JADE WEB SERVICES INTEGRATION GATEWAY (WSIG) GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

Version: 3.1

Last update: 05-Feb-2015.

Authors: JADE Board

TABLE OF CONTENTS

1 INTRODUCTION 3
1.1 Goal 3
1.2 Compliance to standards 3
1.3 Compatibility 3
1.4 Requirements 3

2 ARCHITECTURE 5

3 EXPOSING AGENT SERVICES AS WEB SERVICES 7
3.1 DF Service-Description to WSDL conversion overview 7
3.2 WSDL generation details 8
3.3 Service name prefix 12
3.4 Customizing the WSDL by means of an ontology mapper 12
3.5 Current limitations 17

4 INSTALLATION 19
4.1 Deployment 20
4.2 Configuration 21
4.3 WebSphere Application Server configuration 23
4.4 Security 23
4.4.1 HTTP authentication 23
4.4.2 SSL certificate 24
4.4.3 WS-Security 25

5 ADMINISTRATION GUI 26

6 ADMINISTRATION API
ERRORE. IL SEGNALIBRO NON È DEFINITO.
6.1 Resources: 29
6.1.1 admin/platform 29
6.1.2 admin/platform/{status} 29
6.1.3 admin/configuration 29
6.1.4 admin/services 30
6.1.5 admin/services/{serviceName} 30
6.1.6 admin/services/{serviceName}/wsdl 30
6.2 XML and JSON code examples 31
6.2.1 Configuration 31
6.2.2 Services 32
6.2.3 Service 32

7 APPENDIX I. DESCRIPTION OF THE EXAMPLES 34

1 INTRODUCTION

This document describes the Web Service Integration Gateway (WSIG) add-on that provides support for invocation of JADE agent services from Web service clients. More in details section 2 outlines the architecture of WSIG and shows the mapping between a DF service-description and the corresponding WSDL, section 3 presents useful indications about deploying and configuring WSIG and section 4 gives a step-by-step guidance about how to expose agent services as Web Services by means of WSIG.

The reader is assumed to be familiar with both JADE and the Web Service technology. For those new to JADE we strongly recommend first reading the JADE Administrators Guide and Programmers Guide or the JADE Programming Tutorial, available on the JADE web site (http://jade.tilab.com).

All bugs, issues, contributions and new feature requirements should be posted to the main JADE bug reporting system, and to the standard JADE mailing lists.

Version 2.0 of the WSIG add-on was developed by the JADE Board and is only guaranteed to work with JADE release 3.5 or later.

1.1 Goal

The services web, also known as WEB SERVICES, are becoming one of the most important topics in the panorama of software development and a sort of de facto standard for interconnecting different applications. The objective of WSIG is to expose services provided by agents and published in the JADE DF as web services with no or minimal additional effort, though giving developers enough flexibility to meet specific requirements then may have.

The process involves the generation of a suitable WSDL for each service-description registered with the DF and possibly the publication of the exposed services in a UDDI registry.

1.2 Compliance to standards

The WSDL service description language defines different binding styles (rpc and document) and uses (encoded and literal). WSIG supports the most commonly adopted combinations:

- **rpc/encoded** according to W3C standards (http://www.w3c.org)
- **document/literal wrapped** in compliance with the WS-I basic profile specification (http://www.ws-i.org).

Section 4.2 describes which one to use.

All Date fields are encoded according to the ISO-8601 format.

1.3 Compatibility

WSIG 2.0 has been successfully tested with web services clients developed using AXIS 1.1, 1.2 (http://ws.apache.org/axis) and 2.0 (http://ws.apache.org/axis2) and CXF 2.0 (http://cxf.apache.org) .

1.4 Requirements

The WSIG add-on requires Java JRE v5.0 (http://java.sun.com/javase/), JADE v3.5 or later and a Servlet container such as Jakarta Tomcat (http://jakarta.apache.org/tomcat/). Furthermore, if the automatic UDDI publication feature is turned on, a suitable UDDI registry must be available.

The WSIG makes use of the following third party libraries **already included in the WSIG distribution**:

- Apache Axis v1.4 (http://ws.apache.org/axis/)
- Apache Commons (http://jakarta.apache.org/commons/)
- UDDI4J v2.0.5 (http://uddi4j.sourceforge.net/)
- WSDL4J v1.6.2 (http://sourceforge.net/projects/wsdl4j)
- Eclipse EMF v2.3.0 (http://www.eclipse.org/emf/)

2 ARCHITECTURE

The WSIG add-on supports the standard Web services stack, consisting of WSDL for service descriptions, SOAP message transport and a UDDI repository for publishing Web services using tModels. As depicted in Figure 1 WSIG is a web application composed of two main elements:

