iCOCOA Semantic Services Tutorial

Graham Thomson, Sebastien Bianco

Table of Contents

1 Introduction

In this tutorial we shall look at how to construct a smart home alarm clock application that is based on Semantic Services. The application switches on a radio and brews a cup of coffee when the alarm activates.
This tutorial has several parts: first, we will look at the services – the radio and coffee maker – and how they are described using the Amigo-S semantic service description language; next, we look at how to a task is described and how it composes the services in the way the alarm clock application requires; then, we look at how to create the application and how to use the semantic service repository; and finally, we look at how to deploy and run the application.
For more a more detailed over view of Semantic Services middleware, please refer to Amigo D3.4 Amigo overall middleware: First Prototype implementation & documentation[1].

2 How to create semantic services and descriptions

Our application requires to services to run – a radio service, and a coffee machine service. When the alarm clock activates we want the application to switch on the radio, switch on the coffee machine, and then instruct the coffee machine to brew a cup of coffee. A nice way to start the day in our smart home!
All the code, scripts, descriptions, etc. that feature in this tutorial are available in full in the examples download that accompanies this tutorial. Please explore the contents of this download. The download is in the format of an Eclipse project. It is recommend to use the Eclipse IDE (available at www.eclipse.org) to follow the examples in this tutorial.

2.1 How to create and deploy the services

First we must construct the services we are going to use – the radio and the coffee machine. Below is the listing of some simple Java code for these services. These listings can be found in full in the src\examples\alarmclock folder of the examples download.

The radio interface:

```
package examples.alarmclock;
public interface FmRadio {
	void switchOn();
	void switchOff();
	void increaseVolume();
	void decreaseVolume();
	void findNextStation();
}
```

The coffee machine interface:

```
package examples.alarmclock;
public interface CoffeeMachine {
	void switchOn();
	void switchOff();
	void brew();
	void serve();
}
```

[1] http://www.hitech-projects.com/euprojects/amigo/deliverables/amigo_d3.4_correctedfinal.pdf

The radio implementation:

```
package examples.alarmclock;
public class FmRadioService implements FmRadio {

	private int volume = 5;
	private double[] stations = { 88.6, 99.5, 100.8, 102.5 };
	private int currentStation = 0;
	private boolean switchedOn = false;

	public void switchOn() {
		if (!switchedOn) {
			switchedOn = true;
			System.out.println("The FM Radio has been switched on.");
		}}

	public void switchOff() {
		if (switchedOn) {
			switchedOn = false;
			System.out.println("The FM Radio has been switched
		off.");
		}}

	public void decreaseVolume() {
		if (volume > 0) {
			--volume;
		}
		System.out.println("Volume decreased to " + volume + ".");}

	public void increaseVolume() {
		if (volume < 10) {
			++volume;
		}
		System.out.println("Volume increased to " + volume + ".");}

	public void findNextStation() {
		currentStation = (currentStation + 1) % stations.length;
		System.out.println("Found station " + stations[currentStation]
		+ " FM.");}
}
```

And the coffee machine implementation:

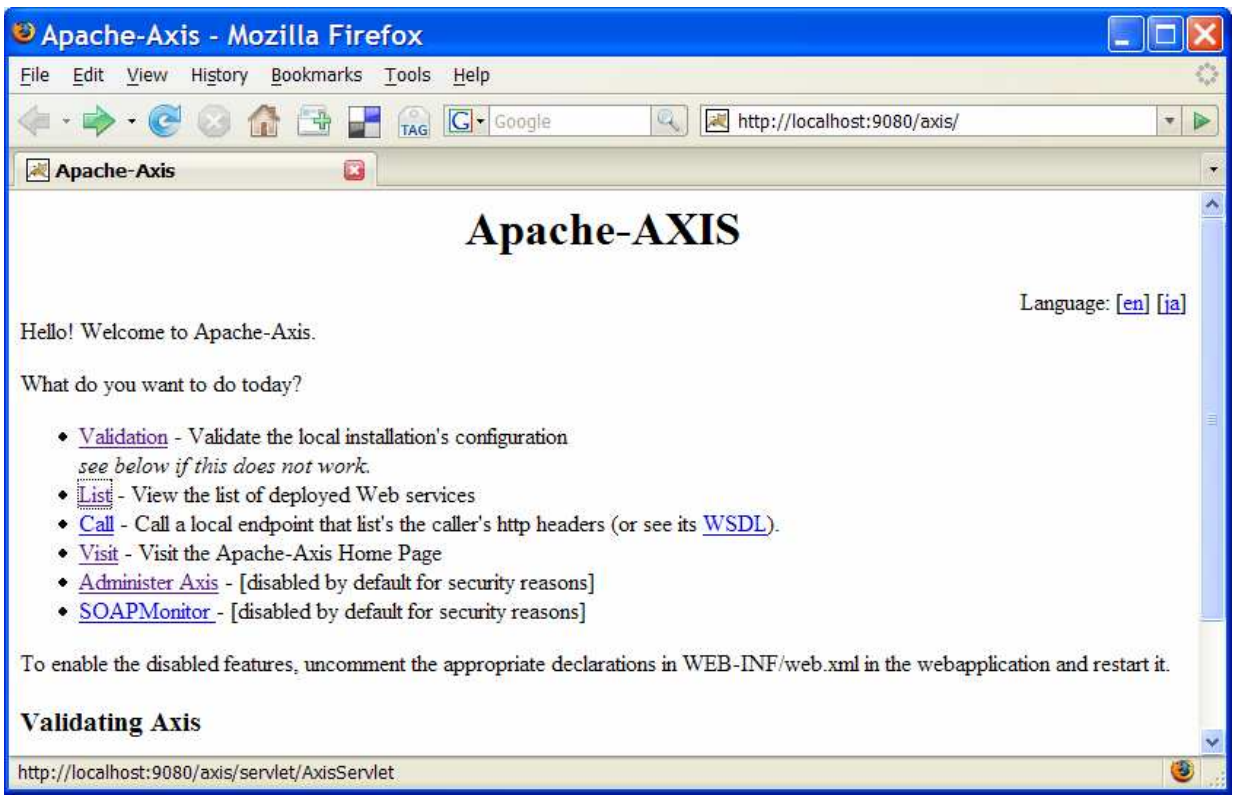
```
package examples.alarmclock;
public class CoffeeMachineService implements CoffeeMachine {

	private boolean switchedOn;
	private boolean brewed;

	public void switchOn() {
		if (!switchedOn) {
			switchedOn = true;
			System.out.println("The Coffee Machine has been switched
			on.");}}

	public void switchOff() {
		if (switchedOn) {
			switchedOn = false;
			System.out.println("The Coffee Machine has been switched
			off."); }}

	public void brew() {
		if (brewed) {
			System.out.println("The coffee is already brewed!"); }
		else { brewed = true; }}

	public void serve() {
		if (!brewed) {
			System.out.println("Please brew the coffee first."); }
		else { brewed = false; System.out.println("The coffee is being
	served."); }}}
```

