SAVARA 1.0

# Getting Started Guide

by Gary Brown and Jeff Yu

# Overview

This is the Getting Started Guide for SAVARA. This guide starts with the installation instructions for the SAVARA tools and runtime modules.

The remainder of the document is organised to reflect phases within the SAVARA Methodology, and how the current tools can be used in support of that methodology. The tools are still in development, and therefore not all phases will have tools, and the tools in some phases will not necessarily be complete.

As an overview, the tools currently include capabilities for:

- Definition of business requirements as scenarios
- Creation of a choreography (global model) to represent the architecture for a system that delivers the requirements
- Generation of documentation based on the choreography
- Generation of service implementation using WS-BPEL
- Generation of service interfaces using WSDL
- Conformance checking a WS-BPEL service implementation against a choreography
- Runtime validation of an ESB service against a choreography description

# Installation

This section describes the installation procedure for SAVARA tools and runtime modules.

## 2.1. Prerequisites

The pre-requisites for the SAVARA Eclipse Tools are:

1. Eclipse JEE (3.5 or higher) http://www.eclipse.org
2. SAVARA (version 1.0.0 or higher), available from http://www.jboss.org/savara/downloads
3. JBoss Tools (3.1 or higher) http://www.jboss.org/tools available from an update site

The pre-requisites for the SAVARA Service Validator (for JBossESB) are:

1. JBossAS (5.1.0.GA or higher) http://www.jboss.org/jbossas
2. JBossAS (4.8 or higher) http://www.jboss.org/jbossesb
3. SAVARA (version 1.0 or higher), available from http://www.jboss.org/savara/downloads

## 2.2. Installation Instructions

The installation instructions for the SAVARA Eclipse tools are:

1. Eclipse
   Download the latest version of Eclipse JEE, and install in your environment.

2. BPMN Modeller
   When Eclipse has been lauched, go to the *Help->Install New Software..* menu item. Select the Eclipse update site for the version of Eclipse (e.g. Galileo or Helios). Within the SOA Development category, select the BPMN Project Feature. Follow the instructions to accept the license and then restart Eclipse after the plugins have been installed.

3. JBoss Tools
   Start up your Eclipse environment, and go to the *Help->Install New Software..* menu item. Select the appropriate update site URL from the JBoss Tools download page, and enter it into the top text field in the dialog window, and press the *Add* button. Once the contents of the update site is available, then select the appropriate components and follow the instructions to install them within your Eclipse environment.

   The *pi4soa core* feature should be selected from the *All JBoss Tools* category.

   If you wish to view the generated BPEL using a BPEL editor, rather than XML, then you should also select the *JBoss BPEL Editor* from the *All JBoss Tools* category.

   NOTE: If you don't install the BPEL Editor, then you will have to install GMF. This can be found on the Galileo/Helios update site, under the *Modeling* category. Select the *Graphical Modeling Framework* entry, and following the instructions to install.

4. Install SAVARA Eclipse plugins
   The Eclipse plugins for SAVARA are installed via an update site referenced on the SAVARA download page.

The installation instructions for the SAVARA Service Validator (for JBossESB) are:

1. JBossAS
   Download the latest version and follow its installation instructions.

2. JBossESB
   Download the latest version and follow the instructions for installing it into the JBossAS environment.

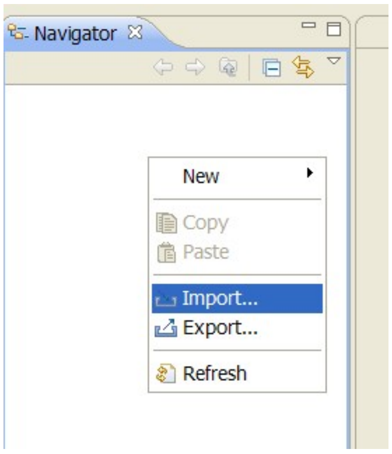
3. SAVARA
   Unpack the SAVARA distribution and edit the `deployment.properties` file in this `${SAVARA}/install` folder. Set the *org.jboss.as.home* property to the root directory where the JBossAS environment is located, and change the *org.jboss.as.config* property from default if you wish to start your JBossAS using a different configuration. Set the *org.jboss.esb.home* property to the root directory where the JBossESB environment is located.

   Start a command window and execute the command *ant deploy*.

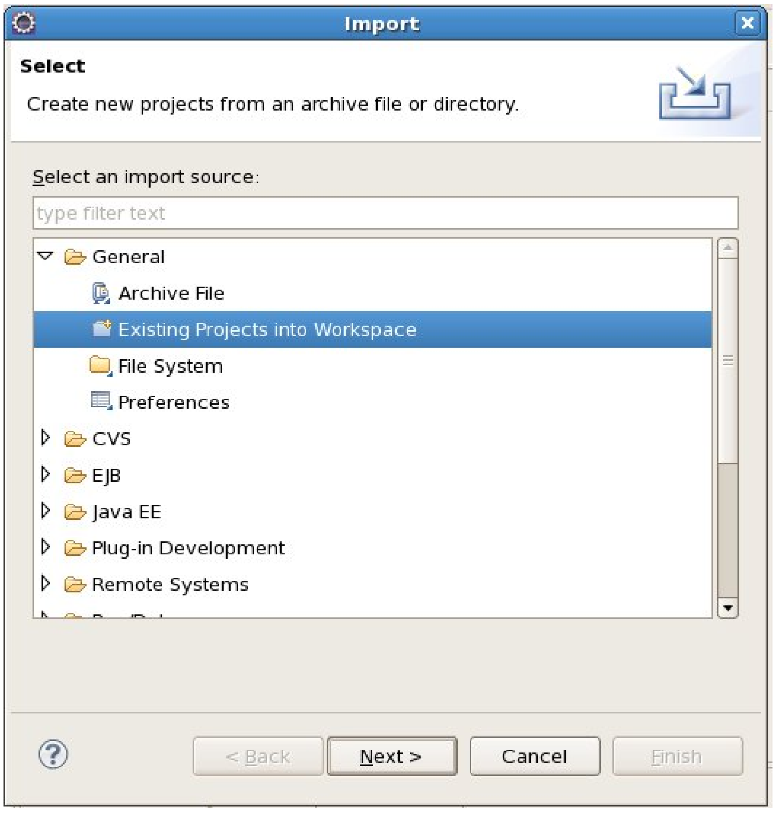
## 2.3. Importing Samples into Eclipse

Once the SAVARA Eclipse Tool 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.

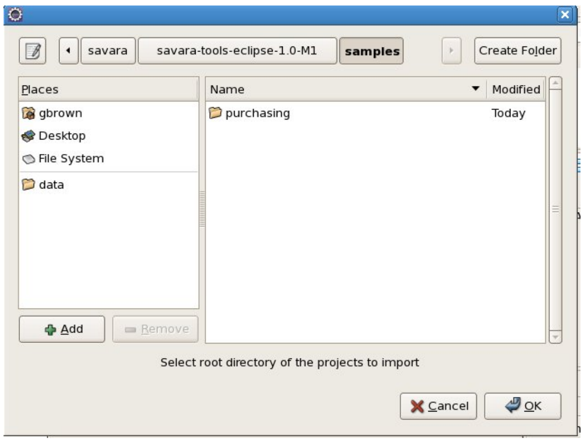
1. Select the 'Import...' menu item, associated with the popup menu on the background of the left panal (Navigator or Package depending on perspective being viewed).