- WSIG Servlet
- WSIG Agent

The WSIG Servlet is the front-end towards the internet world and is responsible for

- Serving incoming HTTP/SOAP requests
- Extracting the SOAP message
- Preparing the corresponding agent action and passing it to the WSIG Agent

Moreover once the action has been served

- Converting the action result into a SOAP message
- Preparing the HTTP/SOAP response to be sent back to the client

The WSIG Agent is the gateway between the Web and the Agent worlds and is responsible for

- Forwarding agent actions received from the WSIG Servlet to the agents actually able to serve them and getting back responses.
- Subscribing to the JADE DF to receive notifications about agent registrations/deregistrations.
- Creating the WSDL corresponding to each agent service registered with the DF and publish the service in a UDDI registry if needed.

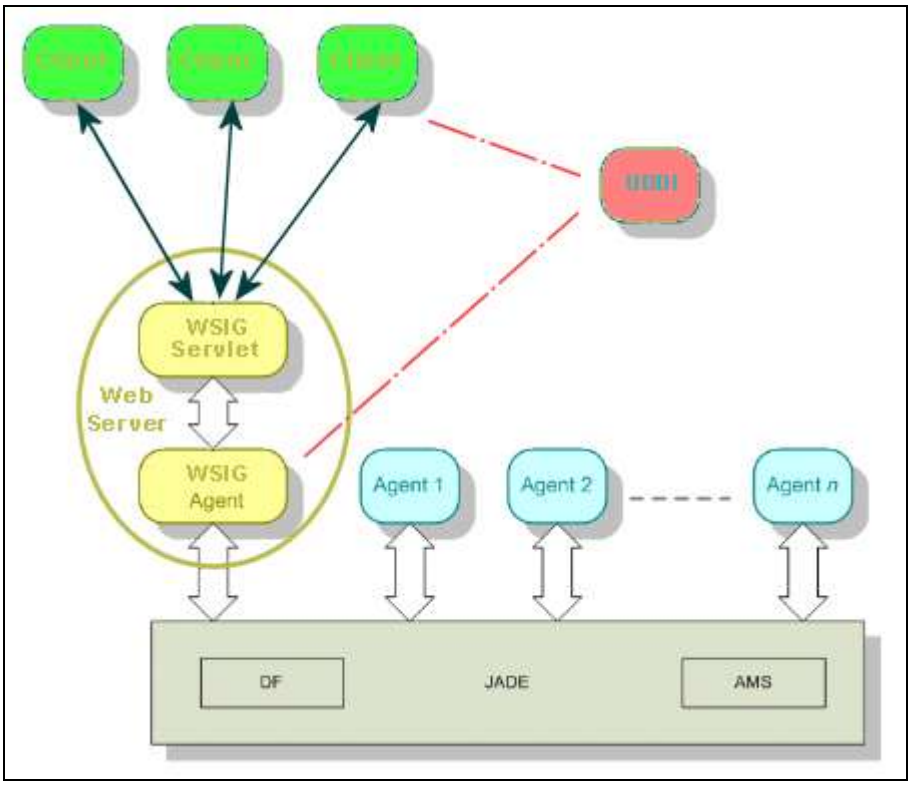

Figure 1. WSIG Architecture

Two main processes are continuously active in the WSIG web application:

- The process responsible for intercepting DF registrations/deregistrations and converting them into suitable WSDLs. As mentioned, this process is completely carried out by the WSIG Agent and is described in section 3.1.

- The process responsible for serving incoming web service requests and triggering the corresponding agent actions. This process is carried out jointly by the WSIG Servlet (performing the necessary translations) and the WSIG Agent (forwarding requests to agents able to serve them).

3 EXPOSING AGENT SERVICES AS WEB SERVICES

JADE agents publish their services in the DF (Directory Facilitator) providing a structure called DF-Agent-Description and defined by the FIPA specification (www.fipa.org). A DF-Agent-Description includes one or more Service-Description each one actually describing a service provided by the registering agent. A Service-Description typically specifies, among others, one or more ontologies that must be known in order to access the published service. The actions the registering agent is actually able to perform are those defined in the specified ontologies.

In order to expose an agent service as a web service it is sufficient to set the `wsig` property to `true` in the properties of the Service-Description at DF registration time as below

```
……
ServiceDescription sd = new ServiceDescription();
……
sd.addProperties(new Property("wsig", "true"));
……
```

3.1 DF Service-Description to WSDL conversion overview

Each Service-Description including the `wsig` property set to `true` will be mapped to a WSDL. All actions defined in the ontologies specified in the Service-Description will be mapped to WSDL operations as depicted in Figure 2.