Once we have these classes compiling and running, we must make them into web services. In this tutorial we will use Apache Tomcat and Apache Axis to host web services, and Active Endpoints ActiveBPEL to executed composed semantic service applications. If you do not already have these installed, grab them from http://tomcat.apache.org/, http://ws.apache.org/axis/ and http://www.active-endpoints.com/ respectively. Get Apache Tomcat 5.5, Apache Axis 1.4 and ActiveBPEL 4.1.
If you are unfamiliar with these projects, documentation is available at the project pages. Here, we'll go over the essentials for using them with the Semantic Services middleware. For debugging for Tomcat or Axis install however, consult the documentation.
In the examples, Tomcat is installed as a Windows service on port 9080. If you install Tomcat on a different port, please update the examples appropriately.
OK – were ready to turn make our services available. Make sure Axis is running, and deploy the services using the scripts provided:

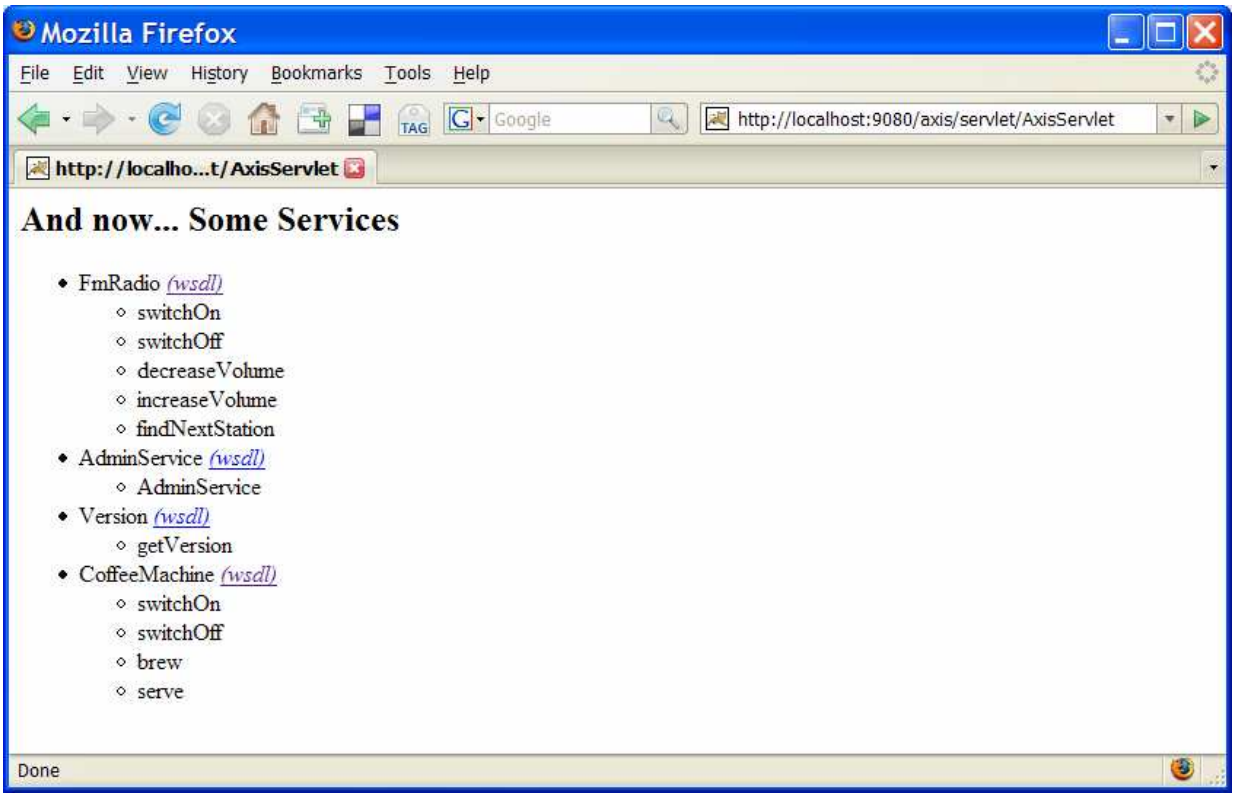
```
cd axis
axisDeploy.bat CoffeeMachineDeploy.wsdd axisDeploy.txt FmRadioDeploy.wsdd
```

Take a look at the .bat script and the .wsdd files and understand what is going on. You will be able to use the script and the .wsdd files as templates for your own services.
Now copy your class files up into the Axis classes folder. This folder will typically be … \Tomcat 5.5\webapps\axis\WEB-INF\classes.

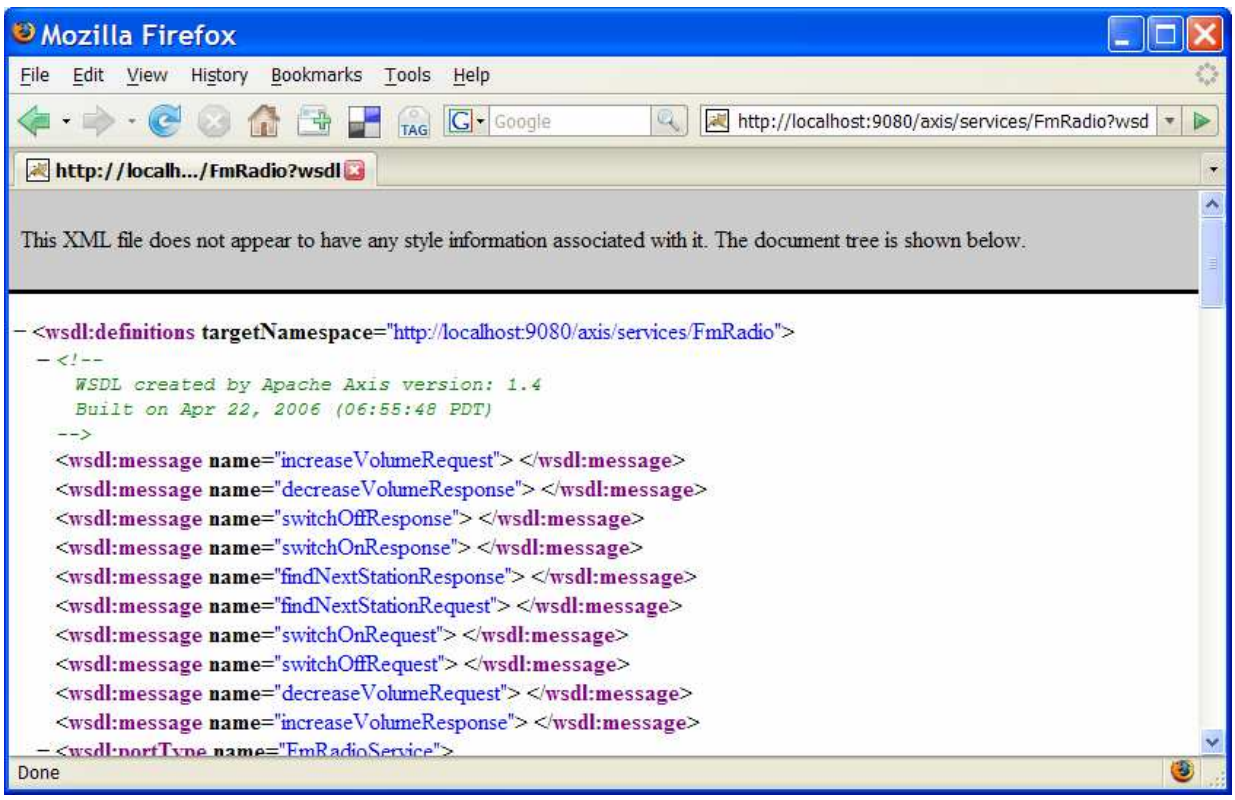
Now go to the Axis page in your browser:



And click on the “List” option:



This lists all the services that are currently deployed on the local Axis instance. We can see the FM radio and coffee machine services.
To create a semantic service description for our services, we need to have the Web Services Description Language (WSDL) file for it. Fortunately, Axis provides these for us! Click on the “(wsdl)” link beside the FM radio service:



This is the WSDL for the service. Save this file locally, giving it a .wsdl extension. Then, go back to the services listing and do the same to get the WSDL for the coffee machine service.
Note that Example WSDL files for the services are already included in the \wsdl folder of the examples download for reference. Make sure to use your own for the services installed on your machine!
We have finished created and deploying our services!

2.2 How to create semantic service descriptions

We have created and deployed our services, but as they are, we can only use them as ordinary web services. In order to use them as Semantic Services, we must create semantic service descriptions for them.
First of all, we must create the semantic concepts we need for the services, their methods – or rather, their capabilities – as well as their return types. The file \owl\AlarmClock.owl contains the ontology files that describe each of these features. We'll look at piece-by-piece below:

```
<?xml version="1.0"?>
<!DOCTYPE uridef[
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
<!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
<!ENTITY objList
"http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl"> <!ENTITY
wsdl "http://localhost:9080/axis/services/CoffeeMachine"> ]>
<rdf:RDF
xmlns:amigo="http://amigo.gforge.inria.fr/owl/Amigo.owl#"
xmlns:lang="http://amigo.gforge.inria.fr/owl/Amigo-S.owl#"
xmlns:capabilities="http://amigo.gforge.inria.fr/owl/Capabilities.owl#"
xmlns:conselec="http://amigo.gforge.inria.fr/owl/ConsumerElectronics.owl#"
xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
xmlns:objList="http://www.daml.org/services/owl-
s/1.1/generic/ObjectList.owl#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns="http://localhost:9080/AlarmClockOntology.owl#"
xml:base="http://localhost:9080/AlarmClockOntology.owl">
<owl:Ontology rdf:about="">
   <owl:imports rdf:resource="http://amigo.gforge.inria.fr/owl/Amigo.owl"/>
   <owl:imports rdf:resource="http://amigo.gforge.inria.fr/owl/Amigo-
S.owl"/>
   <owl:imports
rdf:resource="http://amigo.gforge.inria.fr/owl/Capabilities.owl"/>
   <owl:imports
rdf:resource="http://amigo.gforge.inria.fr/owl/ConsumerElectronics.owl"/>
</owl:Ontology>
```

So far, this is just standard header material – it links to the other ontology files we require. We have additionally included the consumer electronics ontology, and added a namespace declaration and import statement appropriately.

```
<owl:Class rdf:ID="FmRadio">
   <rdfs:subClassOf
rdf:resource="http://amigo.gforge.inria.fr/owl/ConsumerElectronics.owl#Audi
oRenderDevice" />
</owl:Class>

<owl:Class rdf:ID="CoffeeMachine">
   <rdfs:subClassOf
rdf:resource="http://amigo.gforge.inria.fr/owl/ConsumerElectronics.owl#Cons
umerElectronicsDevice" />
```

```
</owl:Class>

<owl:Class rdf:ID="AlarmClock">
   <rdfs:subClassOf
rdf:resource="http://amigo.gforge.inria.fr/owl/ConsumerElectronics.owl#Cons
umerElectronicsDevice" />
</owl:Class>
```

Here we define the concepts for each of the services and for the application as well. Note that an ontology should describe a shared data model – sharing a common data model between different applications and systems is one of the key strengths of using OWL – so first you should search the ontologies available in the Amigo project to see of the concepts you need already exist.
After searching the ontologies, we discover that a Radio does not exist, however an AudioRenderDevice concept, and while neither a Coffee Machine nor Alarm Clock exist, a ConsumerElectronicsDevice concept does. As such, the FmRadio concept is created to be a subclass of AudioRenderDevice, and CoffeeMachine and AlarmClock to be subclasses of ConsumerElectronicsDevice.

```
<!--Input and ouput semantics. -->
<owl:Class rdf:ID="Void" />
```

Next, we define the input and output semantics we require. In simple example, we only require a "void" type for capability outputs.

```
<!--Capability semantics for FmRadio. -->
<owl:Class rdf:ID="FmRadioSwitchOn">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="FmRadioSwitchOff">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="FmRadioIncreaseVolume">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="FmRadioDecreaseVolume">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="FmRadioFindNextStation">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>
```

Above, we define the semantics for each of the capabilities for the FM radio service.

```
<!--Capability semantics for CoffeeMachine. -->
<owl:Class rdf:ID="CoffeMachineSwitchOn">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="CoffeMachineSwitchOff">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<owl:Class rdf:ID="CoffeMachineBrew">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>
```

```
<owl:Class rdf:ID="CoffeMachineServe">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>

<!--Capability semantics for AlarmClock. -->
<owl:Class rdf:ID="AlarmClockActivate">
   <rdfs:subClassOf rdf:resource=
"http://amigo.gforge.inria.fr/owl/Capabilities.owl#ServiceCapability"/>
</owl:Class>
</rdf:RDF>
```

