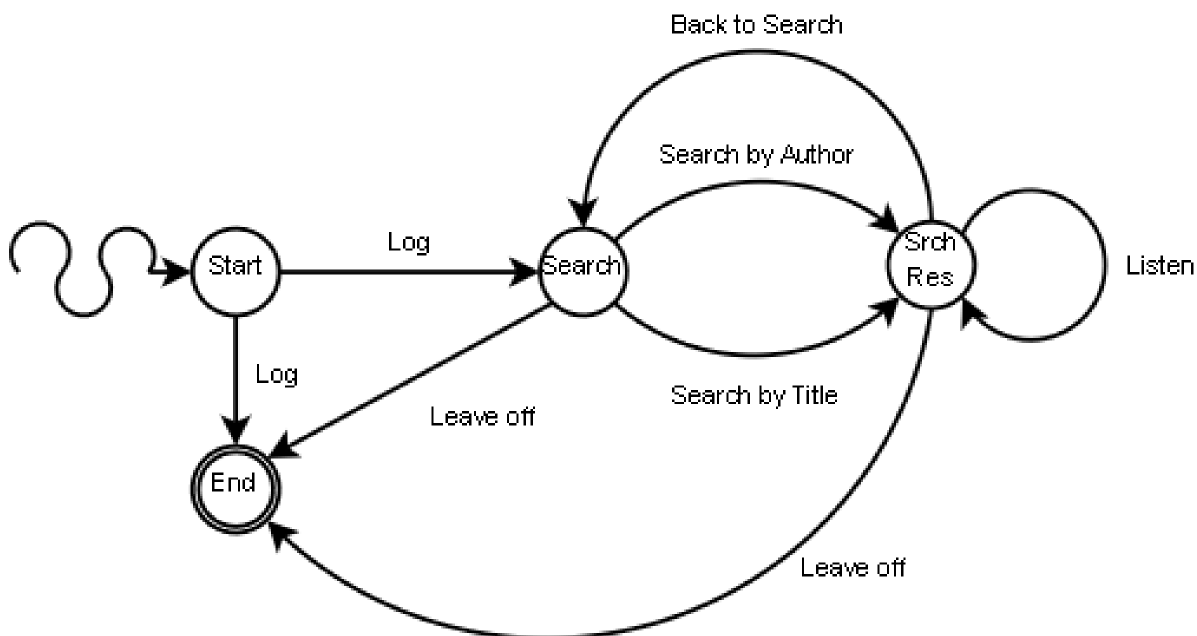

Illustration 10: Modified sample Finite State Machine representing a service

Final states have to turn into nodes without further transitions admitted, and start states must not have reentrant arcs. That is not all: consider that we have a devilish non-deterministic transition, still, that is Log .

Since jPDL documents are used to describe internal processes that are going to be automatically managed by the jBPM engine, then it is obvious that jBPM is not

designed for managing non-deterministic processes: non-determinism in services' Transition Systems is used to hide some internal details, while every concrete program is deterministic, and there is no behavioral information to hide to the process manager.

Hence, jPDL does not admit transitions to have multiple reachable states. In these cases, we can use special nodes called decision-states: they map the choice that the process make among multiple possible evolutions. Later we will see how to make use of them.

Before going on, let us reason about the power to leave the process manager. We may want to use it as

1. a passive flow controller
   - That is to say, it checks that every transition is legally invoked, and nothing more
2. an active flow controller
   - That is, it decides the state to evolve to, when the FSM (execution graph) has a multi-arc
     - decides by itself, in case of devilish (server-dependent) non-determinism, the transition to move through
3. an active flow controller, able to call user-defined actions
   - When the client asks for a transition to proceed, and it's linked to an action, the process manager calls the proper action handler

We choose the third option because it includes the previous ones, make the process manager able to directly manage exceptions thrown during the action (that is, the exception recovery policy and rollback is due to it, not to the calling method), and enriches the focus on actions, rather than using the framework as a simple flow-control instrument.

Beware that jPDL does not require the programmer to write any code line into the definition file: it delegates the action, or the decisions, to proper Java classes defined somewhere else by the programmer. So, the definition remains abstract: it's like putting another labeling (as we will see further).

