# JBoss Overlord CDL 1.0-M2

# User Guide

by Gary Brown and Jeff Yu

**1. Overview** ..... 1
1.1. WS-CDL ..... 1
1.2. pi4soa ..... 2
1.3. SOA Lifecycle Governance ..... 2
1.3.1. Design Time Governance ..... 2
1.3.2. Runtime Governance ..... 3
1.4. First Steps ..... 4
**2. Conversation Validation with CDL** ..... 5
2.1. Overview ..... 5
2.2. Configuration of Conversation Validation ..... 5
2.2.1. Installing the Conversation Validation Mechanism ..... 5
2.2.2. Explicit Configuration ..... 5
2.2.3. Defining the Validator Configuration within a Choreography ..... 7
2.3. Monitoring the Choreography Description ..... 9
2.4. Configuration for Conversation Recording ..... 10
**3. Conversation Aware ESB** ..... 12
3.1. Conversation based Conformance ..... 12
3.1.1. Overview ..... 12
3.1.2. CDL Conformance Checking ..... 12
3.2. Stateful "Conversation Aware" ESB Actions ..... 13
3.2.1. Overview ..... 13
3.2.2. Conversational Service ..... 13
3.2.3. Establishing Correspondance between ESB Configuration and Choreography ..... 16
3.2.4. Managing Sessions ..... 17
3.2.5. Interacting ..... 19
3.2.6. Managing Information ..... 22
3.2.7. Controlling Flow ..... 23
3.3. Stateless "Conversation Aware" ESB Actions ..... 28
3.3.1. Overview ..... 28
3.3.2. Interaction ..... 28
3.3.3. Controlling Flow ..... 30
3.4. Generating a JBossESB Configuration from CDL ..... 32
3.4.1. Overview ..... 32
3.4.2. Generating the JBossESB Configuration ..... 32
3.5. Dealing with Conformance Issues ..... 33
3.5.1. Overview ..... 33
3.5.2. Show referenced description ..... 33
3.5.3. Error: Expecting additional activities as defined in referenced description ..... 33
3.5.4. Error: Type mismatch with referenced description, was expecting '...' ..... 34
3.5.5. Error: Behaviour not present in referenced description ..... 34
3.5.6. Error: Additional unmatched paths in model ..... 34
3.5.7. Error: Additional unmatched paths in referenced description ..... 34
**4. BPEL Governance** ..... 35
4.1. Overview ..... 35

4.2. Generating a BPEL process from CDL ..... 35
4.2.1. Overview ..... 35
4.2.2. Generating the BPEL Process ..... 35
4.2.3. Limitations with the current CDL to BPEL mapping ..... 36

# Overview

The CDL component of the Overlord SOA governance project aims to leverage the concept of a choreography (or conversation) description to provide design-time and run-time governance of an SOA.

A Choreography provides the means to describe the service interactions between multiple parties from a global (or service neutral) perspective. This means that it is possible for an organisation to define how an end-to-end business process should function, regardless of whether orchestrated or peer-to-peer service collaboration will be used.

Although in simple situations, a BPEL process description can provide a description of the interactions between multiple services, this only works where a single orchestrating process is in control. The benefit of the choreography description is that it can be used to provide a global view of a process across multiple orchestrated service domains.

This document will outline how the Choreography Description is being used as part of Project Overlord to provide SOA governance capabilities for each phase of the SOA lifecycle.

When a validated design has been approved by the users, it can be used to generate an initial skeleton of the implementation for each service. The current version of Overlord enables a skeleton implementation to be generated as a JBossESB service configuration file, using 'conversation aware' ESB actions. For more information on these, please see the “Conversational ESB User Guide”.

## 1.1. WS-CDL

WS-CDL, or Web Service Choreography Description Language, is a candidate recommendation from W3C. Although associated with W3C and Web Services, it is important to begin by stating that the Choreography Description Language (CDL) is **not** web service specific.

The purpose of CDL is to enable the interactions between a collection of peer to peer services to be described from a neutral (or global) perspective. This is different to other standards, such as WS-BPEL, that describe interactions from a service specific viewpoint.

In essence a choreography description declares roles which will pass messages between each other, called interactions. The interactions are ordered based on a number of structuring mechanism which enables loops, conditional, choices and parallelism to be described. In CDL variables used for messages and for conditionals are all situated at roles. There is no shared state rather there is a precise description of the state at each role and a precise description of how these roles interact in order to reach some notion of common state in which information is exchanged and processed between them.

In CDL we use interactions and these structuring mechanisms to describe the observable behaviour, the messages exchanges and the rules for those exchanges and any supporting observable state on which they depend, of a system.

## 1.2. pi4soa

*pi4soa* is an open source project established to demonstrate the potential benefits that a global model (as described using CDL) can provide when building an SOA. The open source project is managed by the Pi4 Technologies Foundation, which is a collaboration between industry and academia.

Building complex distributed systems, without introducing unintended consequences, is a real challenge. Although the Choreography Description Language provides a means of describing complex systems at a higher level, and therefore help to reduce such complexity, it does not necessarily guarantee that erronous situations cannot occur due to inappropriately specified interactions. The research, being carried out by members of the Pi4 Technologies Foundation, into the global model and endpoint projection is targeted at identifying potential unintended consequences, to ensure that a global description of a system can be reliably executed and can be free from unintended consequences.

The tool suite currently offers the ability to:

- Define a choreography description
- Export the description to a range of other formats, such as BPMN, UML activity/state/sequence models, and HTML
- Define scenarios (equivalent to sequence diagrams), with example messages, which can then be simulated against an associated choreography
- Generate template endpoint implementations:
  - WS-BPEL for deployment in ActiveBPEL
  - Java stubs for execution with the pi4soa state machine, with deployment options for Apache Axis, J2EE (JBoss, Glassfish) and JBoss ESB

## 1.3. SOA Lifecycle Governance

### 1.3.1. Design Time Governance

Design-time governance is concerned with ensuring that the resulting system correctly implements requirements (whether functional or non-functional). A choreography description can be used to ensure that the implemented system meets the behavioural requirements.

The behavioural requirements can be captured as a collection of scenarios (e.g. sequence diagrams) with associated example messages. This enables an unambiguous representation of the business requirements to be stored in a machine processable form, which can subsequently be used to validate other phases of the SOA lifecycle.

Once the choreography description for the SOA has been defined, it can be validated against the scenarios, to ensure that the choreography correctly handles all of the business requirements.

Once the service enters the implementation phase, it is important to ensure that it continues to adhere to the design and therefore meets the business requirements. Currently this is achieved through the use of techniques such as continuous testing. However this is only as reliable as the quality of the unit tests that have been written.

When a 'structured' implementation language has been used, such as WS-BPEL, jPDL or the new 'conversation aware' ESB actions, it will be possible to infer the behaviour of the service being implemented, to compare it against the choreography description. Currently this has been implemented for the “conversation aware” ESB actions, and is demonstrated using the samples in this Overlord-CDL distribution.

Detecting incorrectly implemented behaviour at the earliest possible time saves on downstream costs associated with finding and fixing errors. By using static validation against the original design, it ensures that the implemented service will deliver its expected behaviour first time. This is important in building large scale SOAs where different services may be implemented in different locations.

There are two other areas where a choreography description can be used as part of design-time governance, that are not currently implemented in Overlord:

- Service lookup – the choreography description can be used to determine if a service already exists in the Service Repository that meets the appropriate behavioural requirements.
- Service unit testing - this can be achieved using the scenarios originally specified to document the behavioural requirements. Rather than develop an independent source of test data, the scenarios can be used to validate the sequence of messages sent to, and received from, a service, as well as validating the contents of the messages returned from the service under test.

## 1.3.2. Runtime Governance

Runtime governance ensures that the SOA executes as expected according to predefined policies. In this context, a choreography description can be used in two ways.

### 1.3.2.1. Service validator

The choreography description represents the interactions between multiple services to deliver a business goal. To validate the behaviour of each individual service, within the choreography description, the behaviour of each service can be derived from the choreography.

The derived behaviour (or “endpoint projection”) of a service can be used within a 'service validator' to monitor the inbound and outbound messages for the service, to ensure they conform to the expected behaviour. If an invalid message is detected, it would be possible to block it, to prevent it from causing subsequent problems in downstream systems. The error can also be reported to a central management capability.

The CDL component of Overlord provides the ability to configure service validators to monitor the behaviour of individual services. An enhanced version of the JBossESB trailblazer example has been included, with the appropriate validator configuration, to demonstrate this mechanism.

#### 1.3.2.2. Process correlation

Validating each service locally can enable errors to be detected quickly, and the effects of the error prevented from contaminating other systems by blocking the erroneous messages.

However local service specific validation may not be adequate to identify errors that would affect the end-to-end business process. Therefore the message activity at each service validator can be reported to a central 'process correlation engine' which can reconstitute a global view of the business transaction, and determine if it matches the expected behaviour as defined in the choreography description.

The benefit of a correlated global view of the distributed business transaction is that it can be further analysed to ensure other governance polices have been followed – e.g. SLAs.