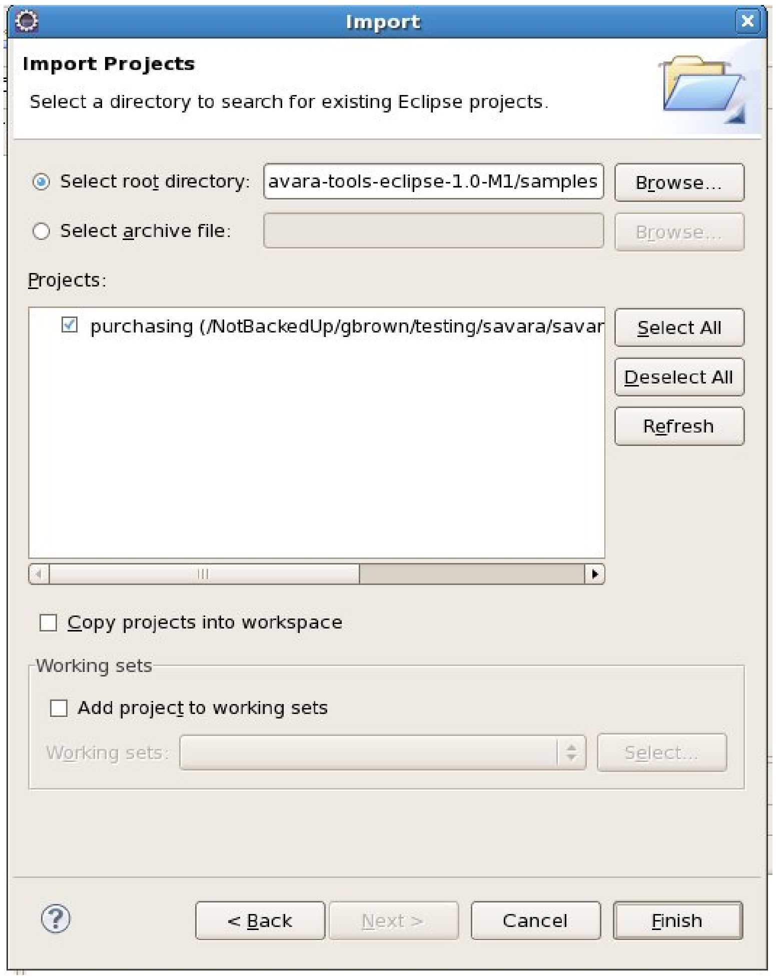
2. When the import dialog appears, select the *General->ExistingProject from Workspace* option and press the 'Next' button.



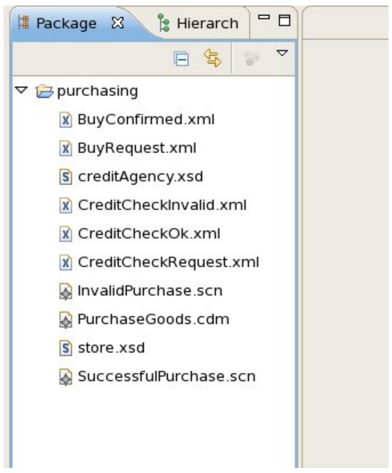
3. Ensuring that the 'Select root directory' radio button is selected, press the 'Browse' button and navigate to the `${SAVARA-Tools}/samples` folder, then press 'Ok'.



4. All of the Eclipse projects contained within the `${SAVARA-Tools}/samples` directory structure will be listed. Press the 'Finish' button to import them all.


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
avara-tools-eclipse-1.0-M1/samples
Browse...
Select archive file:
Browse...
Projects:
purchasing (/NotBackedUp/gbrown/testing/savara/savar
Select All
Deselect All
Refresh
Copy projects into workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish

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

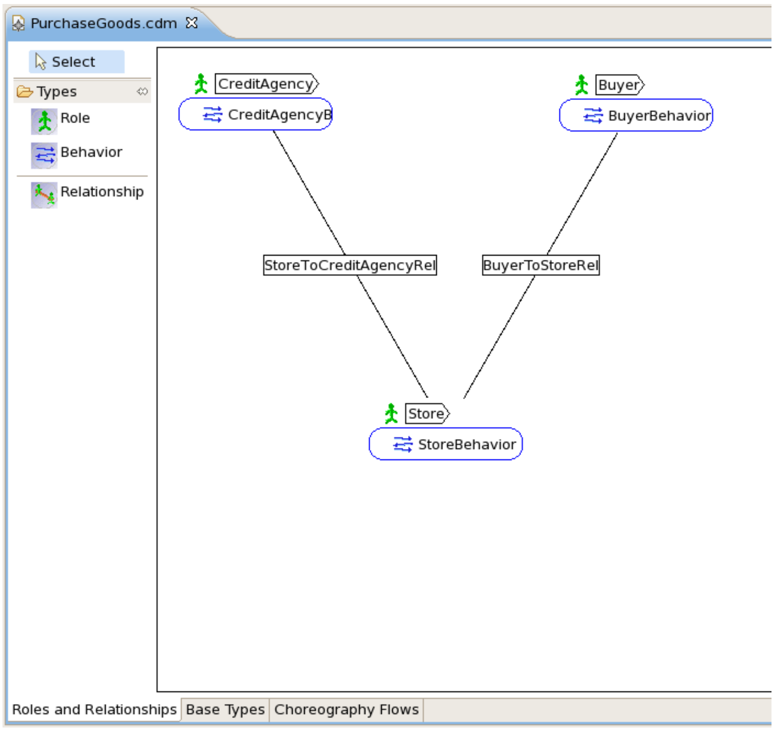


# Business Analysis

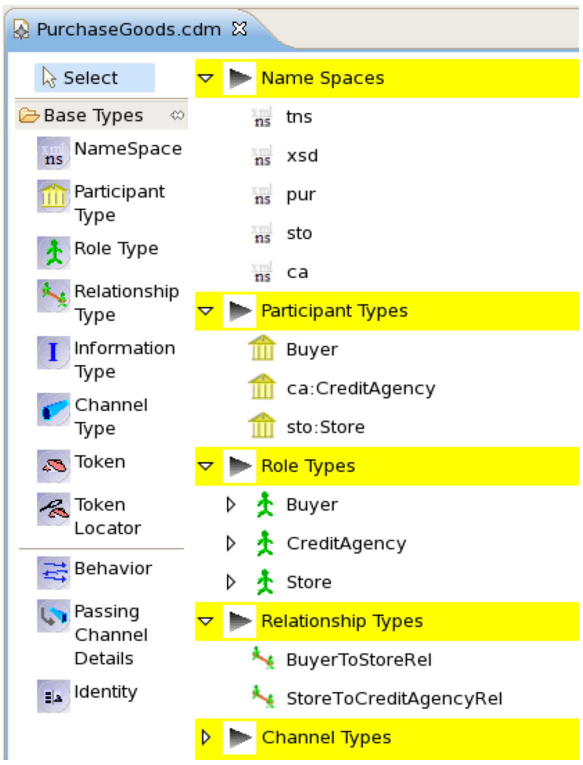
## 3.1. Define Participants

In the current Eclipse tools, that use the pi4soa Scenario and Choreography based models for defining requirements and architectural models, this phase would be achieved by defining the Participants and Roles within the choreography model.

When a choreography description is initially created, using the *New->Other->Choreography->Choreography Description* menu item, the roles and relationships can be defined on the first tab.



Default participant types are automatically created, one per role, and can be found on the *Base Types* tab. For example,



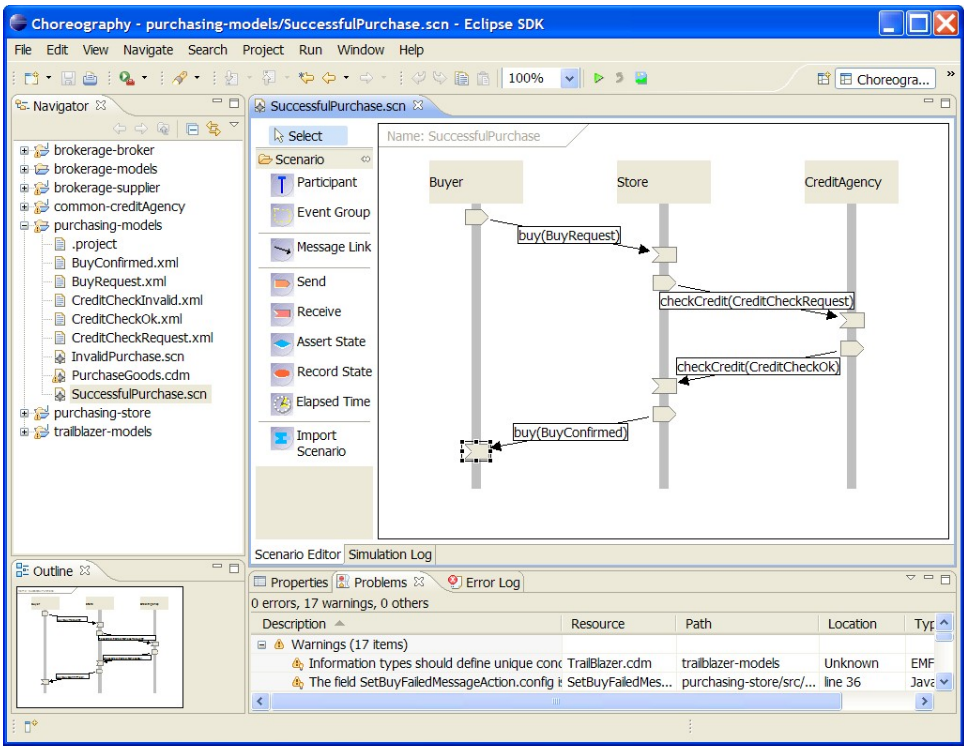
Only these components need to be specified in the choreography model. This enables them to be referenced in the subsequently defined scenarios. Otherwise it would be necessary to return to the scenarios, once the choreography model had been defined in the *Architecture* phase.