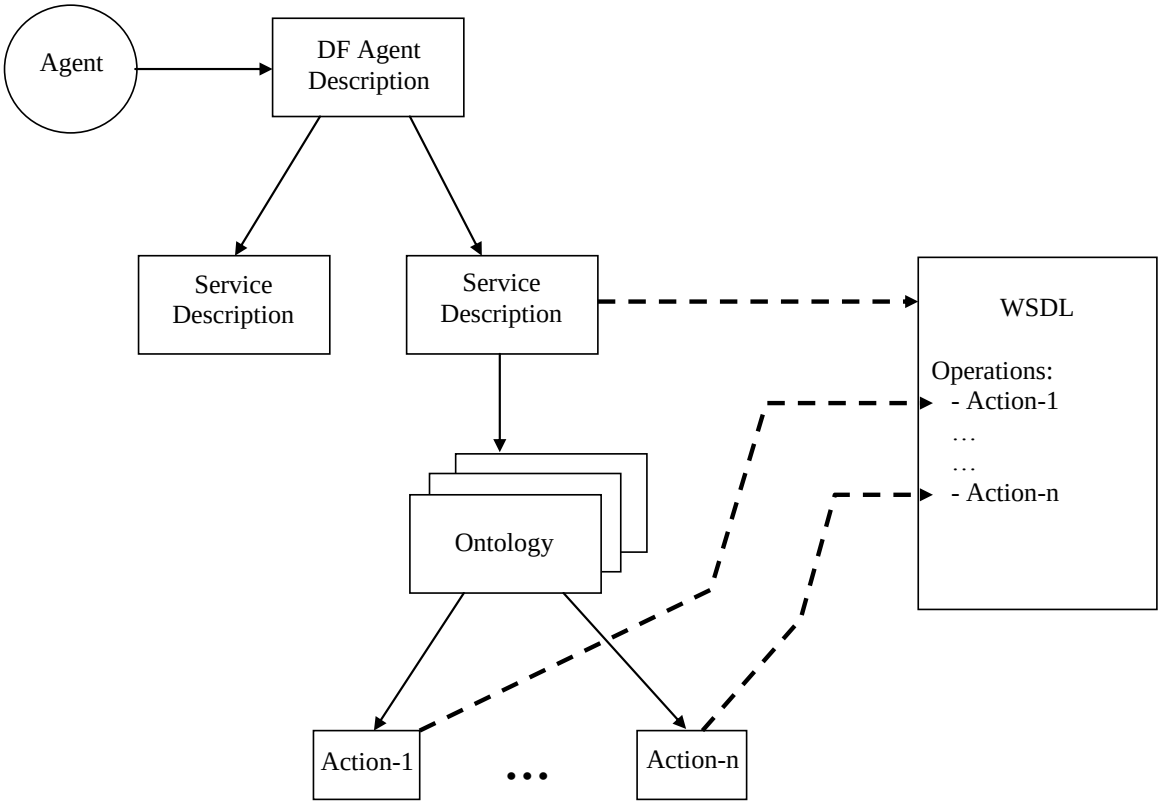

Figure 2. Mapping between DF Service-Description and WSDL

Whether or not a WSDL is also published in a UDDI registry depends on the WSIG configuration as will be presented in 4.2.

Similarly when an agent deregisters from the DF, all its services (if any) are automatically removed from the WSIG.

3.2 WSDL generation details

This section describes in details the default rules that are followed by the WSIG Agent to generate the WSDL corresponding to a Service-Description including the `wsig` property set to `true`.

- WSDL TNS (Target Name Space) = value of the `name` slot of the Service-Description
- For each `AgentActionSchema` defined in the ontology referenced by the ontology slot of the Service-Description an operation is added in the WSDL with
    - operation name = agent action schema name
    - operation parameters (name and type[1]) = agent action schema slots (name and type)
    - operation result [2]= agent action schema result

[1] If the document/literal style is used the type is ignored

[2] Since version 3.5 of JADE, it is possible to associate a result to an agent action by means of the `setResult(TermSchema ts)` and `setResult(TermSchema ts, int cardMin, int cardMax)` methods of the `AgentActionSchema` class.

- Agent action result/slots whose schema is a PrimitiveSchema are mapped to basic types defined in the standard schema http://www.w3.org/2000/10/XMLSchema
- Agent action result/slots whose schema is a ConceptSchema are mapped to complex types defined within the WSDL itself
- Agent action result/slots whose schema is an AggregateSchema are mapped to complex types (labeled as `ArrayOfTtt` where `ttt` is the type of the aggregate elements) defined within the WSDL itself
- The same mappings apply recursively in case of complex result/slots.
- If an agent action slot is optional, the related operation parameter is marked with `minOccurs = "0"`.
- If the `wsdl.writeEnable` configuration property is set to `true` (see 4.2) → WSDL file = value of the `name` slot of the Service-Description plus the `.wsdl` extension

The WSDL document is built using network services definition standard elements:

- **Types** – a container for data type definitions using some type system (such as XSD).
- **Message** – typed definition of the data being communicated.
- **Operation** – description of an action supported by the service.
- **Port Type** – set of operations supported by one or more endpoints.
- **Binding** – a concrete protocol and data format specification for a particular port type.
- **Port** – a single endpoint defined as a combination of a binding and a network address.
- **Service** – a collection of related endpoints.