The pi4soa tool suite includes a simple GUI based monitoring tool to display the information obtained from correlating message events associated with individual services. The trailblazer example has been written to cause out of sequence messages under certain circumstances. See the “Samples Guide” for more information on how to run this example.

## 1.4. First Steps

The first step will be to follow the instructions in the Getting Started Guide to install Overlord.

Once installed, the next step should be to try out the examples in the samples folder. The examples consistent of:

- Service Validation related examples
  The samples folder contains an enhanced version of the trailblazer example from the JBossESB, with the addition of a File Based Bank, and message content including a conversation id to enable the messages to be correlated with a specific session.
- Conversation aware ESB actions, with conformance checking against Choreography
  Two examples have been included, one simple example (purchasing) and the other more advanced (brokerage). Both relate to the business process of purchasing items. The second example introduces the concept of a broker to act on behalf of the customer, interacting with multiple potential suppliers.

  These examples show how a service implementation (built using “conversation aware ESB actions” in this case), can be continuously checked for conformance against a choreography description.

  The final step should be to review all of the documents in the docs folder to understand more about each capability, and then try using the techniques on your own project.

# Conversation Validation with CDL

## 2.1. Overview

Conversation validation is a form of runtime governance concerned with the dynamic behaviour of a system.

When coupled with a choreography description model of a system, this means having the ability to ensure that the way a collection of services interact correctly adheres to a description of the business process being enacted.

This section introduces the choreography description language (CDL) defined by W3C, and the *pi4soa* open source project which provides an editor for creating choreography descriptions, as well as utilizing these descriptions for runtime validation and execution purposes.

## 2.2. Configuration of Conversation Validation

This section explains how to configure the conversation validation mechanism to validate ESB services against a choreography description. The first sub-section describes how the mechanism is hooked into the JBossESB environment. The following two sub-sections explain two alternate ways that relevant endpoint references can be configured for validation.

### 2.2.1. Installing the Conversation Validation Mechanism

The principle mechanism used for validating conversations within an ESB is through the use of a global filter registered with the *jbossesb-properties.xml*. This file is located in the *$JBossESB/server/default/deploy/jbossesb.sar* folder.

```
<properties name="filters">
    ...
    <property name="org.jboss.soa.esb.filter.10"
            value="org.jboss.soa.overlord.validator.jbossesb.ValidatorFilter"/>
</properties>
```

This filter is installed as part of the installation process for the Overlord-CDL distribution.

### 2.2.2. Explicit Configuration

The information concerning which destinations will be validated, and to which model/role they relate, can be explicitly defined within the *validator-config.xml* file, contained within the *overlord-cdl-validator.esb* bundle.

An example of the contents of this file, that would related to the TrailBlazer example, is:

```
<validator mode="monitor" replyToTimeout="10000" >
    <service model="TrailBlazer.cdm"
                role="LoanBrokerParticipant" >
        <output epr="jms:queue/esb-tb-creditAgencyQueue" />
        <input epr="jms:queue/esb-tb-creditAgencyQueue_reply" />
        <output epr="jms:queue/esb-tb-jmsBankRequestQueue" />
        <output epr="jms:queue/esb-tb-fileBankRequestQueue" />
        <input epr="jms:queue/esb-tb-jmsBankResponseQueue" />
        <output epr="jms:queue/esb-tb-customerNotifier" />
        <input epr="jms:queue/esb-tb-fileBankResponseQueue" />
    </service>
    <service model="TrailBlazer.cdm"
                role="CreditAgencyParticipant" >
        <input epr="jms:queue/esb-tb-creditAgencyQueue" />
        <output epr="jms:queue/esb-tb-creditAgencyQueue_reply" />
    </service>
    <service model="TrailBlazer.cdm"
                role="BankParticipant" >
        <input epr="jms:queue/esb-tb-jmsBankRequestQueue" />
        <input epr="jms:queue/esb-tb-fileBankRequestQueue" />
        <output epr="jms:queue/esb-tb-jmsBankResponseQueue" />
        <output epr="jms:queue/esb-tb-fileBankResponseQueue" />
    </service>
    <service model="TrailBlazer.cdm"
                role="NotifierParticipant" >
        <input epr="jms:queue/esb-tb-customerNotifier" />
    </service>
</validator>
```

The 'validator' element has an optional attribute called 'mode', with the possible values of 'monitor' or 'manage'. If the mode is 'monitor' (which is the default), then any messages that result in validation errors being detected will continue to be received or sent, with the errors only be reported for information purposes. If the mode is 'manage', then any erronous messages detected during validation, that conflict with the behaviour as described in the choreography, will be prevented from being received or sent.

### Note

It is important to note that if 'manage' validation mode is used, then the validation mechanism will be an integral part of the message flow. This may have a slight performance impact on the delivery of messages between services.

The optional 'replyToTimeout' (defined in milliseconds) is used to determine how long a dynamic reply-to destination should be monitored for validation purposes. In some message exchanges, the response destination will not always be known in advance. Therefore the configuration can identify such situations, and monitor the reply-to destination for the response. However, if a response is not delivered in a particular time period, we need to be able to discontinue the validation of the dynamic endpoint. If this did not occur, then over time too many endpoints would be monitored, which may result in out-of-memory problems. The default timeout period is 10 seconds.

Within the 'validator' element is a list of 'service' elements, one per service being validated. The behaviour of the service being validated is identified by specifying the model (e.g. choreography description file) and the role (e.g. participant type) within the model. Therefore, within the above configuration, the first set of destinations (eprs) are associated with the *LoanBrokerParticipant* defined within the choreography description model found in the file `TrailBlazer.cdm`, which will be located within the `models` folder contained within the *overlord-cdl-validator.esb* bundle.

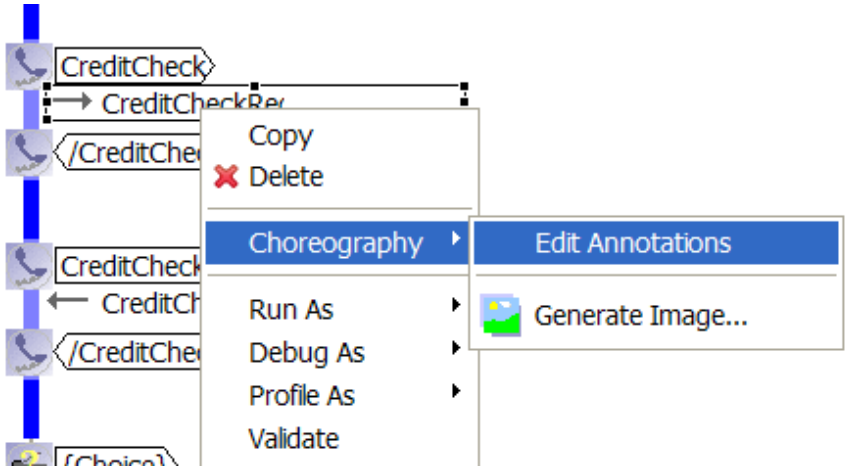
The elements contained within the 'service' element define the *input* and *output* eprs (Endpoint References) that are associated with the service. The *input* eprs are the destinations on which messages will be received and the *output* eprs are the destinations on which messages will be sent by the service.

The format of the 'epr' attribute will be specific to the type of transport used for the ESB aware destination. Currently only JMS is supported, and can be identified by the protocol prefix 'jms:'.

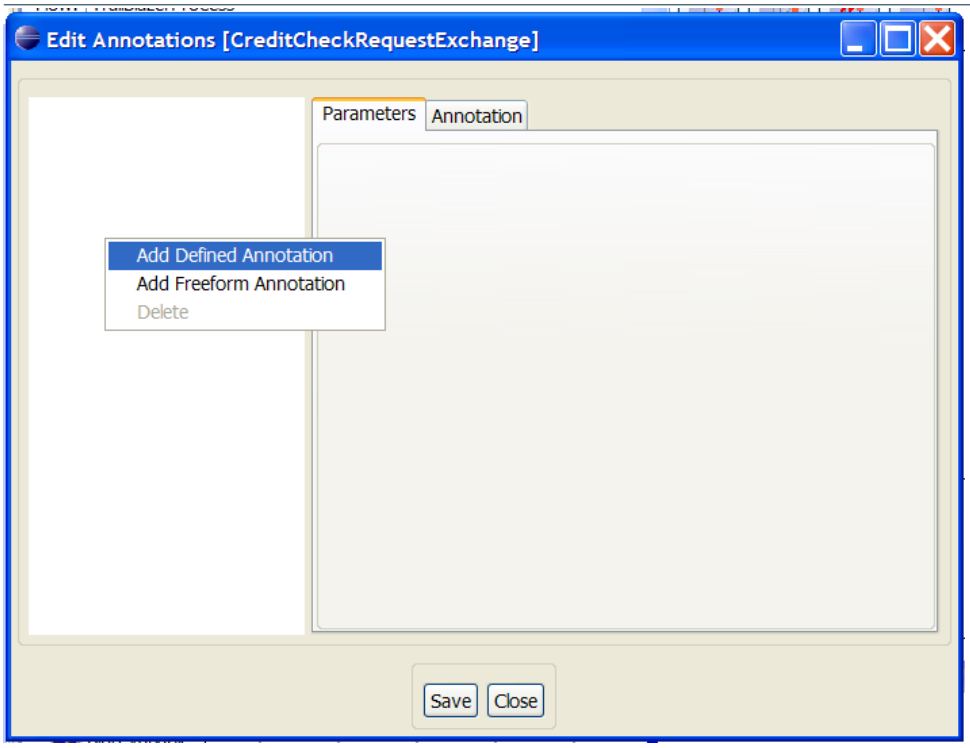
Each 'input' and 'output' element can also define an optional 'dynamicReplyTo' boolean attribute. If defined, it will indicate to the Service Validator that the message on the specified endpoint (epr) will contain a dynamically defined 'reply-to' destination that needs to be monitored for a response.

