Università degli studi di Firenze

Facoltà di Scienze Matematiche, Fisiche e Naturali

Tesi di Laurea in

Informatica

Sperimentazione e confronto di alcuni Engine BPEL

Relatore

Prof. Rosario Pugliese

Corelatore

Dott. Alessandro Lapadula

Candidato

Daniele Nucci

Anno Accademico 2006/2007

Alla mia famiglia, con profonda gratitudine.

INDICE

CAPITOLO 1

INTRODUZIONE

Negli ultimi anni il mondo dell'Information Technology ha visto aumentare l'interesse per le architetture SOA (acronimo di Service Oriented Architecture). Un'architettura orientata ai servizi consente di rendere disponibili in rete differenti risorse richiamabili come servizi a richiesta. Le architetture di tipo SOA semplificano la modifica delle modalità di interazione tra i servizi o della combinazione nella quale i servizi vengono utilizzati nell'applicazione. Esse, inoltre, agevolano l'aggiunta di nuovi servizi o la modifica di processi in risposta a specifiche esigenze. Importante è ricordare come in tale tipo di architettura le applicazioni non sono più vincolate ad una specifica piattaforma o ad un'applicazione ma possono essere intese come componenti di un processo più ampio, e quindi riutilizzate. Le applicazioni che meglio si adattano a far parte di un'architettura SOA sono i web service. Un web service è un sistema software accessibile in rete, il quale, tramite un'interfaccia, mette a disposizione delle funzioni a qualunque applicazione appartenente alla stessa rete. Esso, inoltre, fornisce la descrizione delle proprie caratteristiche, permettendo di capirne l'utilizzo nel momento stesso in cui viene contattato. I web service comunicano con le applicazioni tramite il protocollo HTTP ed altri standard web basati su XML [18]. Ben presto è nata la necessità di poter gestire in modo più rigoroso gli utilizzi dei web service all'interno di un'architettura SOA, portando a sviluppare metodi per il controllo del flusso e creando dei veri e propri linguaggi di programmazione, i quali utilizzano i web service come fossero delle funzioni che prendono, come

parametro, il messaggio da inviare al web service e restituiscono il messaggio di risposta del servizio, come risultato. Il linguaggio che sta ottenendo maggiori consensi per ottenere le funzionalità esposte è WS-BPEL (Web Service Business Process Execution Language) [10]. WS-BPEL è lo standard definito dall'OASIS (Organization for the Advancement of Structured Information Standards) che deriva dal precedente BPEL4WS [14] sviluppato da aziende private come IBM e Microsoft. Esso fornisce un insieme di costrutti che descrivono le composizioni dei web service: alcuni possono essere paragonati con quelli forniti dai linguaggi tradizionali di programmazione, ma, in generale, tutti i costrutti operano ad un livello di astrazione più elevato. I programmi scritti in linguaggio WS-BPEL sono chiamati processi. Per eseguire un processo è necessario un server apposito che ne implementi le funzionalità e resti in attesa di applicazioni client che ne chiedono la creazione di un'istanza: questi server prendono il nome di engine BPEL. Dato che lo standard OASIS [10] è abbastanza recente (aprile 2007), i vari engine BPEL si stanno uniformando alle specifiche in tempi diversi. Inoltre, la stesura delle specifiche in linguaggio naturale porta a delle ambiguità che si riflettono in comportamenti differenti adottati dai vari engine BPEL. Questa tesi inizia con la presentazione degli standard utilizzati dai web service per poi passare alla descrizione del linguaggio WS-BPEL ed, infine, vengono presentati e testati alcuni engine BPEL ritenuti più importanti tra quelli disponibili per un utilizzo gratuito: ActiveBPEL Engine [1], Apache ODE [3] e Oracle BPEL Process Manager [11].

CAPITOLO 2
I WEB SERVICE

Un web service utilizza degli standard web basati su XML [18]. XML è un meta-linguaggio di markup, cioè permette di definire altri linguaggi di markup mediante un insieme di regole sintattiche per modellare la struttura di documenti e dati. Tramite XML sono stati sviluppati gli standard che più ci interessano per il prosieguo della trattazione:

- **XML Schema** [18]
- **WSDL** [17]
- **SOAP** [15]

L'utilizzo di tali standard permette ai web service di interagire con applicazioni residenti su piattaforme hardware e/o software differenti.

2.1 XML Schema

XML Schema si occupa di descrivere la struttura di un documento XML. In particolare definisce:

- i tag;
- gli attributi;
- i tipi di dati (intero, stringa...);
- i valori di default.

Un esempio ci aiuta a capirne il funzionamento. Supponiamo di avere un documento XML come il seguente:

Documento XML

```
<?xml version=¨1.0¨?>
<utente>
        <nickname>Imperatore</nickname>
        <nome>Daniele</nome>
        <cognome>Nucci</congome>
        <email>daniele.nucci@gmail.com</email>
</utente>
```

La descrizione in XML Schema potrebbe essere:

Documento XML Schema

```
<?xml version=¨1.0¨ ?>
< xsd:schema xmlns:xsd=¨http://www.w3.org/2001/XMLSchema¨
  targetNamespace=¨http://www.miatesi.it¨
  xmlns=¨http://www.miatesi.it¨ elementFormDefault=¨qualified¨>
  <xsd:element name=¨utente¨>
      <xsd:complexType>
          <xsd:all>
              <xsd:element name=¨nickname¨ type=¨xsd:string¨/>
              <xsd:element name=¨nome¨ type=¨xsd:string¨/>
              <xsd:element name=¨cognome¨ type=¨xsd:string¨/>
              <xsd:element name=¨email¨ type=¨xsd:string¨/>
          </xsd:all>
      </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Analizziamo il codice nel dettaglio. La prima riga, indica che il documento è scritto in XML. L'elemento schema è la radice del documento; questo elemento può avere degli attributi, nell'esempio abbiamo usato:

- `xmlns:xs`=¨http://www.w3.org/2001/XMLSchema¨: stabilisce che gli elementi con prefisso "xs" appartengono al namespace http://www.w3.org/2001/XMLSchema;

- `targetNamespace`=¨http://www.miatesi.it¨: stabilisce che gli elementi di questo schema appartengo al namespace http://www.miatesi.it;
- `xmlns`=¨http://www.miatesi.it¨ è il namespace di default;
- `elementFormDefault` permette di indicare se gli elementi, definiti nello schema, devono avere o meno un prefisso di default che ne indica il namespace. In questo caso l'attributo ha come valore "qualified" quindi gli elementi devono avere un prefisso.

I tag successivi rappresentano tutti gli elementi che possono essere presenti nel documento XML: nel nostro caso, abbiamo definito quattro elementi semplici (`nickname`, `nome`, `cognome` e `email`) e un elemento complesso (`utente`). Gli elementi semplici si definiscono utilizzando il tag <element> specificando il nome e il tipo, mentre più articolata è la dichiarazione degli elementi complessi. Un elemento complesso viene dichiarato anch'esso tramite il tag <element> ma, per la definizione del tipo, si utilizza il tag <complexType>: al suo interno, si definiscono tutti gli elementi dai quali è composto.
Il tag <all> fa parte degli *indicatori*: denota che gli elementi, in esso contenuti, possono comparire in qualunque ordine ed ogni elemento deve essere utilizzato una sola volta.
La trattazione di XML Schema è volutamente marginale, ma per i nostri scopi è più che sufficiente. Per una visione completa, il lettore può fare riferimento allo standard W3C [18].

2.2 WSDL

Un documento WSDL (Web Service Description Language) si occupa di indicare dove si trovano i servizi e di descrivere le operazioni del web service a cui il documento è associato. Esso è composto da cinque elementi principali:

- <types> definisce i tipi di dati utilizzati;
- <message> definisce un messaggio (per definire più messaggi, si utilizzano più istanze dell'elemento) usato dal web service per comunicare con il client;
- <portType> definisce una porta di comunicazione, le operazioni disponibili ed i messaggi utilizzati nelle operazioni;

- <binding> fornisce le indicazioni su come le operazioni definite da <portType> saranno trasmesse sulla rete;

- <service> indica l'indirizzo HTTP da cui poter accedere al web service.

Adesso vediamo un esempio relativo ad un servizio, il quale, ricevuto un numero intero `idUtente`, restituisce informazioni sull'utente a cui è associato tale numero.

Documento WSDL - definizioni dei namespace e tag <types>

```
<?xml version="1.0"?>
<definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.miatesi.it"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://www.miatesi.it">
<types>
  <xs:schema targetNamespace=http://www.miatesi.it">
  <xs:complexType name="utente">
      <xs:all>
      <xs:element name="nickname" type="xs:string"/>
      <xs:element name="nome" type="xs:string"/>
      <xs:element name="cognome" type="xs:string"/>
      <xs:element name="email" type="xs:string"/>
      </xs:all>
  </xs:complexType>
</xs:schema>
</types>
```

Gli attributi del tag <definitions> determinano i namespace che utilizzeremo. Il tag <types> definisce il tipo complesso utente, mediante la stessa sintassi vista in precedenza per XML Schema.

Documento WSDL - tag <message>

```
<message name="getUserRequest">
  <part name="idUtente" type="xs:integer"/>
</message>
<message name="getUserResponse">
  <part name="utente" type="tns:utente"/>
</message>
```

Il primo messaggio riceve l'intero corrispondente all'`idUtente` inviato dal cliente, mentre il secondo messaggio, di tipo `utente`, contiene la risposta del web service. Il tag <part> (ogni messaggio può averne più di uno) specifica da cosa è composta quella parte di messaggio e le associa un nome.

WSDL - tag <portType>

```
<portType name="gestioneUtentiType">
  <operation name="getUserById">
    <input message="tns:getUserRequest"/>
    <output message="tns:getUserResponse"/>
  </operation>
</portType>
```

Al <portType> si associa un nome e quindi un'operazione tramite <operation> (possono esserci più operazioni). Ogni operazione può essere di quattro tipi:

- `One-way`: riceve un messaggio;
- `Request-response`: riceve un messaggio e invia una risposta;
- `Solicit-response`: invia una richiesta e attende una risposta;
- `Notification`: invia un messaggio.

Nel nostro caso, abbiamo usato un'operazione del tipo `request-response`. L'elemento <binding> si occupa del collegamento con il protocollo di comunicazione, in questo caso SOAP.

WSDL - tag <binding>

```
<binding name="gestioneUtentiBinding" type="tns:gestioneUtentiType">
<soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getUserById">
    <soap:operation soapAction="" style="rpc"/>
    <input>
      <soap:Body use="encoded" namespace="http://www.miatesi.it"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:Body use="encoded" namespace="http://www.miatesi.it"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

Il tag <binding> è del tipo <portType> definito precedentemente. Il secondo binding è relativo a SOAP ed ha due attributi: `style` e `transport`. Nel nostro caso, `style`="rpc" e per `transport` utilizziamo il protocollo HTTP. In seguito, richiamiamo l'operation definita in <portType> e gli associamo una `soapAction` (solitamente viene lasciata vuota). Per finire, tramite i tag <input> e <output>, definiamo il tipo del messaggio. All'interno dei tag si dichiara l'elemento <soap:Body>, il quale specifica come saranno le parti del messaggio all'interno del Body del protocollo SOAP, specificandone, se presente, anche il tipo di codifica.

Infine, abbiamo l'elemento <service>:

WSDL - tag <service>

```
<service name="ServizioUtenti">
  <port name="GestioneUtenti" binding="tns:gestioneUtentiBinding">
    <soap:address location="http://www.miatesi.it/webservice"/>
  </port>
</service>
```

Al tag associamo un nome, colleghiamo una porta al binding e definiamo l'indirizzo HTTP sul quale il nostro web service starà in attesa della richiesta. Per approfondire l'argomento, si può fare riferimento allo standard W3C [17].

2.3 SOAP

Il protocollo SOAP (Simple Object Access Protocol) si occupa di far comunicare, mediante HTTP e XML, applicazioni sviluppate con linguaggi di programmazione diversi e residenti su architetture, sia hardware che software, differenti. I principali tag che compongono un documento SOAP sono:

- <Envelope> dichiara che il documento è un messaggio SOAP, dichiarando gli opportuni namespace;
- <Header>, tag opzionale, permette al client, che si connette al web service, di inviare informazioni aggiuntive;
- <Body> contiene i messaggi da scambiare con il web service;
- <Fault>, tag opzionale, è dichiarato all'interno di <Body> e si occupa di fornire informazioni riguardo ad errori rilevati nei messaggi.