And finally, we define the semantics for the alarm clock application.
Now that we have the semantic concepts we require, we can create the semantic service descriptions. Below is the semantic description for the coffee machine. It can also be found in full at \owl\CoffeeMachine.owl in the examples download. It's a lot to take in, so we'll look at each piece in turn below:

```
<?xml version="1.0"?>
<!DOCTYPE uridef[ <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
<!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
<!ENTITY objList
"http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl">
<!ENTITY wsdl "http://localhost:9080/axis/services/CoffeeMachine"> ]>

<rdf:RDF xmlns:lang="http://amigo.gforge.inria.fr/owl/Amigo-S.owl#"
xmlns:capabilities="http://amigo.gforge.inria.fr/owl/Capabilities.owl#"
xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
xmlns:objList=
"http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:aconto="http://localhost:9080/AlarmClockOntology.owl#"
xmlns="http://localhost:9080/axis/services/CoffeeMachine.owl#"
xml:base="http://localhost:9080/axis/services/CoffeeMachine.owl"
xmlns:wsdl="http://localhost:9080/axis/services/CoffeeMachine.wsdl">
<owl:Ontology rdf:about="">
   <owl:imports rdf:resource="http://amigo.gforge.inria.fr/owl/Amigo-
S.owl"/>
   <owl:imports
rdf:resource="http://amigo.gforge.inria.fr/owl/Capabilities.owl"/>
   <owl:imports
rdf:resource="http://localhost:9080/AlarmClockOntology.owl"/>
</owl:Ontology>
```

Again, this is standard header. You could use this (as well as the complete file) as a template for your own description. Remember to do as has been done here, and include and namespace declaration and import statement for any application specific ontologies you define.

```
<service:Service rdf:ID="CoffeeMachine">
<lang:ServiceType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
http://localhost:9080/AlarmClockOntology.owl#CoffeeMachine
</lang:ServiceType>
```

Here we introduce the service – we state that a service is being defined and that is has the service type "CoffeeMachine". The id here is used only with the ontology file, the name and display name of the service itself are defined later.

```
<service:presents>
   <capabilities:ServiceProfile rdf:ID="CoffeeMachineProfile">

   <!--Provided capability SwitchOn -->
   <lang:hasProvidedCapability>
      <capabilities:CoffeeMachineSwitchOn rdf:ID=
"CoffeeMachineServiceSwitchOnCapability">
         <lang:hasConversation
rdf:resource="#CoffeeMachineServiceSwitchOnConversation"/>
         <lang:hasOutput rdf:resource=
"#CoffeeMachineServiceSwitchOnOutput"/>
      </capabilities:CoffeeMachineSwitchOn>
   </lang:hasProvidedCapability>

   <!--Provided capability SwitchOff -->
   <lang:hasProvidedCapability>
      <capabilities:CoffeeMachineSwitchOff rdf:ID=
"CoffeeMachineServiceSwitchOffCapability">
         <lang:hasConversation rdf:resource=
"#CoffeeMachineServiceSwitchOffConversation"/>
         <lang:hasOutput rdf:resource=
"#CoffeeMachineServiceSwitchOffOutput"/>
      </capabilities:CoffeeMachineSwitchOff>
   </lang:hasProvidedCapability>

   <!--Provided capability Brew -->
   <lang:hasProvidedCapability>
      <capabilities:CoffeeMachineBrew rdf:ID=
"CoffeeMachineServiceBrewCapability">
         <lang:hasConversation rdf:resource=
"#CoffeeMachineServiceBrewConversation"/>
         <lang:hasOutput rdf:resource="#CoffeeMachineServiceBrewOutput"/>
      </capabilities:CoffeeMachineBrew>
   </lang:hasProvidedCapability>

   <!--Provided capability Serve -->
   <lang:hasProvidedCapability>
      <capabilities:CoffeeMachineServe rdf:ID=
"CoffeeMachineServiceServeCapability">
         <lang:hasConversation rdf:resource=
"#CoffeeMachineServiceServeConversation"/>
         <lang:hasOutput rdf:resource= "#CoffeeMachineServiceServeOutput"/>
      </capabilities:CoffeeMachineServe>
   </lang:hasProvidedCapability>
```

This part begins the definition of the service profile. So far we have declared the each of the provided capabilities the service offers. Each capability definition states the semantics of the capability, e.g. `capabilities:CoffeeMachineSwitchOn`, and declarations of its conversation and its inputs and outputs. The bodies of these are defined later.

```
   <service:presentedBy rdf:resource="#CoffeeMachine"/>
   <profile:textDescription
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">This is a semantic
description of the CoffeeMachine service.
   </profile:textDescription>
   <profile:serviceName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string"> Coffee machine
   </profile:serviceName>
</capabilities:ServiceProfile>
```

This part ends the definition of the service profile and includes a textual description of the service, as well as a display name for the service.

```
</service:presents>
<service:supports rdf:resource="#CoffeeMachineGrounding"/>
</service:Service>
```

Here we add a declaration of the grounding for this service.

```
<!—Conversation of method SwitchOn -->
<process:AtomicProcess rdf:ID="CoffeeMachineServiceSwitchOnConversation">
   <process:hasOutput>
      <process:Output rdf:ID="CoffeeMachineServiceSwitchOnOutput">
         <process:parameterType
Rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
http://localhost:9080/AlarmClockOntology.owl#Void
         </process:parameterType>
      </process:Output>
   </process:hasOutput>
</process:AtomicProcess>

<!—Conversation of method SwitchOff -->
<process:AtomicProcess rdf:ID="CoffeeMachineServiceSwitchOffConversation">
   <process:hasOutput>
      <process:Output rdf:ID="CoffeeMachineServiceSwitchOffOutput">
         <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
http://localhost:9080/AlarmClockOntology.owl#Void
         </process:parameterType>
      </process:Output>
   </process:hasOutput>
</process:AtomicProcess>

<!—Conversation of method Brew -->
<process:AtomicProcess rdf:ID="CoffeeMachineServiceBrewConversation">
   <process:hasOutput>
      <process:Output rdf:ID="CoffeeMachineServiceBrewOutput">
         <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
http://localhost:9080/AlarmClockOntology.owl#Void
         </process:parameterType>
      </process:Output>
   </process:hasOutput>
</process:AtomicProcess>

<!—Conversation of method Serve -->
<process:AtomicProcess rdf:ID="CoffeeMachineServiceServeConversation">
   <process:hasOutput>
      <process:Output rdf:ID="CoffeeMachineServiceServeOutput">
         <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
http://localhost:9080/AlarmClockOntology.owl#Void
         </process:parameterType>
      </process:Output>
   </process:hasOutput>
</process:AtomicProcess>
```