## 2.2.3. Defining the Validator Configuration within a Choreography

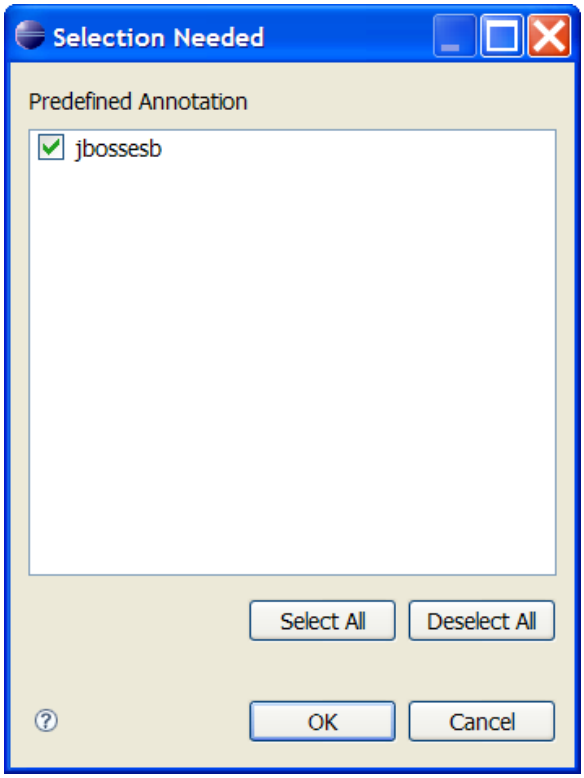
The first step to configuring the validator is to associate the endpoint references (EPRs) against the relevant choreography interactions. This is achieved by defining an annotation for each 'exchange details' component (i.e. each request and response/notification).



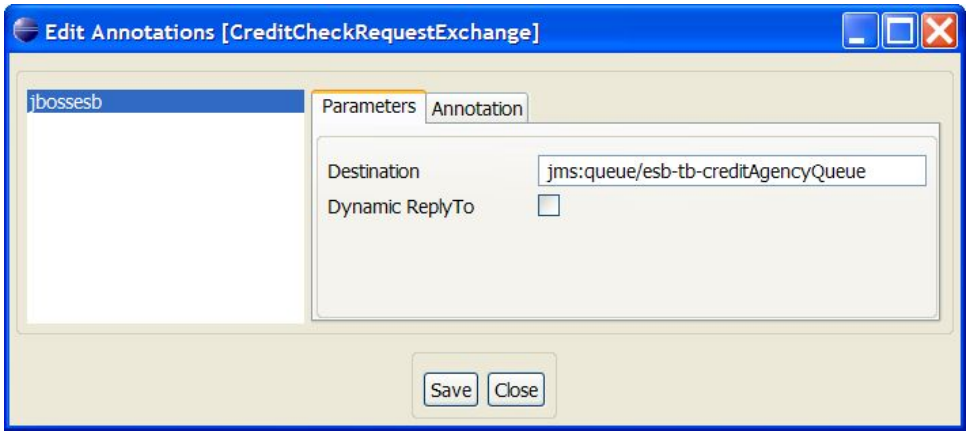

When the annotation editor is displayed for the relevant 'exchange details' component, the *jbossesb* annotation should be added. This is achieved by selecting the popup menu associated with the background of the lefthand panel, and selecting the *Add Defined Annotation* menu item.




When the list of defined annotations is displayed, select the *jbossesb* annotation.




After pressing the *Ok* button, the annotation editor will configure the righthand panel with the parameters associated with this annotation.




To specify the EPR for a particular message exchange, enter the EPR into the *Destination* field. If the exchange is a request, that will result in a response being sent on a dynamically provided "reply-to" destination, then the *Dynamic Reply-To* checkbox should be selected.

Once the annotation has been defined, then press the *Save* button to save the annotation against the interaction's exchange details.

When all of the relevant 'exchange details' components have been configured with a *jbossesb* annotation, defining the EPR to be validated, then the choreography description file can be copied into the `overlord-cdl-validator.esb/models` folder. This will cause the validation mechanism to derive the configuration information from the choreography description model, and begin validating the defined destinations against that choreography description model.

## 2.3. Monitoring the Choreography Description

Once the JBossESB environment has been configured, to perform service validation of a set of ESB services against a choreography description, and the server has been started, then the next step is to launch a tool to view the correlated information from the service validators - and determine if the transactions are being correctly executed.

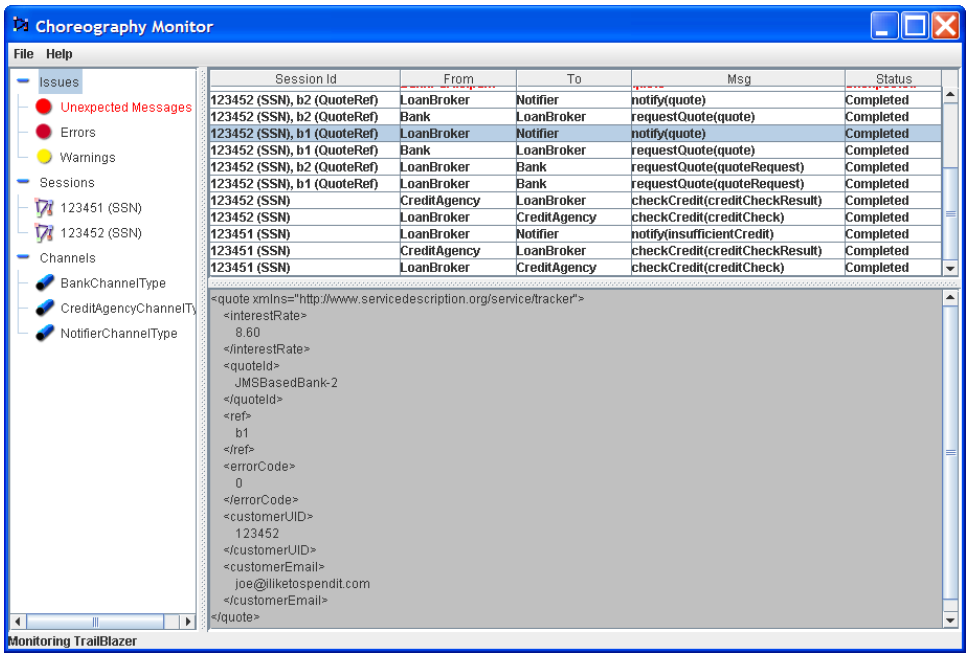
Within an Eclipse Java project, that contains the choreography description to be monitored, a configuration file called *pi4soa.xml* needs to be defined on the project's classpath. This file provides details of the JMS configuration parameters required to subscribe for the information generated by the service validators. The contents of this file is:

```
<pi4soa>
    <tracker>
        <jndi>
         <initialContextFactory>org.jnp.interfaces.NamingContextFactory</initialContextFactory>
                <providerURL>jnp://localhost:1099</providerURL>
                <factoryURLPackages>org.jboss.naming:org.jnp.interfaces</factoryURLPackages>
            </jndi>
        <jms>
                <connectionFactory>ConnectionFactory</connectionFactory>
                <connectionFactoryAlternate>ConnectionFactory</connectionFactoryAlternate>
```

```
                <destination>topic/tracker</destination>
        </jms>
    </tracker>
</pi4soa>
```

The destination defined in this file must match the one configured in the *pi4soa.sar/pi4soa.xml* file within the server.

The next step is to launch the monitoring tool. This is located on the popup menu, for the choreography description (i.e. .cdm) file, by selecting the Choreography->Monitor menu item. Once the tool has been launched, it will load the choreography description, subscribe to the relevant event destination, and then indicate via a message in the bottom status line that it is ready to monitor.




When the information is received, from the service validators representing the different participants (services), it is correlated to show the global status of the business transaction. The list of correlated interactions is show in reverse time order in the image, so in this example a *LoanBroker* sends a *creditCheck* message to a *CreditAgency*, followed by a *creditCheckResult* being returned.

If any *out of sequence* or other error situations arise, these are displayed in red.

