JBoss Overlord CDL 1.0-M2

Getting Started Guide

by Gary Brown and Jeff Yu

1. Installation 1
1.1. Overview 1
1.2. Prerequisites 1
1.3. Installation Instructions 1
1.4. Importing Samples into Eclipse 3
2. SOA Governance with CDL 6
2.1. Design Time Governance 6
2.1.1. Creating a Choreography 6
2.1.2. Design Time Governance With WS-BPEL 9
2.1.3. Design Time Governance With "Conversation Aware" ESB Actions 9
2.1.4. Summary 13
2.2. Runtime Governance using Conversation Validation 14
2.2.1. Service Validator Configuration 14
2.2.2. Deploy the TrailBlazer Example 15
2.2.3. Starting the pi4soa Monitor 16
2.2.4. Running the Example 17
2.2.5. Detecting a Validation Error 18
3. Appendix 19
3.1. Advanced options of installation 19

Installation

1.1. Overview

This section describes the installation procedure for the Overlord CDL based governance capabilities. These capabilities are:

- Conversation aware ESB Actions with conformance checking against a Choreography Description
- ESB Service validation against a Choreography Description

1.2. Prerequisites

1. JBossAS (version 4.2.3.GA or higher), available from http://www.jboss.org/jbossas
2. JBossESB (version 4.5.GA or higher), should download the **jbossesb-4.5.GA.zip**, available from http://www.jboss.org/jbossesb
3. Overlord CDL (version 1.0-M1 or higher), available from http://www.jboss.org/overlord
4. pi4soa (version 2.0.0 or higher), available from http://pi4soa.wiki.sourceforge.net/download

Note

It is recommended that a pre-packaged version is used, which includes all of the necessary Eclipse related plugins. However the plugins can be installed separately into an existing Eclipse environment by following the instructions on the http://www.pi4soa.org download wiki.

5. Ant, available from http://ant.apache.org

1.3. Installation Instructions

1. Install JBossAS
   Unpack the JBossAS installation into the required location.

   NOTE: Before running the server, it is advisable to edit the run.sh/bat script in the `bin` folder to add the following parameter to the JAVA_OPTS variable:

   ```
   -XX:MaxPermSize=128M
   ```

2. Install JBossESB
   Unpack the JBossESB installation into a location alongside the JBossAS installation. Then follow the instructions in the JBossESB installation (install/readme.txt), to deploy JBossESB into the JBossAS environment.

3. Install the Overlord CDL distribution

Unpack the Overlord CDL distribution into a location alongside the JBossAS installation.

- Edit the **install/deployment.properties** file to update the JBossAS and JBossESB location settings.

- From the install folder, run: `ant` to deploy the Overlord CDL to JBossAS.

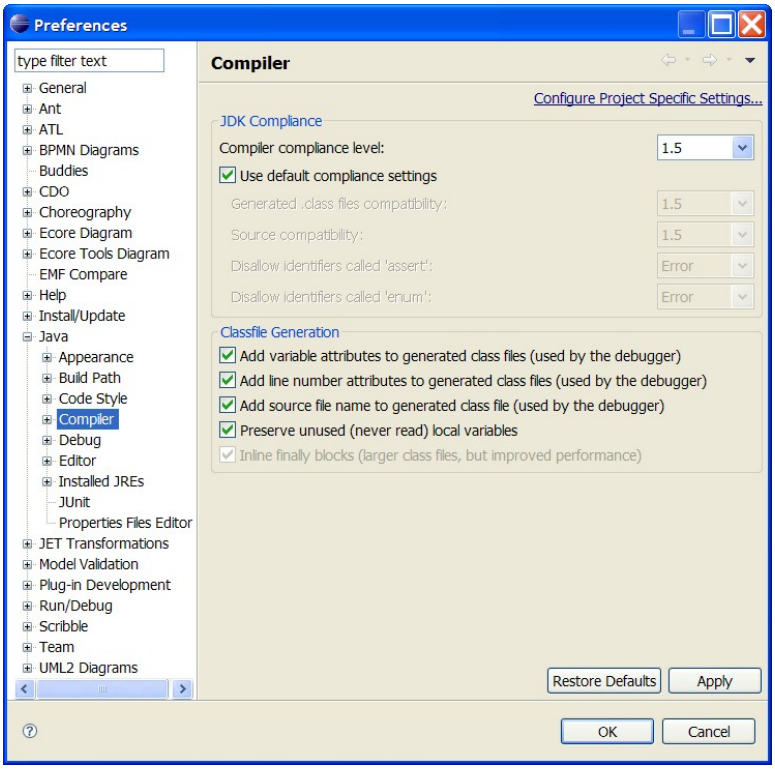
4. Install pi4soa
   Unpack the pi4soa pre-packaged Eclipse version into a location alongside the JBossAS installation. Once unpacked, start the Eclipse environment and update the plugins using the pi4soa update site (http://pi4soa.sourceforge.net/updates) to obtain the most recent version of the plugins. Alternatively, if you wish to use an existing Eclipse environment, instead of the pre-packaged pi4soa Eclipse version, then simply use the update site from your preferred Eclipse environment.

   NOTE: When doing the plugin update, if it complains about unresolved dependencies, then the key components to install/update are the core feature (in the modeller category) and the technology preview feature (in the incubator category).

   If just the service validation capabilities are being used, then no further configuration of the Eclipse environment is necessary. However if the conversational ESB actions, with conformance checking against a Choreography Description, will be used, then the following additional steps will be required:

   - Start the Eclipse environment

   - Overload CDL is currently JDK1.5 compliant. Therefore it is necessary to ensure that the Eclipse environment also compiles classes, used by the ESB "conversation aware" actions, as 1.5 complaint. This can be achieved by selecting the *Windows->Preferences* menu item, and selecting the *Java->Compiler* node, and setting the compliance level to 1.5, as shown in the following image:



5. Install Overlord CDL Eclipse plugins

   - Select the “Help - > Software Updates...” menu item
   - From the **Available Software** tab, press the “Add Site...” button
   - Press the **“Local”** button, browse to locate the **tools** folder in the Overlord CDL distribution, and then press the OK button. This will cause the local Eclipse update site, bundled with the Overlord CDL distribution, to be add to the **Available Software** tab.
   - Select the root node of the newly added local update site, and then press the **“Install”** button and follow the instructions to install the plugins.