The WSDL service description language defines different binding styles (**rpc** and **document**) and uses (**encoded** and **literal**). WSIG only supports the most commonly adopted combinations i.e. **rpc/encoded** and **document/literal wrapped**. Section 4.2 describes which one to use. As an example the WSDL corresponding to the sumcomplex (sum of 2 complex numbers) operation of the MathService included among the WSIG examples is reported below for both supported styles.

WSDL for the document/literal wrapped style

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MathFunctions" targetNamespace="urn:MathFunctions"
xmlns:impl="urn:MathFunctions" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema targetNamespace="urn:MathFunctions" xmlns:impl="urn:MathFunctions"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><xsd:annotation/><xsd:element
name="sumcomplex"><xsd:complexType><xsd:sequence><xsd:element name="firstComplexElement"
type="impl:complex"/><xsd:element name="secondComplexElement"
type="impl:complex"/></xsd:sequence></xsd:complexType></xsd:element><xsd:complexType
name="complex"><xsd:sequence><xsd:element name="real" type="xsd:float"/><xsd:element
minOccurs="0" name="immaginary"
type="xsd:float"/></xsd:sequence></xsd:complexType><xsd:element
name="sumcomplexResponse"><xsd:complexType><xsd:sequence><xsd:element
name="sumcomplexReturn"
type="impl:complex"/></xsd:sequence></xsd:complexType></xsd:element></xsd:schema>
  </wsdl:types>
  <wsdl:message name="sumcomplexRequest">
    <wsdl:part name="parameters" element="impl:sumcomplex">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="sumcomplexResponse">
    <wsdl:part name="parameters" element="impl:sumcomplexResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="MathFunctionsPort">
    <wsdl:operation name="sumcomplex">
      <wsdl:input message="impl:sumcomplexRequest">
    </wsdl:input>
      <wsdl:output message="impl:sumcomplexResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="MathFunctionsBinding" type="impl:MathFunctionsPort">
    <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="sumcomplex">
      <wsdlsoap:operation soapAction="urn:MathFunctionsAction"/>
      <wsdl:input>
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="MathFunctionsService">
    <wsdl:port name="MathFunctionsPort" binding="impl:MathFunctionsBinding">
      <wsdlsoap:address location="http://localhost:8080/wsig/ws/MathFunctions"/>
    </wsdl:port>
```

```
      </wsdl:service>
    </wsdl:definitions>
```

WSDL for the rpc/encoded style

```
    <?xml version="1.0" encoding="UTF-8"?>
    <wsdl:definitions name="MathFunctions" targetNamespace="urn:MathFunctions"
xmlns:impl="urn:MathFunctions" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <wsdl:types>
        <xsd:schema targetNamespace="urn:MathFunctions" xmlns:impl="urn:MathFunctions"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><xsd:annotation/><xsd:complexType
name="complex"><xsd:sequence><xsd:element name="real" type="xsd:float"/><xsd:element
minOccurs="0" name="immaginary"
type="xsd:float"/></xsd:sequence></xsd:complexType></xsd:schema>
      </wsdl:types>
      <wsdl:message name="sumcomplexRequest">
        <wsdl:part name="firstComplexElement" type="impl:complex">
        </wsdl:part>
        <wsdl:part name="secondComplexElement" type="impl:complex">
        </wsdl:part>
      </wsdl:message>
      <wsdl:message name="sumcomplexResponse">
        <wsdl:part name="sumcomplexReturn" type="impl:complex">
        </wsdl:part>
      </wsdl:message>
      <wsdl:portType name="MathFunctionsPort">
        <wsdl:operation name="sumcomplex">
          <wsdl:input message="impl:sumcomplexRequest">
        </wsdl:input>
          <wsdl:output message="impl:sumcomplexResponse">
        </wsdl:output>
        </wsdl:operation>
      </wsdl:portType>
      <wsdl:binding name="MathFunctionsBinding" type="impl:MathFunctionsPort">
        <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="sumcomplex">
          <wsdlsoap:operation soapAction="urn:MathFunctionsAction"/>
          <wsdl:input>
            <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
          </wsdl:input>
          <wsdl:output>
            <wsdlsoap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
          </wsdl:output>
        </wsdl:operation>
      </wsdl:binding>
      <wsdl:service name="MathFunctionsService">
        <wsdl:port name="MathFunctionsPort" binding="impl:MathFunctionsBinding">
          <wsdlsoap:address location="http://localhost:8080/wsig/ws/MathFunctions"/>
        </wsdl:port>
      </wsdl:service>
```