## 2.4. Configuration for Conversation Recording

As well as validating the interactions between a set of services, against a pre-defined choreography description, it is also possible to use the *Service Validators* in a non-validating record mode.

This will be useful in situations where a choreography description does not currently exist, and we wish to use the stream of business events being sent and received by each identified service (or participant type) to gain an understanding of the current business process.

An example of this type of configuration, associated with the TrailBlazer example, is:

```
<validator>
    <service role="LoanBrokerParticipant" validate="false" >
        <output epr="jms:queue/esb-tb-creditAgencyQueue" />
        <input epr="jms:queue/esb-tb-creditAgencyQueue_reply" />
        <output epr="jms:queue/esb-tb-jmsBankRequestQueue" />
        <output epr="jms:queue/esb-tb-fileBankRequestQueue" />
        <input epr="jms:queue/esb-tb-jmsBankResponseQueue" />
        <output epr="jms:queue/esb-tb-customerNotifier" />
        <input epr="jms:queue/esb-tb-fileBankResponseQueue" />
    </service>
    <service role="CreditAgencyParticipant" validate="false" >
        <input epr="jms:queue/esb-tb-creditAgencyQueue" />
        <output epr="jms:queue/esb-tb-creditAgencyQueue_reply" />
    </service>
    <service role="BankParticipant" validate="false" >
        <input epr="jms:queue/esb-tb-jmsBankRequestQueue" />
        <input epr="jms:queue/esb-tb-fileBankRequestQueue" />
        <output epr="jms:queue/esb-tb-jmsBankResponseQueue" />
        <output epr="jms:queue/esb-tb-fileBankResponseQueue" />
    </service>
    <service role="NotifierParticipant" validate="false" >
        <input epr="jms:queue/esb-tb-customerNotifier" />
    </service>
</validator>
```

To define a *Service Validator* in record only mode, the *model* attribute is not specified (because no choreography description exists to be validated against), and the optional *validate* attribute should be set to **false** (by default this attribute is **true**).

# Conversation Aware ESB

## 3.1. Conversation based Conformance

**Warning**

The *conversation aware ESB actions* mechanism should be considered an alpha version only, and subject to change in future releases. Its inclusion within this release is intended to enable the community to experiment with the approach and hopefully provide feedback that can be used to guide the direction of this capability.

### 3.1.1. Overview

The term "conversation" represents a structured set of interactions (or message exchanges) between one or more peer to peer services, to conduct a business transaction. The "conversation" is defined from a service neutral (i.e. global) perspective.

This document explains how such a "conversation" description can be used to ensure conformance of one or more service implementations, within an ESB, during the design and implementation phase of the system.

This section introduces the choreography description language (CDL) defined by W3C, which is a standard notation for defining conversations from a global perspective, and the *pi4soa* open source project which provides an editor for creating choreography descriptions, as well as utilizing these descriptions for conformance checking, monitoring and execution purposes.

Finally the section will provide a brief discussion of how CDL can be used to provide conformance checking of an ESB, through the use of 'conversation aware' ESB actions.

### 3.1.2. CDL Conformance Checking

In general, conformance checking is the procedure for ensuring that a component has been correctly built according to some specification or standard. In terms of CDL, it more specifically ensures that one or more services perform their responsibilities correctly in accordance with the choreography description.

The *pi4soa* tools suite provide the mechanism for producing service endpoint descriptions for each service within a choreography description. The relevant service descriptions can then be compared (for conformance) against their ESB service implementations.

However, to make this possible, it is necessary to be able to derive the communication 'type' structure from the ESB implementation, i.e. where messages (of particular types) are sent and received, where decision points are, where actions are performed concurrently, etc.

This is why a specific set of 'conversation aware' ESB actions have been defined (discussed in a later section), to make it possible to derive the communication 'type' structure from an ESB service implementation.

Once the communication 'type' structure has been obtained from the ESB implementation, it can be compared against the relevant service endpoint description projected from the choreography description, to determine if there are any differences. These can then be reported to the ESB service developer, so that they can fix the problems, before the service is deployed to the runtime environment.

This ensures that all of the services will interaction correctly, as they will all have been validated against the choreography, and therefore work together by design.

In the following two sections, we will describe how ESB services can be described using either *Stateful* or *Stateless* "conversation aware" ESB actions. The benefits of each approach will be discussed.

## 3.2. Stateful "Conversation Aware" ESB Actions

### 3.2.1. Overview

This section outlines the various *stateful* "conversation aware" ESB actions that can be used to make the communication behaviour of a service implementation explicit, thus enabling it to be compared for conformance against a description of the expected behaviour.

The benefit of this *stateful* approach is that the stateful behaviour of a service is explicitly defined, and can be used to perform a precise conformance check against the stateful behaviour expected from a choreography description. The disadvantage of the approach, and the reason why this section describes many more ESB action types than in the following section, is that the ESB service needs to maintain additional state information related to the individual sessions (i.e. a session per business transaction), and this state information needs to be persisted independently from other business information that the service may be accessing. This state information can be used to determine when messages have been received out of order, however this functionality is also available in the conversation validation mechanism.

### 3.2.2. Conversational Service

The top level component is the *Service*, which will have a single endpoint reference (i.e. service category and name) that will be used by external clients (or other services) that interact with this service.

However the service behaviour is stateful, and therefore will need a means of routing the inbound request to the appropriate service descriptor that is (a) capable of handling the request, and (b) the service instance is in an appropriate state where the service descriptor can be executed.

#### 3.2.2.1. Message Router Action

The action used to perform routing of the inbound requests is called *MessageRouterAction*, for example:

```
<service category="ESBBroker.BrokerParticipant" name="ESBBrokerProcess" description="">
    <listeners>
        <jms-listener name="BrokerServiceListener"
                      busidref="BrokerService"
                      maxThreads="1"/>
    </listeners>
```

```
        <actions mep="OneWay">
          <action class="org.jboss.soa.overlord.jbossesb.stateful.actions.MessageRouterAction"
                      process="process" name="s0-1">
                <property name="paths">
                    <route  service-category="ESBBroker.BrokerParticipant"
                            service-name="ESBBrokerProcess.main"
                            initiate="true">
                        <message type="enquiry">
                            <identity type="primary" >
                              <token name="id" locator="//@id" />
                           </identity>
                        </message>
                    </route>
                    <route  service-category="ESBBroker.BrokerParticipant"
                            service-name="ESBBrokerProcess.main.5" >
                        <message type="buy">
                           <identity type="primary" >
                             <token name="id" locator="//@id" />
                           </identity>
                        </message>
                        <message type="cancel">
                           <identity type="primary" >
                             <token name="id" locator="//@id" />
                            </identity>
                        </message>
                    </route>
                </property>
            </action>
        </actions>
    </service>
```

In this example, the 'service' endpoint reference will be associated with the service category "ESBBroker.BrokerParticipant" and name "ESBBrokerProcess". All inbound requests to an instance of this service will be routed via this service descriptor.

This service descriptor therefore only has a single action, which represents the message routing capability. The action class is *org.jboss.soa.overlord.jbossesb.stateful.actions.MessageRouterAction*. This action only defines a single property 'path' which defines one or more 'route' elements. The attributes associated with this 'route' element are:

- service-category and service-name, together identify the service descriptor that should be invoked if this route is selected
- an optional 'initiate' boolean attribute. If the attribute is specified, and its value is 'true', then the route can only be selected if the service instance relevant for the inbound message does not yet exist, and this message will result in a *CreateSessionAction* being invoked to create the service instance. If the attribute value is 'false', or not specified, then a service instance must already exist that is capable of handling the inbound message.

The 'route' element will contain one or more 'identity' elements, and one or more 'messageType' elements.

The 'identity' element is used to extract information from the inbound message that can be used to identify the appropriate service instance. The only attribute on the 'identity' element is the type of identity, which can be:

- primary
  the primary identity field, used to associate a message with the session
- alternate
  an alternative primary identity
- association
  link the message to a session based on an identity previously associated with the session (or parent session). However this identity will not be associated with the current session, it is usually only used to link a child session to a parent session
- derived
  the extracted identity will be placed in reserve for use as the primary identity for a subsequent session. It is not directly associated with the session in which it is discovered

The one or more 'token' definitions contained within the 'identity' element provide the details regarding the structure of the identity. The token has 'name' and 'locator' attributes, the locator being used to provide an expression that locates the identity information within the inbound message. If more than one token is defined, it provides a composite identity (i.e. made up of multiple parts).

The one or more 'messageType' elements, contained within the 'route' element, defines the message type(s) that should be routed to the service descriptor associated with the 'route' element.

**Note**

If a route is marked as *initiate='true'*, with the correct message type for an inbound message, but a service instance already exists for the identity information extracted from the message, then the route will not be selected. The converse is also true.

**Note**

Similarly, even if a message type match is found, if the service instance is not in an appropriate state to invoke the target service descriptor, then the route will not be selected.

If no routes are found for a particular inbound message, then an exception will be reported.