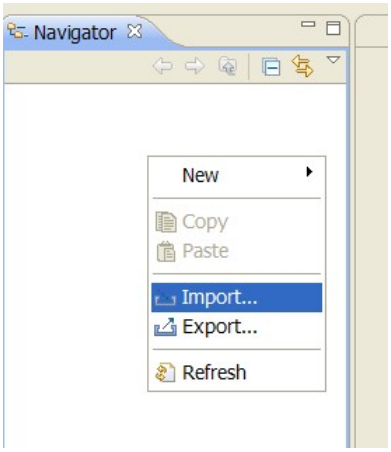
Note

An eclipse issue occasionally causes the nodes under the checked root node to become unchecked, resulting in the software update manager indicating that no plugins need to be installed. If this happens, simply uncheck the root node, and then re-check the root node and press the **“Install”** button again.

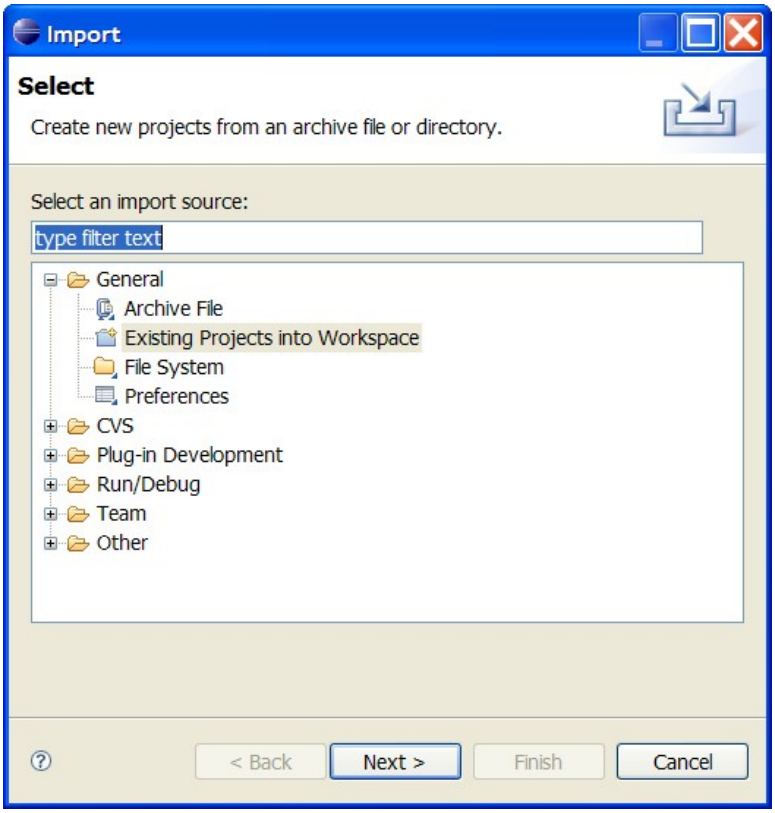
1.4. Importing Samples into Eclipse

Once the Overlord CDL distribution has been correctly installed, if you wish to try out any of the examples then the following steps should be followed to import the relevant projects into the previously configured Eclipse environment.

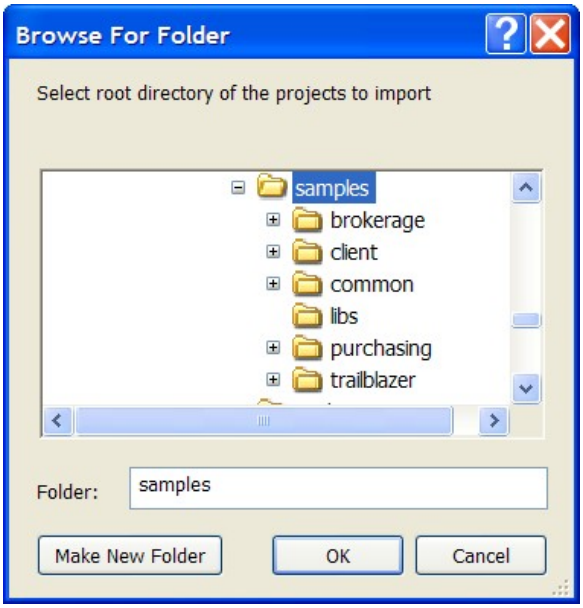
1. Select the 'Import...' menu item, associated with the popup menu on the background of the left panal (Navigator or Package depending on perspective being viewed).



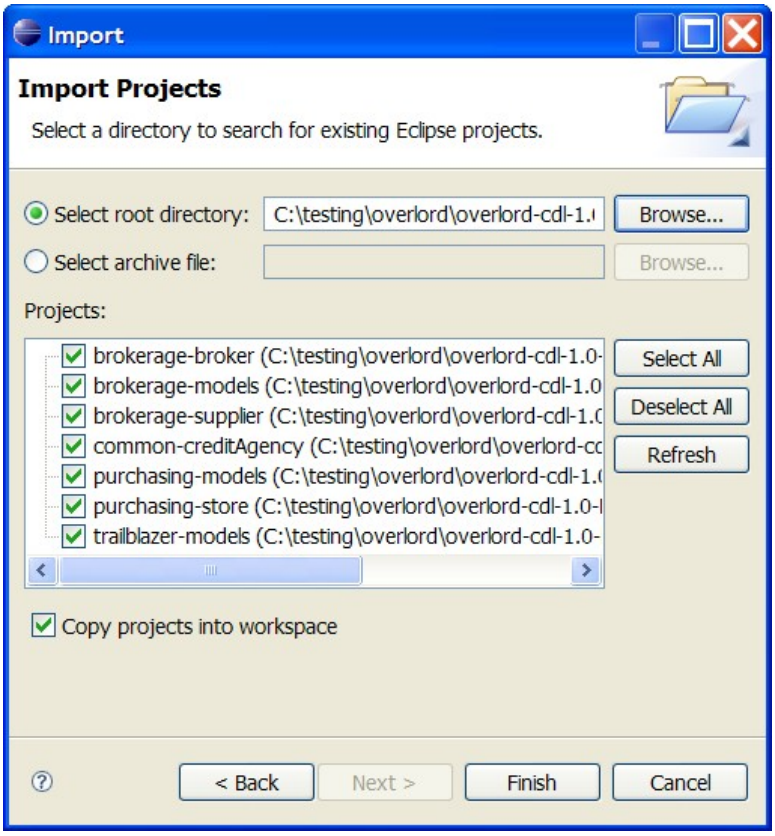
2. When the import dialog appears, select the *General->ExistingProject from Workspace* option and press the 'Next' button.