In this part, we define the conversations of each of the capabilities of the service. In this example, all of the provided capabilities are atomic – that is, they are single capability, not a workflow of capabilities – and so here we need only define the capability's output.

```
<!—Grounding (global properties) -->
<grounding:WsdlGrounding rdf:ID="CoffeeMachineGrounding">
   <grounding:hasAtomicProcessGrounding
rdf:resource="#CoffeeMachineServiceSwitchOnGrounding"/>
   <grounding:hasAtomicProcessGrounding
rdf:resource="#CoffeeMachineServiceSwitchOffGrounding"/>
   <grounding:hasAtomicProcessGrounding
rdf:resource="#CoffeeMachineServiceBrewGrounding"/>
   <grounding:hasAtomicProcessGrounding
rdf:resource="#CoffeeMachineServiceServeGrounding"/>
</grounding:WsdlGrounding>
```

```
<!—Grounding for method SwitchOn -->
<grounding:WsdlAtomicProcessGrounding
rdf:ID="CoffeeMachineServiceSwitchOnGrounding">
   <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine.wsdl
   </grounding:wsdlDocument>
   <grounding:owlsProcess
rdf:resource="#CoffeeMachineServiceSwitchOnConversation"/>
   <grounding:wsdlOperation>
      <grounding:WsdlOperationRef>
         <grounding:portType rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine
         </grounding:portType>
         <grounding:operation rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOn
         </grounding:operation>
      </grounding:WsdlOperationRef>
   </grounding:wsdlOperation>
   <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOnRequest
   </grounding:wsdlInputMessage>
   <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOnResponse
   </grounding:wsdlOutputMessage>
</grounding:WsdlAtomicProcessGrounding>

<!—Grounding for method SwitchOff -->
<grounding:WsdlAtomicProcessGrounding
rdf:ID="CoffeeMachineServiceSwitchOffGrounding">
   <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine.wsdl
   </grounding:wsdlDocument>
   <grounding:owlsProcess
rdf:resource="#CoffeeMachineServiceSwitchOffConversation"/>
   <grounding:wsdlOperation>
      <grounding:WsdlOperationRef>
         <grounding:portType rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine
         </grounding:portType>
         <grounding:operation rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOff
         </grounding:operation>
      </grounding:WsdlOperationRef>
   </grounding:wsdlOperation>
   <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOffRequest
   </grounding:wsdlInputMessage>
   <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#switchOffResponse
   </grounding:wsdlOutputMessage>
</grounding:WsdlAtomicProcessGrounding>

<!--Provided capability Brew -->
<grounding:WsdlAtomicProcessGrounding
rdf:ID="CoffeeMachineServiceBrewGrounding">
   <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine.wsdl
   </grounding:wsdlDocument>
   <grounding:owlsProcess
rdf:resource="#CoffeeMachineServiceBrewConversation"/>
   <grounding:wsdlOperation>
      <grounding:WsdlOperationRef>
         <grounding:portType rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine
         </grounding:portType>
         <grounding:operation rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#brew
         </grounding:operation>
      </grounding:WsdlOperationRef>
   </grounding:wsdlOperation>
   <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#brewRequest
```

```
    </grounding:wsdlInputMessage>
    <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#brewResponse
    </grounding:wsdlOutputMessage>
</grounding:WsdlAtomicProcessGrounding>

<!--Provided capability Serve -->
<grounding:WsdlAtomicProcessGrounding
rdf:ID="CoffeeMachineServiceServeGrounding">
    <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine.wsdl
    </grounding:wsdlDocument>
    <grounding:owlsProcess
rdf:resource="#CoffeeMachineServiceServeConversation"/>
    <grounding:wsdlOperation>
       <grounding:WsdlOperationRef>
          <grounding:portType rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine
          </grounding:portType>
          <grounding:operation rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#serve
          </grounding:operation>
       </grounding:WsdlOperationRef>
    </grounding:wsdlOperation>
    <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#serveRequest
    </grounding:wsdlInputMessage>
    <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
http://localhost:9080/axis/services/CoffeeMachine#serveResponse
    </grounding:wsdlOutputMessage>
</grounding:WsdlAtomicProcessGrounding>
</rdf:RDF>
```

This part defines the grounding of the service. Essentially we must map the atomic capabilities to the operations in the WSDL files. At first it may be a little overwhelming, but the structure is very regular, and so simpler than first meets the eye. Each capability has `grounding:WsdlAtomicProcessGrounding` entry that described the input and output message of the associated WSDL operation, as mapping to its port types, operations, and parts. Again, this example could serve as a template for your own services.

2.3 Generating the service description using amigosgen

I'm sure you'll agree that writing all of this description by hand is tedious! But fortunately, a service description template can be automatically generated for a service using the amigosgen tool.
The tool is included in the Semantic Repository bundle, and can be run from the command line as such:

```
java tools.amigosgen CoffeeMachine.wsdl CoffeeMachine.owl
```

This takes the WSDL file as input, and generates the corresponding template in CoffeeMachine.owl. The file produced looks like this:

```
<service:Service rdf:ID="@@ SEMANTIC SERVICE NAME @@">
<lang:ServiceType rdf:datatype= "http://www.w3.org/2001/XMLSchema#string">
@@ SERVICE TYPE SEMANTIC URI @@
</lang:ServiceType>
<service:presents> <capabilities:ServiceProfile rdf:ID= "@@ SEMANTIC
SERVICE NAME @@Profile">
<!--Provided capabilities --> <lang:hasProvidedCapability> <capabilities:@@
switchOn CAPABILITY SEMANTIC @@ rdf:ID=
"CoffeeMachineServiceSwitchOnCapability"> <lang:hasConversation
rdf:resource= "#CoffeeMachineServiceSwitchOnConversation"/> <lang:hasOutput
rdf:resource= "#CoffeeMachineServiceSwitchOnOutput"/> </capabilities:@@
switchOn CAPABILITY SEMANTIC @@> </lang:hasProvidedCapability>
```

Only part of the file is shown, but the rest follows a similar pattern. Almost all of the service description has been created automatically. All that remains to be filled in are the few details in the @@ … @@ tags. Each tag indicates what information must be filled in at that point. Using amigosgen can considerable decrease the time is takes to create semantic service descriptions.
The examples download include the full semantic service description for the FmRadio service too. It can be found in the \owl folder.
Now that we have completed the semantic service descriptions, we can register the semantic services with the Amigo Semantic Service Repository. The code to this is included in the example in the \src\tools\addservices.java. Run this class to add the services. If you wish to programmatically register semantic service within your own code, you can use similar code to that used here.
If at any point you wish to remove the services in the repository, you can use either the clearRepository() method of the Semantic Repository or by simply deleting the \repository folder.
At this point, we have completed the description and registration of the radio and coffee machine semantic services.