## 3.2.3. Establishing Correspondance between ESB Configuration and Choreography

All actions (and therefore service descriptors) for a particular service implementation must be defined within the same ESB configuration file. This is a current requirement, to ensure that all of the inter-related service descriptors are available to support the static validation (e.g. conformance checking).

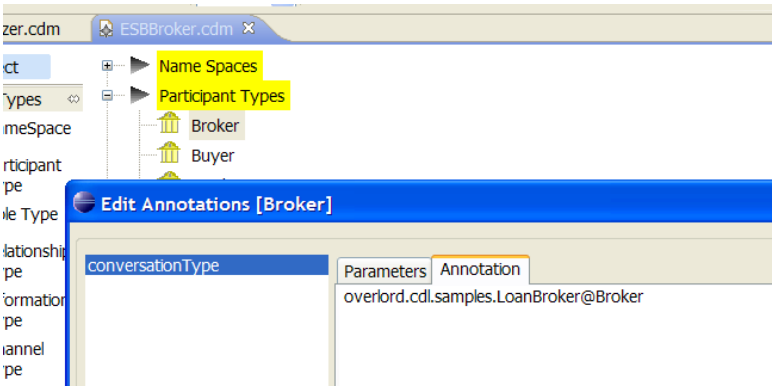
The `jboss-esb.xml` file must be defined within either the same Eclipse project as the *pi4soa* choreography description (.cdm file), or in a project that has a 'project reference' to the project containing the choreography description.

To establish the link between the choreography description and the `jboss-esb.xml`, both must 'implement' a common *conversation type*.

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.CreateSessionAction"
                process="process" name="...">
    ....
    <property name="conversationType"
        value="overlord.cdl.samples.LoanBroker@Broker" />
    ....
</action>
```

The *conversation type* is specified in the *CreateSessionAction* ESB action, described later in this section, and is associated as a semantic annotation against the relevant Participant Type or Participant Instance within the choreography description.

To associate the *conversation type* with either the Participant Type or Instance, open the choreography description in the *pi4soa* choreography designer. Then select the "Choreography->Edit Annotations" menu item from the popup menu associated with the Participant Type or Instance. In the lefthand panel, select "Add Freeform Annotation" from the popup menu, then specify "conversationType" as the annotation type and press 'Ok'. Then select the 'Annotation' tab in the righthand panel, and enter the conversation type, e.g. "overlord.cdl.samples.LoanBroker@Broker". Note the '@' is important, as the following word indicates the 'role' associated with the conversation type which precedes the '@' symbol.



## 3.2.4. Managing Sessions

A "session" represents a specific instantiation of a service definition that is involved in a business transaction (or conversation). The "session" will be distinguished based on unique identity information relevant to the business transaction, that is derived from specific properties within the messages being exchanged by the interacting services.

The service description can be composed from other reusable sub-service descriptions, although the service can only define a single 'root' (i.e. top level) definition.

### 3.2.4.1. Business State and Logic

Each top level or sub-service description will be associated with a business state 'pojo' class that contains its business relevant information and logic (i.e. methods). This class is identified by the property 'session', which will be associated with the initial conversation based ESB actions in a service (or sub-service) description, or in any subsequent conversation based ESB actions that require the information to determine the session instance.

As well as representing the business state and logic for a session (or sub-session), the pojo can define an annotation that provides information about the session. The annotation type is `org.jboss.soa.overlord.jbossesb.stateful.actions.Service`.

```
package org.jboss.soa.overlord.samples.jbossesb.loan.broker;

import org.jboss.soa.overlord.jbossesb.stateful.actions.Service;

@Service(name="{http://www.jboss.org/overlord/loanBroker}Broker",
  conversationType="overlord.cdl.samples.LoanBroker@Broker",
     root=true)
public class BrokerMain {
  ....
}
```

The 'name' attribute represents a name associated with the complete service description, encompassing both top level and composed sub-session behaviour. The 'root' attribute indicates whether this pojo is related to the top level session, or a sub-session.

For pojos that represent the 'root', or top level session, for a service description, they should also specify the optional 'conversationType' attribute. This expresses the type that will be checked for conformance, and is comprised of two parts separated by the '@' symbol. The first part is a fully qualified name of the conversation type. The second part represents the role being played within the conversation type.

If the *Service* annotation is not defined on the pojo, then the information (service description name, root and optional conversation type) must be explicitly defined in the relevant conversation based ESB actions.

### 3.2.4.2. Instantiating top level and child sessions

A session, whether top level or a child sub-session, will be instantiated by defining a service descriptor that starts with a *CreateSessionAction*. For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.CreateSessionAction"
                        process="process" name="...">
    <property name="session"
        value="org.jboss.soa.overlord.samples.jbossesb.loan.broker.BrokerMain" />
</action>
```

where the 'session' property references the pojo defined previously. What distinguishes whether the session is a top level or child sub-session is the value of the 'root' attribute on the pojo annotation (or property on the *CreateSessionAction* if no annotation is defined on the pojo), i.e. if root is true, then the service descriptor will represent the top level session.

Only a single top level (root) session can be defined per service description.

### 3.2.4.3. Retrieving an existing session

Certain service descriptors, and actions within those service descriptors, will need to be able to access the session in which they are executing. In many situations the session will automatically be available due to prior activities that may have occurred, causing the session reference to be cached and retrieved when required.

However, in some situations (such as receiving a response message from another service) there may not be enough context information to understand which session should be retrieved from the database to handle the incoming message.

The solution to this problem is to ensure that the first "conversation aware" ESB action in the receiving service descriptor has the additional information required to resolve the context. This information will include:

- Message identities
  Message identities refer to the extraction of information from the message content to be used to uniquely identify a session instance. This information can be associated with any "conversation aware" ESB action, and has the following format:

```
<property name="identities" >
    <identity type="primary" >
        <token name="<name>" locator="<xpath expression>" />
    </identity>
</property>
```

Multiple identities can be derived from an incoming message. Each identity will have a type, either *primary*, *alternate*, *derived* or *association*.

Each identity can have one or more tokens defined. If more than one is defined, then it will represent a composite identity.

The name of the token is not relevant, just used for information purposes. The main purpose of the token is to locate a value from the message content.

- Session pojo
  The session pojo class can be used to narrow down the specific sub-session, within a service instance, that the message should be routed to.
- Service description name (which may be available on a *Service* annotation on the Session pojo).

## 3.2.5. Interacting

Services interact by sending and receiving messages, whether synchronously or asynchronously.

JBossESB is designed to anonymously handle inbound messages (possibly requests), without explicitly defining restrictions on message type, and then optionally returning responses (again without explicitly defining the response message type).

Although this is sufficient for a runtime mechanism, where issues related to unexpected message types can be handled with suitable exceptions/faults, it does not enable the communication type structure to be understood by examination of the JBossESB configuration.

Therefore, the set of "conversation aware" ESB actions include actions for sending and receiving messages. Although these actions are not strictly necessary for the ESB to process messages, they make the communication behaviour of the ESB service explicit, which enables it to be statically checked (validated) against a description of the expected behaviour.

### 3.2.5.1. Sending a message

When sending a message, the first thing to consider is the type of the message. This can be declared as a property on the *SendMessageAction*. If dealing with RPC style interactions, then we may also want to optionally specify an operation name.

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SendMessageAction"
            process="process" name="...">
    <property name="operation" value="getQuote" />
    <property name="messageType" value="requestForQuote" />
    ....
</action>
```

In this example, the message type has no 'namespace'. If a namespace is appropriate, it can be defined inside {} curly braces (e.g. "{http://www.jboss.org/examples}requestForQuote" for an XML type, or "{java:org.jboss.example}Quote" for a Java type).

The next important aspect is to define the destination of the message. This will be dependent upon whether the message being sent is a request or a response/notification.

- Sending a request
  When sending a request, we need to identify the destination service category/name. This is done by either specifying the category and name explicitly, using the *serviceCategory* and *serviceName* properties:

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SendMessageAction"
            process="process" name="...">
    ....
    <property name="serviceName" value="RequestForQuote.main" />
    <property name="serviceCategory" value="ESBBroker.SupplierParticipant" />
    ....
</action>
```

  or based on expressions using the *serviceCategoryExpression* and *serviceNameExpression* properties:

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SendMessageAction"
            process="process" name="...">
    ....
    <property name="serviceNameExpression" value="supplier.serviceName" />
    <property name="serviceCategoryExpression" value="supplier.serviceCategory" />
    ....
</action>
```

  The second approach enables the category/name details to be obtained from the session pojo, using an expression (see later for details of specifying expressions).

  If a response is expected from the request, then the optional *responseServiceName* and *responseServiceCategory* can be used to specify the service descriptor that should receive the response message.

- Sending a response

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SendMessageAction"
            process="process" name="...">
```

```
        ....
        <property name="clientEPR" value="buyer" />
        ....
    </action>
```

When sending a response, the destination will not be directly available. The destination would have been received as a 'reply to' EPR (Endpoint Reference) in a previously received request (see *ReceiveMessageAction* for details of how to store the 'reply to' EPR).