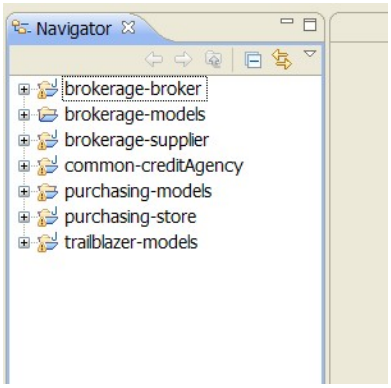
3. Ensuring that the 'Select root directory' radio button is selected, press the 'Browse' button and navigate to the `${OverlordCDL}/samples` folder, then press 'Ok'.



4. All of the Eclipse projects contained within the `samples` directory structure will be listed. Press the 'Finish' button to import them all.



Once imported, the Eclipse navigator will list the sample projects:



SOA Governance with CDL

The Choreography Description Language (CDL) provides a means of describing a process, that executes across a distributed set of services, from a global (or service independent) perspective.

SOA Governance, using CDL, is about ensuring a process is correctly implemented (as part of design-time governance), and executes as expected (part of runtime governance).

In this chapter we will take you through a worked example associated with each of these aspects.

Note

Before proceeding, please make sure that the Overlord CDL distribution has been correctly installed and that the samples have been imported into the Eclipse environment.

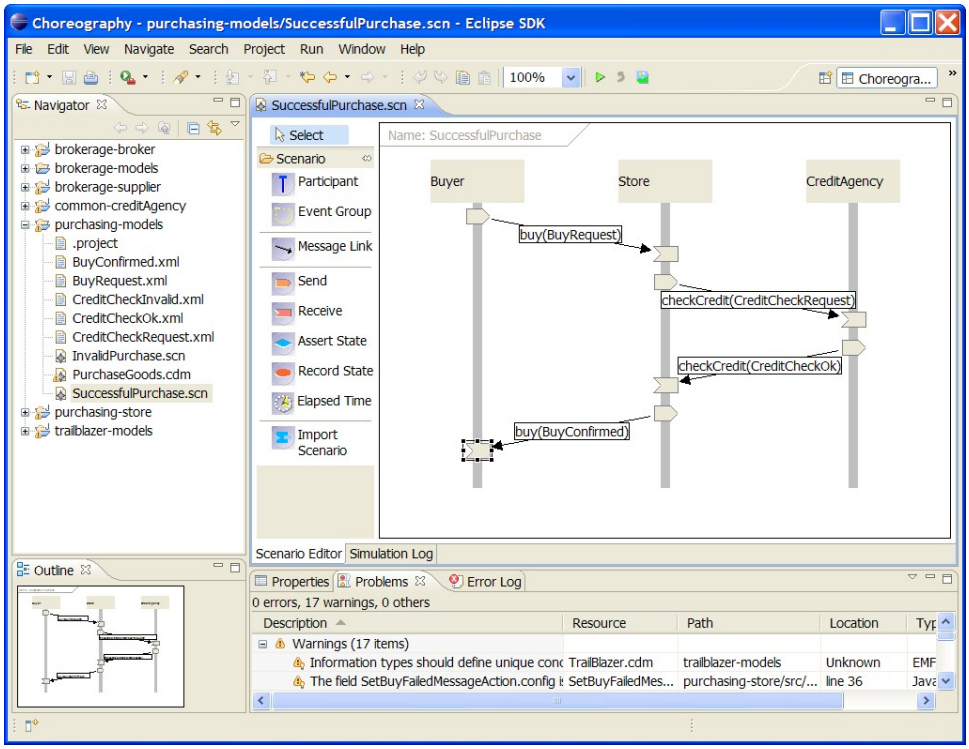
2.1. Design Time Governance

2.1.1. Creating a Choreography

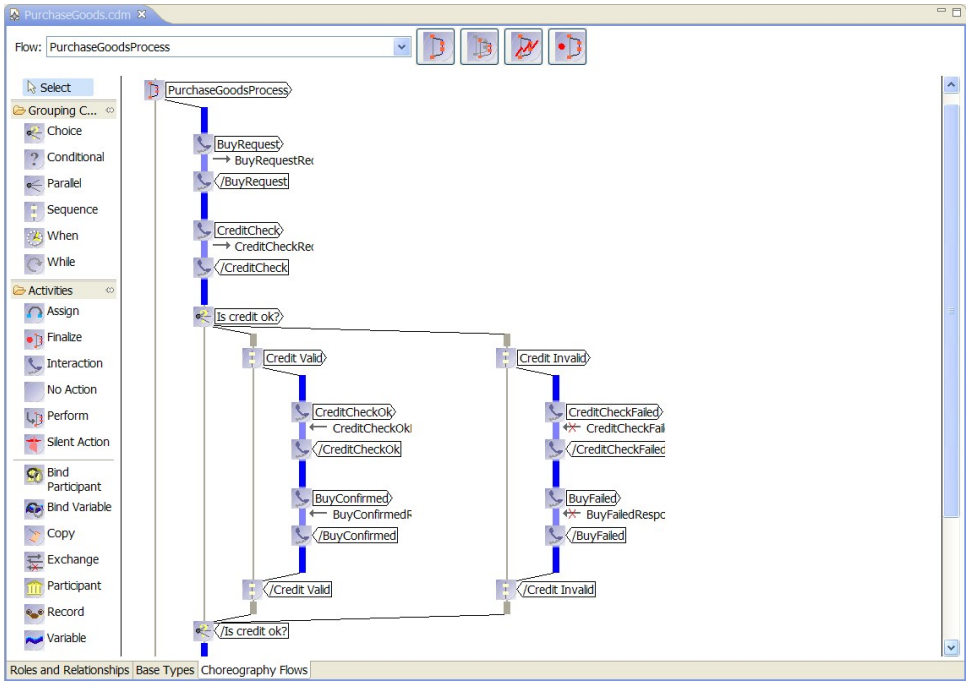
When designing a system, it is necessary to capture requirements. Various approaches can be used for this, but currently there are no mechanisms that enable the requirements to be documented in such a way to enable an implementation to be validated back against the requirements.