```
    </wsdl:definitions>
```

3.3 Service name prefix

Considering the mapping described in the previous section it is clear that if two agents register two Service-Descriptions with the same name there is a conflict due to the fact that two WSDL definitions have the same name. To avoid this conflict it is possible to use the `wsig-prefix` Service-Description property (as exemplified in the code snippet below) to specify a label that will be used to prefix both the Target Name Space and the WSDL file name (if generated).

```
……
ServiceDescription sd = new ServiceDescription();
……
sd.addProperties(new Property("wsig", "true"));
sd.addProperties(new Property("wsig-prefix", "prefix"));
……
```

3.4 Customizing the WSDL by means of an ontology mapper

In many cases it is highly desirable to customize the exposed web services operations. For instance in case of an action with several optional slots that are only used in particular conditions, one may wish to expose an operation with only those parameters that are meaningful when the operation is invoked by an external Web Service client. Similarly in many cases an agent may be able to support (or may whish to expose as web service operations) only a subset of the actions included in an ontology. In other cases, it would be appreciated the possibility of personalizing the response of the web service operation adding slots (we recall that in Jade ontologies only actions with just one output parameter are allowed) or modifying parameters format.

To meet these customization requirements the WSIG add-on provides a flexible mechanism based on so called "ontology mapper" classes. An ontology mapper is a class providing:

1) a set of methods with the following signature

```
public <action-class> to<action-name>(<parameters>) {
  …
}
```

used to customize the SOAP request associated to the specific operation.

2) a set of inner classes with the following structure

```
@ResultConverter({
@ApplyTo(action="<action-name>",operation="<op-name>")
…
})
public class Xxx {
  …
}
```

used to customize the SOAP response associated to the specific operation.

3) a set of inner classes with the following structure

```
@FaultConverter({
@ApplyTo(action="<action-name>",operation="<op-name>")
…
})
public class Yyy {
  …
}
```

used to customize the SOAP FAULT response associated to the specific operation.

When an ontology mapper is specified in a DF Service-Description, for each action `my-action` defined in the ontology referenced by the Service-Description and associated to an AgentAction class `MyActionClass`, the WSIG agent searches for a method with signature

`public MyActionClass toMyAction(…)`

in the mapper class. If one such method is found, the slots of the `my-action` action are ignored and the parameters of the `toMyAction()` method are used (names and types) as parameters of the `my-action` web service operation. Furthermore if the `toMyAction(...)` method is annotated with the `@OperationName` annotation (available in the `wsigAnnotations.jar` package), the value of the `name` attribute of that annotation will be used (instead of `my-action`) to name the exposed web service operation.

It is possible to use the annotations @Slot e @AggregateSlot (availables in `jade.jar` package) to specify if a parameter is mandatory and, for the aggregates, the cardinality and the type of the contained element.

```
// <soapenv:Body>
//      <Add>
//        <first>?</first>
//        <second>?</second>
//        <!--Optional-->
//        <third>?</third>
//     </Add>
// </soapenv:Body>

@OperationName(name="Add")
public Sum toSum(   @Slot(mandatory=true) float first,
                    @Slot(mandatory=true) float second,
                    @Slot(mandatory=false) Float third){
       ……
}
……
//  <soapenv:Body>
//      <Multiplication>
//           <!--1 to 5 repetitions-->
//           <float>?</float>
//      </Multiplication>
//  </soapenv:Body>

public Multiplication toMultiplication(
       @AggregateSlot(cardMin=1, cardMax=5, type=Float.class) List numbers) {
       ……
}
```

Several methods `to<action-name>` can be declared, provided that the name of the web-service operation (declared in the @OperationName) is unique.

When the web service operation is invoked, the WSIG agent extracts the operation parameters from the SOAP message and uses them to call the `toMyAction()` method of the ontology mapper. This method is responsible for building the `MyActionClass` action object that will be forwarded to the target agent.

By means of the `@SuppressOperation` annotation (available in the `wsigAnnotations.jar` package) it is then possible not to expose as web service operation a given ontology action, as exemplified in the code snippet below.