Therefore, to indicate which EPR to respond to, a property called 'clientEPR' is specified with the name of the stored EPR. This must match up to a previously stored EPR name within a *ReceiveMessageAction*.

The final part of the *SendMessageAction* is the identity declaration. See discussion on Message Identities early in this chapter.

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SendMessageAction"
            process="process" name="...">
    ....
    <property name="identities" >
        <identity type="primary" >
            <token name="id" locator="//@id" />
            <token name="supplierId" locator="//@supplierId" />
        </identity>
    </property>
</action>
```

The identity information extracts values from the message content being sent, and determines whether it correlates with the identity already associated with the session (or sub-session). If the message is correctly correlated to the session, then it will be sent. Otherwise an exception will be generated.

It is possible that multiple identities can be defined, associating new 'alternate' or 'derived' identities with the session (or sub-session). These can be used to link subsequent messages (send or received) to the same session.

### 3.2.5.2. Receiving a message

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.ReceiveMessageAction"
            process="process" name="s1-2">
    <property name="operation" value="makeEnquiry" />
    <property name="messageType" value="enquiry" />
    <property name="clientEPR" value="buyer" />
    <property name="identities" >
        <identity type="primary" >
            <token name="id" locator="//@id" />
        </identity>
```

```
        </property>
    </action>
```

The *ReceiveMessageAction* is used to explicitly define the message type that should be received. If an RPC style has been used, then the optional operation name can also be defined.

Unlike the *SendMessageAction*, which will actually send a message to the specified service category/name, the *ReceiveMessageAction* primarily serves to provide explicit details about the expected message and to perform any relevant validation of the message content, including conforming that the extracted identity details correlate correctly with the session (or sub-session). If an incorrect message type is received, or an appropriate session cannot be located, then an error will be logged.

The optional 'clientEPR' property is used to store any specific "reply to" EPR (associated with the message) against the specified name. This makes the EPR accessible to any subsequent *SendMessageAction* activities that need to return a response or send a notification.

## 3.2.6. Managing Information

### 3.2.6.1. Manipulating State Information

As previously mentioned, each session or sub-session is associated with a business state pojo which contains the information required by the session. The relevant class is identified by the 'session' property on appropriate "conversation aware" ESB actions.

When an action needs to access (read) information defined on this business state pojo, it will simply define an expression that can access (and navigate) the properties and invoke methods on the object where appropriate.

However we also need to have the ability to modify the state information. This is achieved using the *SetStateAction*. For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SetStateAction" name="...">
    <property name="variable" value="supplierIndex" />
    <property name="stateExpression" value="nextSupplier()" />
</action>
```

With this action, it is possible to specify a target 'variable' on the pojo business object, associated with the session (or sub-session), and have the value associated with an expression assigned to that variable.

In the example above, the expression is defined using the 'stateExpression' property. This expression will be applied to the pojo business object associated with the session (or sub-session), to extract information from variables, or invoke methods on the object.

The other type of expression that can be used is defined in the 'messageExpression' property. This will apply the expression to the content of the current message on the ESB action pipeline.

#### 3.2.6.2. Manipulating Message Information

The other action that can be used to manipulate information is *SetMessageAction*. This action can be used to set the contents or a header property of the message on the ESB action pipeline. For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SetMessageAction"
            process="process" name="...">
    <property name="headerProperty" value="quoteList" />
    <property name="stateExpression" value="quotes" />
</action>
```

This example shows how information from the session's associated business pojo state can be placed in a header property of the current message passing through the action pipeline.

If the 'headerProperty' property is not specified, then the value extracted from the state object will be placed in the default entry within the message contents, replacing any existing value.

### 3.2.7. Controlling Flow

This section describes the various control flow mechanisms that are supported by the "conversation aware" ESB actions.

The default control flow, supported natively by the ESB action pipeline design, is a sequence. As the name implies, the actions are performed one at a time in the order they defined in the action pipeline, i.e. in a sequence.

#### 3.2.7.1. Selecting paths based on a decision

The action associated with the 'selection of a path based on a decision' is the *IfAction*. An example of this construct is:

```
   <action class="org.jboss.soa.overlord.jbossesb.stateful.actions.IfAction" process="process"
name="...">
       <property name="paths">
           <if expression="isCreditHistoryAvailable()"
                   service-category="ESBBroker.CreditAgency"
                   service-name="CreditAgency.decision1"
                   immediate="true" />
           <elseif expression="isBadCreditHistory()"
                   service-category="ESBBroker.CreditAgency"
                   service-name="CreditAgency.decision2"
                   immediate="true" />
           <else service-category="ESBBroker.CreditAgency"
                   service-name="CreditAgency.decision3"
                   immediate="true" />
       </property>
   </action>
```

This construct defines a 'path' property with one or more elements, representing the *if*, *elseif* and *else* aspects of the traditional if/else construct. Only the *if* element is mandatory, and can be followed by zero or more *elseif* elements before ending with the optional *else* element.

The *if* and *elseif* elements can define an 'expression' attribute to be evaluated at runtime, to determine if the associated 'service-category' and 'service-name' should be invoked.

If the 'immediate' boolean attribute is defined as true on any of the elements, it means that the associated service category/name should be invoked immediately (i.e. it represents a control link). However, if this attribute is false, or not specified, then the associated service category/name is expected to be triggered upon the receipt of a message from an external source.

### 3.2.7.2. Selecting paths based on the type of a received message

The action used to select paths based on a received message type is *SwitchAction*. For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.SwitchAction"
                process="process" name="...">
    <property name="paths">
        <case service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.7" >
            <message type="buy" />
        </case>
        <case service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.6" >
            <message type="cancel" />
        </case>
    </property>
</action>
```

The 'paths' property defines one or more 'case' elements. These elements identify a service category and name that should be invoked upon receipt of one or more message types, as specified by 'message' elements contained within the 'case' elements.

The 'type' attribute on the 'message' element defines the type of message that can be routed to the specified service category/name. In the example above, the message types have no namespace. However if they have a namespace, this can be defined in curly braces, e.g. "{http://www.jboss.org/samples}buy".

Once a path has been selected, the associated service category/name will be invoked immediately. If none of the paths specified within this action are relevant to the received message, then a runtime exception will be thrown.

### 3.2.7.3. Executing multiple paths concurrently

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.ParallelAction"
                process="process" name="...">
    <property name="paths">
        <path service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.2"
                immediate="true" />
        <path service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.3"
                immediate="true" />
        <join service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.4" />
    </property>
</action>
```

The *ParallelAction* is used to initiate multiple concurrent threads of activity. These are represented by the 'path' elements within the 'paths' property. Each path identifies its service category/name and also whether it should be invoked immediately (i.e. 'immediate' attribute set to 'true'), or will be triggered by a message from an external source.

The other element contained within the 'paths' property is the 'join' element, which represents the service category/name that acts as a convergence point across all of the concurrent threads. When all of the initiated threads have completed, the service category/name associated with the 'join' element will be invoked.

### 3.2.7.4. Repetition

The mechanism for performing repetition is through the use of two actions, *WhileAction* and *ScheduleStateAction*. The *WhileAction* defines the condition to be evaluated, and the two relevant paths, i.e. one that enters the loop body and one that represents the activities that occur after the while loop, when the conditions have not been met. The *ScheduleStateAction* is used within the loop body to return back to the service descriptor containing the *WhileAction*, to cause the loop to be re-evaluated.

For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.WhileAction"
                process="process" name="...">
    <property name="paths">
        <while  expression="hasNextSupplier()"
            service-category="ESBBroker.BrokerParticipant"
            service-name="ESBBrokerProcess.main.2"
            immediate="true" />
        <exit service-category="ESBBroker.BrokerParticipant"
            service-name="ESBBrokerProcess.main.3"
            immediate="true" />
    </property>
</action>
```

The *WhileAction* defines a 'paths' property which contains two elements, a 'while' and an 'exit' element. Both elements define the service category and name they will invoke, and whether the service descriptors will be triggered immediately or based on an incoming message from an external source.

Additionally, the 'while' path defines the condition that will decide whether to enter or skip the loop.

Then at some point in the activities directly or indirectly associated with the service descriptor "ESBBroker.BrokerParticipant/ESBBrokerProcess.main.2", there would be,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.ScheduleStateAction"
                process="process" name="...">
    <property name="serviceCategory" value="ESBBroker.BrokerParticipant" />
    <property name="serviceName" value="ESBBrokerProcess.main.1" />
    <property name="immediate" value="true" />
</action>
```

while results in the service descriptor associated with the *WhileAction* being re-evaluated (i.e. executed again). The *ScheduleStateAction* will define the service category and name to invoke, and whether to invoke them immediately, or whether the service descriptor will be triggered via an incoming message from an external source.

### 3.2.7.5. Blocking awaiting a decision

The *WhenAction* is used to block one or more awaiting paths, until an expression associated with one of the paths becomes true.