Vediamo un esempio di richiesta, compatibile con il documento WSDL descritto in precedenza:

Documento SOAP

```
<?xml version=¨1.0¨ ?>
<SOAP-ENV:Envelope
xmlns:xsi=¨http://www.w3.org/2001/XMLSchema-instance¨
  xmlns:xs=¨http://http://www.w3.org/2001/XMLSchema¨
  xmlns:SOAP-ENV=¨http://schemas.xmlsoap.org/soap/envelope/¨>
    <SOAP-ENV:Body xmlns:tns=¨http://www.miatesi.it¨>
      <tns:getUserById
SOAP-ENV:encodingStyle=¨http://schemas.xmlsoap.org/soap/envelope/¨>
        <idUtente xsi:type=¨xsd:integer¨>29</idUtente>
      </tns:getUserById>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Il tag <Envelope> definisce i namespace del documento, mentre, nel <Body>, inseriamo il namespace dichiarato nel documento WSDL: così facendo, il web service saprà come interpretare gli elementi presenti all'interno del messaggio. Il messaggio SOAP richiama l'operazione `getUserById` del web service, la quale aspetta un dato di tipo intero (il tag input dell'operazione) e la risposta conterrà il dato di tipo `utente`, corrispondente all' `idUtente` inviato (il tag output dell'operazione). Per fare arrivare il messaggio al web service, un'ipotetica applicazione client deve incapsulare il documento SOAP nel protocollo HTTP ed inviarlo all'indirizzo specificato nell'attributo `location`, dichiarato nel tag <service> di WSDL. Anche in questo caso, per una conoscenza più approfondita, si consiglia la lettura dello standard [15]. Per inviare messaggi SOAP ai web service, esiste un ottimo tool open source, *soapUI*, scaricabile all'indirizzo: *http://www.soapui.org/*. Questa utility ha molte funzioni, per conoscerne tutte le funzionalità si raccomanda di fare riferimento alla *soapUI User Guide* [2]. Per testare gli esempi contenuti in questa tesi, è sufficiente:

- creare un nuovo progetto WSDL, non specificando alcun file WSDL;
- cliccare con il tasto destro il nome del progetto e scegliere "add WSDL from url";
- inserire l'url del servizio web. Esempio: *http://127.0.0.1:8080/ode/processes/InstanceManagement?wsdl*;
- rispondere "sì" alla domanda "create default request for all operations".

Così facendo, *soapUI* interrogherà il web service e metterà a disposizione tutte le operazioni eseguibili. Scegliendo un'operazione e la relativa `request`, *soapUI* creerà un messaggio SOAP, con dei punti interrogativi, dove dobbiamo inserire i valori da inviare al web service; una volta inviato il messaggio, il tool mostrerà la risposta ricevuta da parte del servizio.

Abbiamo accennato come è possibile creare un'applicazione raggiungibile da qualsiasi parte del mondo e come poter comunicare con essa. E' naturale domandarsi se vi è un modo di far sì che più web service possano essere utilizzati per creare nuove applicazioni o, perché no?, nuovi web service, i quali sfruttano le peculiarità di ognuno di essi. La risposta positiva a tale domanda, ha visto negli ultimi anni l'affermarsi del linguaggio *WS-BPEL* [10].

CAPITOLO 3

WS-BPEL

WS-BPEL [10], acronimo di Web Services Business Process Execution Language, è un linguaggio di programmazione basato su XML e incentrato, come dice il nome, sulle attività di business. WS-BPEL è in grado di eseguire attività in parallelo e correlate tra loro, manipolare dati e gestire eventuali eccezioni. Un processo di business, tramite l'utilizzo di WS-BPEL, può comporre più web service creando un'applicazione la cui interfaccia pubblica può essere esibita agli utenti finali.

3.1 Struttura di un processo WS-BPEL

Un processo WS-BPEL è un documento XML-like che può contenere alcune sezioni, tutte opzionali, prima della definizione del flusso del processo. Queste sezioni sono:

- <extensions> qui vengono dichiarate le estenzioni al linguaggio WS-BPEL;
- <import> specifica i documenti WSDL e gli XML Schema utilizzati dal processo;
- <partnerlinks> definisce le relazioni tra i vari partner;
- <variables> dichiara le variabili utilizzate;

- <correlationSets> definisce le correlazioni tra le parti: servono per assicurare che i messaggi siano smistati alle giuste istanze dei web service;
- <faultHandlers> definisce il comportamento del processo nel caso si presentino degli errori;
- <eventHandlers> si comporta come il faultHandlers, però gestisce degli eventi attesi invece che degli errori.

3.1.1 Attività di WS-BPEL

Il flusso di un processo WS-BPEL viene implementato tramite le attività di WS-BPEL. Esse possono essere catalogate in tre macro gruppi: attività di base, attività short-lived ed attività strutturate. Di seguito, per ogni attività viene presentata la sintassi XML-like dichiarata dallo standard OASIS [10]. Per facilitarne la lettura, riportiamo parte delle definizioni presenti nelle specifiche:

- la sintassi appare come un'instanza XML, ma i valori indicano i tipi di dati invece dei valori;
- I caratteri presenti alla fine degli elementi e degli attributi hanno il seguente significato: ¨?¨ (0 o 1), ¨*¨ (0 o più), ¨+¨ (1 o più);
- gli elementi e gli attributi separati da ¨|¨ e raggruppati da ¨(¨ e ¨)¨ rappresentano una scelta alternativa (or esclusivo);
- Qname e NCName sono definiti in XML Schema Part 1 [20]: un NCName è una sequenza di lettere, cifre, underscore ma non può iniziare con una cifra. Un Qname è composto da un prefisso che deve corrispondere al prefisso di un namespace, dichiarato in precedenza, seguito da un NCName;
- gli `standard-attributes` sono:

```
name="NCName"?
suppressJoinFailure="yes|no"?
```

- gli `standard-elements` sono:

```
<targets>?
 <joinCondition
 expressionLanguage="anyURI"?>?
     bool-expr
 </joinCondition>
 <target linkName="NCName"/>+
</targets>
<sources>?
 <source linkName="NCName">+
   <transitionCondition
   expressionLanguage="anyURI"?>?
       bool-expr
   </transitionCondition>
 </source>
</sources>
```

Vediamo, adesso, le attività di base:

- <receive> attende un messaggio da parte di un servizio. Inoltre ha la possibilità di iniziare un processo, creando l'istanza del processo stesso;

```
<receive partnerLink="NCName"
 portType="QName"?
 operation="NCName"
 variable="BPELVariableName"?
 createInstance="yes|no"?
 messageExchange="NCName"?
 standard-attributes>
 standard-elements
  <correlations>?
    <correlation set="NCName"
     initiate="yes|join|no"?>+
  </correlations>
  <fromParts>?
   <fromPart part="NCName"
    toVariable="BPELVariableName"/>+
  </fromParts>
```

```
</receive>
```

- <reply> invia un messaggio di risposta al partner che ha inviato il messaggio all'attività <receive>;

```
<reply partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   variable="BPELVariableName"?
   faultName="QName"?
   messageExchange="NCName"?
   standard-attributes>
 standard-elements
 <correlations>?
   <correlation set="NCName"
    initiate="yes|join|no"?>+
 </correlations>
 <toParts>?
   <toPart part="NCName"
    fromVariable="BPELVariableName"/>+
 <toParts>
</reply>
```

- <invoke> invoca un web service per eseguire un'operazione;

```
<invoke partnerLink="NCName"
   portType="QName"?
   operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?
   standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="NCName"
     initiate="yes|join|no"?
     pattern=
      "request|response|request-response"?/>+
```

```
  </correlations>
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    faultMessageType="QName"?
    faultElement="QName"?>*
      activity
  </catch>
  <catchAll>?
      activity
  </catchAll>
  <compensationHandler>?
      activity
  </compensationHandler>
  <toParts>?
    <toPart part="NCName"
     fromVariable="BPELVariableName" />+
  </toParts>
  <fromParts>?
    <fromPart part="NCName"
     toVariable="BPELVariableName" />+
  </fromParts>
</invoke>
```

- <validate> convalida i valori delle variabili, confrontandoli con il documento WSDL associato;

```
<validate variables="BPELVariableNames"
 standard-attributes>
   standard-elements
</validate>
```

- <wait> sospende il processo per un determinato periodo di tempo o fino a quando non si verifica un determinato evento;

```
<wait standard-attributes>
   standard-elements
   (
```

```
   <for expressionLanguage="anyURI"?>
     duration-expr
   </for>
   |
   <until expressionLanguage="anyURI"?>
     deadline-expr
   </until>
   )
</wait>
```

- <compensateScope> lancia la compensazione di una attività <scope> completata con successo;

```
<compensateScope target="NCName"
 standard-attributes>
   standard-elements
</compensateScope>
```

- <compensate> lancia la compensazione di tutte le attività <scope> completate con successo;

```
<compensate standard-attributes>
   standard-elements
</compensate>
```

- <extensionActivity> permette di creare nuove attività.

```
<extensionActivity>
   <anyElementQName standard-attributes>
      standard-elements
   </anyElementQName>
</extensionActivity>
```

Le attività short-lived, sono quelle attività che hanno un'esecuzione di breve durata e possono essere viste come operazioni atomiche:

- <assign> può essere usata per copiare valori da una variabile ad un'altra oppure per assegnare nuovi valori;

```
<assign validate="yes|no"? standard-attributes>
   standard-elements
   (
   <copy keepSrcElementName="yes|no"?
    ignoreMissingFromData="yes|no"?>
      from-spec
      to-spec
   </copy>
   |
   <extensionAssignOperation>
      assign-element-of-other-namespace
   </extensionAssignOperation>
   )+
</assign>
```

- <empty> non fa niente ed è paragonabile ad un no operation;

```
<empty standard-attributes>
   standard-elements
</empty>
```

- <exit> termina il processo immediatamente;

```
<exit standard-attributes>
   standard-elements
</exit>
```

- <throw> lancia un segnale di errore;

```
<throw faultName="QName"
   faultVariable="BPELVariableName"?
   standard-attributes>
   standard-elements
</throw>
```

- <rethrow> passa il compito di gestire un errore precedentemente segnalato al fault handler che lo racchiude.

```
<rethrow standard-attributes>
   standard-elements
</rethrow>
```

Le attività strutturate possono contenere al loro interno attività appartenenti a qualsiasi macrogruppo:

- <sequence> esegue le attività contenute al suo interno in ordine lessicale;

```
<sequence standard-attributes>
   standard-elements
   activity+
</sequence>
```

- <flow> esegue tutte le attività contenute al suo interno in parallelo;

```
<flow standard-attributes>
   standard-elements
   <links>?
      <link name="NCName" />+
   </links>
   activity+
</flow>
```

- <while> esegue una data attività ripetutamente, finché si verifica una certa condizione;

```
<while standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>
      bool-expr
   </condition>
   activity
</while>
```

- <repeatUntil> si comporta come <while> ma l'attività è eseguita almeno una volta perché la condizione viene verificata alla fine dell'esecuzione;

```
<repeatUntil standard-attributes>
   standard-elements
   activity
   <condition expressionLanguage="anyURI"?>
     bool-expr
   </condition>
</repeatUntil>
```

- <pick> esegue un'attività scelta in base al verificarsi di un determinato evento. Può iniziare un processo, creando un'istanza di un servizio;

```
<pick createInstance="yes|no"?
 standard-attributes>
   standard-elements
   <onMessage partnerLink="NCName"
      portType="QName"? operation="NCName"
      variable="BPELVariableName"?
      messageExchange="NCName"?>+
      <correlations>?
         <correlation set="NCName"
          initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
         <fromPart part="NCName"
          toVariable="BPELVariableName" />+
      </fromParts>
      activity
   </onMessage>
   <onAlarm>*
      (
      <for expressionLanguage="anyURI"?>
         duration-expr
      </for>
```

```
      |
      <until expressionLanguage="anyURI"?>
         deadline-expr
      </until>
      )
      activity
   </onAlarm>
</pick>
```

- <if> esegue un'attività se la condizione è vera, se la condizione è falsa esegue un'altra attività;

```
<if standard-attributes>
   standard-elements
   <condition expressionLanguage="anyURI"?>
      bool-expr
   </condition>
   activity
   <elseif>*
      <condition expressionLanguage="anyURI"?>
         bool-expr
      </condition>
      activity
   </elseif>
   <else>?
      activity
   </else>
</if>
```

- <forEach> esegue l'attività <scope>, presente al suo interno, per un determinato numero di volte. Le iterazioni possono essere svolte contemporaneamente in parallelo;

```
<forEach counterName="BPELVariableName"
   parallel="yes|no"
   standard-attributes>
   standard-elements
```

```
   <startCounterValue
    expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </startCounterValue>
   <finalCounterValue
    expressionLanguage="anyURI"?>
      unsigned-integer-expression
   </finalCounterValue>
   <completionCondition>?
      <branches expressionLanguage="anyURI"?
         successfulBranchesOnly="yes|no"?>?
         unsigned-integer-expression
      </branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

- <scope> può essere vista come un sottoprocesso con le proprie variabili ed i propri partnerlink: le sezioni definite al suo interno hanno solo valore locale.

```
<scope isolated="yes|no"?
   exitOnStandardFault="yes|no"?
   standard-attributes>
   standard-elements
   <partnerLinks>?
      ...
   </partnerLinks>
   <variables>?
      ...
   </variables>
   <correlationSets>?
      ...
   </correlationSets>
   <faultHandlers>?
      ...
   </faultHandlers>
```

```
    <compensationHandler>?
        ...
    </compensationHandler>
    <terminationHandler>?
        ...
    </terminationHandler>
    <eventHandlers>?
        ...
    </eventHandlers>
    activity
</scope>
```

3.1.2 Esempio di un processo WS-BPEL

Come esempio concreto di processo scritto in linguaggio WS-BPEL vediamo ora un processo per la richiesta di un prestito. Il processo attende una richiesta di prestito di un certo valore. Se l'ammontare è minore di 10000 euro, viene richiamato il web service di consulenza, il quale decide se il richiedente fa parte dei soggetti a basso o ad alto rischio: se è a basso rischio, il prestito viene confermato. Nel caso in cui il richiedente sia ad alto rischio oppure desideri un prestito maggiore di 10000 euro, viene richiamato il web service del finanziatore che decide se elargire il prestito oppure no. Inoltre, nel caso accada qualcosa di inaspettato durante l'esecuzione del processo, è prevista come risposta un codice numerico di errore.
Definiamo, innanzitutto, lo XML Schema per il codice di errore:

XML Schema - codice errore

```
<?xml version=¨1.0¨?>
<xs:schema xmlns=¨http://www.miatesi.it/messaggioErrore¨
  xmlns:xs=¨http://www.w3.org/2001/XMLSchema¨
  targetNamespace=¨http://www.miatesi.it/messaggioErrore¨
  elementFormDefault=¨qualified¨
    <xs:element name=¨codice¨ type=¨xs:integer¨/>
</xs:schema>
```

Il listato è molto semplice: dichiara un nuovo elemento "codice" di tipo intero. Il documento WSDL è molto più corposo; iniziamo con la dichiarazione dei namespace, dei tipi e dei messaggi:

WSDL del Processo Prestito - parte I

```
<?xml version=¨1.0¨?>
<wsdl:definitions
  targetNamespace=¨http://www.miatesi.it¨
  xmlns:ens=¨http://www.miatesi.it/messaggioErrore¨
  xmlns:plnk=¨http://docs.oasis-open.org/wsbpel/2.0/plnktype¨
  xmlns:soap=¨http://schemas.xmlsoap.org/wsdl/soap/¨
  xmlns:tns=¨http://www.miatesi.it¨
  xmlns:wsdl=¨http://schemas.xmlsoap.org/wsdl/¨
  xmlns:xsd=¨http://www.w3.org/2001/XMLSchema¨>
    <wsdl:types>
      <xsd:schema elementFormDefault=¨qualified¨>
        <xsd:import
        namespace=¨http://www.miatesi.it/messaggioErrore¨
        schemaLocation=¨messaggioErrore.xsd¨/>
      </xsd:schema>
    </wsdl:types>
    <wsdl:message name=¨messaggioRichiestaCredito¨>
      <wsdl:part name=¨nome¨ type=¨xsd:string¨/>
      <wsdl:part name=¨cognome¨ type=¨xsd:string¨/>
      <wsdl:part name=¨valore¨ type=¨xsd:integer¨/>
    </wsdl:message>
    <wsdl:message name=¨messaggioApprovazione¨>
      <wsdl:part name=¨accetta¨ type=¨xsd:string¨/>
    </wsdl:message>
    <wsdl:message name=¨messaggioLivelloDiRischio¨>
      <wsdl:part name=¨livello¨ type=¨xsd:string¨/>
    </wsdl:message>
    <wsdl:message name=¨messaggioErrore¨>
      <wsdl:part name=¨codiceErrore¨ element=¨ens:codice¨/>
    </wsdl:message>
```

Nella parte iniziale, dichiariamo i namespace; in particolare, quello con prefisso plnk serve per utilizzare i partnerlink di WS-BPEL. L'unico tipo di dato lo definiamo importando l'XML Schema per il codice di errore; gli altri dati utilizzano tipi base (interi e stringhe). I messaggi di cui abbiamo bisogno sono: un messaggio per la richiesta del credito che conterrà il nome, il cognome e l'ammontare desiderato, un messaggio per la risposta alla richiesta, un messaggio contenente il livello di rischio e un messaggio per l'eventuale errore di esecuzione.
Proseguiamo la stesura del file WSDL, definendo i portType:

WSDL del Processo Prestito - parte II

```
<wsdl:portType name="servizioPrestitoPT">
  <wsdl:operation name="richiesta">
    <wsdl:input message="tns:messaggioRichiestaCredito"/>
    <wsdl:output message="tns:messaggioApprovazione"/>
    <wsdl:fault name="impossibileGestireRichiesta"
    message="tns:messaggioErrore"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="gestioneRischioPT">
  <wsdl:operation name="controllo">
    <wsdl:input message="tns:messaggioRichiestaCredito"/>
    <wsdl:output message="tns:messaggioLivelloDiRischio"/>
    <wsdl:fault name="erroreProcesso"
    message="tns:messaggioErrore"/>
  </wsdl:operation>
</wsdl:portType>
<wsdl:portType name="accettaPrestitoPT">
  <wsdl:operation name="approva">
    <wsdl:input message="tns:messaggioRichiestaCredito"/>
    <wsdl:output message="tns:messaggioApprovazione"/>
    <wsdl:fault name="erroreProcesso"
    message="tns:messaggioErrore"/>
  </wsdl:operation>
</wsdl:portType>
```

I tre portType sono tutti di tipo request-response e dichiarano un'operazione ciascuno: le operazioni `richiesta` e `approva` hanno gli stessi messaggi di input e output ma una viene utilizzata per la risposta al cliente, quando richiede un prestito minore di 10000 euro e il suo livello di rischio è basso, l'altra, per la richiesta al finanziatore; l'operazione `controllo` si occupa di richiedere il livello di rischio del cliente.
Adesso scriviamo gli elementi <binding> e <ervice> relativi alle chiamate ai web service del finanziatore e della consulenza:

WSDL del Processo Prestito - parte III

```
<wsdl:binding name="accettaPrestitoBinding"
type="tns:accettaPrestitoPT">
  <soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="approva">
    <soap:operation soapAction="" style="rpc"/>
    <wsdl:input>
      <soap:Body use="encoded" namespace="urn:accettaprestito"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:input>
    <wsdl:output>
      <soap:Body use="encoded" namespace="urn:accettaprestito"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="AccettaPrestito">
  <wsdl:port name="accettaPrestitoPort"
  binding="tns:accettaPrestitoBinding">
    <soap:address
    location="http://miatesi.it:8080/servizi/FinanziatoreWebService"/>
  </wsdl:port>
</wsdl:service>
```

Abbiamo fatto un collegamento al portType `accettaPrestitoPT`, in modo da permettere al protocollo SOAP di utilizzare i messaggi dell'operazione `approva` e poi abbiamo specificato dove è collocato il web service del

finanziatore. In egual modo, ci comportiamo per creare il collegamento con il portType `gestioneRischioPT` e fornire l'indirizzo del web service di consulenza:

WSDL del Processo Prestito - parte IV

```
<wsdl:binding name="gestioneRischioBinding"
type="tns:gestioneRischioPT">
  <soap:binding style="rpc"
  transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="controllo">
    <soap:operation soapAction="" style="rpc"/>
    <wsdl:input>
      <soap:Body use="encoded" namespace="urn:consulenteprestito"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:input>
    <wsdl:output>
      <soap:Body use="encoded" namespace="urn:consulenteprestito"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="ConsulenzaPrestito">
  <wsdl:port name="consulenzaPrestitoPort"
  binding="tns:gestioneRischioBinding">
    <soap:address
    location="http://www.miatesi.it:8080/servizi/ConsulenteWebService"/>
  </wsdl:port>
</wsdl:service>
```

I partnerLinkType, dei quali tra poco vedremo la dichiarazione, rappresentano un'estensione del documento WSDL, definita dal namespace `plnk` inserito nella parte iniziale del documento. Un partnerLinkType definisce il rapporto che c'è tra due servizi, stabilendo il ruolo corrispondente a ciascuno di essi: quello del processo e quello del web service. Nel nostro caso, tutti i partnerLinkType hanno un solo ruolo perché c'è un solo partner che offre il servizio. Per conoscere quali tipi di messaggi utilizzare, ad ogni ruolo è associato un portType.

WSDL del Processo Prestito - parte V

```
<plnk:partnerLinkType name="partnerPrestitoLT">
  <plnk:role name="servizioPrestito" portType="tns:servizioPrestitoPT"/>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="approvaPrestitoLT">
  <plnk:role name="finanziatore" portType="tns:accettaPrestitoPT"/>
</plnk:partnerLinkType>
<plnk:partnerLinkType name="gestioneRischioLT">
  <plnk:role name="consulente" portType="tns:gestioneRischioPT"/>
</plnk:partnerLinkType> </wsdl:definitions>
```

Definiti i file XML Schema e WSDL, andiamo a vedere un possibile processo WS-BPEL in grado di utilizzarli per offrire un servizio di accettazione prestiti.

Come detto, anche il linguaggio WS-BPEL ha una sintassi XML-like; iniziamo con dichiarare i namespace all'interno del tag <process> ed importare il documento WSDL appena creato. La funzione dell'attributo `suppressJoinFailure` la spiegheremo in seguito.

Processo WS-BPEL - namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:pns="http://www.miatesi.it/prestito.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ProcessoPrestito"
suppressJoinFailure="yes"
targetNamespace="http://www.miatesi.it/prestito.bpel">
  <import importType="http://schemas.xmlsoap.org/wsdl/"
  location="prestito.wsdl"
  namespace="http://www.miatesi.it/prestito.wsdl"/>
```

La definizione dei partnerLink fa riferimento ai partnerLinkTypes del file WSDL. I partnerLink definiscono il ruolo dei web service, tramite l'attributo `partnerRole`, e quello del processo WS-BPEL stesso, tramite `myRole`.

Processo WS-BPEL - partnerLinks

```
<partnerLinks>
  <partnerLink myRole="servizioPrestito"
  name="cliente" partnerLinkType="pns:partnerPrestitoLinkType"/>
  <partnerLink name="finanziatore"
  partnerLinkType="pns:approvaPrestitoLinkType"
  partnerRole="finanziatore"/>
  <partnerLink name="consulente"
  partnerLinkType="pns:gestioneRischioLinkType"
  partnerRole="consulente"/>
</partnerLinks>
```

Le variabili utili per il processo sono tre, corrispondenti ai tipi di messaggi definiti nel documento WSDL; la variabile per il messaggio di errore è definita all'interno di <faultHandlers>.

Processo WS-BPEL - variabili e faultHandlers

```
<variables>
  <variable messageType="pns:messaggioRichiestaCredito"
  name="richiesta"/>
  <variable messageType="pns:messaggioLivelloDiRischio"
  name="rischio"/>
  <variable messageType="pns:messaggioApprovazione"
  name="approvato"/>
</variables>
<faultHandlers>
  <catch faultMessageType="pns:messaggioErrore"
  faultName="pns:erroreProcesso" faultVariable="errore">
    <reply faultName="pns:impossibileGestireRichiesta"
    operation="richiesta" partnerLink="cliente"
    portType="pns:servizioPrestitoPT" variable="errore"/>
  </catch>
</faultHandlers>
```

L'elemento <faultHandlers> si occupa di gestire le situazioni di errore; nel nostro esempio, invia, come risposta alla richiesta di prestito, un numero che

rappresenta il codice di errore. Riesce a farlo mediante l'uso dell'elemento <catch>, il quale intercetta tutti gli errori il cui nome è dato dal valore dell'attributo `faultName`. Se l'attributo `faultName` non è specificato, il tag intercetta tutti gli errori che sono del tipo `faultMessageType`, inoltre, è possibile definire più istanze <catch>. Il tag <catchAll>, non presente nell'esempio, si occupa di gestire tutti gli errori che non sono stati catturati dai <catch> più specifici.

Siamo arrivati alla parte in cui si inseriscono le attività del processo. Vogliamo che le attività siano disponibili contemporaneamente, quindi le inseriamo in un'attività <flow> che, come già detto, esegue in parallelo tutte le attività presenti al suo interno. I *link* connettono due attività e ne controllano l'ordine di esecuzione tramite la creazione di dipendenze; sono presenti solo all'interno dell'attività <flow>. Per controllare l'ordine di esecuzione, si utilizzano i tag <source> e <target>: un'attività, che ha all'interno del tag <target> un *link* con nome "nomeLink", potrà essere eseguita solo dopo la terminazione dell'attività che contiene, nel tag <source>, il *link* con lo stesso nome. Per il nostro esempio abbiamo bisogno di sei *link*.

Processo WS-BPEL - <flow> e <links>

```
<flow>
  <links>
    <link name=¨richiesta-consulente¨/>
    <link name=¨richiesta-finanziatore¨/>
    <link name=¨consulente-messaggio¨/>
    <link name=¨consulente-finanziatore¨/>
    <link name=¨messaggio-risposta¨/>
    <link name=¨finanziatore-risposta¨/>
  </links>
```

Il processo resta in attesa di una richiesta di prestito; l'attività <receive> è ciò di cui abbiamo bisogno per questo scopo. Dato che il processo inizia il suo percorso solo dopo che arriva una tale richiesta, l'attività avrà l'attributo `createInstance`="yes"; inoltre controlla se la richiesta è minore o maggiore di 10000 euro, attivando il *link richiesta-consulente*, nel primo caso, e il *link richiesta-finanziatore*, nell'altro. Le espressioni per la valutazione del credito sono scritte in XPath [19].

Processo WS-BPEL - <receive>