## 3.2. Outline Scenarios

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:



0 errors, 17 warnings, 0 others

| Description | Resource | Path | Location | Typ |
|---|---|---|---|---|
| Warnings (17 items) | | | | |
| Information types should define unique conc | TrailBlazer.cdm | trailblazer-models | Unknown | EMF |
| The field SetBuyFailedMessageAction.config i | SetBuyFailedMes... | purchasing-store/src/... | line 36 | Java |

The business requirements can therefore defined as a set of scenarios, each demonstrating a specific use-case, or path through the business process being enacted.

## 3.3. Create Example Messages

The next step is to create the example messages required by the scenarios.

Some previously defined examples can be found in the `process-models` Eclipse project. For example, the Buy request is defined as:

```
<tns:BuyRequest xmlns:tns="http://www.jboss.org/examples/store"
            id="1" />
```

Although a schema may not have been defined at this stage, unless one previously existed that is being reused, it is a good idea to define a namespace for the message type. This is because it will be used within the scenarios and architectural models defined in the following stage. If the namespace was not specified at this stage, then the example messages, scenarios and architectural models would need to be updated at a later stage.

Although this phase has been defined following the definition of the scenarios, in practice these phases are iterative. So scenarios and example messages would be defined concurrently. Similarly, new participants may be added in an evolutionary manner, as scenarios are created that require them.

# Architecture

## 4.1. Define Information Model

One of the stages within the architecture phase is to define the information model for the message types associated with the messages exchanges between the interacting participants.

This involves defining message schema for each example message. The schema could already exist and be reused, it could be based on existing schema and just need to be upgraded to support new requirements, or it may need to be defined from scratch.

An example of a schema associated with the purchasing model is the `store.xsd` shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.jboss.org/examples/store"
        xmlns:tns="http://www.jboss.org/examples/store"
        elementFormDefault="qualified">

    <element name="BuyRequest" type="tns:StoreType"></element>
    <element name="BuyConfirmed" type="tns:StoreType"></element>
    <element name="BuyFailed" type="tns:StoreType"></element>

    <complexType name="StoreType">
        <attribute name="id" type="string"></attribute>
    </complexType>
</schema>
```

Once the schema has been defined, then the example messages need to be updated to reference the schema, as shown in the following `BuyRequest.xml` example message:

```
<tns:BuyRequest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:tns="http://www.jboss.org/examples/store"
            xsi:schemaLocation="http://www.jboss.org/examples/store store.xsd "
            id="1" />
```

### 4.1.1. Validating Example Messages against Schema

Once the association between example messages and the schema has been established, it is possible to validate the messages against the schema.

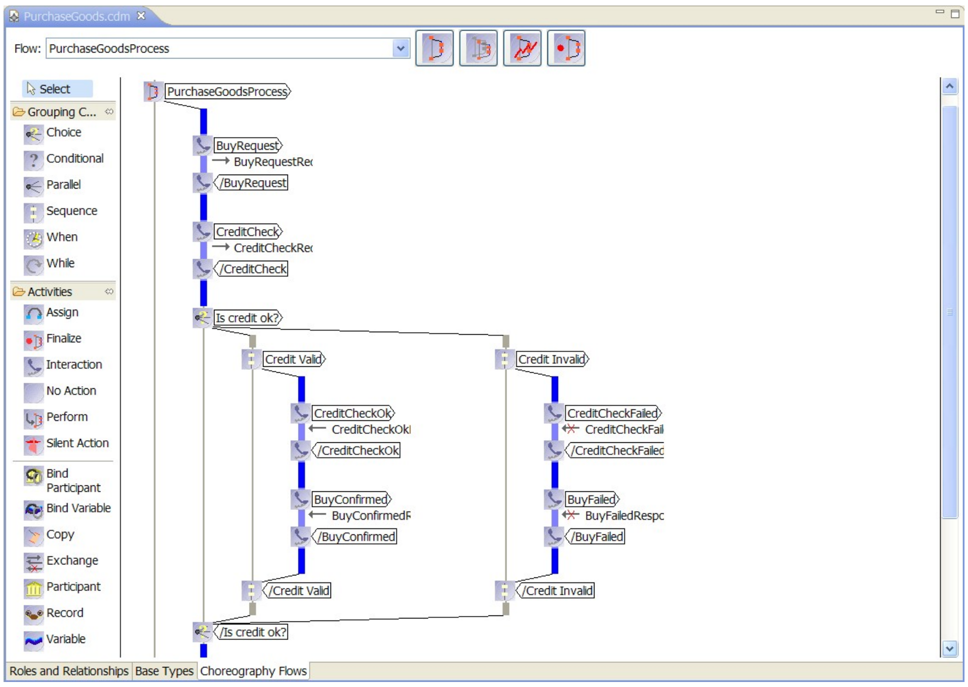
For information on how to use the validation capabilities within Eclipse, please read the Eclipse XML Validation Tutorial.

## 4.2. Define Choreography Model

The next step in the development process is to specify a Choreography Model to implement the requirements described within the set of scenarios.

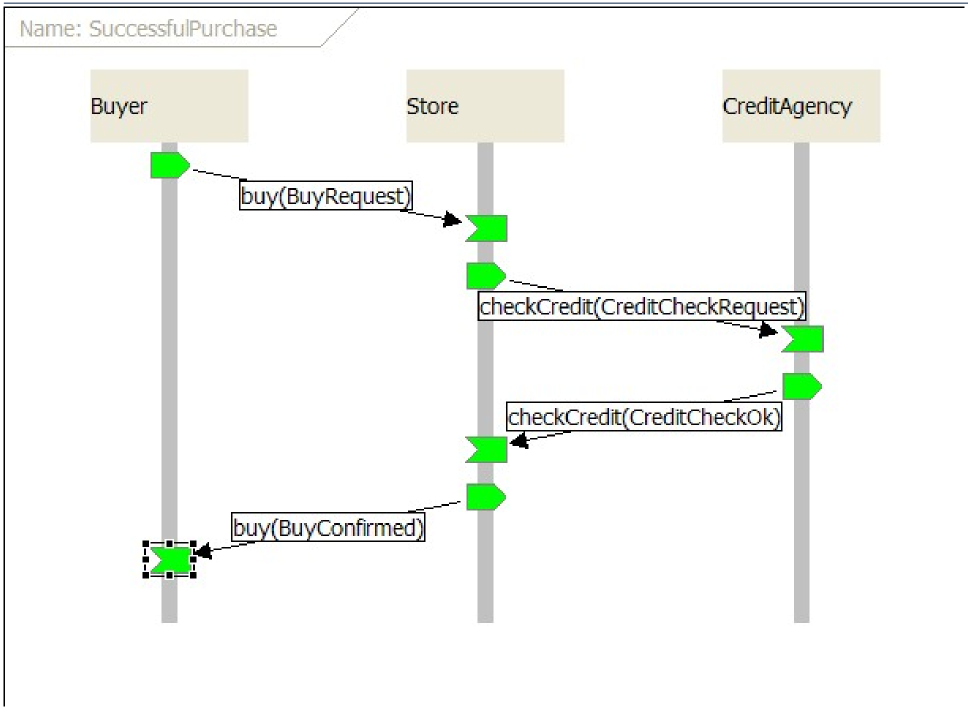
The current representation used to define Choreography Models within SAVARA is the W3C Web Service Choreography Description Language (WS-CDL). The pi4soa tools provide a WS-CDL (or choreography description) editor. Although this standard is associated with web services, it does not mean that a system specified using this standard needs to be implemented using web services. The actual WS-CDL language is used for defining the interactions between any distributed system.

The choreography description 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:

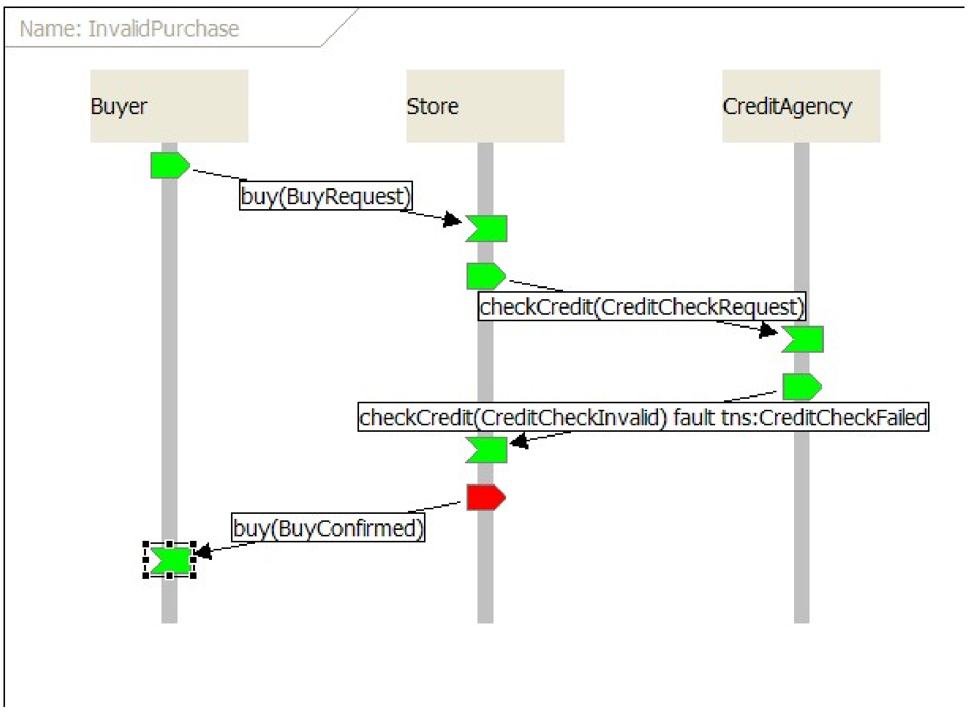


### 4.2.1. Validating Requirements against Choreography Model

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.


Name: InvalidPurchase
Buyer
Store
CreditAgency
buy(BuyRequest)
checkCredit(CreditCheckRequest)
checkCredit(CreditCheckInvalid) fault tns:CreditCheckFailed
buy(BuyConfirmed)

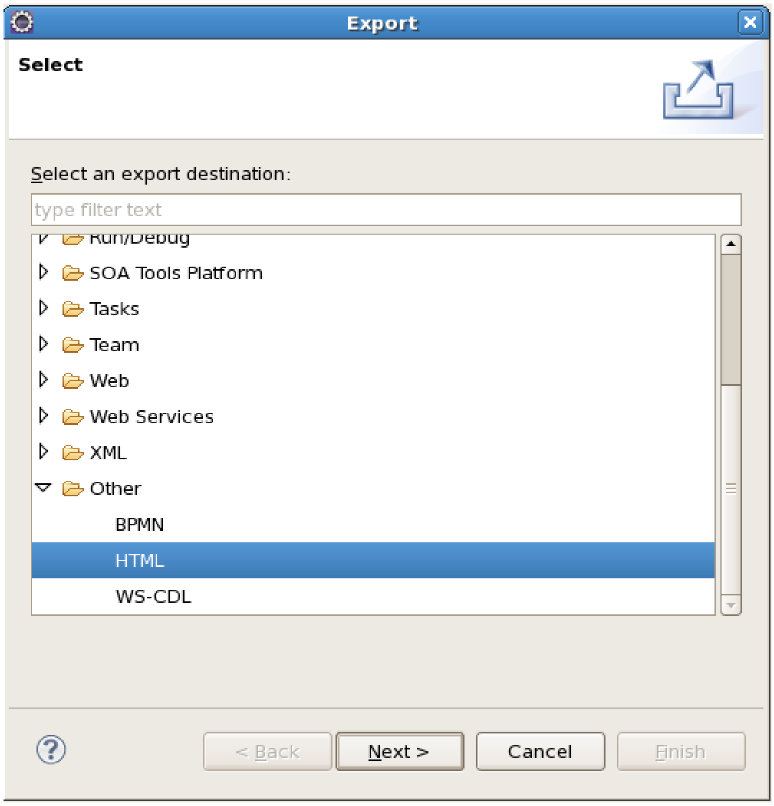
You will notice that the *Store* participant has a red 'send' node, indicating that this action was not expected behaviour when compared with the choreography description. The reason this is considered an error, is that the *Store* participant should only send a *BuyFailed* message following an invalid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

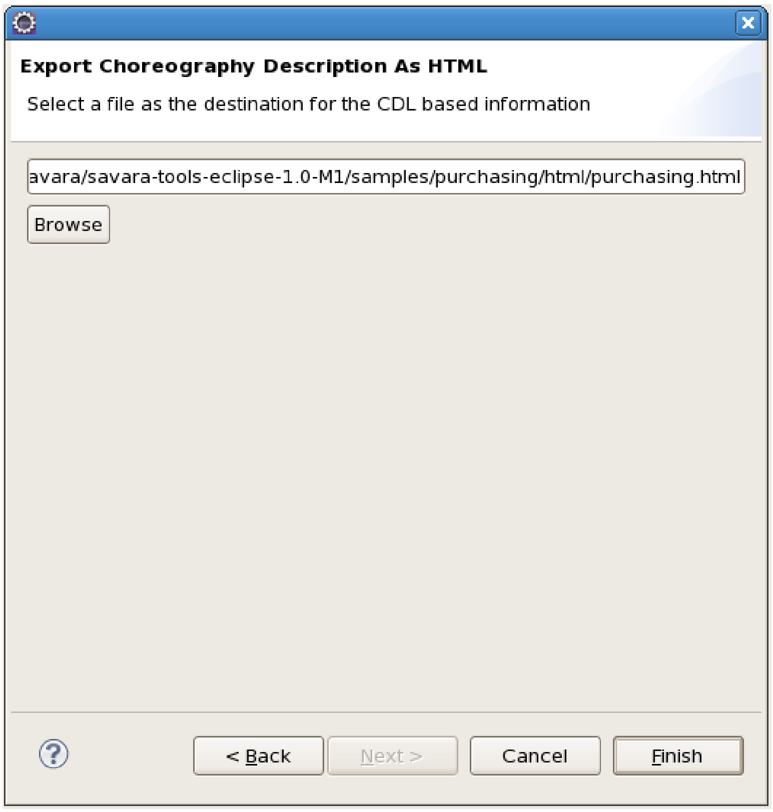
### 4.2.2. Create Documentation

Once the choreography description has been successfully tested against the scenarios, the next step may be to obtain approval to proceed to the analysis/design phase. To help support this effort, the pi4soa tools provide the means to export the choreography description to a range of representations. HTML documentation generated is discussed below, and BPMN diagram generation is discussed in the Service Oriented Analysis and Design section.

To generate HTML documentation, select the *Export->Other->HTML* menu item associated with the choreography description file.



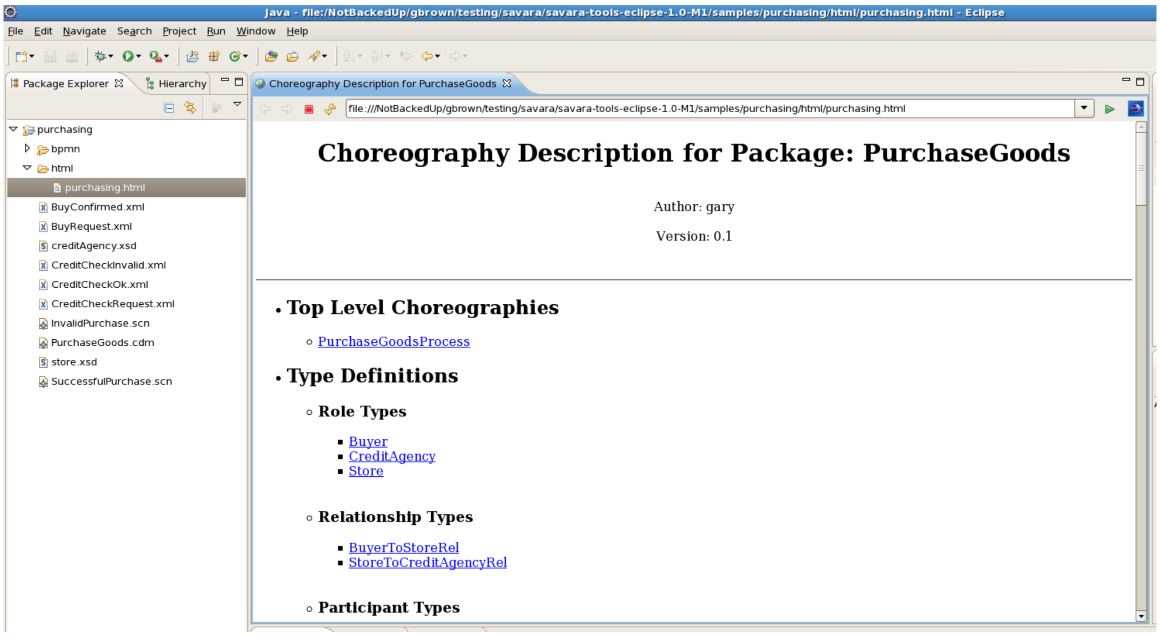
The next step is to provide the location and name of the HTML file to be generated.


**Export Choreography Description As HTML**

Select a file as the destination for the CDL based information

avara/savara-tools-eclipse-1.0-M1/samples/purchasing/html/purchasing.html

If the HTML has been generated within the scope of Eclipse project, then refresh the relevant folder to show the file and open the file with the Eclipse web browser (as shown below). If outside the Eclipse project, then use a normal web browser to view the file.


Java - file:/NotBackedUp/gbrown/testing/savara/savara-tools-eclipse-1.0-M1/samples/purchasing/html/purchasing.html - Eclipse

- purchasing
  - bpmn
  - html
    - purchasing.html
  - BuyConfirmed.xml
  - BuyRequest.xml
  - creditAgency.xsd
  - CreditCheckInvalid.xml
  - CreditCheckOk.xml
  - CreditCheckRequest.xml
  - InvalidPurchase.scn
  - PurchaseGoods.cdm
  - store.xsd
  - SuccessfulPurchase.scn

file:///NotBackedUp/gbrown/testing/savara/savara-tools-eclipse-1.0-M1/samples/purchasing/html/purchasing.html

**Choreography Description for Package: PurchaseGoods**

Author: gary

Version: 0.1

- **Top Level Choreographies**
  - PurchaseGoodsProcess
- **Type Definitions**
  - **Role Types**
    - Buyer
    - CreditAgency
    - Store
  - **Relationship Types**
    - BuyerToStoreRel
    - StoreToCreditAgencyRel
  - **Participant Types**

# Service Oriented Analysis and Design

At this point in the lifecycle, various activities would occur related to reviewing services (i.e. in a SOA Repository) and understanding whether existing services meet requirements, need to be modified, or whether new services need to be developed from scratch.

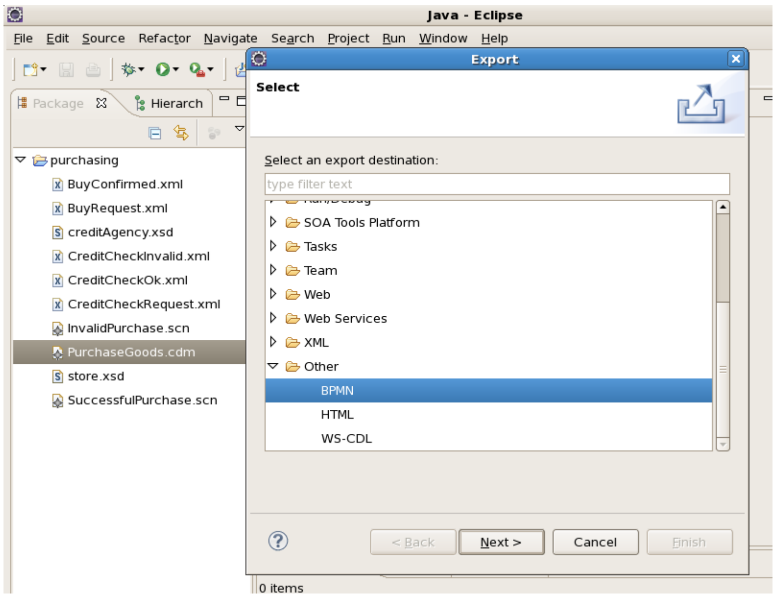
## 5.1. Service Oriented Design

In the current SAVARA tooling, the main functionality in the Service Oriented Design phase is the generation of BPMN (version 1) diagrams. These diagrams can be used as guidance for the development teams that are implementing the individual services.

It is also possible to extend the generated BPMN (version 1) diagrams to include service logic. However it should be noted that changes to the choreography or BPMN diagrams will not be synchronized/merged. So changes in the choreography will not be checked for conformance against previously generated BPMN diagrams, and it will be necessary to generate new 'service contract' BPMN (version 1) diagrams to reflect changes in behaviour of a service within the updated choreography.