```
@SuppressOperation
public MyActionClass toMyAction() {
  return null;
}
```

After the processing of the action performed by the target agent, the WSIG checks if the mapper contains a ResultConverter related to the specific operation and, in this case, uses it to prepare the SOAP response message.

The Result Converter is an inner-class of the mapper annotated as it follows:

```
@ResultConverter({
@ApplyTo(action="<action-name>",operation="<op-name-1>"),
@ApplyTo(action="<action-name>",operation="<op-name-2>")
…
})
```

where the multiple annotation `@ApplyTo` (available in the `wsigAnnotations.jar` package) specifies the association between a specific web-service operation `<op-name-xxx>` (handled by the ontological action <action-name>) and the ResultConverter. The same ResultConverter can be applied to several operations and/or actions.

If the operation attribute is not specified, the ResultConverter is applied to the all web-service operations associated to the ontological action `<action-name>`.

Any name can be used for an inner class representing a ResultConverter (even if, the ResultConverter suffix is recommended) and its structure is the one typical of the java beans. The constructor, instead, must have one of the following signature:

```
public Xxx(<ontology-result-type> yyy) {
      …
}
```

or

```
public Xxx(<ontology-result-type> yyy, ACLMessage message) {
      …
}
```

where the <ontology-result-type> is the java type returned by the execution of the ontological action. Such java type can be defined by means of setResult(...), for traditional ontologies, or by means of the @Result(...) annotation for bean-ontologies. (Refer to the Jade documentation for details).

In the inner-class all methods, having the following signature:

```
public <java-type> get<element-name>(){…}
public boolean is<element-name>(){…}
```

contribute to the construction of the response message. Each method creates, in the SOAP message, an element, named <element-name>, whose value is the one returned by the evaluation of the method itself.

```
//  <soapenv:Body>
//      <AddResponse>
//           <result>4</result>
//           <squareRoot>2</squareRoot>
//           <power>16</power>
//      </AddResponse>
//   </soapenv:Body>

@ResultConverter({
       @ApplyTo(action="add")
})
public class AddResultConverter {
       private double result;

       public AddResultConverter(double ontologyResult) {
              this.result = ontologyResult;
       }

       public double getResult() {
              return result;
       }

       public double getSquareRoot() {
              return Math.sqrt(result);
       }

       public double getPower() {
              return Math.pow(result, 2.0);
       }
}
```

The ResultConverter methods can be annotated using the @Slot, @AggregateSlot and @SuppressSlot annotations for further customizing the output message. The meanings of the @Slot, @AggregateSlot annotations have been described above; the @SuppressSlot annotation, instead, allows to suppress a get/is method in the construction of the output message.

When executor agent responds FAILURE, REFUSE or NOT_UNDERSTOOD the WSIG checks if the mapper contains a FaultConverter related to the specific operation and, in this case, uses it to prepare the SOAP FAULT response message.

The Fault Converter is an inner-class of the mapper annotated as it follows:

```
@FaultConverter({
```

```
      @ApplyTo(action="<action-name>",operation="<op-name-1>"),
      @ApplyTo(action="<action-name>",operation="<op-name-2>")
      …
      })
```

The significance of the annotations is the same as the ResultConverter. The same FaultConverter can be applied to several operations and/or actions.

If the operation attribute is not specified, the FaultConverter is applied to the all web-service operations associated to the ontological action `<action-name>`.

Any name can be used for an inner class representing a FaultConverter (even if, the FaultConverter suffix is recommended) but its internal structure must be the following:

```
      public Xxx(ACLMessage fauilureMessage) {
            …
      }

      public String getFaultCode() {
            return ....;
      }

      public String getFaultString() {
            return ....;
      }

      public String getFaultActor() {
            return ....;
      }
```

```
//  <soapenv:Body>
//      <soapenv:Fault>
//           <faultCode>Server</faultCode>
//           <faultString>My message</faultString>
//           <faultActor>My actor</faultActor>
//      </soapenv:Fault>
//  </soapenv:Body>

@FaultConverter({
      @ApplyTo(action="add")
})
public class AddFaultConverter {

      public AddFaultConverter(ACLMessage message) {
      }

      public String getFaultCode() {
            return SOAPException.FAULT_CODE_SERVER;
      }

      public String getFaultString() {
            return "My message";
```

```
        }

        public String getFaultActor() {
                return "My actor";
        }
}
```

Summarizing, the role of an ontology mapper is twofold;

- At service registration time, to define the I/O parameters (and possibly the name) of the web service operations to be exposed.
- At service invocation time, to create the objects representing the actions the target agent is actually requested to perform and to convert the result returned by the target agent in a customized structure.

It should be noted that parameters of the `toXXX()` methods and the return types of the methods metodi getXXX() or isXXX(), defined by the ResultConverter in an ontology mapper class, must be either primitive types, complex types or array of the above types.

WSIG annotations (`@OperationName`, `@SuppressOperation`, `@ResultConverter` e `@ApplyTo`) are available including in the project classpath the library wsigAnnotations.jar. When necessary this library can be found in the folder `add-ons/wsig/utils`. (See chapter 4 - INSTALLATION for details).

The other annotation (`@Slot`, `@AggregateSlot` e `@SuppressSlot`) are available in the jade.jar library.

In order to specify an ontology mapper the `wsig-mapper` service description property must be used as exemplified in the code snippet below.