For example,

```
<action class="org.jboss.soa.overlord.jbossesb.stateful.actions.WhenAction"
            process="process" name="s4-1">
    <property name="session"
        value="org.jboss.soa.overlord.samples.jbossesb.loan.broker.BrokerMain" />
    <property name="paths">
        <when expression="receivedAllQuotes()"
                service-category="ESBBroker.BrokerParticipant"
                service-name="ESBBrokerProcess.main.4" />
    </property>
</action>
```

In this example, there is only one path defined, which will block until the *receivedAllQuotes()* expression becomes true. When this expression evaluates to true, the service category and name defined for the *when* element will be scheduled.

### 3.2.7.6. Composing reusable sub-conversions into a higher level conversation

The ability to compose reusable modules into higher level functions is a useful capability in any language. This mechanism is also supported in the "conversation aware" ESB actions, using the *PerformAction* to invoke a sub-session. For example,

```
<actions mep="OneWay">
    <action class="org.jboss.soa.overlord.jbossesb.stateful.actions.PerformAction"
                process="process" name="...">
        <property name="serviceCategory" value="ESBBroker.BrokerParticipant" />
        <property name="serviceName" value="RequestForQuote.main" />
        <property name="returnServiceCategory" value="ESBBroker.BrokerParticipant" />
        <property name="returnServiceName" value="ESBBrokerProcess.main.9" />
        <property name="bindDetails" >
            <bind from-expression="getCurrentSupplier()"
                        to-variable="supplier" />
        </property>
        <property name="parentReference" value="parent" />
    </action>
</actions>
```

The 'serviceCategory' and 'serviceName' properties define the service descriptor that will be invoked (i.e. that represents the sub-session). This service descriptor must begin with a *CreateSessionAction*, to indicate that it represents the start of a sub-session.

The optional 'returnServiceCategory' and 'returnServiceName' represent the service descriptor that will be invoked once the sub-session has completed, to enable the parent session to continue. If these properties are not defined, then it means that sub-session is being performed asynchronously, and therefore the parent session will not wait for it to complete.

The optional 'bindDetails' provides the means for the parent session to assign specific information from its state to the newly created pojo associated with the child session.

**Note**

Currently the bound details from the parent will be copied into the child pojo, and therefore modifications will not be reflected back into the parent pojo.

The optional 'parentReference' is used to set a reference, on the child session's pojo, to the parent session's pojo. The value of the 'parentReference' identifies the property on the child session's pojo that will be used to reference the parent pojo.

## 3.3. Stateless "Conversation Aware" ESB Actions

### 3.3.1. Overview

After the M1 release, we've found some problems in the *Stateful* "conversation aware" ESB actions approach, mainly due to its complexity and the need to have services recording state information. Therefore, we've introduced the *Stateless* "conversation aware" ESB actions, which we will see it later on. With this new approach, users can still validate the message exchange in the runtime, but it doesn't need us to store the state. So it simplified a lot without losing its functionality.

Thus the benefit of this approach is its simplicity, and therefore it has less impact on the way that users create ESB service descriptors. The disadvantage with this approach is that only fragments of stateful behaviour can be derived from the ESB jboss-esb.xml. However, this is useful enough to perform static conformance checking of the service implementation against a choreography description, and when used in conjunction with the dynamic conversation validation mechanism, it is still possible to determine out of sequence messages.

### 3.3.2. Interaction

Firstly, let's see the two basic actions for interaction. Services interact by sending and receiving messages, whether synchronously or asynchronously.

JBossESB is designed to anonymously handle inbound messages (possibly requests), without explicitly defining restrictions on message type, and then optionally returning responses (again without explicitly defining the response message type).

Although this is sufficient for a runtime mechanism, where issues related to unexpected message types can be handled with suitable exceptions/faults, it does not enable the communication type structure to be understood by examination of the JBossESB configuration.

#### 3.3.2.1. Sending a message

When sending a message, the first thing to consider is the type of the message. This can be declared as a property on the *SendMessageAction*. If dealing with RPC style interactions, then we may also want to optionally specify an operation name.

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.SendMessageAction"
            process="process" name="...">
    <property name="operation" value="getQuote" />
    <property name="messageType" value="requestForQuote" />
    ....
</action>
```

- Sending a request

When sending a request, we need to identify the destination service category/name. This is done by either specifying the category and name explicitly, using the *serviceCategory* and *serviceName* properties:

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.SendMessageAction"
            process="process" name="...">
    ....
    <property name="serviceName" value="RequestForQuote.main" />
    <property name="serviceCategory" value="ESBBroker.SupplierParticipant" />
    ....
</action>
```

- Sending a response

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.SendMessageAction"
            process="process" name="...">
    ....
    <property name="clientRole" value="buyer" />
    <property name="eprStore" value="orgization.your.impl.EPRStorageImpl" />
    ....
</action>
```

When sending a response, the destination will not be directly available. The destination would have been received as a 'reply to' EPR (Endpoint Reference) in a previously received request (see *ReceiveMessageAction* for details of how to store the 'reply to' EPR). Therefore, to indicate which EPR to respond to, a property called 'clientEPR' is specified with the name of the stored EPR. This must match up to a previously stored EPR name within a *ReceiveMessageAction*.

The value of *storageClass* should be a class that implements the *org.jboss.soa.overlord.jbossesb.EPRStorage* interface, basically this class is responsible for registering, getting EPR from roleName.

### 3.3.2.2. Receiving a message

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.ReceiveMessageAction"
           process="process" name="s1-2">
    <property name="operation" value="makeEnquiry" />
    <property name="messageType" value="enquiry" />
    <property name="clientRole" value="buyer" />
    <property name="eprStore" value="com.acme.services.creditAgency.MemoryEPRStorage" />
</action>
```

The *ReceiveMessageAction* is used to explicitly define the message type that should be received. If an RPC style has been used, then the optional operation name can also be defined.

Unlike the *SendMessageAction*, which will actually send a message to the specified service category/name, the *ReceiveMessageAction* primarily serves to provide explicit details about the expected message and to perform any relevant validation of the message content. If an incorrect message type is received, then an error will be logged.

The optional 'clientRole' and 'storageClass' properties are used to store any specific "reply to" EPR (associated with the message) against the specified name. This makes the EPR accessible to any subsequent *SendMessageAction* activities that need to return a response or send a notification.

## 3.3.3. Controlling Flow

This section describes the two control flow mechanisms that are supported by the stateless ESB actions.

The default control flow, supported natively by the ESB action pipeline design, is a sequence. As the name implies, the actions are performed one at a time in the order they defined in the action pipeline, i.e. in a sequence.

### 3.3.3.1. Selecting paths based on a decision

The action associated with the 'selection of a path based on a decision' is the *IfAction*. An example of this construct is:

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.IfAction" name="..."
                    process="process">
    <property name="paths">
        <if service-category="PurchaseGoods.CreditAgency"
            service-name="CreditAgency.decision1"
            decision-class="org.jboss.soa.overlord.jbossesb.TestDecision"/>
        <elseif service-category="PurchaseGoods.CreditAgency"
            service-name="CreditAgency.decision2"
            decision-class="org.jboss.soa.overlord.jbossesb.Test2ndDecision"/>
        <else service-category="PurchaseGoods.CreditAgency"
            service-name="CreditAgency.decision3"/>
    </property>
</action>
```

This construct defines a 'path' property with one or more elements, representing the *if*, *elseif* and *else* aspects of the traditional if/else construct. Only the *if* element is mandatory, and can be followed by zero or more *elseif* elements before ending with the optional *else* element.

The *if* and *elseif* elements can define an 'decision-class' attribute to be evaluated at runtime, to determine if the associated 'service-category' and 'service-name' should be invoked. the value of decision-class must implement the interface of *org.jboss.soa.overlord.jbossesb.Decision*.

### 3.3.3.2. Message router action

The action used to select paths based on a received message type is *SwitchAction*. In the stateless esb actions approach, we are using this as the *Message Router*. The configuration of *SwithAction* is like:

```
<action class="org.jboss.soa.overlord.jbossesb.stateless.actions.SwitchAction"
                    name="..." process="process">
    <property name="serviceDescriptionName"
        value="{org.pi4soa.esbbroker.esbbroker}ESBBrokerProcess-Broker"/>
    <property name="conversationType" value="overlord.cdl.samples.LoanBroker@Broker"/>
    <property name="paths">
        <case service-category="org.pi4soa.esbbroker.esbbroker"
                service-name="ESBBrokerProcess_Broker__1">
            <message type="enquiry"/>
        </case>
        <case service-category="org.pi4soa.esbbroker.esbbroker"
                service-name="ESBBrokerProcess_Broker__7">
            <event description="Event trigger to send quoteList message type(s)"/>
        </case>
        <case service-category="org.pi4soa.esbbroker.esbbroker"
                service-name="ESBBrokerProcess_Broker__8">
            <message type="buy"/>
        </case>
    </property>
</action>
```

In this approach, the *SwitchAction* typically is the point of contact for other services. Also true for the internal services. In the same manner as the *Stateful* ESB actions, it is necessary to specify the 'conversation type' that the service represents, to enable conformance checking to be performed against a choreography description that provides the associated conversation type. In the *Stateless* ESB actions, this initial *SwitchAction* defines this information in the *conversationType* property.

The 'paths' property defines one or more 'case' elements. These elements identify a service category and name that should be invoked upon receipt of one or more message types, as specified by 'message' elements contained within the 'case' elements.

The 'type' attribute on the 'message' element defines the type of message that can be routed to the specified service category/name. In the example above, the message types have no namespace. However if they have a namespace, this can be defined in curly braces, e.g. "{http://www.jboss.org/samples}buy".

The 'event' element is defined for paths with no associated message type. These paths are expected to be triggered by a different sort of event, for example an internal timeout managed by the service implementation. However these internal events are triggered, and directed to the appropriate service descriptor is a implementation issue for the service. The inclusion of this path in the *SwitchAction* symbolises the exist of this asynchronously triggered path in the behaviour of the service.

Once a path has been selected, the associated service category/name will be invoked immediately. If none of the paths specified within this action are relevant to the received message, then a runtime exception will be thrown.

**Tip**

In this M2 release, we've just had these four actions in the stateless esb action approach. we might add other actions in the subsequent release. If you have any comments and feedback, you can go to the forum or issue track to raise your request.

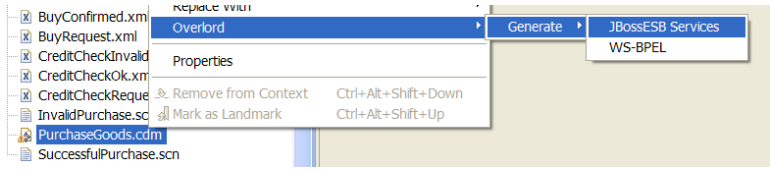
# 3.4. Generating a JBossESB Configuration from CDL

## 3.4.1. Overview

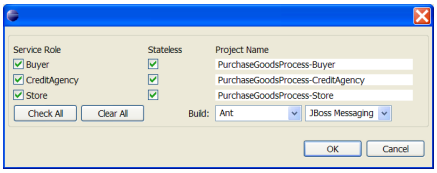
This section explains how to generate a template JBossESB configuration file from a pi4soa choreography description (.cdm) file.

## 3.4.2. Generating the JBossESB Configuration

When the choreography description has been completed, and has no errors, the user should select the "Overlord->Generate->JBossESB Services" menu item from the popup menu associated with the choreography description (.cdm) file.

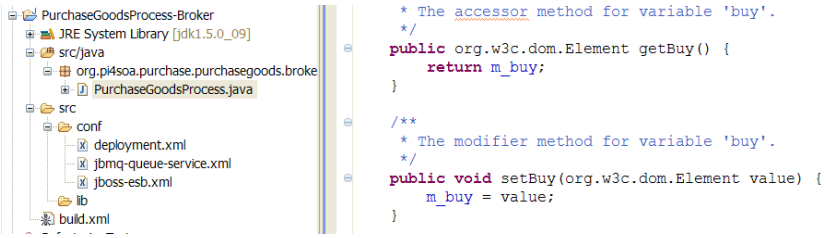



When the dialog window is displayed, it will contain the list of services that can be generated, along with the project names that will be created. The user can unselect the services they do not wish to generate (also using the 'Check All' or 'Clear All' buttons). There is also a checkbox to determine whether a stateful or stateless (the default) implementation should be generated.




The user can also select their preferred build system, which will create the relevant build structure and script in the generated Java project to enable the JBossESB service to be deployed, as well as the appropiate messaging system to use.

If there is a problem with the name of the project select, such as invalid characters used in the name, or the project name already exists, then it will be displayed in red.



In the above image, on the left it shows the generated project structure for the Broker service role. On the right it shows a portion of a generated session class, with the accessor and modifier for one of the variables associated with that session.

Although an initial build script will be created, depending on the build system selected, the user may need to edit the script to set certain properties, or change the way the build occurs. For example, with the Ant build, the deploy target (which is the default) will attempt to place the deployed .esb file into a location defined by the `${org.jboss.esb.server.deploy.dir}`. If this variable has not been defined, then a folder called `${org.jboss.esb.server.deploy.dir}` will be created in the root folder of the project, containing the .esb file.

## 3.5. Dealing with Conformance Issues

### 3.5.1. Overview

Conformance checking can be used to determine whether an ESB configuration, containing "conversation aware" ESB actions, matches the expected behaviour as defined within a choreography description (.cdm file). The Eclipse environment will report any conformance issues as errors in the *Problems* view.

This section will explain the types of conformance errors that may be reported and how to deal with them. Not all errors, associated with an ESB configuration, will be discussed in this section. Many errors may be detected that will indicate problems associated with the way that behaviour has been specified in the configuration file (e.g. conversation type has not been defined). These are not directly related to conformance.

### 3.5.2. Show referenced description

When an error occurs, related to conformance between the ESB configuration file and a choreography description, it will have an associated *quick fix* resolution that can be used to display the relevant activity being referred to within the choreography description.

### 3.5.3. Error: Expecting additional activities as defined in referenced description

This error message indicates that the reference description contains activities that were not found in the ESB configuration.

This error has an associated *quick fix* to enable the missing activities to be inserted in the appropriate location within the ESB configuration.

**Note**

When this resolution is selected, if it displays an error “Could not insert activities found in referenced description”, this means that it was not possible to insert the additional activities automatically.

### 3.5.4. Error: Type mismatch with referenced description, was expecting '...'

This error occurs when an activity contains a type that does not match with the equivalent activity in the choreography description. A common example would be an interaction, where the message types are not compatible.

This error has an associated *quick fix* to enable the type to be updated in the relevant activity within the ESB configuration.

**Note**

When this resolution is selected, if it displays an error “Could not update activity with information from referenced description”, this means that it was not possible to update the information automatically.

### 3.5.5. Error: Behaviour not present in referenced description

This error occurs when there are extra activities within the ESB configuration that do not appear within the choreography description.

This error has an associated *quick fix* to enable the unwanted activities to be removed from the ESB configuration.

**Note**

When this resolution is selected, if it displays an error “Could not delete activities from the model”, this means that it was not possible to delete the activities automatically.

### 3.5.6. Error: Additional unmatched paths in model

This error indicates that a grouping contruct (e.g. *ParallelAction*, *IfAction* or *SwitchAction*) in the ESB configuration has additional paths that do not match the equivalent grouping construct in the choreography description.

### 3.5.7. Error: Additional unmatched paths in referenced description

This error indicates that a grouping contruct (e.g. *Choice* or *Parallel*) in the choreography description has additional paths that do not match the equivalent grouping construct in the ESB configuration.

# BPEL Governance

## 4.1. Overview

This section will describe governance features related to WS-BPEL. This initial release provides basic support for generating BPEL processes from a choreography description. Subsequent releases will also provide conformance checking, to ensure that changes to a BPEL process are validated to ensure the process remains conformant with the choreography.

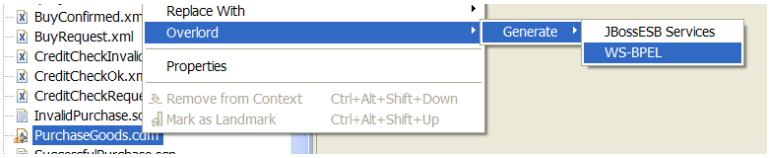
## 4.2. Generating a BPEL process from CDL

### 4.2.1. Overview

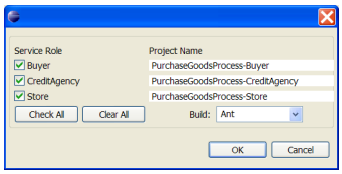
This section explains how to generate a template BPEL process from a pi4soa choreography description (.cdm) file.

### 4.2.2. Generating the BPEL Process

When the choreography description has been completed, and has no errors, the user should select the "Overlord->Generate->WS-BPEL" menu item from the popup menu associated with the choreography description (.cdm) file.



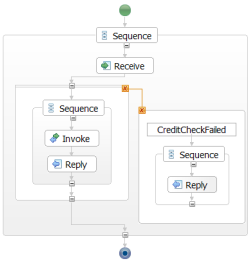

When the dialog window is displayed, it will contain the list of services that can be generated, along with the project names that will be created. The user can unselect the services they do not wish to generate (also using the 'Check All' or 'Clear All' buttons).




The user can also select their preferred build system, which will create the relevant build structure.

If there is a problem with the name of the project select, such as invalid characters used in the name, or the project name already exists, then it will be displayed in red.

Once the BPEL is generated, it can be viewed using the Eclipse BPEL editor, e.g.




### 4.2.3. Limitations with the current CDL to BPEL mapping

This initial version of the BPEL generation is primarily targeted at generating the interactions and grouping constructs. These are the important components that will be required when doing conformance checking as part of the next milestone.

This means that assignments and conditional expressions are not currently generated. The 'when' construct in CDL (also known as the blocking workunit) is also not currently handled. This may possible be implemented using flow links.