In future versions of the SAVARA, based on BPMN2, it will be possible to formally check BPMN2 process models for conformance against a choreography model, and potentially synchronize differences in *externally observable behaviour* between them.

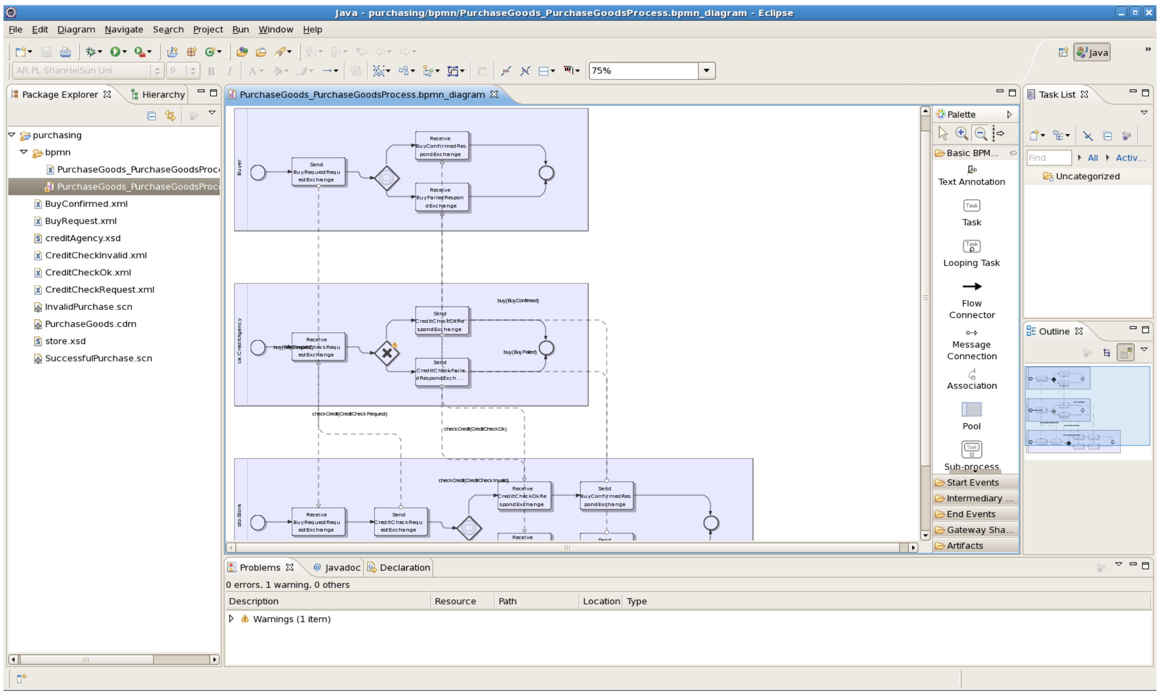
To generate a BPMN (version 1) diagram from a choreography, select the *Export* menu item associated with the choreography file, and select the *Other->BPMN* option.



Once the option has been selected, you will be asked to select the location where the generated BPMN diagrams should be stored. A diagram will be created containing all of the participants involved in the choreography in a single collaboration diagram.

Select a folder that is located within a project in your Eclipse workspace. Once the folder has been chosen, the diagrams will be generated. To see them within the Eclipse project, you will need to *refresh* the relevant folder.

The generated diagram will appear as two files, one contains the underlying BPMN model (i.e. the information about the tasks, control links, message links, etc.) and the other file contains the diagram information (i.e. node positions, etc). Double click on the file with the `.bpmn_diagram` suffix to view the diagram in the Eclipse BPMN editor.


Java - purchasing/bpmn/PurchaseGoods_PurchaseGoodsProcess.bpmn_diagram - Eclipse
File Edit Diagram Navigate Search Project Run Window Help
75%
Java
Package Explorer
Hierarchy
purchasing
bpmn
PurchaseGoods_PurchaseGoodsProc
PurchaseGoods_PurchaseGoodsProc
BuyConfirmed.xml
BuyRequest.xml
creditAgency.xsd
CreditCheckInvalid.xml
CreditCheckOk.xml
CreditCheckRequest.xml
InvalidPurchase.scn
PurchaseGoods.cdm
store.xsd
SuccessfulPurchase.scn
PurchaseGoods_PurchaseGoodsProcess.bpmn_diagram
Buyer
Send BuyRequestRequestExchange
Receive BuyConfirmedResponseExchange
Receive BuyFailedResponseExchange
ca.CreditAgency
Receive CreditCheckRequestExchange
Send CreditCheckOkResponseExchange
Send CreditCheckFailedResponseExchange
buy(BuyConfirmed)
buy(BuyFailed)
checkCredit(CreditCheck Request)
checkCredit(CreditCheckOk)
checkCredit(CreditCheckInvalid)
sto:Store
Receive BuyRequestRequestExchange
Send CreditCheckRequestExchange
Receive CreditCheckOkResponseExchange
Send BuyConfirmedResponseExchange
Palette
Basic BPM...
Text Annotation
Task
Looping Task
Flow Connector
Message Connection
Association
Pool
Sub-process
Start Events
Intermediary ...
End Events
Gateway Sha...
Artifacts
Task List
Find
All
Activ...
Uncategorized
Outline
Problems
Javadoc
Declaration
Description
Resource
Path
Location
Type

0 errors, 1 warning, 0 others

Warnings (1 item)

# Service Development

Services can be developed by generating initial development artifacts, based on artifacts created in preceding phases (e.g. global model or service contracts/designs).

To ensure that the services continue to conform to the artifacts defined in the previous phases, the tools perform conformance checking between the service implementation and the existing architecture/design artifacts. This is not possible with all implementation languages - they must provide the means to extract the communication structure for comparison.

The following sections explain how the generation and conformance checking can be achieved for the WS-BPEL implementation language.

## 6.1. WS-BPEL

This tools include a capability to generate a service implementation, for a participant in a choreography, using WS-BPEL. A completed version of the *PurchasingGoods* example can be found in the samples directory (which can be imported into Eclipse).

However if you wish to generate the example from scratch, the follow the instructions in this section. More information about how to use this feature can be found in the User Guide.

### 6.1.1. Generating WS-BPEL based Services

When a choreography description has been created, it is possible to generate a BPEL Process (and associated WSDL files and deployment descriptor) for each of the participants defined within the choreography. To try this out, select the *Savara->Generate->WS-BPEL* 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* participant, 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* and *CreditAgency* participants.

Each project will contain a single `bpel` folder containing the WS-BPEL process definition for the participant, a list of relevant WSDL files and a deployment descriptor file for use with RiftSaw. Howeve the WS-BPEL and WSDL files are standard, so can be deployed to any WS-BPEL 2.0 compliant engine.

### 6.1.2. Adding implementation details to CreditAgency

#### 6.1.2.1. Deployment Descriptor

When generated, the deployment descriptor initially has the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<deploy  xmlns="http://www.apache.org/ode/schemas/dd/2007/03"  xmlns:ns1="http://www.jboss.org/
examples/creditAgency">
    <process name="ns1:PurchaseGoodsProcess_CreditAgency">
        <active>
            true
        </active>
        <provide partnerLink="StoreToCreditAgency">
            <service/>
        </provide>
    </process>
</deploy>
```