```
……
ServiceDescription sd = new ServiceDescription();
……
sd.addProperties(new Property("wsig", "true"));
sd.addProperties(new Property("wsig-
mapper","com.tilab.wsig.examples.MathOntologyMapper"));
……
```

The `MathOntologyMapper` class included among the examples packaged with the WSIG distribution provides an example of ontology mapper class.

3.5 Current limitations

Version 2.0 of the WSIG add-on has the following known limitations.

- The JADE main container must be already up and running when the WSIG web application is started.
- Even if a Service-Description can reference several ontologies, WSIG is able to handle only one.

- In rpc/encoded style the MULTIREF soap request is not supported. Note that AXIS 1.x use this formalism as default → In order to create AXIS 1.x based clients able to access Web services exposed by WSIG it is necessary to disable the MULTIREF option (see the AXIS documentation for details).

4 INSTALLATION

The Web Service Integration Gateway is a JADE add-on and, as such, requires JADE 3.5 or later to be already installed. Furthermore, being WSIG a web application, it is necessary to have a Servlet Container such as Apache Tomcat, properly installed and configured (the details on servlet container installation and configuration are out of the scope of this document).

In order to install the WSIG add-on the following steps must be performed.

Download the WSIG distribution file from the add-ons area of the JADE web site (http://jade.tilab.com).

Unzip the WSIG distribution file in the JADE home directory. You should end-up with a directory structure like that depicted in Figure 3.

Figure 3. WSIG directory structure

A brief description of the content of each directory is presented hereafter.

- `bin`: contains a number of startup scripts both in `bat` (Windows) and `sh` (Linux/Unix) form.
- `context`: contains the WSIG web application context file that can be used for customized installations
- `examples`: contains the source files of the WSIG example (`MathAgent`)
- `lib`: contains third party libraries used by the WSIG add-on (including the related licenses) mentioned in 1.4

- `src`: contains the WSIG source code
- `utils`: contains the WSIG annotations library (wsigAnnotation.jar)
- `webapp`: contains the WSIG webapp with nothing custom configurations (wsig.war)
- `webModule`: contains the structure of the WSIG web application as it will appear in the `wsig.war` file.
- `build.xml/build.properties`: this is the ANT build file by means of which it is possible to compile, and deploy the WSIG web application as will be described in the following section

4.1 Deployment

As mentioned WSIG is a web application that must be deployed in a servlet container such as Apache Tomcat and will be executed within the JVM of the servlet container. As a consequence **both the WSIG own classes, the third party libraries used by WSIG, and all ontologies and mapper classes the WSIG will have to deal with must be included in the WSIG web application classpath**.

Considering the above requirement the typical process to deploy the WSIG add-on in a real application involves the following steps:

1. Prepare the WSIG web application content in the `webModule` directory. This step can be performed by means of the `build` target of the ANT build file included in the WSIG distribution.
2. Copy ontology and mapper application specific jar files in the `webModule/WEB-INF/lib` directory.
3. Edit the `webModule/conf/wsig.properties` file to specify required WSIG configurations as will be described in 4.2.
4. Edit the `webModule/WEB-INF/web.xml` file to specify required WSIG security configurations as will be described in 4.24.
5. Create the WSIG war file by zipping the content of the `webModule` directory. This step can be performed by means of the `war` target of the ANT build file included in the WSIG distribution. The produced war file is put in the `webapp` directory.
6. Copy the WSIG war file (`wsig.war`) into the `webapps` directory of the servlet container.

Alternatively one can:

1. Create the (application agnostic) WSIG war file by means of the `war-base` target of the ANT build file included in the WSIG distribution.
2. Copy the WSIG war file (`wsig.war`) into the `webapps` directory of the servlet container.
3. Copy application specific ontology and mapper jar files directly in the `wsig/WEB-INF/lib` directory of the servlet container.
4. Edit WSIG configurations `wsig.properties` directly in the `wsig/conf` directory of the servlet container as will be described in 4.2.
5. Edit the `web.xml` directly in the `wsig/webModule/WEB-INF` directory of the servlet container as will be described in 4.24.

Eventually, it is also possible to customize and deploy a WSIG application following these steps:

1. Prepare a zip file, containing the configuration file and the libraries and the classes, used in the ontologies and in the mappers
2. Use the ant target `customize-war` to build the webapp war.

The structure of the zip file is showed in Figure 4, where both the configuration files ((`wsig.properties` e `web.xml`) and the folders (`lib` e `classes`) are optionals.