3 How to create a task description

In this section we will look at how to create the task for the alarm clock application that will orchestrate the execution of the switch on radio, switch of coffee machine, and brew coffee capabilities.
The Alarm clock application simply has a single capability “activate” that is executed when the alarm goes off. This capability must order the sequence of execution of the other composite semantic services’ capabilities.
To do this, we create a task description. The alarm clock task can be found in full in the \owl folder, and may serve as a basis for your own task descriptions. A task description is simple a service description with a few key differences – a service describes a real, existing service and the capabilities it provides, and their grounding, while a task describe and abstract service that may not yet exist, and the capabilities that are required. As a task is abstract, is does not require a grounding. Services are matched against a task and provide the capabilities the task requires.
Let’s look at the profile for the task:

```
<service:presents>
   <capabilities:ServiceProfile rdf:ID="AlarmClockProfile">

      <!--Required capabilities (high level capability) -->
      <lang:hasRequiredCapability>
         <capabilities:AlarmClockActivate rdf:ID=
"AlarmClockActivateCapability">
            <lang:hasConversation rdf:resource=
"#AlarmClockActivateConversation"/>
            <lang:hasOutput rdf:resource=
"#AlarmClockServiceActivateOutput"/>
         </capabilities:AlarmClockActivate>
      </lang:hasRequiredCapability>

      <!--Required sub-capability SwitchOn for FM Radio -->
      <lang:hasCapability>
         <capabilities:FmRadioSwitchOn rdf:ID="FmRadioSwitchOnCapability">
            <lang:hasConversation rdf:resource=
"#FmRadioSwitchOnConversation"/>
            <lang:hasOutput rdf:resource="#FmRadioServiceSwitchOnOutput"/>
         </capabilities:FmRadioSwitchOn>
      </lang:hasCapability>
```

```
      <!--Required sub-capability SwitchOn for Coffee Machine -->
      <lang:hasCapability>
         <capabilities:CoffeeMachineSwitchOn rdf:ID=
"CoffeeMachineSwitchOnCapability">
            <lang:hasConversation rdf:resource=
"#CoffeeMachineSwitchOnConversation"/>
            <lang:hasOutput rdf:resource="#CoffeeMachineSwitchOnOutput"/>
         </capabilities:CoffeeMachineSwitchOn>
      </lang:hasCapability>

      <!--Required sub-capability Brew  for Coffee Machine -->
      <lang:hasCapability>
         <capabilities:CoffeeMachineBrew rdf:ID=
"CoffeeMachineBrewCapability">
            <lang:hasConversation rdf:resource=
"#CoffeeMachineBrewConversation"/>
            <lang:hasOutput rdf:resource="#CoffeeMachineBrewOutput"/>
         </capabilities:CoffeeMachineBrew>
       </lang:hasCapability>

      <service:presentedBy rdf:resource="#AlarmClock"/>
      <profile:textDescription rdf:datatype=
"http://www.w3.org/2001/XMLSchema#string"> This is a semantic description
of the AlarmClock service.
      </profile:textDescription>
      <profile:serviceName rdf:datatype=
"http://www.w3.org/2001/XMLSchema#string"> Alarm clock
      </profile:serviceName>
   </capabilities:ServiceProfile>
```

This is similar to the service description, but here we describe the required capability as well as sub-capabilities. The activate capability is a composite capability. That is, it executes a workflow of sub-capabilities, rather than a single, atomic capability. In this example, the subcapabilities – written simply `lang:hasCapability` – are executed in order: switch on radio, then switch coffee machine, then brew coffee.
Next we'll look at the conversation section of the task, where this workflow is defined:

```
<!--Conversations -->
<process:CompositeProcess rdf:ID="AlarmClockActivateConversation">
   <process:hasOutput>
      <process:Output rdf:ID="AlarmClockServiceActivateOutput">
         <process:parameterType rdf:datatype=
"http://www.w3.org/2001/XMLSchema#anyURI">
http://localhost:9080/AlarmClockOntology.owl#Void
         </process:parameterType>
      </process:Output>
   </process:hasOutput>
   <process:composedOf>
      <process:Sequence>
         <process:components>
            <process:ControlConstructList>
               <objList:first>
                  <process:Perform rdf:ID="FmRadioSwithOnPerform">
                     <!--The first capability of the workflow -->
                     <process:process
rdf:resource="#FmRadioSwitchOnCapability"/>
                  </process:Perform>
               </objList:first>
               <objList:rest>
                  <process:ControlConstructList>
                     <objList:first>
                        <process:Perform
rdf:ID="CoffeeMachineSwitchOnPerform">
                           <!--The second capability of the workflow -->
                           <process:process rdf:resource=
"#CoffeeMachineSwitchOnCapability"/>
                        </process:Perform>
                     </objList:first>
```

```
                    <objList:rest>
                       <process:ControlConstructList>
                           <objList:first>
                              <process:Perform
rdf:ID="CoffeeMachineBrewPerform">
                                 <!--The last capability -->
                                 <process:process rdf:resource=
"#CoffeeMachineBrewCapability"/>
                              </process:Perform>
                           </objList:first>
                           <!—always end with #nil (LISP mode) -->
                           <objList:rest rdf:resource="&objList;#nil"/>
                       </process:ControlConstructList>
                    </objList:rest>
                 </process:ControlConstructList>
              </objList:rest>
           </process:ControlConstructList>
        </process:components>
     </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>
```

The conversations for the atomic sub-capabilities are defined similar to there service description counter-parts.
We have now completed the alarm clock task description!

4 How to create a semantic service application

In this section, we'll look at how to create the alarm clock semantic service application.
First of all, we need create the Activator for our new application. The example Activator looks like this:

```
package examples.alarmclock;
import org.apache.log4j.Logger;
import org.osgi.framework.BundleContext;
import org.ungoverned.gravity.servicebinder.GenericActivator;

public class AmigoCoreActivator extends GenericActivator {
      public void start(BundleContext bundleContext) throws Exception {
            super.start(bundleContext);
            Logger logger = Logger.getLogger(this.getClass());
            logger.info("Alarm clock activator started!");
      }
}
```

Remember to change the package to suit your own example, and include the appropriate log4j and OSGi bundles in your class path when building. The full code of the Activator can be found in the file src\examples\alarmclock\AmigoCoreActivator.java.