The only change necessary is to add some attributes to the *service* element:

```
            <service name="ns1:CreditAgencyService" port="CreditAgencyInterfacePort"/>
```

### 6.1.2.2. BPEL Process Definition

The generated BPEL process for the CreditAgency participant is as follows:

```
<process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:ca="http://www.jboss.org/examples/creditAgency"
        xmlns:pur="http://www.jboss.org/examples/purchasing"
        xmlns:sto="http://www.jboss.org/examples/store"
        xmlns:tns="http://www.jboss.org/savara/examples"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:ns0="http://www.scribble.org/conversation"
        ns0:conversationType="savara.samples.Common@CreditAgency"
        name="PurchaseGoodsProcess_CreditAgency"
        targetNamespace="http://www.jboss.org/examples/creditAgency"
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
    <import importType="http://schemas.xmlsoap.org/wsdl/"
                location="PurchaseGoodsProcess_CreditAgency.wsdl"
                namespace="http://www.jboss.org/examples/creditAgency"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
                location="PurchaseGoodsProcess_Store.wsdl"
                namespace="http://www.jboss.org/examples/store"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
                location="CreditAgencyPartnerLinkTypes.wsdl"
                namespace="http://www.jboss.org/examples/creditAgency"/>
    <partnerLinks>
        <partnerLink myRole="CreditAgencyService" name="StoreToCreditAgency"
                    partnerLinkType="ca:StoreToCreditAgencyServiceLT"/>
    </partnerLinks>
    <variables>
        <variable messageType="ca:CreditCheckRequest" name="creditCheckRequestVar"/>
        <variable messageType="ca:CreditCheckOk" name="creditCheckOkVar"/>
        <variable messageType="ca:CreditCheckInvalid" name="creditCheckInvalidVar"/>
    </variables>
    <sequence>
        <receive createInstance="yes" operation="checkCredit"
                partnerLink="StoreToCreditAgency" portType="ca:CreditAgencyInterface"
                        variable="creditCheckRequestVar"/>
        <if>
            <sequence>
                <reply operation="checkCredit" partnerLink="StoreToCreditAgency"
                    portType="ca:CreditAgencyInterface" variable="creditCheckOkVar"/>
            </sequence>
            <else>
                <sequence>
                    <reply faultName="ca:CreditCheckFailed" operation="checkCredit"
                            partnerLink="StoreToCreditAgency"
                            portType="ca:CreditAgencyInterface"
                            variable="creditCheckInvalidVar"/>
                </sequence>
            </else>
        </if>
    </sequence>
</process>
```

There are three changes required, the first being to add a condition following the *if* element:

```
        <if>
            <condition>
                $creditCheckRequestVar.CreditCheckRequest/pur:amount &lt;= 500
            </condition>
            ....
```

The next two changes relate to taking the information provided in the request and constructing an appropriate normal and fault response. In this simple example we only echo back the information received in the request, however more complicated processing could be performed before returning either response.