The pi4soa tools provide a means of describing requirements, representing specific use cases for the interactions between a set of cooperating services, using scenarios - which can be considered similar to UML sequence diagrams that have been enhanced to include example messages.

In the `purchasing-models` Eclipse project, the `SuccessfulPurchase.scn` scenario looks like this:

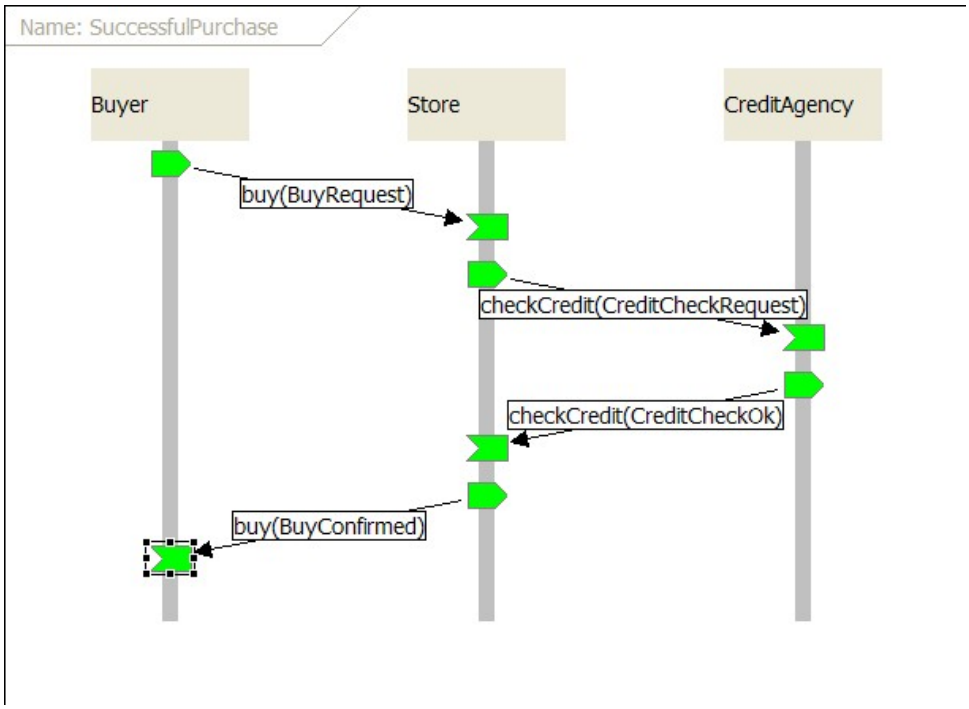


The next step in the development process is to specific a Choreography Description to implement the requirements described within the set of scenarios. The choreography for the Purchasing example can be found in `purchasing-models/PurchaseGoods.cdm`. When the choreography editor has been launched, by double-clicking on this file within the Eclipse environment, then navigate to the *Choreography Flows* tab to see the definition of the purchasing process:

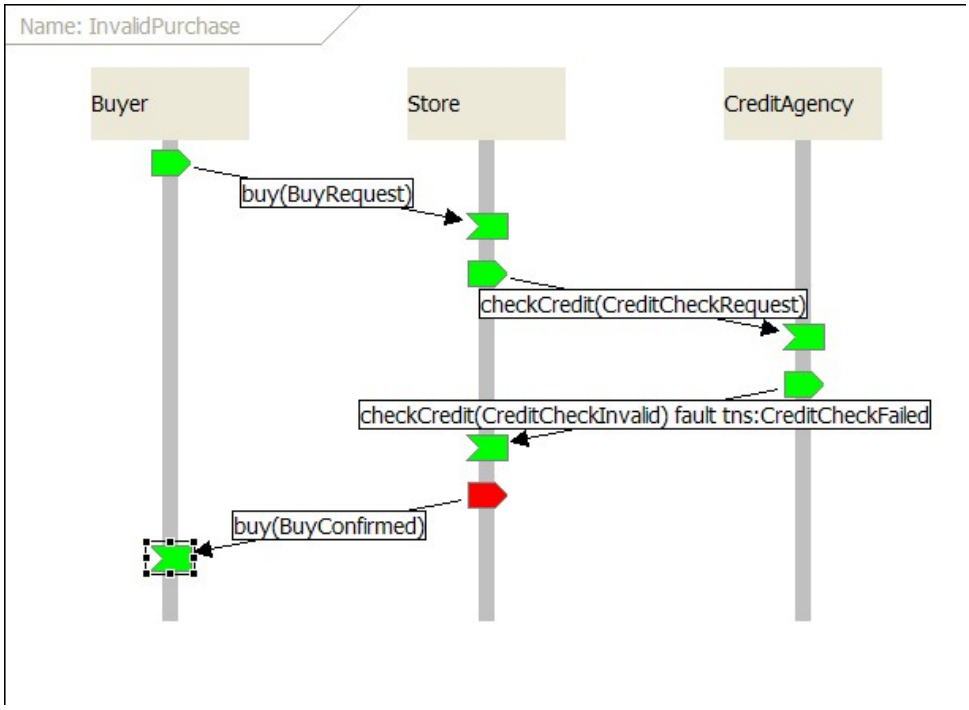


The pi4soa tools can be used to test the scenarios against the choreography description, to ensure that the choreography correctly implements the requirements. To test the `SuccessfulPurchase.scn` scenario

against the choreography, launch the scenario editor by double-clicking on the scenario file, and then pressing the green *play* button in the toolbar. When complete, the scenario should look like the following image, indicating that the scenario completed successfully.



To view a scenario that demonstrates a test failure, open the `InvalidPurchase.scn` scenario by double-clicking on the file, and then initiate the test using the green *play* button in the toolbar. When complete, the scenario should look like the following image.



You will notice that the *Store* participant has a red 'send' node, indicating that this action was not expected behaviour when compared with the choreography description. The reason this is considered an error, is that the *Store* participant should only send a *BuyFailed* message following an invalid credit check.

When an error is detected in a scenario, the choreography designer can then determine whether the scenario is wrong (i.e. it does not correctly describe a business requirement), or whether the choreography is wrong and needs to be updated to accomodate the scenario.