Next we want to create the application itself. The code for is looks like:

```
package examples.alarmclock;
import java.util.Arrays;
import org.apache.log4j.Logger;
import org.osgi.framework.BundleContext;
import org.osgi.framework.BundleException;
import org.ungoverned.gravity.servicebinder.Lifecycle;
import org.ungoverned.gravity.servicebinder.ServiceBinderContext;
import com.francetelecom.amigo.core.AmigoException;
import com.francetelecom.amigo.core.AmigoLdapLookup;
import com.francetelecom.amigo.core.AmigoService;
import fr.inria.icocoa.common.PropertyManager;
import fr.inria.icocoa.common.Tools;
import fr.inria.icocoa.semrep.CapabilityInvoker;
import fr.inria.icocoa.semrep.SemanticRepository;
```

```
public class SemanticRepositoryClientComponent implements Lifecycle {

   private Logger log;
   private AmigoLdapLookup lookup;
   private BundleContext context;

   public SemanticRepositoryClientComponent(ServiceBinderContext sbc) {
      log = Logger.getLogger(this.getClass().getName());
      context = sbc.getBundleContext();
   }

   public void bindLookupService(AmigoLdapLookup lookup) {
      this.lookup = lookup;
   }

   public void unbindLookupService(AmigoLdapLookup lookup) {
      this.lookup = lookup;
   }

   public void activate() {
      log.info("Activating Semantic Repository Client Service...");
      if (lookup != null) {
         AmigoService[] services;
         try {
            services = lookup.lookup("urn:amigo",
"SemanticServiceRepository", 5);
            if (services.length == 0) {
                log.warn( "Unable to find services matching
'SemanticServiceRepository'."); }
            else if (services.length > 1) {
                log.error("Found multiple semantic repositories: "
+ Arrays.toString(services)); }
            else {
               SemanticRepository semrep = (SemanticRepository)
services[0].getSpecificStub(SemanticRepository.class);
               try {
                  String taskFileUri = Tools.getURIStringFromFilePath(
PropertyManager.LOCAL_DIRECTORY + "/owl/AlarmClock.owl");

                  // Find services for task
                  log.info("Discovering services...");
                  String[] matchingServices =
semrep.getServicesForTask(taskFileUri);

                  if (matchingServices.length > 0) {
                     String serviceWsdlUri =
semrep.getWsdlForService(matchingServices[0]);
                     log.info("Activating alarm clock (invoker)...");
                     CapabilityInvoker invoker = new
CapabilityInvoker(serviceWsdlUri);
                     invoker.invoke("AlarmClockActivate");
                     log.info("Activating alarm clock (stub)...");
                     AlarmClock alarmClock = new
AlarmClockStub(serviceWsdlUri);
                     alarmClock.activate();
                  }else {
                     log.info("No services were found matching " +
taskFileUri + ".");
                  }
               } catch (AmigoException e) {
                  log.warn( "Lookup failed for services of type
'SemanticServiceRepository'."); }
      } else {
         log.warn( "No lookup service found for Semantic Repository Client
bundle."); }

         new Thread() {
            public void run() {
               try {
                  context.getBundle().stop();
               } catch (BundleException e) {
                   log.warn("Unable to stop bundle.", e); } } }.start(); }
```

```
    public void deactivate() {
        log.info("Stopping the Semantic Repository Client bundle.");
    }
}
```

Again, this can be used as a template. Much of this code will be the same in your client service. In fact, in most case, you'll only need to fill in the details of the "activate" method!
If you are interested in the details of the other methods here, such as the bind and unbind methods, or the binder context objects in the constructor, please refer to the Amigo OSGi Programming Framework documentation, available at https://gforge.inria.fr/frs/?group_id=160.
The line after `// Find services for task` is where we call upon the semantic service repository to compose available service to meet the requirements of the alarm clock task.
This will return a list of WSDL files of services that meet the task's requirements.
Now that you have a matching service, we want to invoke the alarm clock activate capability. To this, we have a choice – use code like the following:

```
log.info("Activating alarm clock (invoker)...");
CapabilityInvoker invoker = new CapabilityInvoker(serviceWsdlUri);
invoker.invoke("AlarmClockActivate");
```

to use the dynamic capability invoker. This allows any of capabilities of your task description to be invoked, even if the task description was generated dynamically at runtime!
If you would prefer to use a Java interface to invoke the capabilities, you must first have designed and built a suitable Java interface and stub class. These are straightforward to write
– simply include each of the task's capabilities in the interface, and then write a stub that extends the AbstractStub class, and use doInvoke to invoke the task capability. As an example, an interface and stub for the alarm clock application are provided below. The full source code for both of these is available in \src\examples\alarmclock.
The interface looks like:

```
package examples.alarmclock;
public interface AlarmClock {
      void activate();
}
```

And the stub looks like this:

```
package examples.alarmclock;
import fr.inria.icocoa.semrep.AbstractStub;
import fr.inria.icocoa.semrep.excp.InvocationException;

public class AlarmClockStub extends AbstractStub implements AlarmClock {

   public AlarmClockStub(String wsdl) { super(wsdl); }

   public void activate() {
      try { doInvoke("AlarmClockActivate"); }
      catch (InvocationException e) {
         System.err.println("An invocation error occurred: " + e); } }
}
```

Then we can invoke the capabilities of the services returned by the Semantic Repository as so:

```
log.info("Activating alarm clock (stub)...");
AlarmClock alarmClock = new AlarmClockStub(serviceWsdlUri);
alarmClock.activate();
```

The repository can be configured to match services to tasks in different ways. By default, the repository will matches only identical concepts. That is, the capability semantic for a service's provided capability must be identical to the capability semantic of the task's required capability, for there to be a match between them. This is termed "syntactic" matching. The repository can be set to use this type of matching as follows:

```
semrep.setMatcher("syntactic");
```

In addition, the repository offers two other looser modes of matching. First, is “exact” matching – this will match concepts that are identical, but also concepts that are either explicitly equivalent (tagged with the `owl:equivalentClass` relation), or implicitly equivalent (it has similar properties of another class tagged with the `owlowl:equivalentProperty` relation). The repository can be set to use this type of matching as follows:

```
semrep.setMatcher("exact");
```

Second, is “weak” matching – this will match all concepts that exact matching does, but also this will match concepts that are identical, but also concepts that are within a stated radius of the to be matched concept. For example, if we have a radius of 1, the concept would be matched with either an immediate super-class or an immediate sub-class. The repository can be set to use this type of matching as follows:

```
semrep.setMatcher("weak 1");
```

Take a look at the Semantic Repository interface and try out the over methods that are available. For example, you can look up semantic services based on type, or discover the Quality of Service properties of a semantic service.
We have completed the implementation of the alarm clock application.

5 How to deploy a semantic service application

To deploy the alarm clock application, we must first package it up as an service bundle. To do this, we must first create a manifest file and a metadata file for the service. The manifest for the example looks like:

```
ManifestVersion: 1.0
BundleDescription: An alarm clock application based on Semantic Services.
BundleName: AlarmClock
BundleActivator: examples.alarmclock.AmigoCoreActivator
BundleClassPath: .
CreatedBy: Graham Thomson
BundleVendor: INRIARocq
ImportPackage: com.francetelecom.amigo.core, org.ungoverned.gravity.servicebinder,
org.apache.log4j, fr.inria.icocoa.semrep, fr.inria.icocoa.semrep.excp,
fr.inria.icocoa.common
BundleCopyright: LGPL
Metadatalocation: metadata.xml
```