The following XML code should be added before the normal response (i.e. just inside the *sequence* element following the condition:

```
                <assign name="CopyPurchaseDetails">
                    <copy>
                        <from>$creditCheckRequestVar.CreditCheckRequest</from>
                        <to>$creditCheckOkVar.CreditCheckOk</to>
                    </copy>
                </assign>
```

The following XML code should be added before the fault response (i.e. just inside the *sequence* element that is contained in the *else* element:

```
                    <assign name="CopyPurchaseDetails">
                        <copy>
                            <from>$creditCheckRequestVar.CreditCheckRequest</from>
                            <to>$creditCheckInvalidVar.CreditCheckInvalid</to>
                        </copy>
                    </assign>
```

### 6.1.3. Adding implementation details to Store

#### 6.1.3.1. Deployment Descriptor

When generated, the deployment descriptor initially has the following content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<deploy  xmlns="http://www.apache.org/ode/schemas/dd/2007/03"  xmlns:ns1="http://www.jboss.org/
examples/store">
    <process name="ns1:PurchaseGoodsProcess_Store">
        <active>
            true
        </active>
        <provide partnerLink="BuyerToStore">
            <service/>
        </provide>
        <invoke partnerLink="StoreToCreditAgency">
            <service/>
        </invoke>
    </process>
</deploy>
```

The only changes necessary are, (1) to add a namespace prefix definition,

```
                xmlns:ns2="http://www.jboss.org/examples/creditAgency"
```

and (2) to add some attributes to the *service* element:

```
        <provide partnerLink="BuyerToStore">
            <service name="ns1:StoreService" port="StoreInterfacePort"/>
        </provide>
        <invoke partnerLink="StoreToCreditAgency">
            <service name="ns2:CreditAgencyService" port="CreditAgencyInterfacePort"/>
        </invoke>
```

### 6.1.3.2. BPEL Process Definition

The generated BPEL process for the Store participant is as follows:

```
<process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        xmlns:ca="http://www.jboss.org/examples/creditAgency"
        xmlns:pur="http://www.jboss.org/examples/purchasing"
        xmlns:sto="http://www.jboss.org/examples/store"
        xmlns:tns="http://www.jboss.org/savara/examples"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:ns0="http://www.scribble.org/conversation"
        ns0:conversationType="savara.samples.Purchasing@Store"
        name="PurchaseGoodsProcess_Store"
        targetNamespace="http://www.jboss.org/examples/store"
        xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
    <import importType="http://schemas.xmlsoap.org/wsdl/"
            location="PurchaseGoodsProcess_Store.wsdl"
            namespace="http://www.jboss.org/examples/store"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
            location="PurchaseGoodsProcess_CreditAgency.wsdl"
            namespace="http://www.jboss.org/examples/creditAgency"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
            location="StorePartnerLinkTypes.wsdl"
            namespace="http://www.jboss.org/examples/store"/>
    <partnerLinks>
        <partnerLink myRole="StoreService" name="BuyerToStore"
                partnerLinkType="sto:BuyerToStoreServiceLT"/>
        <partnerLink name="StoreToCreditAgency"
                partnerLinkType="sto:StoreToCreditAgencyLT"
                partnerRole="CreditAgencyRequester"/>
    </partnerLinks>
    <variables>
        <variable messageType="sto:BuyRequest" name="buyRequestVar"/>
        <variable messageType="ca:CreditCheckRequest" name="creditCheckRequestVar"/>
        <variable messageType="ca:CreditCheckOk" name="creditCheckOkVar"/>
        <variable messageType="sto:BuyConfirmed" name="buyConfirmedVar"/>
        <variable messageType="sto:BuyFailed" name="buyFailedVar"/>
    </variables>
    <sequence>
        <receive createInstance="yes" operation="buy" partnerLink="BuyerToStore"
                        portType="sto:StoreInterface" variable="buyRequestVar"/>
        <scope>
            <faultHandlers>
                <catch faultMessageType="ca:CreditCheckInvalid"
                        faultName="ca:CreditCheckFailed" faultVariable="creditCheckInvalidVar">
                    <sequence>
                        <reply faultName="sto:BuyFailed" operation="buy"
                                partnerLink="BuyerToStore" portType="sto:StoreInterface"
                                variable="buyFailedVar"/>
                    </sequence>
                </catch>
            </faultHandlers>
            <sequence>
                <invoke inputVariable="creditCheckRequestVar" operation="checkCredit"
                           outputVariable="creditCheckOkVar" partnerLink="StoreToCreditAgency"
                            portType="ca:CreditAgencyInterface"/>
               <reply operation="buy" partnerLink="BuyerToStore" portType="sto:StoreInterface"
                                variable="buyConfirmedVar"/>
            </sequence>
        </scope>
    </sequence>
</process>
```

There are three changes required. The first being to add an assignment statement within the *catch* element's *sequence* prior to the *reply*:

```
                        <assign name="CopyPurchaseDetails">
                            <copy>
                                <from>$creditCheckInvalidVar.CreditCheckInvalid</from>
                                <to>$buyFailedVar.BuyFailed</to>
                            </copy>
                        </assign>
```

The remaining two changes relate to taking the information received in the initial request, to construct a request to the credit agency, and then extracting the information from the credit agency response, to return it to the *Store* client. The following snippet shows the two assignment statements either side of the *invoke* statement:

```
                <assign name="CopyPurchaseDetails">
                    <copy>
                        <from>$buyRequestVar.BuyRequest</from>
                        <to>$creditCheckRequestVar.CreditCheckRequest</to>
                    </copy>
                </assign>
                <invoke inputVariable="creditCheckRequestVar" operation="checkCredit"
                           outputVariable="creditCheckOkVar" partnerLink="StoreToCreditAgency"
                                portType="ca:CreditAgencyInterface"/>
                <assign name="CopyPurchaseDetails">
                    <copy>
                        <from>$creditCheckOkVar.CreditCheckOk</from>
                        <to>$buyConfirmedVar.BuyConfirmed</to>
                    </copy>
                </assign>
```

## 6.2. Summary

This section has provided a brief introduction to the design-time SOA governance features provided within the SAVARA Eclipse Tools distribution.

The aim of these capabilities are to enable verification of an implementation, initially defined just using BPEL process definitions, 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 a later section (Runtime Validation).

# Runtime Validation

**Note**

Before you can deploy and run the runtime validation example, you will need to install the SAVARA Validator module for JBossESB.

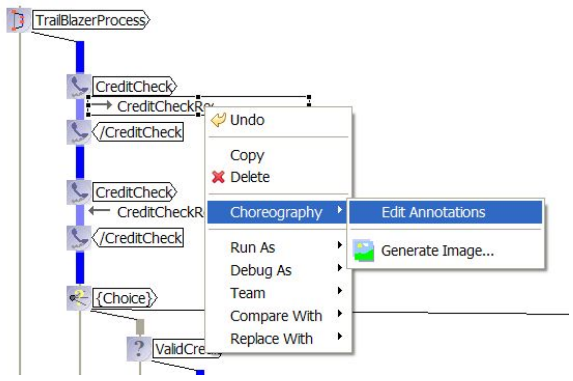
Once services have been deployed, as mentioned in the previous section, we still need to be able to verify that the services continue to conform to the choreography description. The *Conversation Validation* capability within the SAVARA distribution can be used to validate the behaviour of each service.

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

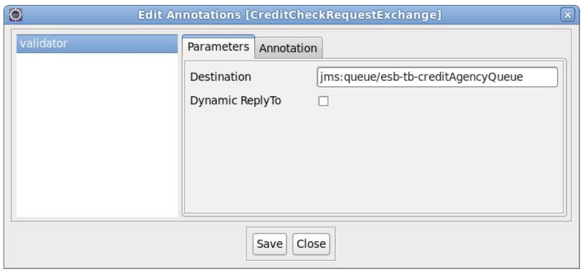
## 7.1. Service Validator Configuration

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 'validator'. 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).



Destination: jms:queue/esb-tb-creditAgencyQueue
Dynamic ReplyTo: ☐

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

Once 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/savara-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 `savara-validator.esb/validator-config.xml` within the JBossAS environment is modified, or choreography description files added, removed or updated within the `savara-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.

## 7.2. 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**.

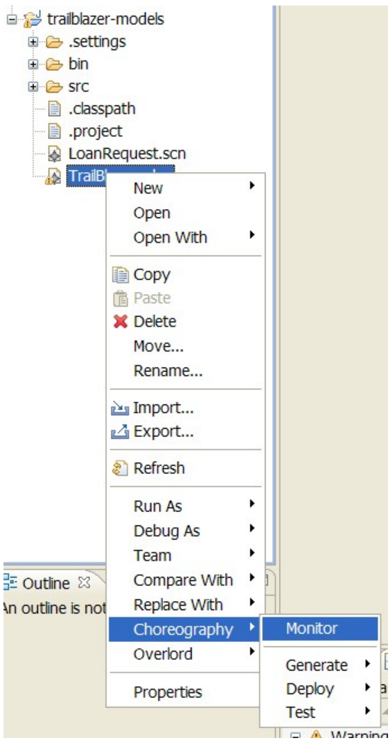
3. From the `trailblazer/banks` folder, execute the command to start the JMS Bank service: **ant runFileBank**.

## 7.3. Starting the pi4soa Monitor

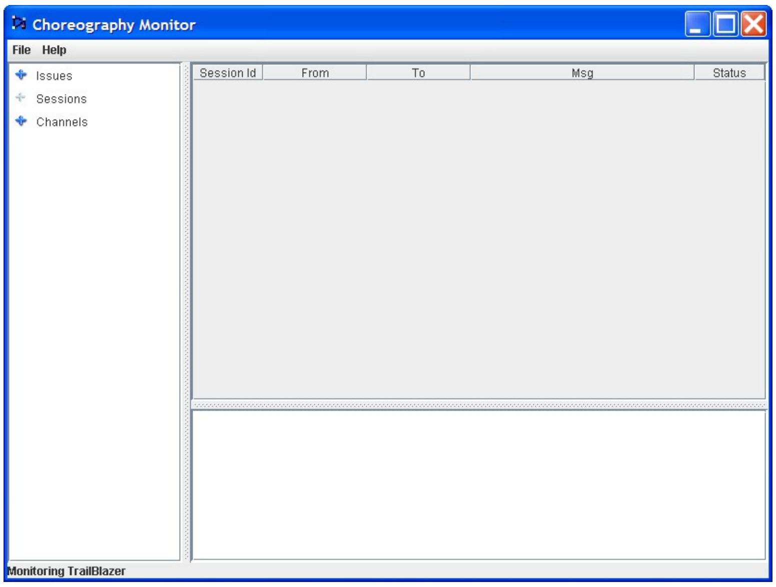
The pi4soa Monitor is used to observe a correlated view of the executing business transactions. Each service validator can be configured to report activites (i.e. sent and received messages) that it validates, to enable the correlator to reconstitute a global interpretation of each transaction.

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.

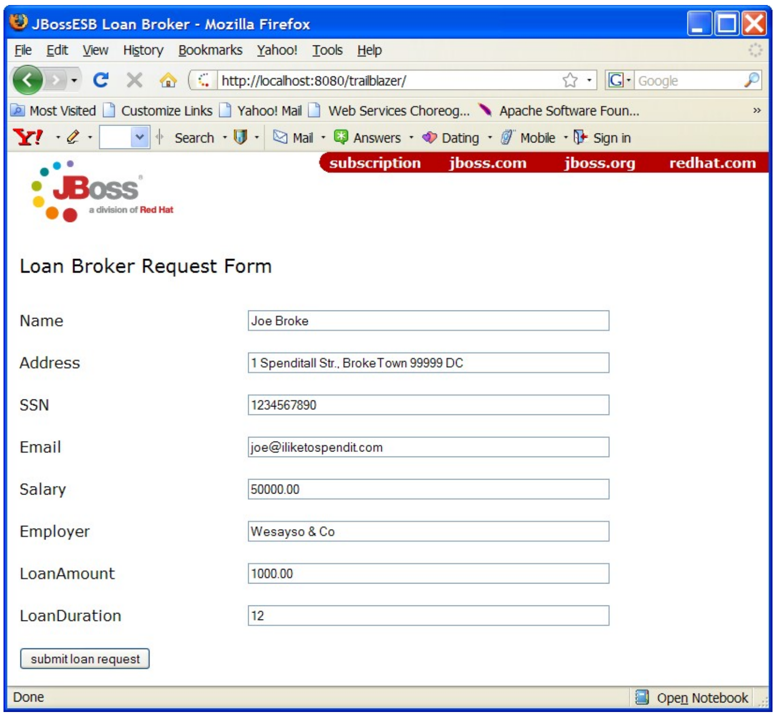


Wait for the monitor window to start, and indicate that the choreography is being monitored, shown in the status line at the bottom of the window.



## 7.4. 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:


JBossESB Loan Broker - Mozilla Firefox
http://localhost:8080/trailblazer/
subscription
jboss.com
jboss.org
redhat.com
submit loan request
Done

Loan Broker Request Form

| | |
|---|---|
| Name | Joe Broke |
| Address | 1 Spenditall Str., BrokeTown 99999 DC |
| SSN | 1234567890 |
| Email | joe@iliketospendit.com |
| Salary | 50000.00 |
| Employer | Wesayso & Co |
| LoanAmount | 1000.00 |
| LoanDuration | 12 |

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.

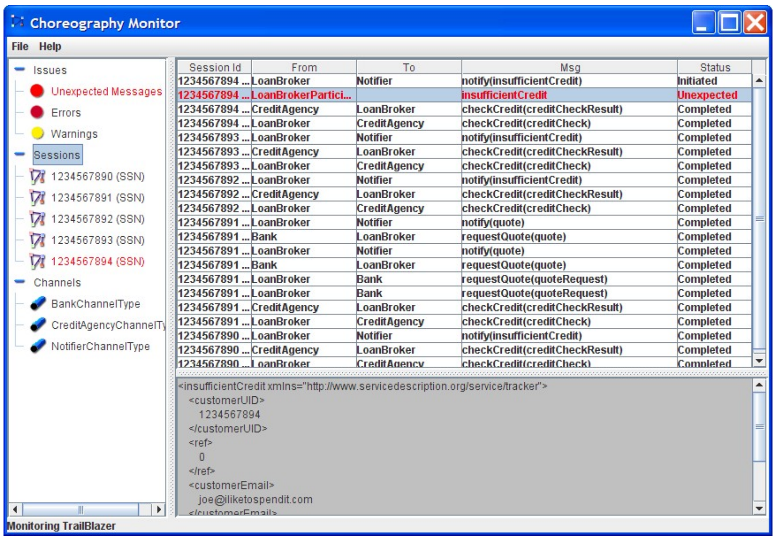
## 7.5. Detecting a Validation Error

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 `${SAVARA}/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.



| Session Id | From | To | Msg | Status |
|---|---|---|---|---|
| 1234567894 ... | LoanBroker | Notifier | notify(insufficientCredit) | Initiated |
| 1234567894 ... | LoanBrokerPartici... | | insufficientCredit | Unexpected |
| 1234567894 ... | CreditAgency | LoanBroker | checkCredit(creditCheckResult) | Completed |
| 1234567894 ... | LoanBroker | CreditAgency | checkCredit(creditCheck) | Completed |
| 1234567893 ... | LoanBroker | Notifier | notify(insufficientCredit) | Completed |
| 1234567893 ... | CreditAgency | LoanBroker | checkCredit(creditCheckResult) | Completed |
| 1234567893 ... | LoanBroker | CreditAgency | checkCredit(creditCheck) | Completed |
| 1234567892 ... | LoanBroker | Notifier | notify(insufficientCredit) | Completed |
| 1234567892 ... | CreditAgency | LoanBroker | checkCredit(creditCheckResult) | Completed |
| 1234567892 ... | LoanBroker | CreditAgency | checkCredit(creditCheck) | Completed |
| 1234567891 ... | LoanBroker | Notifier | notify(quote) | Completed |
| 1234567891 ... | Bank | LoanBroker | requestQuote(quote) | Completed |
| 1234567891 ... | LoanBroker | Notifier | notify(quote) | Completed |
| 1234567891 ... | Bank | LoanBroker | requestQuote(quote) | Completed |
| 1234567891 ... | LoanBroker | Bank | requestQuote(quoteRequest) | Completed |
| 1234567891 ... | LoanBroker | Bank | requestQuote(quoteRequest) | Completed |
| 1234567891 ... | CreditAgency | LoanBroker | checkCredit(creditCheckResult) | Completed |
| 1234567891 ... | LoanBroker | CreditAgency | checkCredit(creditCheck) | Completed |
| 1234567890 ... | LoanBroker | Notifier | notify(insufficientCredit) | Completed |
| 1234567890 ... | CreditAgency | LoanBroker | checkCredit(creditCheckResult) | Completed |
| 1234567890 ... | LoanBroker | CreditAgency | checkCredit(creditCheck) | Completed |

```
<insufficientCredit xmlns="http://www.servicedescription.org/service/tracker">
  <customerUID>
    1234567894
  </customerUID>
  <ref>
    0
  </ref>
  <customerEmail>
    joe@iliketospendit.com
  </customerEmail>
```

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