```
<receive createInstance="yes" operation="richiesta"
partnerLink="cliente" portType="pns:servizioPrestitoPT"
variable="richiesta">
  <sources>
    <source linkName="richiesta-consulente">
      <transitionCondition>
        ($richiesta.valore &lt; 10000)
      </transitionCondition>
    </source>
    <source linkName="richiesta-finanziatore">
      <transitionCondition>
        ($richiesta.valore &gt;= 10000)
      </transitionCondition>
    </source>
  </sources>
</receive>
```

Nel caso in cui la richiesta è inferiore ai 10000 euro, il processo deve passare i dati del cliente al web service adibito alla consulenza. Per le chiamate ai web service, usiamo l'attività <invoke> inviando, come dato, la richiesta del prestito e aspettando, come risposta, il livello di rischio del cliente, utilizzando l'operazione `controllo`. Dato che non possiamo utilizzare il web service finché non abbiamo una richiesta di prestito, inseriamo un tag <target> per il *link richiesta-consulente*. Poi, se il livello di rischio è basso, attiviamo il *link consulente-messaggio*, altrimenti attiviamo il *link consulente-finanziatore*. Se, per il web service, il livello di rischio del cliente è basso, allora il processo può accettare di elargire il prestito, dando alla variabile `approvato` il valore "sì". Assegnare valori alle variabili è compito dell'attività <assign>, mediante l'utilizzo dei tag <from>, nel quale va messo il valore da copiare, e <to>, contenente la variabile di destinazione; questi tag sono contenuti, a loro volta, nell'elemento <copy>.

Anche questa attività ha delle restrizioni per poter essere eseguita: basterà mettere il *link richiesta-consulente*, all'interno del suo tag <target>. Infine, in <source>, inseriamo il *link messaggio-risposta*.

Processo WS-BPEL - chiamata al web service di consulenza

```
<invoke inputVariable="richiesta" operation="controllo"
outputVariable="rischio" partnerLink="consulente"
portType="pns:gestioneRischioPT">
  <targets>
    <target linkName="richiesta-consulente"/>
  </targets>
  <sources>
    <source linkName="consulente-messaggio">
      <transitionCondition>
        ($rischio.livello = 'basso')
      </transitionCondition>
    </source>
    <source linkName="consulente-finanziatore">
      <transitionCondition>
        ($rischio.livello != 'basso')
      </transitionCondition>
    </source>
  </sources>
</invoke>
<assign>
  <targets>
    <target linkName="consulente-messaggio"/>
  </targets>
  <sources>
    <source linkName="messaggio-risposta"/>
  </sources>
  <copy>
    <from>'sì'</from>
    <to part="accetta" variable="approvato"/>
  </copy>
</assign>
```

L'altro web service, quello del finanziatore, verrà eseguito quando l'ammontare del prestito è superiore ai 10000 euro oppure, quando il livello di rischio del cliente è stato valutato alto dal web service del consulente. Allora,

questa attività deve avere due elementi <target>: uno per il *link richiesta-finanziatore* e l'altro per il *link consulente-finanziatore*. Adesso, possiamo spiegare il significato dell'attributo `suppressJoinFailure` utilizzato nella dichiarazione del processo. Questo attributo, se impostato a "no", permette l'esecuzione di una attività solamente se tutte le attività, che hanno almeno un *link* nei tag <sources> presente nei suoi elementi <target>, sono state completate. Nel nostro caso, dovrebbero essere completate sia l'attività <receive>, con richiesta di prestito maggiore di 10000 euro, sia la chiamata al web service di consulenza, con risposta di livello di rischio alta: questo non può mai accadere e il processo restituirebbe un errore. Impostando l'attributo a "yes", si richiede che almeno una attività (tra quelle collegate tramite *link*) sia completata prima di poter eseguire l'attività di nostro interesse. Il web service del finanziatore viene utilizzato, come quello del consulente, tramite l'attività <invoke>: inviamo i dati del prestito richiesto e attendiamo la risposta, di accettazione oppure di rifiuto, tramite l'operazione `approva`. Aggiungiamo inoltre un elemento <source>, contenente il *link finanziatore-risposta*.

Processo WS-BPEL - chiamata al web service del finanziatore

```
<invoke inputVariable="richiesta" operation="approva"
outputVariable="approvato" partnerLink="finanziatore"
portType="pns:accettaPrestitoPT">
  <targets>
    <target linkName="richiesta-finanziatore"/>
    <target linkName="consulente-finanziatore"/>
  </targets>
  <sources>
    <source linkName="finanziatore-risposta"/>
  </sources>
</invoke>
```

A questo punto, siamo arrivati alla fine del nostro processo. Dobbiamo solo rendere noto al cliente l'esito della richiesta del prestito. Per rispondere ad una attività di <receive>, in caso di `request-response` è obbligatorio utilizzare l'attività <reply>: il nostro esempio utilizza l'operazione `richiesta`, per restituire il valore della variabile `approvato`. Il processo fornirà una

risposta positiva se il cliente ha chiesto un prestito minore di 10000 euro ed ha un livello di rischio "basso": in questo caso abbiamo bisogno del tag <target> contenente il *link messaggio-risposta*. Nel caso in cui sia necessaria la decisione del finanziatore, abbiamo bisogno di un elemento <target> contenente il *link fianziatore-risposta*. Infine, chiudiamo l'attività <flow> ed il processo.

Processo WS-BPEL - risposta al cliente

```
<reply operation=¨richiesta¨ partnerLink=¨cliente¨
portType=¨pns:servizioPrestitoPT¨ variable=¨approvato¨>
  <targets>
    <target linkName=¨messaggio-risposta¨/>
    <target linkName=¨finanziatore-risposta¨/>
  </targets>
</reply>
</flow>
</process>
```

CAPITOLO 4

ENGINE WS-BPEL

Per usufruire dei servizi offerti da un processo WS-BPEL, tale processo deve essere, in qualche modo, reso eseguibile: questa azione viene chiamata *deploying* del processo. I server, che permettono il deploy di processi WS-BPEL, sono conosciuti come engine WS-BPEL. Solitamente, ogni engine offre anche un ambiente integrato di sviluppo (IDE), permettendo di creare processi WS-BPEL mediante interfacce grafiche: in questo modo, la conoscenza approfondita della sintassi del linguaggio WS-BPEL può passare in secondo piano, dato che la stesura del codice avverrà tramite l'IDE stesso. Passiamo ad esaminare alcuni degli engine WS-BPEL tra quelli disponibili gratuitamente.

4.1 ActiveBPEL Engine

ActiveBPEL Engine [1] è il motore open source della Active Endpoints, sviluppato in JAVA. Per essere eseguito, necessita di un qualunque server web, che supporti la tecnologia servlet [9] come Apache Tomcat, e il Java Development Kit (JDK) appropriato: per esempio, se utilizziamo Tomcat 5.5 avremo bisogno di JDK 1.5. L'engine può essere scaricato dall'indirizzo *http://activevos.com/community-open-source-terms-conditions.php*: al momento della stesura di questo documento, è disponibile la versione 4.1 del motore ma a breve sarà ultimata la versione 5.0. Dato che l'engine è sviluppato in JAVA, può essere eseguito su qualunque piattaforma e, per semplificare l'installazione, sono disponibili due script: un file con estensione

`.bat`, per i sistemi operativi Windows, e uno con estensione `.sh`, per i sistemi operativi Unix-like. Questi script, però, sono utili solo se abbiamo installato Tomcat come server web, dato che, al loro interno, fanno riferimento alla variabile di ambiente `CATALINA_HOME`, la quale contiene il path di questo server. L'ambiente di sviluppo, di supporto a questo engine, si chiama ActiveBPEL Designer ed è disponibile solo per i sistemi operativi Windows XP e Windows 2000 e può essere recuperato all'indirizzo:
http://www.activebpel.org/foundation/abl_jboss_4.html.
Dopo aver installato l'engine e avviato il server web, possiamo vedere la pagina di gestione, digitando il seguente URL in un qualunque browser:
http://localhost:8080/BpelAdmin.
L'home page (figura 4.1) visualizza i seguenti dettagli:

- la data e l'ora di avvio del motore;
- il nome dei processi di cui è stato fatto il deploy; l'engine è impostato per aggiornare questo numero ogni 20 secondi;
- la configurazione dell'engine; di default è `In-Memory`: significa che tutti i processi si fermano, quando viene fermato il motore;
- lo stato del motore: avviato o fermo;
- la versione dell'engine.

Il menù, presente sulla sinistra, ci permette di fare alcune operazioni:

- modificare la configurazione;
- vedere i dettagli dei processi di cui è stato fatto il deploy, possiamo visualizzare il sorgente WS-BPEL, il documento WSDL e lo schema grafico del processo;
- sospendere o fermare i processi avviati;
- analizzare gli step di esecuzione di un processo avviato;
- attivare e scaricare il logging dei processi avviati o completati.

Per fermare o avviare il motore, c'è l'apposito pulsante nella parte destra dello schermo. Nel menù di configurazione, è consigliabile impostare il logging level su "execution". Questa impostazione permette di seguire il percorso fatto da un processo, scegliendolo tra quelli presenti nella lista "Active Processes".

Figura 4.1: ActiveBPEL - pagina di gestione

4.1.1 Deploying di un processo

Come detto in precedenza, l'operazione di rendere disponibile l'esecuzione di un processo, su un engine WS-BPEL, è detta *deploying*. Ogni engine ha le proprie regole per il deploy di un processo. Qui vediamo in dettaglio quelle di ActiveBPEL engine.

Questo engine richiede la creazione di un archivio JAR, contenente tutti i file necessari per la completa descrizione del processo. La struttura dell'archivio è fissata e deve contenere le seguenti tre directory:

- bpel: deve contenere il file `.bpel` che descrive il comportamento del processo;
- META-INF: deve racchiudere il file *wsdlCatalog.xml*, il quale elenca i file WSDL e XML Schema utilizzati dal processo;
- wsdl: deve contenere i file WSDL e XML Schema utili al processo.

All'esterno delle tre directory, deve esserci un file con estensione `.pdd`, il quale rappresenta il *Process Deployment Descriptor* utilizzato per contenere delle direttive per l'engine.

Struttura di wsdlCatalog.xml

Il file *wsdlCatalog.xml* deve contenere un tag <wsdlEntry> per ogni documento WSDL utilizzato dal processo ed un tag <schemaEntry> per ogni file XML Schema.
Entrambi i tag devono specificare gli attributi `location` e `classpath`. Vediamo il file *wsdlCatalog.xml*, per il nostro ProcessoPrestito:

ActiveBPEL engine - wsdlCatalog.xml

```
<?xml version=¨1.0¨ encoding=¨UTF-8¨ ?>
<catalog
xmlns=¨http://schemas.active-endpoints.com/catalog/2006/07/catalog.xsd¨>
  <wsdlEntry location=¨project:/ProcessoPrestito/prestito.wsdl¨
  classpath=¨wsdl/prestito.wsdl¨/>
  <schemaEntry location=¨project:/ProcessoPrestito/messaggioErrore.xsd¨
  classpath=¨wsdl/messaggioErrore.xsd¨/>
</catalog>
```

Struttura del Process Deployment Descriptor

ActiveBPEL definisce il seguente schema per il file `.pdd`:

```
<process location="bpel/Tutorial/tutorial.bpel"
   persistenceType="none"
     name="Qname"
     xmlns="http://schemas.active-
       endpoints.com/pdd/2005/09/pdd.xsd"
     xmlns:bpelns="anyNamespace"
     xmlns:wsa="http://
      schemas.xmlsoap.org/ws/2003/03/addressing">
  <partnerLinks>
    <partnerLink name="ncname">
      <partnerRole
          endpointReference=
          "static|dynamic|invoker|principal"
          customInvokeHandler="java:class:args"?>
        [... ws-addressing....]?
      </partnerRole>?
      <myRole service="name"
          allowedRoles="namelist"?
          binding="MSG|RPC|EXTERNAL"/>?
    </partnerLink>+
  </partnerLinks>
  <References>
    <wsdl namespace="uri" location="uri"/>+
  </References>?
</process>
```

Il `ws-addressing` [16] è uno standard per costruire `endpoint reference` utilizzati per rappresentare l'indirizzo di un web service. Il Process Deployment Descriptor del nostro esempio è il seguente:

Process Deployment Descriptor - parte I

```
<?xml version="1.0" encoding="UTF-8"?>
<process
xmlns="http://schemas.active-endpoints.com/pdd/2006/08/pdd.xsd"
xmlns:bpelns="http://www.miatesi.it/prestito.bpel"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
location="ProcessoPrestito/ProcessoPrestito.bpel"
name="bpelns:ProcessoPrestito" persistenceType="none">
  <partnerLinks>
    <partnerLink name="finanziatore">
      <partnerRole endpointReference="static">
        <wsa:EndpointReference xmlns:s="http://www.miatesi.it"
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
          <wsa:Address>
            http://miatesi.it:8080/servizi/FinanziatoreWebService
          </wsa:Address>
          <wsa:ServiceName PortName="accettaPrestitoPort">
            s:AccettaPrestito
          </wsa:ServiceName>
          </wsa:EndpointReference>
      </partnerRole>
    </partnerLink>
    <partnerLink name="consulente">
      <partnerRole endpointReference="static">
        <wsa:EndpointReference xmlns:s="http://www.miatesi.it"
        xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
          <wsa:Address>
            http://www.miatesi.it:8080/servizi/ConsulenteWebService
          </wsa:Address>
          <wsa:ServiceName PortName="consulenzaPrestitoPort">
            s:ConsulenzaPrestito
          </wsa:ServiceName>
        </wsa:EndpointReference>
      </partnerRole>
    </partnerLink>
```

Process Deployment Descriptor - parte II