Once the choreography description has been successfully tested against the scenarios, and therefore is shown to meet the business requirements, the next step is to design and implement each service involved in the choreography. The pi4soa tools provide the means to export BPMN, UML or HTML documentation to aid the implementation phase. However there is special support for a concept called "Conversation Aware" ESB Actions.

2.1.2. Design Time Governance With WS-BPEL

This milestone release includes a basic capability to generate a service implementation, for a participant in a choreography, using WS-BPEL.

Subsequent milestone releases will include more capabilities as part of the service generation, including the generation of WSDL, as well as supporting conformance checking back against the choreography description.

More information about how to use this feature can be found in the User Guide.

2.1.3. Design Time Governance With "Conversation Aware" ESB Actions

2.1.3.1. What are "Conversation Aware" ESB Actions?

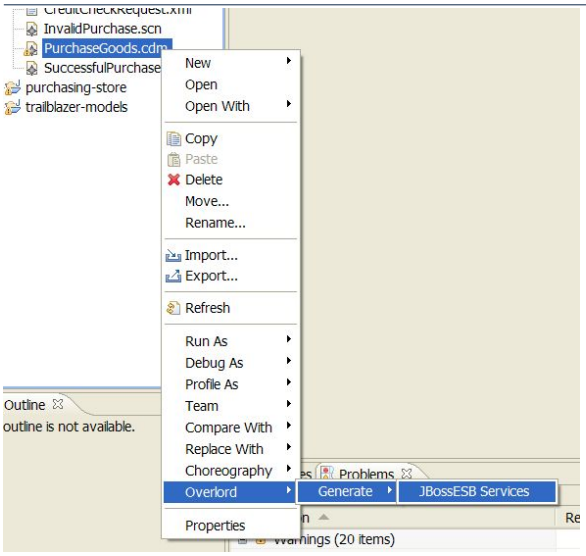
Conversation aware ESB actions refer to a set of pre-defined ESB actions that enable the structure (or behaviour) of a service to be inferred.

For example, there are actions that explicitly define the sending and receiving of messages. These actions define a property that declares the type of the message being sent or received. Other actions describe grouping constructs such as if/else, parallel and while loop.

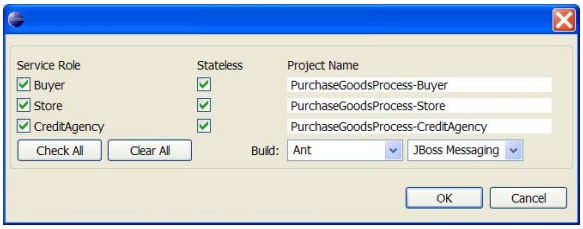
The benefit of making these concepts explicit within the ESB service configuration, is that it makes it possible to check the implementation correctly matches the expected behaviour as defined within the choreography. This will be demonstrated in the following sub-section discussing conformance checking.

2.1.3.2. Generating an ESB Service using "Conversation Aware" ESB Actions

Once we have a choreography description, it is possible to generate an ESB Service (with *conversation aware* ESB actions), for each of the participants defined within the choreography. To try this out, select the *Overlord->Generate->JBossESB Services* menu item from the popup menu associated with the `PurchaseGoods.cdm`.

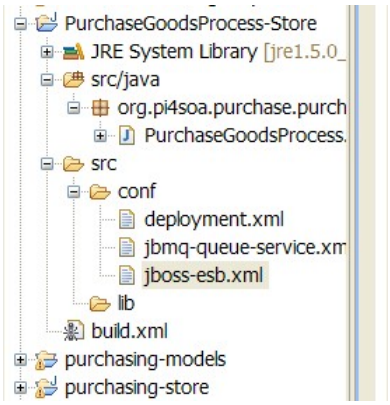


This will display a dialog listing the possible services that can be generated from this choreography, with a proposed Eclipse project name.



To test out this feature, uncheck the *Buyer* and *CreditAgency* participants, leave the build system as *Ant*, select the messaging system appropriate for your target environment and press the 'Ok' button. This will create a single new project for the *Store* participant.

Depending upon the value of the 'stateless' checkbox, the generated ESB service artefact (jboss-esb.xml) will either use a stateful or stateless approach for encoding the behaviour of the service. The difference relates to whether the Overlord infrastructure explicitly maintains state information about session instances, to help police the behaviour of individual transactions, or whether this is implicitly performed using other capabilities (such as the Conversation Validation mechanism described in the following section, which can be used as an external monitor to ensure the service behaves as expected).



The generated project includes the ESB configuration file (in the `src/conf` folder) and the relevant Java classes in the `src/java` folder. The contents of this project represents a template of the service. Before it can be executed, the ESB configuration file will need to be enhanced to include internal implementation details for the service. The contents of this generated project should be compared to the completed version in the `purchasing-store` project.

Note

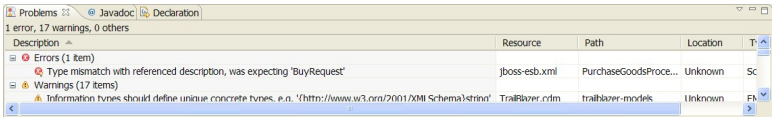
When the project is generated, if errors are reported against the `jboss-esb.xml`, then simply double-click on the error to launch the ESB configuration file. Then make a minor change, such as adding a new line and then removing it, and save the file again (to force re-validation). This should cause the errors to be cleared. This occurs because the Eclipse tasks that validate the `jboss-esb.xml` file and compiling the new Java classes in the project sometimes gets confused, causing the classes not to be present when the validation rules attempt to access them. This issue is being investigated.

2.1.3.3. Conformance Checking "Conversation Aware" ESB Services

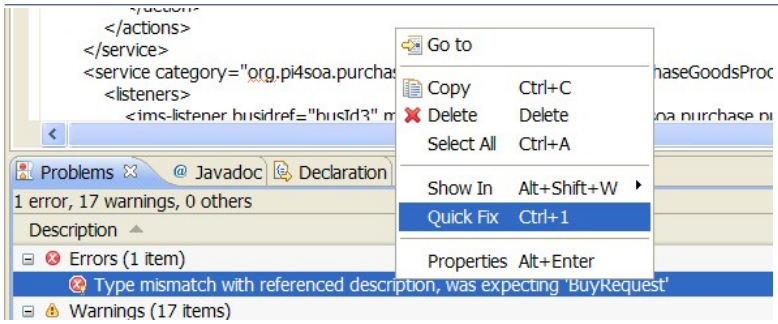
To demonstrate the conformance checking mechanism, where the behaviour of the ESB service is verifed against its responsiblities as defined within the choreography description, open the `src/conf/jboss-esb.xml` in the `PurchaseGoodsProcess-Store` generated in the previous sub-section.