This is why we can consider jPDL files as descriptors not only useful to be the sole input for the process manager, but also to be shown as contracts for the clients, or orchestrators, at the same time.

Once the proper delegated classes are encoded, then there is an interpretation for the schema.

Considering this all, we have to enlarge the algebra representing Transition Systems for internal representation, by adding three new labeling functions and three new sets, as follows:

$$PTS = \langle \Sigma, S, s_0, \delta, F, A, \alpha, H, \eta, D, \omega \rangle$$

where

- A is the set of action-handlers (labels)
- D is the set of decision-states, in S (so that transitions can start from elements in D, too, without extending the definition of δ)
- H is the set of decision-handlers (labels)
- α is the partial labeling function that, given a state s and an action a, returns an action-handler e, element of A (we assume that only if δ is defined on s and a, then α is defined as well),
- η is the labeling function that, given a decision-state d (in D) returns a decision-handler (element of H)
- ω is the transition function that, given a decision-state d (in D) and an action a, returns a state s

Even though everything looks much more complicated, it immediately appears easier when reading the following mapping table, from the Process Transition System to the jPDL syntax.

$d \in D$	<decision name="*d*"/>
$\alpha(s, a) = e$	<state name="*s*"> <transition to="*δ(s, a)*" name="*a*"> <action class="*e*"/> </transition> </state>
$\eta(d) = h$	<decision name="*d*"><handler class="*h*"/></decision>
$\omega(d, a) = s$	<decision name="*d*"> <handler class="*α(d, a)*"/> <transition to="*s*" name="*a*"/> </decision>

The sample service shown until now will look, once transposed in a jBPM valid process, as in Illustration 11