```
<partnerLink name=¨cliente¨>
  <myRole allowedRoles=¨¨ binding=¨RPC¨
  service=¨servizioPrestito¨/>
</partnerLink>
</partnerLinks>
<references>
  <wsdl location=¨project:/ProcessoPrestito/prestito.wsdl¨
  namespace=¨http://www.miatesi.it¨/>
</references>
</process>
```

Dichiarati i namespace, definiamo all'interno degli elementi <parterLink>, tramite `ws-addressing`, gli `endpoint reference` che permettono di conoscere gli indirizzi e le porte utili per comunicare con i web service utilizzati dal processo. Infine, all'interno di <references> dichiariamo il path del file WSDL utilizzato dal processo.

Create le directory e copiati i giusti file all'interno di esse, possiamo generare il file JAR, di estensione `.bpr` (Business Process Archive), con il seguente comando: *jar cf prestito.bpr *.pdd META-INF bpel wsdl*. L'archivio deve essere copiato nella directory `bpr` di Tomcat: in questo modo, ActiveBPEL farà automaticamente il deploy del processo.
Utilizzando l'ambiente di sviluppo ActiveBPEL Designer, il deploy è semplificato, dato che l'IDE si occupa di creare tutti i file necessari per l'archivio `bpr` e il relativo invio al web server.

4.2 Apache ODE

Apache ODE [3] è l'engine open source della Apache Software Fondation e, come ActiveBPEL, è sviluppato in JAVA: per questo motivo, anch'esso necessita di un server web che supporti i servlet, tipo Apache Tomcat. Il motore è scaricabile all'indirizzo: *http://ode.apache.org/getting-ode.html*. Durante la stesura di questa tesi, è disponibile la versione 1.1.1. Sono presenti due distribuzioni, una come web application, tramite un web archive (un web archive è un file con estensione `.war`, assolutamente analogo ad un archivio JAVA, che contiene tutti i file di una applicazione web già or-

Figura 4.2: Apache ODE - pagina di gestione

ganizzati secondo la corretta struttura di deployment) e una per la Java Business Integration (distribuzione JBI); nel seguito della trattazione, faremo riferimento alla distribuzione WAR. L'installazione è molto semplice, una volta decompresso il pacchetto, bisogna copiare il file *ode.war* nella directory *webapps* di Tomcat. A differenza di ActiveBPEL, Apache ODE non ha un ambiente IDE proprietario, però esiste un ambiente di sviluppo Business Process Management Suite (BPMS) open source che utilizza ODE come engine WS-BPEL. Si tratta di *Intalio|Designer* [6], scaricabile all'indirizzo *http://bpms.intalio.com/downloads.html*.

A questo punto, avviato il web server, digitando in un browser l'indirizzo *http://127.0.0.1:8080/ode*, verrà visualizzata la pagina di benvenuto di Apache Axis2 (figura 4.2). Axis2 [4] è un ulteriore engine sul quale si basa il motore ODE. Nella home page, sono presenti tre link:

- Services: permette di vedere i processi dei quali è stato fatto il deploy;
- Validate: controlla se il sistema funziona correttamente;
- Administration: conduce alla console di configurazione di Axis2, i dati di default per il login sono: "admin", per l'username, e "axis2", per la password.

Il link Services, se cliccato, darà un errore; il progetto sembra essere ancora in una fase preliminare: controllando attentamente le eccezioni descritte da

JAVA, si vede che non riesce a trovare il file *commons-fileupload-1.1.1.jar*. Infatti, nella directory *webapps/ode/WEB-INF/lib* di Tomcat, c'è una versione precedente del file: *commons-fileupload-1.0.jar*; per correggere l'errore, il file mancante può essere copiato nella directory, scaricandolo da *http://www.ibiblio.org/maven/commons-fileupload/jars/*.

Nella console di configurazione, è possibile attivare o disattivare i servizi presenti. Per avere maggiori informazioni riguardo ai servizi dei quali è stato fatto il deploy, ODE non ha strumenti già pronti ma mette ha disposizione delle funzionalità di *management API*. Inoltre, sono presenti due processi di default che utilizzano i metodi offerti dalle *API*: *ProcessManagement* e *InstanceManagement* [5]. Come per qualsiasi altro web service, possiamo richiedere le informazioni a questi due processi tramite messaggi SOAP. Alcune delle operazioni eseguibili dal service *InstanceManagemet* sono:

- `listAllInstances`: elenca tutte le instanze dei processi presenti, indicandone: iid (identificatore dell'istanza), pid (identificatore del processo), stato, nome e ora di avvio;
- `suspend`: sospende l'esecuzione del processo, indicato tramite il proprio iid (identificatore dell'istanza);
- `resume`: riprende l'esecuzione del processo;
- `terminate`: termina l'esecuzione del processo;
- `fault`: il processo termina, riportando l'errore generato. Se nel processo è definito un opportuno <faultHandler>, l'esecuzione continua con la gestione dell'errore;
- `delete`: elimina una o più istanze di processo completate.

Vediamo un esempio di messaggio SOAP da inviare per terminare un'istanza di processo:

Ode Engine - messaggio SOAP per l'operazione terminate

```
<soapenv:Envelope
xmlns:soapenv=¨http://schemas.xmlsoap.org/soap/envelope/¨
xmlns:pmap=¨http://www.apache.org/ode/pmapi¨>
  <soapenv:Header/>
  <soapenv:Body>
    <pmap:terminate>
      <iid>“numero iid del processo”</iid>
    </pmap:terminate>
  </soapenv:Body>
</soapenv:Envelope>
```

L'operazione più importante del service *ProcessManagement* è `listAllProcesses`: restituisce la lista di tutti i processi con l'aggiunta di molte informazioni: stato, versione, riepilogo delle istanze (terminate, completate, sospese, riprese, fallite) ed i file utilizzati dal processo. Un esempio di chiamata SOAP, per questa operazione, potrebbe essere:

Ode Engine - messaggio SOAP per l'operazione listAllProcesses

```
<soapenv:Envelope
xmlns:soapenv=¨http://schemas.xmlsoap.org/soap/envelope/
xmlns:pmap=¨http://www.apache.org/ode/pmapi¨>
  <soapenv:Header/>
  <soapenv:Body>
    <pmap:listAllProcesses/>
  </soapenv:Body>
</soapenv:Envelope>
```

4.2.1 Deploying di un processo

Il deploy dei processi in ODE engine, consiste nel creare il file XML *deploy.xml* e nel copiare l'intera directory, contenente i file utilizzati dal processo, *deploy.xml* compreso, nella directory *webapps/ode/WEB-INF/processes* del server Tomcat. Il file di deploying dichiara i collegamenti tra i partnerLink e i web service; l'XML Schema del documento è la seguente:

```
<deploy>
```

```
 <process name = QName  fileName = String?>
  <(provide | invoke) partnerLink=NCName>*
   <service name = QName port = NCName?/>
  </(provide | invoke)>
 </process>
</deploy>
```

Il file di deploying nel nostro esempio è:

Apache Ode Engine - deploy.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<deploy xmlns="http://www.apache.org/ode/schemas/dd/2007/03"
xmlns:main="http://www.miatesi.it"
xmlns:mws="http://www.miatesi.it/prestito.wsdl">
  <process name="main:ProcessoPrestito">
    <provide partnerLink="partnerPrestitoLinkType">
      <service name="mws:servizioPrestito"
      port="servizioPrestitoPT"/>
    </provide>
    <invoke partnerLink="approvaPrestitoLinkType">
      <service name="mws:AccettaPrestito"
      port="accettaPrestitoPort"/>
    </invoke>
    <invoke partnerLink="gestioneRischioLinkType">
      <service name="mws:ConsulenzaPrestito"
      port="consulenzaPrestitoPort"/>
    </invoke>
  </process>
</deploy>
```

All'inizio dichiariamo i namespace e il nome del processo. Poi, nell'elemento <provide> definiamo il servizio fornito dal processo tramite gli attributi `partnerLink`, `name` e `port`. Infine all'interno degli elementi <invoke>, utilizziamo gli stessi attributi per definire i servizi utilizzati dal processo.

Utilizzando Intalio|Designer [6], il deploy verrà gestito automaticamente dal tool di sviluppo.

4.3 Oracle BPEL Process Manager

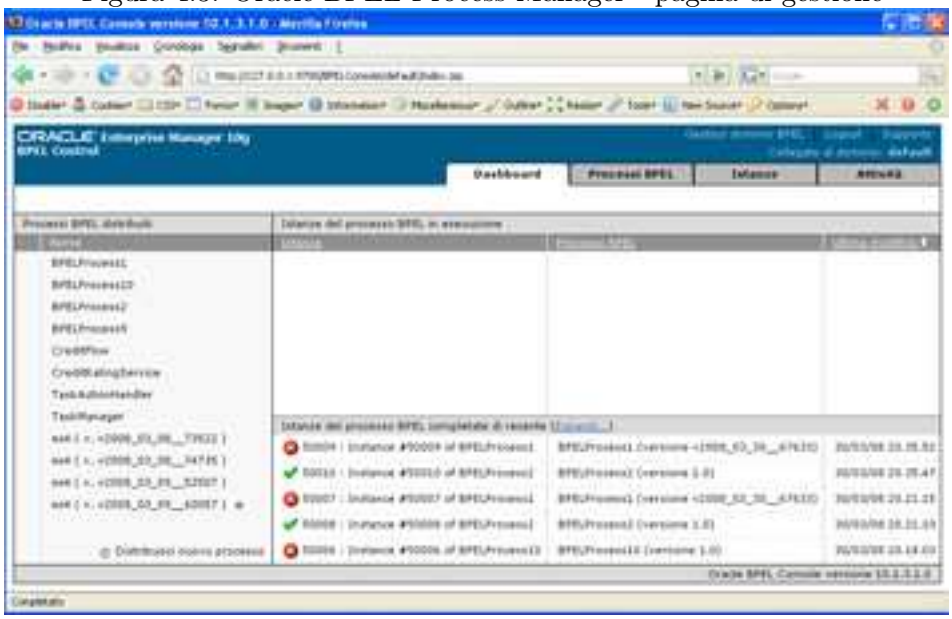
Oracle BPEL Process Manager [11] è l'engine WS-BPEL, scritto in JAVA, della nota società di sviluppo software Oracle; a differenza dei precedenti motori, non è open source. Inoltre, l'utilizzo commerciale dell'engine non è permesso, obbligando, chi abbia interesse ad un utilizzo in tal senso, ad acquistare una licenza. La versione attuale del motore è numerata 10.1.3.1.0 ed è scaricabile al seguente indirizzo:
http://www.oracle.com/technology/products/ias/bpel/index.html. I sistemi operativi supportati sono: Windows 2000/2003/XP, Red Hat Enterprise Linux AS/ES 3.0/4.0 e Suse Linux Enterprise Server 9; nel seguito, faremo riferimento alla versione per sistemi operativi Windows.

Terminata l'installazione, il server è avviabile tramite l'apposita voce presente nel menù start del sistema operativo. Per raggiungere la pagina di gestione (figura 4.3), inseriamo l'indirizzo *http://localhost:9700/BPELConsole* in un qualunque browser. L'engine dispone di due pannelli di controllo: uno per la gestione dei processi WS-BPEL (BPEL Console: user "default", password "welcome1") e l'altro per la configurazione del server (BPEL admin: user "bpeladmin", password "welcome1"). Per un utilizzo standard, non è necessario modificare la configurazione del server. Nella BPEL Console, sono elencati i processi dei quali è stato fatto il deploy: cliccando su un processo, appare la pagina per avviare una nuova istanza. Oracle BPEL Process Manager, oltre ad accettare i normali messaggi SOAP, fornisce un client interno per il testing dei processi; è possibile inserire i dati sia attraverso un form HTML, sia tramite messaggi XML. Terminata l'esecuzione di un'istanza di un determinato processo, abbiamo la possibilità di studiare il comportamento del processo stesso tramite tre funzioni offerte dall'engine:

- `Visual Flow`: mostra lo schema dell'esecuzione del processo, in modalità grafica;
- `Instance Auditing`: permette di vedere lo scambio dei messaggi e la gestione delle variabili, effettuati durante l'esecuzione del processo;
- `Instance Debug`: mostra il sorgente del processo, con la possibilità di cliccare sulle variabili e vederne il contenuto.

`Visual Flow` e `Instance Debug` sono disponibili anche quando il processo non è ancora completato. La BPEL Console permette anche di attivare e

Figura 4.3: Oracle BPEL Process Manager - pagina di gestione



disattivare i processi, controllare le varie istanze create per ogni processo e fare il deploy di un nuovo processo.

4.3.1 Deploying di un processo

Il deploy dei processi si basa sulla definizione di un file XML (solitamente chiamato *bpel.xml*), il quale contiene i seguenti dettagli:

- il nome del file WS-BPEL del processo;
- il nome del processo WS-BPEL (ID);
- la localizzazione dei file WSDL di ogni web service utilizzato dal processo;
- altre configurazioni opzionali.