When the ESB configuration has been loaded into an editor, locate the first *ReceiveMessageAction* ESB action, which should have a property called *messageType* with a value of *BuyRequest*. To cause a conformance checking error, simply append an 'X' to the end of the message type value, as shown in the following screenshot:

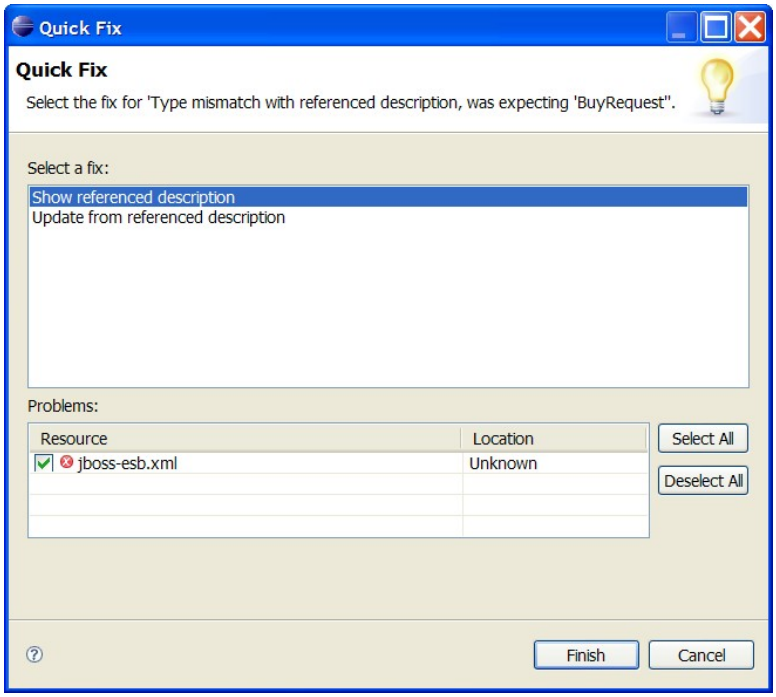
This results in an error message being reported:



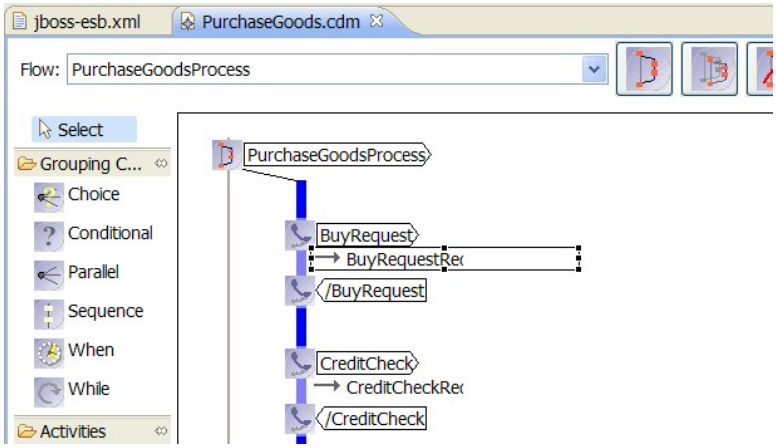
To fix conformance issues, some of the error messages will provide *Quick Fix* solutions. These can be access using the popup menu associated with the error message:



This will display the *Quick Fix* dialog listing the available resolutions.



If the *Show Referenced Description* resolution is selected, then it will cause the choreography description to be launched and the specific interaction to be focused.



If the *Update from Referenced Description* is selected, then the `jboss-esb.xml` will be automatically updated to remove the appended 'X' from the message type.

2.1.3.4. Running "Conversation Aware" ESB Services

The purchasing example describes the interactions between a Buyer, Store and Credit Agency. The flow for this example would be:

- Buyer send a 'buy' request to Store
- Store send a 'credit check' request to the Credit Agency.
- If the Credit Agency returns a successful message, then the Store will send a 'BuyConfirmed' to user.
- If the Credit Agency returns a failed message, then the Store will send a 'BuyFailed' to user.

There are two alternate implementations of the services involved in the purchasing example, demonstrating both the stateful and stateless "conversation aware" ESB actions. The `$Overlord/samples` contains a sub-folder for each variation.

To run the *purchasing* example, firstly ensure that the JBoss Application Server has been fully configured as described in the *Installation* chapter, and then do the following:

1. In a command window, go to the `$Overlord/samples/stateful` (or `$Overlord/samples/stateless` if using the stateless approach) folder and execute **ant deploy-purchasing**
2. In a command window, go to the `$Overlord/samples/client` folder and execute **ant runPurchasingClient** (or **ant runStatelessPurchasingClient** if using the stateless services), which will send a 'BuyRequest' message to the *Store*, which will then perform the credit check before returning a response to the client.

In this example, the conversation ESB actions will do the validation in the runtime. As we've said, the client send the 'buyRequest' message to the store, firstly the store service will check the received message based on its messageType attribute. and then send another message to the 'credit agency' service it goes through its validation. If the messageType that store service received is not as same as the one defined in the conversational esb actions, it will throw out the exception and ends its flow.

2.1.4. Summary

This section has provided a brief introduction to the design-time SOA governance features provided within the Overlord CDL distribution.

The aim of these capabilities is to enable verification of an implementation, defined using *conversation aware* ESB actions in this example, against a choreography, which in turn has been verified against business requirements defined using scenarios. Therefore this helps to ensure that the implemented system meets the original business requirements.

Being able to statically check that the implementation should send or receive messages in the correct order is important, as it will reduce the amount of testing required to ensure the service behaves correctly. However

it does not enable the internal implementation details to be verified, which may result in invalid decisions being made at runtime, resulting in unexpected paths being taken. Therefore, to ensure this situation does not occur, we also need runtime governance, which is discussed in the following section.