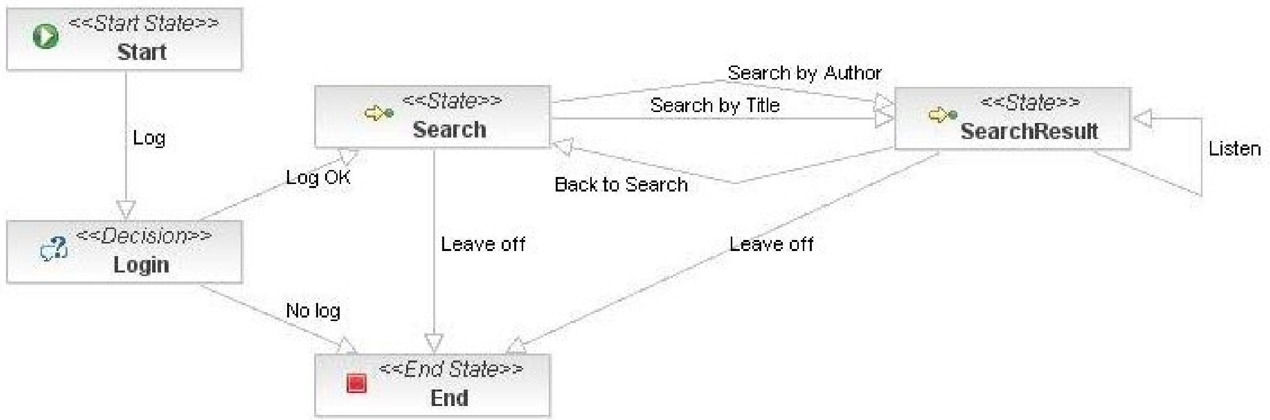

Illustration 11: The process representing the sample service

This graphic representation has been drawn with the graphical designer, included in the jBPM Plug-In for Eclipse, free to download, alone or inside the JBoss Tools Plug-In for Eclipse.

```
3<process-definition  xmlns="urn:jbpm.org:jpdl-3.2"  name="WSterProcess">
4
5    <description>
6        This example process is designed for describing a Napster-like application.
7    </description>
8
9    <start-state name="Start">
10        <transition to="Login" name="Log"></transition>
11    </start-state>
12
13    <decision name="Login">
14        <handler class="it.uniroma1.dis.sis.wster.service.action.LoginDecision"/>
15        <transition to="Search" name="Log OK"></transition>
16        <transition to="End" name="No log"></transition>
17    </decision>
18
19    <state name="Search">
20        <transition to="SearchResult" name="Search by Title">
21            <action class="it.uniroma1.dis.sis.wster.service.action.SearchByTitleAction"/>
22        </transition>
23        <transition to="SearchResult" name="Search by Author">
24            <action class="it.uniroma1.dis.sis.wster.service.action.SearchByAuthorAction"/>
25        </transition>
26        <transition to="End" name="Leave off"></transition>
27    </state>
28
29    <state name="SearchResult">
30        <transition to="Search" name="Back to Search"></transition>
31        <transition to="SearchResult" name="Listen">
32            <action class="it.uniroma1.dis.sis.wster.service.action.ListenAction"/>
33        </transition>
34        <transition to="End" name="Leave off"></transition>
35    </state>
36
37    <end-state name="End"></end-state>
38
39</process-definition>
```

Illustration 12: The sample process jPDL definition

The same process, encoded in jPDL XML syntax, is shown in Illustration 12

The classes mentioned inside the `class` attributes of `decision/handler` and `action`

nodes must be referred to classes that the programmer will have actually implemented inside the same WAR package the jPDL descriptor is. Respectively, those classes must implement the `org.jbpm.graph.node.DecisionHandler` and `org.jbpm.graph.def.ActionHandler` interfaces.

HennessyMilner / XPath

Now that we studied how does jPDL works, we can see some examples of HennessyMilner formulas translated into XPath queries.

As usual, we consider the example diagram.

First, if we want to verify whether, from Search state, we can reach SearchResult through Search by Title transition, we can write (knowing that the result should be TRUE):

- HM

  $Search \rightarrow \langle Search{\cdot}by{\cdot}Title \rangle T \wedge [Search{\cdot}by{\cdot}Title] SearchResult$

- XPath

```
boolean(
        not(//state[@name="Search"])
        or (
                //transition[
                        parent::state[@name="Search"]
                        and @name="Search by Title"
                        and @to="SearchResult"]
        )
)
```

If we want to verify whether the only available transition from Search is Search by Title (and we know that this is FALSE):

- HM

  $Search \rightarrow \langle any \rangle T \wedge [any - Search{\cdot}by{\cdot}Title] F$

- XPath

```
boolean(
        not(//state[@name="Search"])
        or (
                //transition[parent::state[@name="Search"]]
                        and not(
                                //transition[parent::state[@name="Search"]
```

```
                    and not(@name="Search by Title")
                    ]
                )
        )
)
```

Lastly, if we want to verify whether the only available transition from the start-state Start is Log (TRUE):

- HM

  $Start \rightarrow \langle any \rangle T \wedge [any - Log]F$

- XPath

```
boolean(
        not(//start-state[@name="Start"])
        or (
                //transition[parent::start-state[@name="Start"]]
                and not(
                        //transition[
                        parent::start-state[@name="Start"]
                        and not(@name="Log")]
                )
        )
)
```