L'XML Schema del file *bpel.xml* è il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<BPELSuitcase>
   <BPELProcess src="BPEL source path"
   id="process name">
     <partnerLinkBindings>
```

```
      <partnerLinkBinding name="Qname">+
         <property name="wsdlLocation">
            indirizzo file wsdl
         </property>
      </partnerLinkBinding>
     </partnerLinkBindings>
     <configurations>?
      <property name="Qname">*
       ...
      </property>
     </configurations>
   </BPELProcess>
</BPELSuitcase>
```

Le proprietà di configurazione opzionali sono `testIntroduction` e `default-Input`, le quali permettono, rispettivamente, la visualizzazione di un testo nella console dell'engine, quando il processo viene avviato, e l'inserimento dei dati di input di default tramite messaggi XML. Il file di deploying, nel nostro esempio, è:

Oracle BPEL Process Manager - bpel.xml

```
<?xml version=¨1.0¨ encoding=¨UTF-8¨ ?>
<BPELSuitcase>
  <BPELProcess src=¨ProcessoPrestito.bpel¨
  id=¨ProcessoPrestito¨>
    <partnerLinkBindings>
      <partnerLinkBinding name=¨cliente¨>
        <property name=¨wsdlLocation¨>
          prestito.wsdl
        </property>
      </partnerLinkBinding>
      <partnerLinkBinding name=¨accettaPrestitoBinding¨>
        <property name=¨wsdlLocation¨>
          http://miatesi.it:8080/servizi/FinanziatoreWebService?wsdl
        </property>
<!--continua nella pagina seguente-->
```

Oracle BPEL Process Manager - bpel.xml

```
<!--segue dalla pagina precedente-->
    </partnerLinkBinding>
    <partnerLinkBinding name=¨gestioneRischioBinding¨>
     <property name=¨wsdlLocation¨>
      http://www.miatesi.it:8080/servizi/ConsulenteWebService?wsdl
     </property>
    </partnerLinkBinding>
   </partnerLinkBindings>
  </BPELProcess>
</BPELSuitcase>
```

All'inizio definiamo il nome del file che contiene il sorgente WS-BPEL del processo e il nome che il processo avrà sull'engine. Poi dichiariamo per ogni web service la localizzazione dei file WSDL all'interno degli elementi <property>. Gli attibuti `name` degli elementi <partnerLinkBinding> fanno riferimento allo stesso attributo presente nei tag <binding> del documento WSDL del processo.

Definito il file per il deploy, occorre creare un archivio JAR, contenente i file necessari al funzionamento del processo. L'engine mette a disposizione un'utility a riga di comando che si occupa di creare l'archivio e di farne il deploy in un dominio dell'engine (di solito tale dominio è *default*). L'utility si chiama *bpelc* e si trova nella directory */bpel/bin* dell'engine: il comando da utilizzare è *bpelc -deploy default bpel.xml*. Oltre all'utility *bpelc*, ci sono altri due metodi per fare il deploying dell'archivio JAR:

- copiare manualmente l'archivio nella directory del dominio del server;
- caricare l'archivio dalla BPEL Console dell'engine.

Come per gli altri engine, è possibile fare il deploy automatico utilizzando il tool di sviluppo a corredo dell'engine, in questo caso, Oracle JDeveloper.

4.4 Altri engine BPEL

Gli engine di cui abbiamo parlato, non sono gli unici disponibili però sono i più completi e in continuo sviluppo tra quelli che prevedono un utilizzo o il testing gratuiti. Ne presentiamo brevemente alcuni:

- bexee [12] è un engine open source sviluppato da studenti dell'università di Berna, i quali dopo la fine degli studi hanno abbandonato il progetto. Allo stato attuale l'engine supporta solamente una parte delle vecchie specifiche WS4BPEL [14];

- Virtuoso Universal Server [13] ha l'obiettivo di essere un server universale open source, però l'engine BPEL che offre supporta solo il linguaggio WS4BPEL [14]. Il progetto è attivo ma l'engine BPEL non viene aggiornato da molto tempo e non c'è nessun riferimento a progetti futuri per l'adeguamento allo standard OASIS [10];

- Open ESB (acronimo di Enterprise Service Bus) [7] offre molti strumenti per lo sviluppo in ambito SOA, tra cui un engine BPEL. Il progetto, anch'esso open source, è interessante nonostante l'engine sia ancora in fase di sviluppo. Purtroppo non è stato possibile testare le funzionalità compatibili con lo standard OASIS [10] a causa di complicazioni avvenute sia durante l'installazione che all'avvio dell'engine: il server restava in servizio pochi secondi e poi eseguiva lo shutdown autonomamente. Probabilmente sui computer nei quali è stato installato l'engine, erano presenti software con cui entrava in conflitto.

CAPITOLO 5

TESTING DEGLI ENGINE WS-BPEL

Lo standard WS-BPEL di OASIS [10] è stato pubblicato nell'aprile 2007. A distanza di un anno, i vari engine hanno cercato di rispettare, chi più e chi meno, i dettami dello standard. Bisogna premettere che lo standard stesso è descritto in linguaggio naturale e, quindi, alcune sue parti possono essere ambigue e interpretate in modi differenti.

In questo capitolo, ci occupiamo di testare come i vari engine si comportano in relazione allo standard; procediamo con due diversi tipi di analisi: statica e dinamica. L'analisi statica consiste nel verificare quali attività e loro attributi sono supportate dai motori, mentre l'analisi dinamica controlla il comportamento dei server in determinati case study di particolare interesse.

5.1 Analisi statica

Per effettuare l'analisi statica, seguiamo la struttura del processo WS-BPEL presentata nel Capitolo 2:

- la sezione <extensions> è supportata da tutti gli engine;
- <import> non è presente in Oracle BPEL Process Manager, infatti la localizzazione dei file WSDL viene definita nel file *bpel.xml*;
- l'attributo `initializePartnerRole` del tag <parnerLink>, apparte-

nente alla sezione <partnerlinks>, viene ignorato da Oracle BPEL Process Manager;

- in <variables> solo ActiveBPEL consente l'inizializzazione delle variabili durante la loro dichiarazione, riconoscendo la direttiva `from-spec`;
- <correlationSets> è supportata da tutti gli engine;
- in <faultHandlers> Oracle BPEL ignora gli attributi (opzionali) `faultMessageType` e `faultElement` dell'elemento <catch>;
- in <eventHandlers> Oracle BPEL, al posto dell'elemento <onEvent>, utilizza <onMessage>. Quest'ultimo, però, non riconosce gli attributi `messageType`, `element`, `messageExchange` e l'elemento <fromParts> (quest'ultimo non è riconosciuto neanche da Apache ODE). Inoltre, lo standard prevede che all'interno di <onEvent> e <onAlarm> si possa utilizzare soltanto l'attività <scope>, mentre Oracle BPEL permette un'attività qualsiasi. Per finire, Oracle BPEL utilizza una sintassi diversa per il tag <onAlarm>, riportata di seguito:

```
<onAlarm for="duration-expr"?
         until="deadline-expr"?>*
 activity
</onAlarm>
```

Adesso, analizziamo le varie attività WS-BPEL. Premettiamo che Oracle BPEL e Apache ODE non riconoscono i tag <fromParts> e <toParts> presenti nelle specifiche delle attività dello standard OASIS [10]. Nel seguito non segnaleremo più questa lacuna, però sappiamo che questi tag vengono ignorati dai due engine.

- <receive>: Ode e Oracle BPEL non supportano l'avvio multiplo delle attività; questo, implica che non è possibile avere un processo con più <receive> in parallelo aventi l'attributo `createIstance`="yes". Inoltre, lo standard impone che tutte le attività in parallelo in un certo istante, se il processo non è già stato avviato da una attività precedente, abbiano la possibilità di creare un'istanza del processo, altrimenti deve essere segnalato un errore; questa direttiva è seguita soltanto da ActiveBPEL;

- <reply> è implementata da tutti gli engine;
- <invoke> è implementata da tutti gli engine;
- <validate> è supportata solamente da ActiveBPEL;
- <assign> è completamente supportata da ActiveBPEL. Apache ODE e Oracle BPEL non riconosco gli attibuti `validate`, `keepSrcElement-Name`, `ignoreMissingFromData` e l'elemento <extensionAssignOperation>;
- <throw> è supportata da tutti gli engine;
- <rethrow> non è implementata da Oracle BPEL;
- <exit> non è implementata da Oracle BPEL, al suo posto utilizza l'attività <terminate>;
- <wait>: Oracle BPEL utilizza una sintassi diversa, che mostriamo di seguito:

```
<wait (for="duration-expr" |
       until="deadline-expr")
 standard-attributes>
   standard-elements
</wait>
```

- <empty> è correttamente implementata da tutti gli engine;
- <compensateScope> non è presente in Oracle BPEL;
- <compensate> è supportata solo da ActiveBPEL. Oracle BPEL la utilizza per le attività <scope>, avendo un attributo `scope` dove va indicato il nome dell'attività da compensare. ODE la implementa con l'identica sintassi di <compensateScope>;
- <extensionActivity> non è implementata da Oracle BPEL;
- <sequence> è implementata da tutti gli engine;
- <flow> è implementata da tutti gli engine;
- <while>: Oracle BPEL utilizza una sintassi diversa:

```
<while condition="bool-expr"
       standard-attributes>
 standard-elements
 activity
</while>
```

- <repeatUntil> non è supportata da Oracle BPEL;
- <pick>: Oracle BPEL e ODE presentano le stesse peculiarità evidenziate nell'attività <receive>;
- <if> non è supportata da Oracle BPEL, il quale la sostituisce con l'attivita <switch>, della quale riportiamo la sintassi:

```
<switch standard-attributes>
 standard-elements
 <case condition="bool-expr">+
  activity
 </case>
 <otherwise>?
  activity
 </otherwise>
</switch>
```

- <forEach> non è implementata da Oracle BPEL;
- <scope>: Apache ODE non riconosce gli attributi `isolated` e `exitOnStandardFault`; Oracle BPEL oltre a non supportare tali attributi, non permette l'utilizzo dei tag <partnerLinks>, <messageExchanges> e <terminationHandler>.

Con queste differenze, è evidente che il porting dei processi tra i vari engine non è facile da effettuare. Inoltre, Apache ODE e Oracle BPEL hanno sviluppato alcune proprietà e attività non previste dallo standard OASIS [10]: per esempio, l'attività <SMS> di Oracle BPEL che permette l'invio di messaggi SMS durante l'esecuzione del processo, oppure l'utilizzo di variabili esterne al processo BPEL, contenute in database, da parte di ODE. Comunque, la maggioranza dei processi scritti per Oracle BPEL, che non

utilizzino le attività non previste dallo standard, riescono ad essere eseguiti da ActiveBPEL e Apache ODE, essendo questi ultimi retrocompatibili con la versione precedente di WS-BPEL (BPEL4WS [14]). Infatti, Oracle BPEL supporta unicamente le vecchie specifiche.

5.2 Analisi dinamica

Gli engine, oltre ad utilizzare alcune sintassi diverse dallo standard WS-BPEL, possono avere comportamenti differenti durante l'esecuzione dei processi. I case study, che trattiamo adesso, sono presi da esempi scritti in BLite [8]. Blite è un linguaggio, sviluppato dall'università di Firenze, che si occupa di dare una base formale al linguaggio WS-BPEL per porre rimedio alle ambiguità presenti nello standard OASIS [10] causate dal linguaggio naturale con cui è scritto.

Identiche <receive> consecutive

Lo standard OASIS [10] non impedisce l'utilizzo sequenziale di più attività <receive>. Questo può comportare dei problemi se il processo ha più di un'istanza in esecuzione. Infatti ad un ipotetico client, che esegua due <invoke> consecutive, può accadere di inviare un messaggio ad un'istanza del processo e l'altro messaggio ad un'altra istanza. Per prevenire questo effetto, possiamo utilizzare i <correlationSet> che si occupano di far pervenire i messaggi alla giusta istanza del processo. Se nel processo le <receive> sono identiche il problema si fa più gravoso.

Analisi dinamica - identiche <receive> consecutive

```
...
<bpel:sequence>
<bpel:receive createInstance="yes" operation="inizia"
partnerLink="partnerLT" portType="ns:servizioPT"
variable="variabile"/>
<bpel:receive operation="inizia"
partnerLink="partnerLT" portType="ns:servizioPT"
variable="variabile"/>
</bpel:sequence>
...
```

Vediamo come si comportano gli engine alla ricezione da parte del client di due <invoke> identiche. ActiveBPEL e ODE inviano i due messaggi alla stessa istanza del processo: il primo messaggio arriva alla prima <receive> che crea l'istanza e il secondo messaggio è gestito dalla seconda <receive>. Invece, Oracle BPEL crea due istanze del processo che vanno in conflitto, bloccandone l'esecuzione.

Terminazione forzata

Quando un processo, durante l'esecuzione delle attività strutturate <sequence> o <flow> deve effettuare una attività <exit> oppure <throw>, lo standard ([10] sezione 12.6) prevede che "le attività strutturate devono terminare, causando anche la terminazione di tutte le attività attive all'interno di esse". L'ultima parte è interpretata in modo diverso dagli engine.

Analisi dinamica - terminazione forzata

```
...
<bpel:flow>
  <bpel:sequence>
    <bpel:empty/>
    <bpel:exit/>
  </bpel:sequence>
  <bpel:sequence>
    <bpel:assign>
      <bpel:copy>
        <bpel:from variable=¨x2¨/>
        <bpel:to variable=¨x1¨/>
      </bpel:copy>
    </bpel:assign>
    <bpel:assign>
      <bpel:copy>
<!-- continua nella pagina seguenta -->
```

Analisi dinamica - terminazione forzata