2.2. Runtime Governance using Conversation Validation

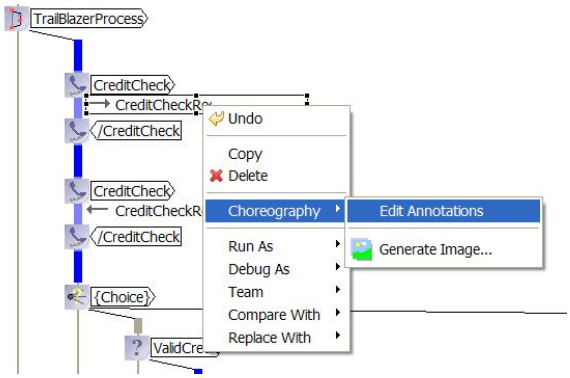
Once services have been deployed, as mentioned in the previous section, we still need to be able to verify that the services continue to conform to the choreography description. The *Conversation Validation* capability within the Overlord CDL distribution can be used to validate the behaviour of each service.

In this section, we will use the Trailblazer example found in the `$Overlord/samples/trailblazer` folder and the `trailblazer-models` Eclipse project.

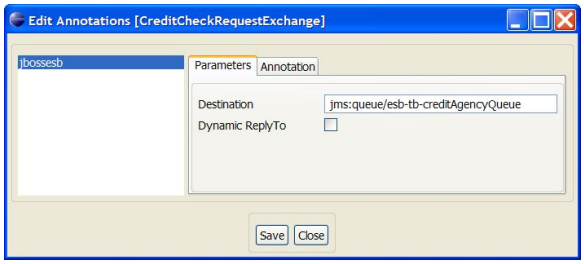
2.2.1. Service Validator Configuration

The JBossESB service validator configuration is defined using jbossesb specific annotations, that are associated with the 'exchange details' components (contained within interactions), within the choreography description.

To view the pre-configured service validator configuration defined for the Trailblazer example, edit the `TrailBlazer.cdm` file, navigate to the *Choreography Flows* tab and then select the *Choreography->Edit Annotations* menu item associated with the first 'exchange details' component (as shown below).



This will display the annotation editor, with the single configured annotation called 'jbossesb'. This annotation defines the information required for the Service Validator to monitor this specific message exchange (i.e. the JMS destination on which the message will be passed).



Once an annotation has been defined, it will also be displayed as part of the tooltip for the associated model component, for example:

One the jbossesb annotations have been defined for all relevant 'exchange details' components in the choreography description, the choreography file can be copied to the `$JBossAS/server/default/deploy/overlord-cdl-validator.esb/models` folder in the JBossAS environment. The service validator configuration for the *trailblazer* example has been preconfigured to be deployed as part of the installation procedure.

Note

If the `overlord-cdl-validator.esb/validator-config.xml` within the JBossAS environment is modified, or choreography description files added, removed or updated within the `overlord-cdl-validator.esb/models` sub-folder, then the changes will automatically be detected and used to re-configure the service validators without having to restart the JBossESB server.

2.2.2. Deploy the TrailBlazer Example

The first step to deploying the Trailblazer example is to configure the JBossAS environment:

1. Update the `$JBossAS/server/default/deploy/jbossesb.sar/jbossesb-properties.xml` file, in the section entitled "transports" and specify all of the SMTP mail server settings for your environment.
2. Update the `trailblazer/trailblazer.properties`

   Update the file.bank.monitored.directory and file.output.directory properties. These are folders used by the File Based Bank, and are set to `/tmp/input` and `/tmp/output` by default.
3. Update the `trailblazer/esb/conf/jboss-esb.xml`

   There is a *fs-provider* block, update the directory attribute value to be the same as the file.output.directory value in `trailblazer.properties` file.
4. Start the JBossAS server

One the server has been started, the next step is to deploy the relevant components into the JBossAS environment. This is achieved by:

1. From the `trailblazer` folder, execute the following command to deploy the example to the ESB: **ant deploy**

   this should deploy the ESB and WAR files to your JBoss AS `server/default`.
2. From the `trailblazer/banks` folder, execute the command to start the JMS Bank service: **ant runJMSBank**.

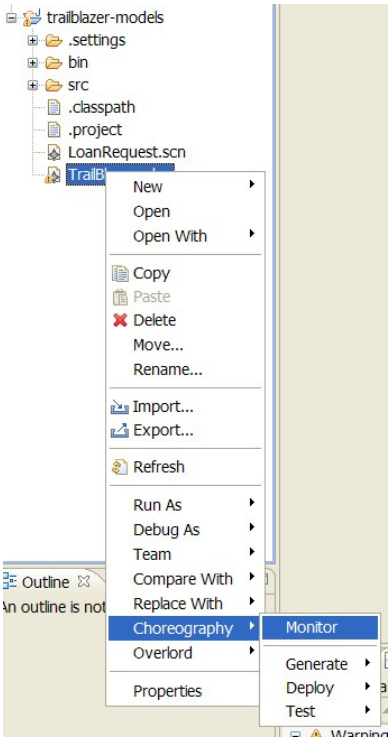
3. From the `trailblazer/banks` folder, execute the command to start the JMS Bank service: **ant runFileBank**.

2.2.3. Starting the pi4soa Monitor

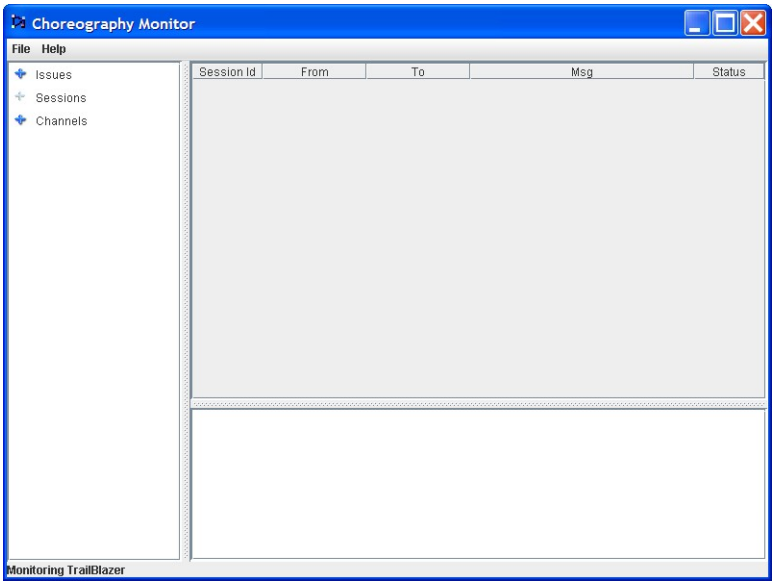
The pi4soa Monitor is used to observe a correlated view of the executing business transactions. Each service validator can be configured to report activites (i.e. sent and received messages) that it validates, to enable the correlator to reconstitute a global interpretation of each transaction.