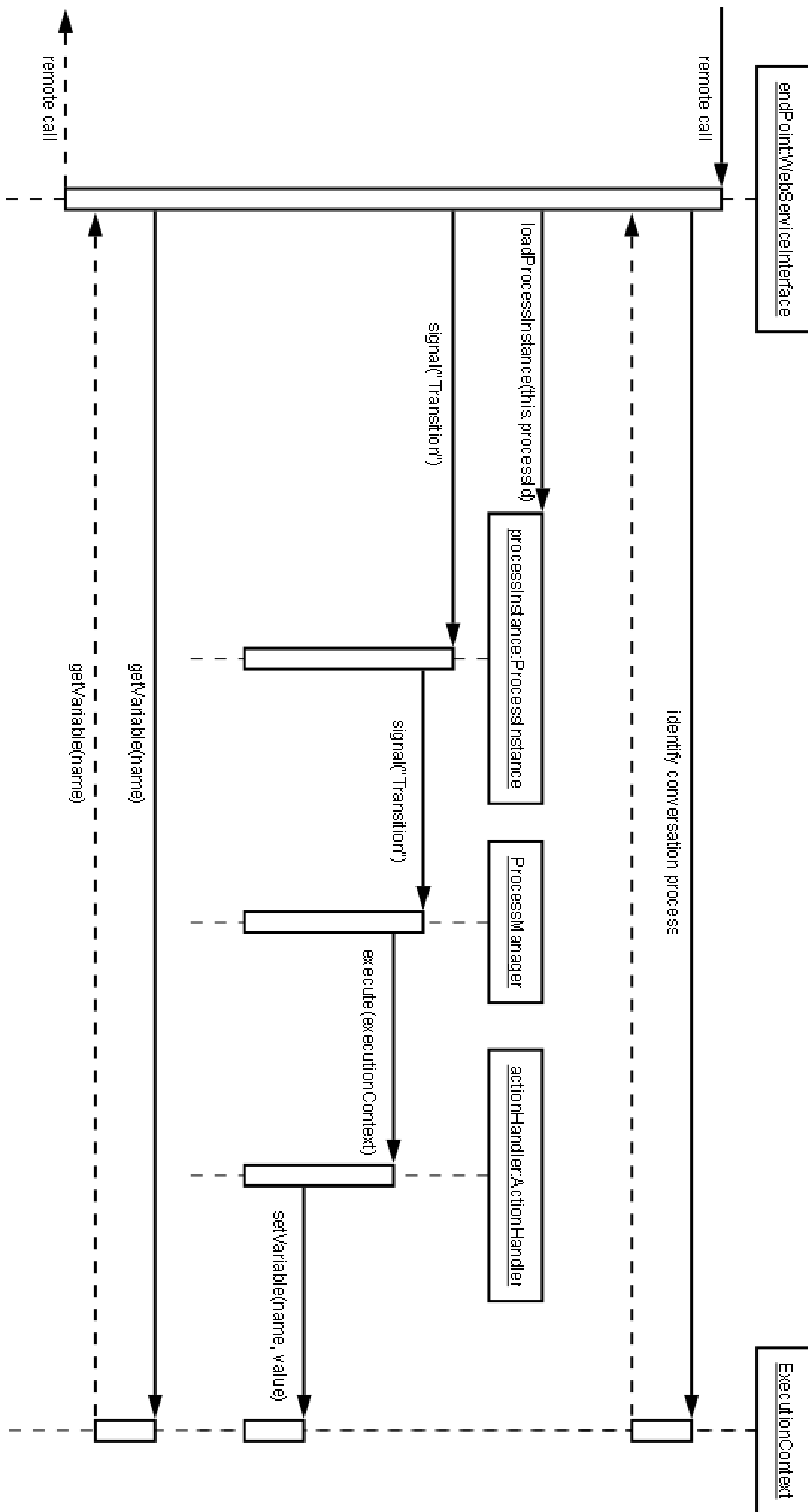

Illustration 13: Sequence diagram of a request processing

Web Service / jBPM interaction

In the framework we are describing now, web services act like a façade for the real service core. The service core is spread over jBPM, that is the process manager, action-handlers and decision-handlers.

Illustration 13 shows a sequence diagram representing the processing of a remote request call to a Web Service, in our framework.

As we can see, Web Service operations' implementing code

1. identifies the process related to the conversation;
2. invokes the Jbpm Context methods for activating proper transitions;
3. waits for the Jbpm Context to complete its business
   - registering the transition for updating the current state on the specified process
   - activating eventual decisions and actions;
4. returns the computations' results back to the client.

Differently from BPEL, there is no information about variables to update and parameters to call inside the process descriptors, due to information hiding, and Actions and Web Service Endpoints don't share any reference, but the so called jBPM Context, in order to have encapsulation. They both access temporary runtime variables provided by the jBPM Context, and jBPM framework guarantees persistence, coherence, concurrency over process runtime variables.

Thanks to the completely independent computation of Web Services operations and Action- (or Decision-) Handlers, the same process could stand behind many services, offering different interfaces.