```
<!--segue dalla pagina precedente -->
    <bpel:from variable="x1"/>
    <bpel:to variable="x3"/>
   </bpel:copy>
  </bpel:assign>
 </bpel:sequence>
 <bpel:sequence>
  <bpel:wait>
   <bpel:for>'PT5S'</bpel:for>
  </bpel:wait>
  <bpel:empty/>
 </bpel:sequence>
</bpel:flow>
...
```

Nell'esempio, Oracle BPEL Process Manager quando esegue l'attività <terminate> (come sappiamo non ha <exit>) blocca l'esecuzione dell'attività <wait> contenuta nella terza <sequence> in parallelo ma riesce a completare l'attività <sequence> che contiene le due <assign> (figura 5.1). ActiveBPEL (figura 5.2) e Apache ODE invece, oltre a bloccare l'esecuzione dell'attività <wait>, interrompono anche la sequenza di <assign> completando solamente la prima attività.

Inoltre le specifiche dicono che la compensazione e la gestione degli errori possono completare il loro comportamento durante una terminazione forzata. ActiveBpel e ODE rispettano la direttiva mentre Oracle BPEL non lo fa. L'esempio è composto da un'attività <scope> che ha al suo interno due <scope> in parallelo, una delle quali contiene una sequenza di due <scope>. La prima <scope> all'interno della sequenza esegue una <assign> e implementa la compensazione, l'altra <scope> della sequenza e quella in parallelo eseguono una <assign> e poi terminano il processo tramite un'attività <throw>. Possiamo vedere (figura 5.3) che ActiveBPEL esegue la compensazione della <scope> prima della terminazione del processo causata dalla <scope> in parallelo.

Figura 5.1: Oracle BPEL - terminazione forzata

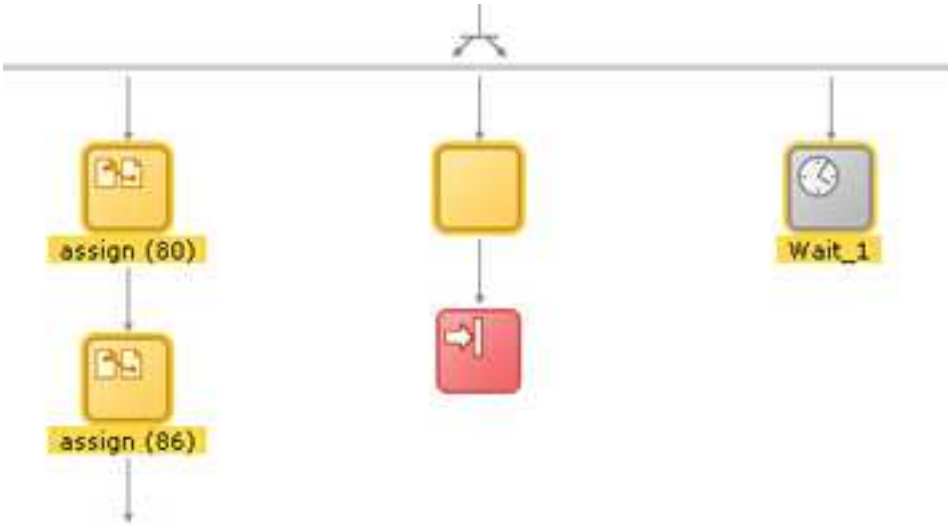


Figura 5.2: ActiveBPEL - terminazione forzata

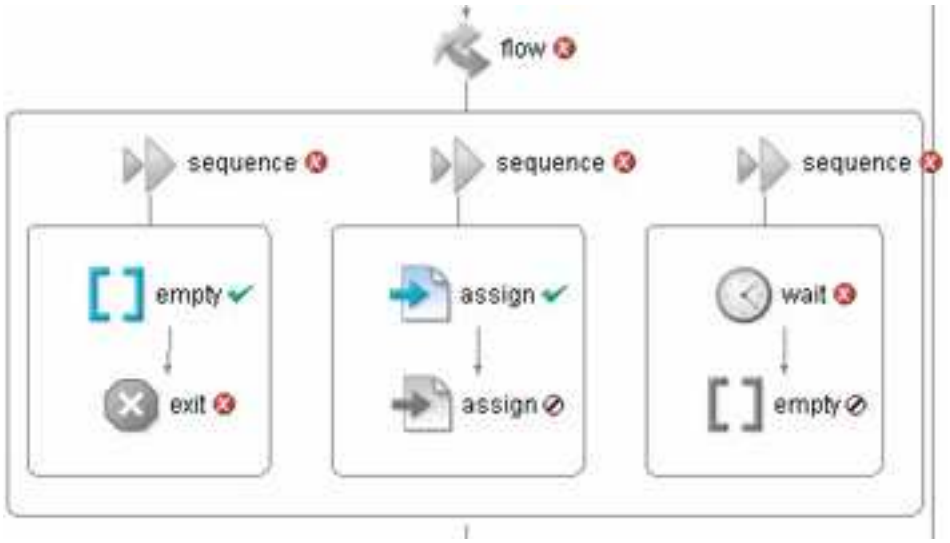
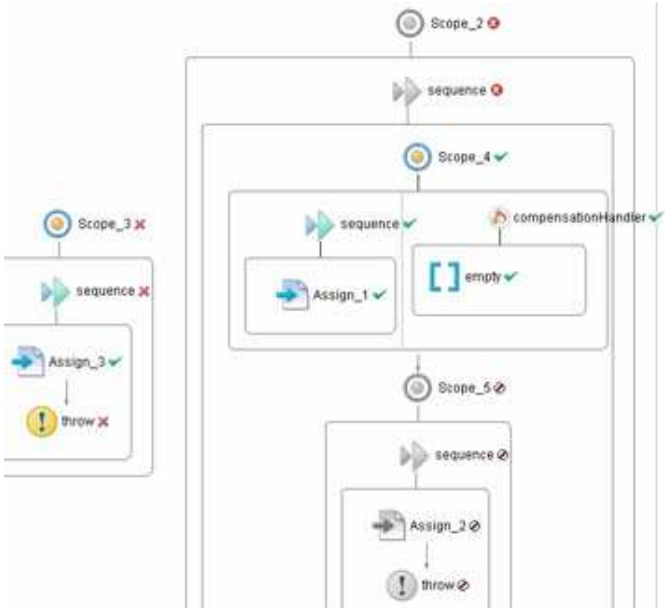


Figura 5.3: ActiveBPEL - compensazione



Analisi dinamica - compensazione

```
...<bpel:flow>
    <bpel:scope name="Scope_3">
      <bpel:sequence>
        <bpel:assign name="Assign_3">
          <bpel:copy>
            <bpel:from>
            concat( $approvalMessage.accept ,'2')
            </bpel:from>
            <bpel:to part="accept" variable="approvalMessage"/>
          </bpel:copy>
        </bpel:assign>
        <bpel:throw faultName="bpel:correlationViolation"/>
      </bpel:sequence>
    </bpel:scope>
    <bpel:scope name="Scope_2">
      <bpel:sequence>
        <bpel:scope name="Scope_4">
          <bpel:compensationHandler>
            <bpel:empty/>
<!--continua nella pagina seguente-->
```

Analisi dinamica - compensazione

```
<!--segue dalla pagina precedente–>
    </bpel:compensationHandler>
    <bpel:sequence>
     <bpel:assign name=¨Assign_1¨>
       <bpel:copy>
         <bpel:from>
         concat( $approvalMessage.accept ,'2')
         </bpel:from>
         <bpel:to part=¨accept¨ variable=¨approvalMessage¨/>
       </bpel:copy>
     </bpel:assign>
    </bpel:sequence>
   </bpel:scope>
   <bpel:scope name=¨Scope_5¨>
    <bpel:sequence>
     <bpel:assign name=¨Assign_2¨>
       <bpel:copy>
         <bpel:from>concat
         ( $approvalMessage.accept ,'2')
         </bpel:from>
         <bpel:to part=¨accept¨ variable=¨approvalMessage¨/>
       </bpel:copy>
     </bpel:assign>
     <bpel:throw faultName=¨bpel:correlationViolation¨/>
    </bpel:sequence>
   </bpel:scope>
  </bpel:sequence>
 </bpel:scope>
</bpel:flow>
...
```

Attività in parallelo

L'attività <flow> ([10] sezione 11.6) dovrebbe permettere l'esecuzione concorrente delle attività contenute al suo interno, senza seguire un ordine

prestabilito. Il seguente esempio, inizializzate tre variabili, esegue tre attività <assign> in parallelo, ognuna copia il valore di una variabile in un'altra. Ad ogni esecuzione del processo dovremmo avere le variabili con dei valori non prevedibili, però non è quello che otteniamo dai vari engine.

Analisi dinamica - parallelo

```
...
<bpel:flow>
  <bpel:assign>
    <bpel:copy>
      <bpel:from variable="x2"/>
      <bpel:to variable="x1"/>
    </bpel:copy>
  </bpel:assign>
  <bpel:assign>
    <bpel:copy>
      <bpel:from variable="x3"/>
      <bpel:to variable="x2"/>
    </bpel:copy>
  </bpel:assign>
  <bpel:assign>
    <bpel:copy>
      <bpel:from variable="x1"/>
      <bpel:to variable="x3"/>
    </bpel:copy>
  </bpel:assign>
</bpel:flow>
```

Supponiamo che le variabili siano inizializzate nel seguente modo: x1="uno", x2="due" e x3="tre". Dopo l'esecuzione dell'attività <flow>, l'engine ActiveBPEL avrà le variabili contenenti i seguenti valori: x1="due", x2="tre" e x3="due". Questo accade per ogni esecuzione del processo; ciò significa che in pratica l'engine esegue le attività, all'interno di <flow>, seguendo l'ordine lessicale in cui si presentano. In Oracle BPEL, invece, per ogni esecuzione del processo, le variabili avranno i seguenti valori: x1="uno", x2="uno" e x3="uno". Cioè, l'engine esegue le attività in ordine inverso rispetto a quello seguito da ActiveBPEL. Apache ODE è l'unico motore che non sembra

avere un ordine predefinito di esecuzione per le attività contenute da <flow>. Infatti, eseguendo più volte la <flow>, le variabili avranno valori che non sempre si ripetono, per esempio: x1="tre", x2="tre" e x3="uno" e nella prova seguente x1="due", x2="tre" e x3="due".

Ancora sulla terminazione

Analisi dinamica - priorità della terminazione

```
...
<bpel:flow>
  <bpel:sequence>
  <assign name="numero">
    <copy>
      <bpel:from variable="x1"/>
      <bpel:to variable="x2"/>
      </copy>
    </assign>
    <assign name="numero">
      <copy>
        <bpel:from variable="x3"/>
        <bpel:to variable="x4"/>
      </copy>
    </assign>
  </bpel:sequence>
  <bpel:throw faultName="bpel:conflitto"
  faultVariable="erroe"/>
  <bpel:sequence>
    <bpel:wait>
      <bpel:for>'PT5S'</bpel:for>
    </bpel:wait>
    <bpel:empty/>
  </bpel:sequence>
</bpel:flow>
```

Lo standard WS-BPEL ([10] sezione 12.6) impone che le attività di terminazione devono terminare immediatamente tutte le attività già avviate. L'esempio effettua una <flow> contenente tre rami: una sequenza di <assign>,

Figura 5.4: ActiveBPEL - terminazione

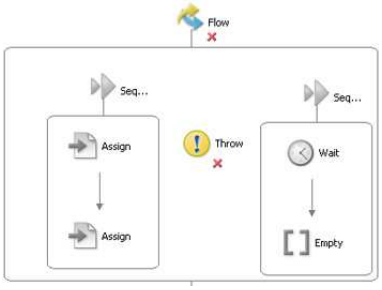
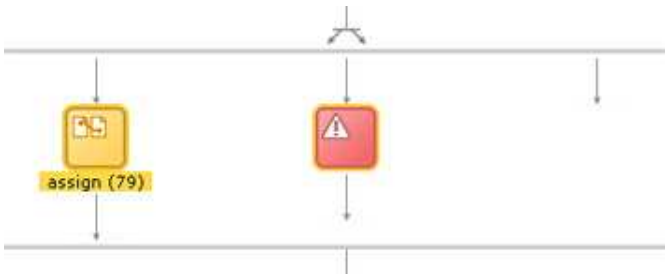


Figura 5.5: Oracle BPEL - terminazione



una <throw> e una sequenza con all'interno una <wait> e una <empty>. In questo caso l'attività <throw> dovrebbe avere la precedenza. Vediamo che ActiveBPEL dà la precedenza all'attività di terminazione (figura 5.4); Oracle BPEL e Apache ODE possono, invece, iniziare ad eseguire uno qualsiasi dei rami in parallelo, così da riuscire a completare qualche attività, prima di dare il segnale di terminazione. In figura 5.5 vediamo che Oracle BPEL riesce a completare una <assign>.

Gestione degli eventi dell'attività <pick>

L'attività <pick> aspetta il verificarsi di determinati eventi per eseguire delle attività. Lo standard ([10] sezione 11.5) impone che al verificarsi di un evento, gli altri eventi non devono più essere monitorati, lasciando inutilizzate le attività a loro associate. Inoltre, se la <pick> è usata per creare un'istanza del processo, deve contenere solo eventi <onMessage> e nessun <onAlarm>. Tutti gli engine seguono queste direttive. L'esempio seguente utilizza la <pick> per attendere un messaggio, se il messaggio non arriva

entro 20 secondi il processo viene terminato tramite l'esecuzione dell'attività <exit> (figura 5.6).

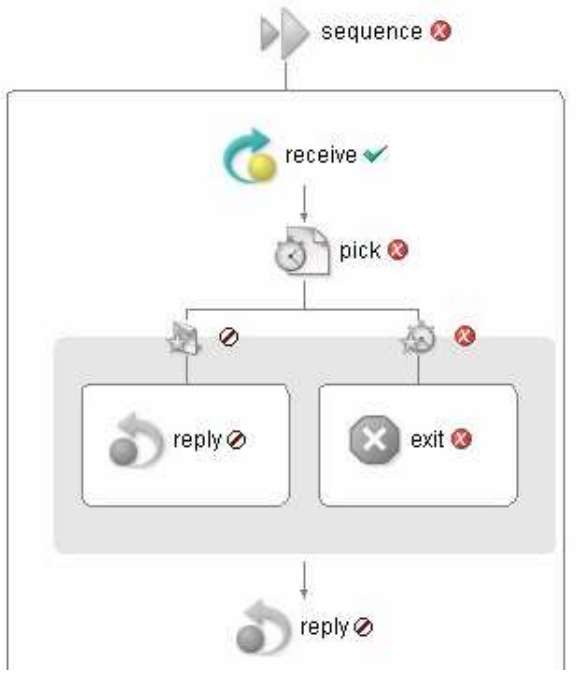
Analisi dinamica - attività <pick>