This correlated view of each transaction can be used to understand where each transaction is within the process. It can also be used to report *out of sequence*, *unexpected messages* and more general errors in the context of the business process.

A simple monitoring tool is currently provided with the pi4soa tools, to enable the correlated global view of the transactions to be observed. Once the Trailblazer example has been deployed to the JBossAS environment, and the server is running, then the monitoring tool can be launched from the Eclipse environment by selecting the *Choreography->Monitor* menu item from the popup menu associated with the `TrailBlazer.cdm` file.

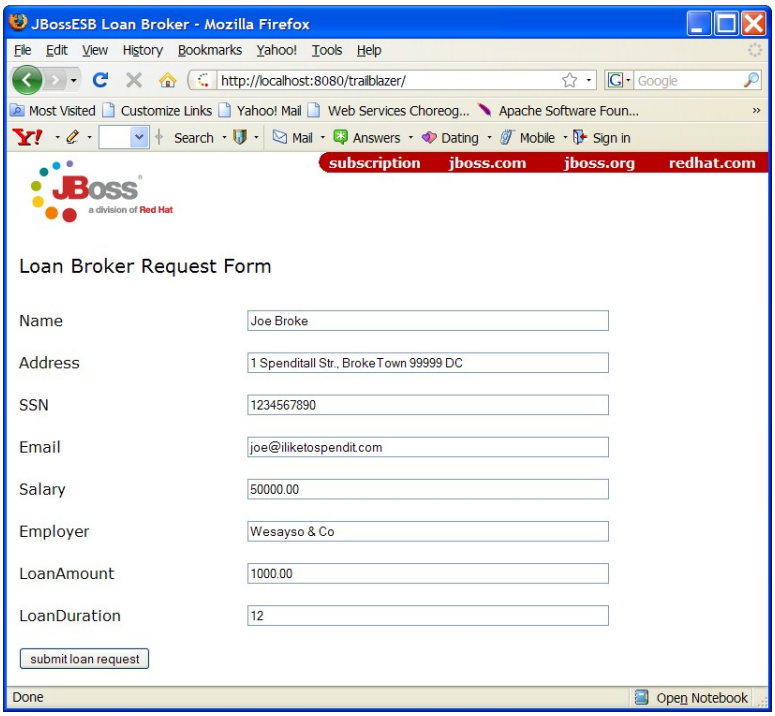


Wait for the monitor window to start, and indicate that the choreography is being monitored, shown in the status line at the bottom of the window.



2.2.4. Running the Example

To run the example, you need to start a browser and select the URL localhost:8080/trailblazer. This will show the following page, if the server has been configured correctly and the TrailBlazer example deployed:



Now you can submit quotes, You will see either a loan request rejected (single email) because the score is less than 4, or two emails (one from JMS bank and one from FileBased bank) with valid quotes. When entering subsequent quotes, make sure that the quote reference is updated, so that each session has a unique id.

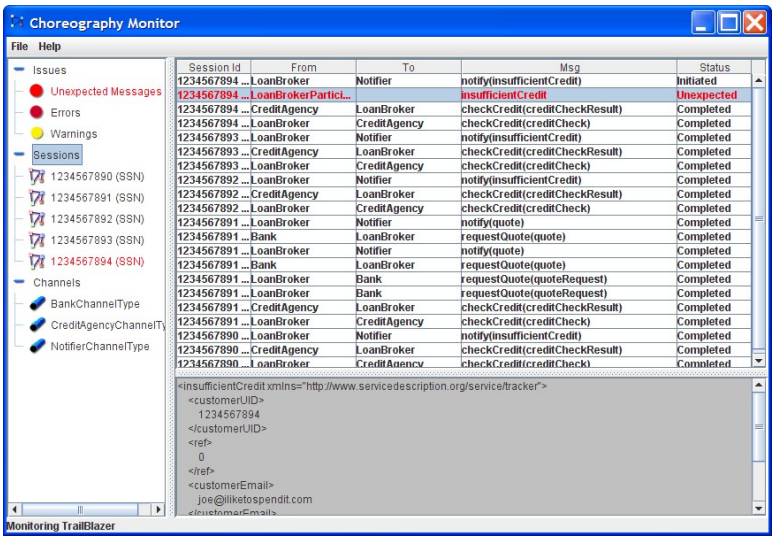
2.2.5. Detecting a Validation Error

To demonstrate the detection of validation errors, there is an alternative implementation of the trailblazer modules that behaviour differently to the choreography that is being monitored. Specifically, the credit score threshold used to determine whether a loan request should be issued to the banks, is raised from 4 to 7.

To deploy the version of the TrailBlazer example that results in validation errors, then:

- From the `$Overlord/samples/trailblazer` folder, execute the following command to deploy the example to the ESB: **ant deploy-error-client**.

The next step is to issue more transactions, until a credit check score occurs that is between 4 and 6 inclusive. This will result in a *insufficientCredit* interaction being reported, which would be unexpected in terms of the choreography.



When errors, such as unexpected messages, are detected by the service validators and reported to the Choreography Monitor, they are displayed in red.

Appendix

3.1. Advanced options of installation

The Overlord CDL has two separate modules. One is **Validator** module, which is used from management perspective; The other is the **Runtime** module, which is used for execution of flow/interaction sections of CDL file.

You can deploy each of them separately in the **Install** folder. From the install folder.

- Run: `ant deploy-overlord-cdl-runtime` to deploy just the conversational ESB actions support. Or
- Run: `ant deploy-overlord-cdl-validator` to to deploy just the service validation capability.

By default, when you run `ant` or `ant deploy`, it will deploy both of two modules.

Also, you can undeploy modules by running `ant undeploy`. Or remove them module by module through running: `ant undeploy-overlord-cdl-runtime` and `ant undeploy-overlord-cdl-validator`