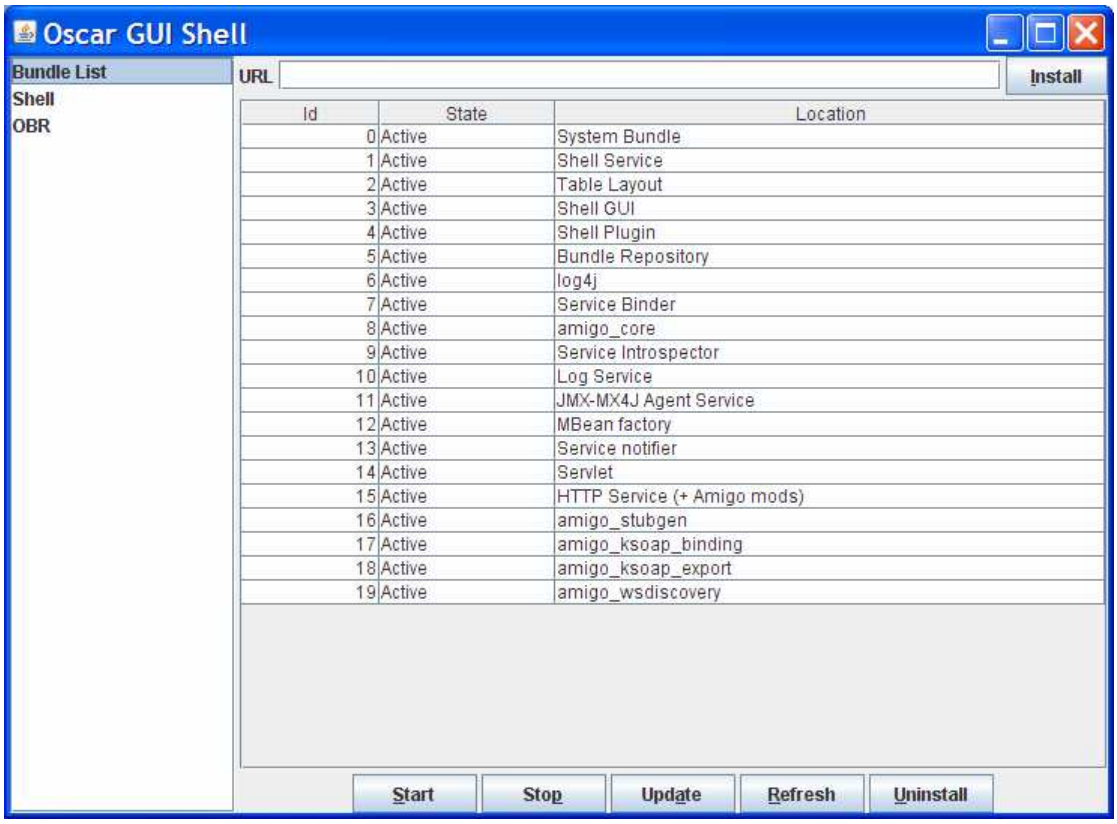
And the metadata looks like:

```
<?xml version="1.0" encoding="UTF8"?>
  <bundle>
    <component class="examples.alarmclock.SemanticRepositoryClientComponent">
      <requires service="com.francetelecom.amigo.core.AmigoLdapLookup"
filter="" cardinality="1..1" policy="dynamic" bindmethod="bindLookupService" unbind-
method="unbindLookupService"
/> </component> </bundle>
```

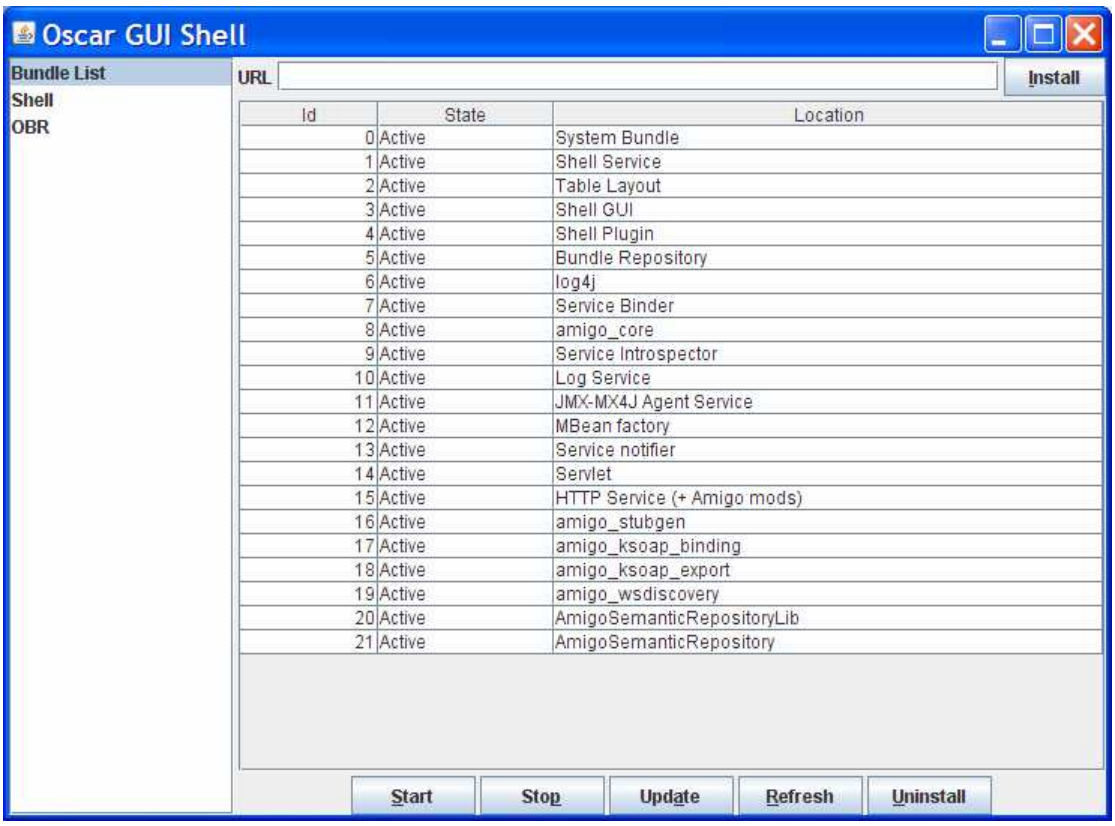
The full source of both of these files can be found in \src\examples\alarmclock. Again, these files can be used as a template, and further details about these files can be found the Amigo OSGi Programming Framework documentation, available at http://amigo.gforge.inria.fr/home/index.html.
Next we must bundle these files up together with the class files of the service. This can be done from the command line, by running the make_osgi_bundle.bat in the \scripts folder:

```
cd scripts make_osgi_bundle.bat
```

The resulting jar file in the \osgi folder is the service bundle for you new service.
Next start up oscar by running oscar.bat in the examples download root directory. The screenshot below shows the minimal set of packages that must be installed to run the Semantic Repository. Any missing bundles can be added from the OBR tab.



Install the Semantic Repository bundles semantic-repository-lib-1.0.0.jar and semanticrepository-service-1.0.0.jar by entering their URL in the bar at the top and clicking Install, and then start them. Oscar should now look like this:



You are now ready to install and run alarm clock application! Similarly, enter the URL or your service in to the bar, click Install, then Start, and watch the console for your service's output.

You can now use the Semantic Service Repository! Go create your world!