```
...
<bpel:sequence>
  <bpel:receive operation="testAssign" partnerLink="PL1"
  portType="ns1:TestAssignPortType" variable="receive"/>
  <bpel:pick>
    <bpel:onMessage operation="risposta" partnerLink="PL2"
    portType="ns1:rispostaPortType" variable="messaggio">
      <bpel:reply operation="risposta" partnerLink="PL2"
      variable="messaggio"/>
    </bpel:onMessage>
    <bpel:onAlarm>
      <bpel:for>'PT20S'</bpel:for>
      <bpel:exit/>
    </bpel:onAlarm>
  </bpel:pick>
  <bpel:reply operation="testAssign" partnerLink="PL1"
  portType="ns1:TestAssignPortType" variable="receive"/>
</bpel:sequence>
...
```

Esecuzione dell'attività <foreach> in parallelo

L'attività <foreach> esegue un numero predefinito di iterazioni della <scope> in essa contenuta. Lo standard OASIS ([10] sezione 11.7) obbliga ad eseguire le iterazioni in modalità concorrente quando l'attributo `parallel` è impostato a "yes". L'esempio crea 10 istanze dell'attività <scope> contenuta nella <foreach>.

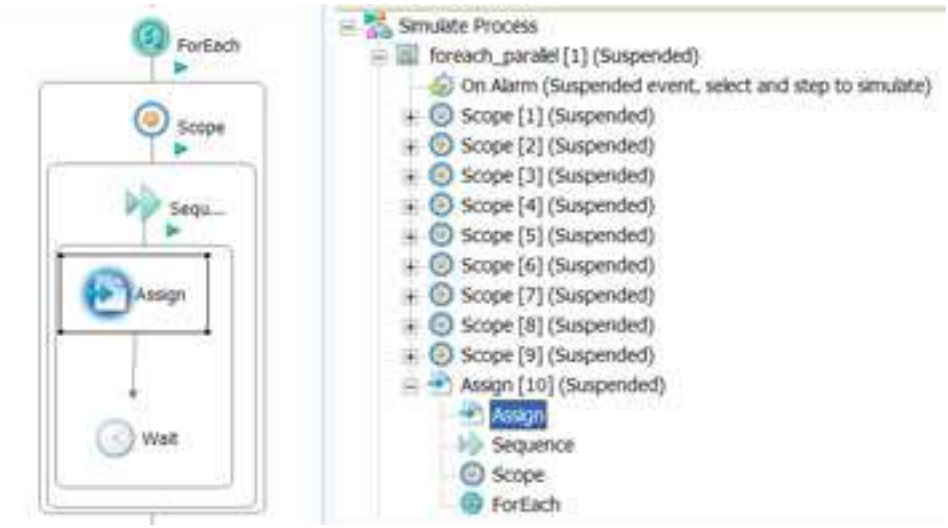
Figura 5.6: ActiveBPEL - attività <pick>



Analisi dinamica - attività <foreach>

```
...
<forEach counterName="counter" parallel="yes">
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>10</finalCounterValue>
  <scope>
    <sequence>
      <bpel:assign>
        <bpel:copy>
          <bpel:from variable="x2"/>
          <bpel:to variable="x1"/>
        </bpel:copy>
<!--continua nella pagina seguente-->
```

Figura 5.7: ActiveBPEL - <foreach> in parallelo



Analisi dinamica - attività <foreach>

```
<!--segue dalla pagina precedente-->
    </bpel:assign>
    <wait>
      <for>'PT5S'</bpel:for>
    </wait>
   </sequence>
  </scope>
</forEach>
...
```

Come vediamo dalla figura 5.7, le 10 attività <scope> vengono subito istanziate ed eseguite in modalità concorrente. ActiveBPEL e ODE consentono l'esecuzione in parallelo della <foreach>. Oracle BPEL non implementa l'attività <foreach> però ha l'attività <flowN> non definita nelle specifiche, sia in quelle attuali [10] che nelle precedenti [14], che emula l'attività <foreach> eseguita in parallelo. La sostanziale differenza consiste nel permettere l'utilizzo di qualsiasi attività all'interno del ciclo, al contrario della <foreach> che impone la presenza dell'attività <scope>.

Analisi dinamica - attività <flowN>

```
...
<flowN name=¨FlowN¨ N=¨getVariableData('numero_iterazioni')¨
indexVariable=¨Index¨>
  <sequence>
    <bpel:assign>
      <bpel:copy>
        <bpel:from variable=¨x2¨/>
        <bpel:to variable=¨x1¨/>
      </bpel:copy>
    </bpel:assign>
    <wait>
      <for>'PT5S'</bpel:for>
    </wait>
  </sequence>
</flowN>
...
```

5.3 Considerazioni

Dopo la trattazione delle analisi statica e dinamica, possiamo dire che le specifiche dello standard OASIS [10] sono maggiormente supportate dall'engine ActiveBPEL, seguito da Apache ODE. Oracle BPEL Process Manager rispetta la versione precedente dello standard [14] e presenta solo alcune delle novità apportate dalle nuove specifiche. Nonostante ciò, Oracle BPEL è attualmente l'engine più utilizzato dalle aziende che vogliono basare la propria architettura SOA sul linguaggio WS-BPEL. Questo è dovuto sicuramente ad un pronto sviluppo dell'engine BPEL, quando ancora il linguaggio era poco conosciuto al di fuori delle realtà accademiche e al nome di sicuro prestigio che la società si è fatta durante anni di sviluppo software. Un'altra importante differenza tra Oracle BPEL e gli altri due engine è la richiesta di risorse di sistema che i server necessitano per operare: a parità di processi BPEL installati Oracle BPEL utilizza all'incirca 600 MB di memoria RAM contro i circa 100 MB del server Tomcat con gli altri due engine installati su di esso. Anche le esecuzioni delle istanze di processo BPEL hanno tempi maggiori su Oracle BPEL che aumentano se la macchina su cui risiede

l'engine ha prestazioni limitate: gli engine sono stati installati su un pc desktop con processore Intel Core 2 Duo E6320 con 3 GB di memoria Ram e su un pc portatile con processore Intel Pentium 4 2.4 GHz con 1 GB di memoria RAM. Le esecuzioni delle istanze di processo di Oracle BPEL erano sensibilmente più lente sul pc portatile, mentre per gli altri engin BPEL non si sono riscontrate grandi differenze tra le due macchine. Comunque Apache ODE è leggermente più performante di ActiveBPEL però non fornisce un ambiente di sviluppo proprietario per i processi WS-BPEL e questo può rappresentare un problema per la diffusione di questo engine, inoltre Intalio|Designer [6] è meno semplice da usare degli IDE degli altri engine BPEL.

A seguito di queste ultime considerazioni, l'engine ActiveBPEL rappresenta la scelta migliore per le aziende che vogliono un'architettura SOA basata sul linguaggio WS-BPEL. Infatti ActiveBPEL è l'engine che maggiormente rispetta lo standard OASIS [10], permette la visualizzazione del flusso delle istanze dei processi, ha una richiesta di risorse di sistema contenuta, presenta delle buone prestazioni sia su macchine di ultima generazione che su quelle meno recenti ed ha un ambiente di sviluppo proprietario completo e di semplice utilizzo.

CAPITOLO 6

CONCLUSIONI

I web service sono l'ideale per lo sviluppo di architetture SOA, grazie all'impiego di protocolli standard liberi e indipendenti dalla piattaforma di utilizzo. L'orchestrazione dei web service affidata a WS-BPEL si presenta come un'ottima scelta date le funzionalità offerte dal linguaggio: gestione di variabili, controllo del flusso del processo, gestione di eventi e di errori, esecuzione concorrente di alcune parti del processo. L'assenza di semantica formale nello standard OASIS [10] comporta una disomogeneità nel comportamento degli engine BPEL che si trasforma in una maggiore difficoltà per gli sviluppatori di processi in WS-BPEL. Lo scenario ideale sarebbe l'esistenza di engine conformi ad uno standard scritto con una semantica formale privo di ambiguità. Purtroppo siamo ancora lontani dal raggiungimento di tale traguardo: le differenze tra gli engine sono palesi e il rispetto delle specifiche è imputabile solo ad ActiveBPEL. Una grave carenza dello standard è la fase che tratta il deploy di un processo: la decisione di non imporre un metodo uguale per tutti preclude ogni possibilità allo scambio di archivi di processi BPEL pronti per l'utilizzo tra entità che utilizzano engine differenti. Infatti, anche quando tutti gli engine implementeranno completamente lo standard, la portabilità avrà un collo di bottiglia durante il deploy dei processi: gli sviluppatori saranno costretti a creare i file necessari per l'utilizzo dei processi per ogni engine specifico.

Un fattore da non tralasciare è anche la gestione delle risorse di sistema utilizzate da ogni engine. La scelta dell'engine deve essere preceduta da

uno studio sull'hardware disponibile, sulla velocità di risposta attesa per l'esecuzione dei processi e sul limite massimo di istanze, superato il quale, è permesso un calo delle prestazioni. Dai test risulta che in uno scenario in cui si prospetta un alto carico di lavoro, la scelta più indicata ricada su Apache ODE. Se invece la priorità è la sicurezza offerta da un'assistenza tecnica dedicata, Oracle è garanzia di professionalità e quindi è meglio scegliere Oracle BPEL. La scelta obbligata però sembra essere ActiveBPEL che, oltre a rispettare lo standard, offre prestazioni paragonabili a Apache ODE e una comunità online, attiva e propositiva, nella quale è possibile trovare esperti pronti ad aiutare nei casi di difficoltà.

La trattazione degli argomenti di questa tesi non può considerarsi conclusa dato lo stato di costante sviluppo degli engine BPEL: a breve uscirà una nuova versione di ActiveBPEL, sul forum di Oracle BPEL ci sono delle indiscrezioni che parlano dell'implementazione di alcune attività di WS-BPEL già presenti nella versione beta della prossima release e il team di sviluppo di Apache ODE sta lavorando alla versione 1.2 ma, probabilmente, verrà numerata 2.0 per evidenziare le molte novità aggiunte. Inoltre, non va dimenticato l'engine Open ESB di cui per problemi tecnici è stato impossibile testarne l'efficienza ma che, almeno sulla carta, presenta un buon potenziale per entrare a far parte dei migliori engine BPEL utilizzabili gratuitamente. Per ovviare alla mancanza delle specifiche, un progetto interessante potrebbe essere lo sviluppo di un'applicazione che si occupa di produrre i file per il deploy dei processi BPEL per i vari engine: la portabilità dei processi tra i vari engine sarebbe agevolata con l'esistenza di un software dedicato alla creazione dei documenti per il deploy.

BIBLIOGRAFIA

[1] Active Endpoints. ActiveBPEL Engine.
http://activevos.com/community-open-source.php.

[2] eviware.com. soapUI User Guide.
http://www.soapui.org/userguide/index.html.

[3] The Apache Software Fondation. Apache ODE.
http://ode.apache.org/index.html.

[4] The Apache Software Foundation. Apache Axis2.
http://ws.apache.org/axis2/1_3/contents.html.

[5] The Apache Software Foundation. ODE BPEL Management API Specification.
http://ode.apache.org/bpelmanagementapispecification.html.

[6] Intalio. Intalio|Designer.
http://bpms.intalio.com/.

[7] java.net. Open ESB.
https://open-esb.dev.java.net/.

[8] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *Proc. 10th international conference on Coordination Models and Languages (COORDINATION'08)*, Lecture Notes in Computer Science. Springer.

[9] Sun microsystems. Java Servlet Technology.
http://java.sun.com/products/servlet/.

[10] OASIS. Standard BPEL 2.0.
http://docs.oasis-open.org/wsbpel/2.0.

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

[12] P.Fornasier and P.Kowalski. bexee - BPEL Execution Engine.
http://bexee.sourceforge.net/.

[13] OPENLINK software. Virtuoso Universal Server.
http://virtuoso.openlinksw.com/index.html.

[14] T.Andrews, F.Curbera, H.Dholakia, Y.Goland, J.Klein, F.Leymann, K.Liu, D.Roller, D.Smith, S.Thatte, I.Trickovic, and S.Weerawarana. Business Process Execution Language for Web Services 1.1.
http://xml.coverpages.org/BPELv11-May052003Final.pdf.

[15] W3C. Standard SOAP.
http://www.w3.org/TR/soap.

[16] W3C. Standard Web Services Addressing.
http://www.w3.org/Submission/ws-addressing/.

[17] W3C. Standard WSDL.
http://www.w3.org/TR/wsdl.

[18] W3C. Standard XML Schema.
http://www.w3.org/XML/Schema.

[19] W3C. XML Path Language.
http://www.w3.org/TR/xpath.

[20] W3C. XML Schema Part 1: Structures Second Edition.
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.