In Illustration 14 we can see the typical source code of a Web Service implementation: after initializing (`initSessionState()`, method in which we will read the conversational id and load the associated process id see the attached code for further information), it asks the `jbpmContext` reference to load the process instance related to the current conversation, sets the variables the decision-handler will read, and then ask the process instance to activate the transition named Log (`processInstance.signal(...)`). Then, it reads the updated value of the return parameter, stored into the Process Context Instance, and returns it to the client (given that the transition was callable at the moment otherwise, a `JbpmException` is thrown).

```
145    public BooleanDto login(String username, String password) {
146        this.initSessionState();
147        log.info("Trying to log in through: " + username + " / " + password);
148
149        ProcessInstance processInstance =
150            this.jbpmContext.loadProcessInstance(this.processId);
151        ContextInstance ctxInstance = processInstance.getContextInstance();
152
153        ctxInstance.setVariable(LoginDecision.USER_VARNAME,
154                new UserData(username, password));
155
156        Boolean result = null;
157
158        try {
159            processInstance.signal("Log");
160            result =
161                ((Boolean)
162                        (ctxInstance.getVariable(
163                                LoginDecision.LOG_DECISION_VARNAME)));
164        }
165        catch(JbpmException jbpmEx) {
166            log.error("Illegal state exception: ", jbpmEx);
167            ErrorDto error = new ErrorDto(new IllegalStateException(jbpmEx));
168            BooleanDto errorResult = new BooleanDto(false);
169            errorResult.error = error;
170            return errorResult;
171        }
172        finally { this.jbpmContext.close(); }
173
174        return new BooleanDto(result);
175    }
```

Illustration 14: Web Service operation implementing code

An example: WSter

WSter is a simple application that uses the framework applied to the example showed in this document. It's divided into three logical units, and physically deployed in three correspondent archives, following the MVC architecture:

1. A WAR with presentation modules (View)
    - JSP files, presentation logic, clients of the main Web Service
    1. Conversational WS-Addressing MessageHandlers
2. A WAR with control modules (Control)
    - Service definition files
        - jPDL descriptor, message handlers stack descriptor, WSDL
    - Service implementation classes
        - Servlet, ActionHandlers, DecisionHandlers, MessageHandlers
    - Conversational WS-Addressing MessageHandlers

3. A JAR with persistence modules (Model)
    - EJB 3.0 - based Web Service (for remote access)
    - JPA-connected Database

Since it contains all the source code whose snippets are illustrated here, and shows many J2EE frameworks, it is strongly recommended to install it, use it, and carefully read the code (most of all, the WAR with control modules, the core of the framework).

Bibliography

ASCVS: Daniela Berardi, Fahima Cheikh, Giuseppe De Giacomo, Fabio Patrizi, Automatic Service Composition Via Simulation, 2007

MDWSD: Karim Baina, Boualem Benatallah, Fabio Casati, and Farouk Toumani, Model-Driven Web Service Development, 2004

ASCBOBD: Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini and Massimo Mecella, Automatic Service Composition Based On Behavioral Descriptions, 2005

ASGBMDB: Sebastian Sardina, Fabio Patrizi and Giuseppe De Giacomo, Automatic Synthesis of a Global Behavior from Multiple Distributed Behaviors, 2007

BCPF: Sebastian Sardina, Fabio Patrizi and Giuseppe De Giacomo, Behavior Composition in the Presence of Failure, 2008

DCWSBea: Bea, Designing Conversational Web Services, , http://e-docs.bea.com/workshop/docs81/doc/en/workshop/guide/converse/navMaintainingStatewithConversations.html

MDWSD: Karim Baina, Boualem Benatallah, Fabio Casati, and Farouk Toumani, Model-Driven Web Service Development,

WSAW3C: W3C, Web Services Addressing 1.0 - Core, 2006, http://www.w3.org/TR/ws-addr-core/

WSAT: JBoss.org, WS-Addressing Test, 2005, http://www.java2s.com/Open-Source/Java-Document/JBoss/jbossws-2.0.1.GA/org.jboss.test.ws.jaxws.samples.wsaddressing.htm

WSAJBoss: JBoss.org, WS-Addressing Guide, 2005, http://jbossws.jboss.org/mediawiki/index.php/JAX-WS_User_Guide#WS-Addressing

JBPM: JBoss.org, JBoss jBPM - Workflow in Java, 2008, http://docs.jboss.com/jbpm/v3.2/userguide/html/index.html

PSP: Massimo Mecella, Putting Services into Practice, 2008