```
myZip/
  |---wsig.properties
  |---web.xml
  |---lib/
  |     |---myLib1.jar
  |     |---myLib2.jar
  |---classes/
  |     |---myClass1.class
  |     |---myPackage/
  |     |     |---myClass2.class
```

Figure 4. Customize-zip directory structure

When invoked, the ant target `customize-war` interactively asks to the user the path of the zip file and produces, in the `webapp` folder, the WSIG war, ready to be deployed and customized using the libraries, the classes and the configurations files, included in the zip file.

The ant target `customize-war` can also been called from an ant file, programmatically setting the zip file path, as follows in the example:

```
<target name="my-custom-target">
   <property name="wsig.dir" value="<WSIG-installation-path>"></property>

   <ant antfile="${wsig.dir}/build.xml"
        dir="${wsig.dir}"
        target="customize-war">
      <property name="zipPath" value="<zip-path>"></property>
    </ant>
</target>
```

4.2 Configuration

All WSIG configurations can be set by editing the `conf/wsig.properties` file. This file includes both the configurations of the JADE container that will host the WSIG agent (such as the host and port of the Main Container) and the WSIG specific configurations. For the

configurations of the JADE container the reader is redirected to the JADE Administrator's guide since all JADE configuration options can be used. WSIG specific configurations are summarized in Table 1.

Parameter	Description	Default value
JADE configuration section		
host	Host name of JADE main-container	localhost
port	Port of JADE main-container	1099
container-name	Name of WSIG container	WSIG-Container
local-port	Port of WSIG container	Automatically assigned by platform
SL-preserve-java-types	Enable the java type preservation in message communications	false
WSIG configuration section		
wsig.servicesURL	URL to invoke the WSIG SOAP services	http://<webapp-url>/ws
wsig.servicesPath	SOAP endpoint path	ws
wsig.REST.servicesURL	URL to invoke the WSIG REST services	http://<webapp-url>/wsRest
wsig.REST.servicesPath	REST endpoint path	wsRest
wsig.admin.servicesURL	URL to invoke the WSIG ADMIN services	http://<webapp-url>/admin
wsig.admin.servicesPath	ADMIN endpoint path	admin
wsig.agent	Fully qualified class name of the WSIG agent in case one needs to extend/modify the base functionality of the WSIGAgent class	com.tilab.wsig.agent.WSIGAgent
wsig.style	Defines the style to use. Supported values are rpc (indicates rpc/encoded) and document (indicates document/literal wrapped)	document
wsig.timeout	Timeout (in milliseconds) for the execution of agent actions corresponding to WSIG invocations	30000
wsdl.localNamespacePrefix	TNS prefix used in WSDL construction	impl
wsdl.writeEnable	Tells WSIG whether or not to store generated WSDL descriptions to .wsdl filess	false
wsdl.directory	The directory, relative to the WSIG web application root (webModule) where to store generated WSDL files. Has no effect if wsdl.writeEnable is set to false	wsdl
wsig.traceClientIP	Add client IP in UserDefinedParameter	True
wsig.traceHttpHeaders	Add http headers in UserDefinedParameter	True
UDDI registry configuration section		
uddi.enable	Flag that specifies whether or not generated WSDLs must be published in a UDDI registry too. If this option is set to false all other options in this section are ignored	false
uddi.queryManagerURL	UDDI inquiry URL	
uddi.lifeCycleManagerURL	UDDI publication URL	
uddi.businessKey	UDDI business key	
uddi.userName	UDDI username	
uddi.userPassword	UDDI password	
UDDI4j configuration section		
org.uddi4j.logEnabled	Flag used to turn on/off UDDI4j logs	true

org.uddi4j.TransportClassName	The class implementing the protocol used to interact with the UDDI registry	org.uddi4j.transport.ApacheAxisTransport
Ontologies section		
onto.<ontology-name>		

Table 1. WSIG configuration properties

The ontologies section includes a number of application specific properties describing the ontologies the WSIG will have to deal with. In particular for each such ontology there must be a property of the form

`onto.<ontology-name>=<ontology-fully-qualified-classname>`

For instance if the WSIG is expected to expose as web services agent services referring to the `math-ontology` implemented by the `com.tilab.wsig.examples.MathOntology` class, the `wsig.properties` file must include a property of the form

`onto.math-ontology=com.tilab.wsig.examples.MathOntology`

4.3 WebSphere Application Server configuration

Only when deploying WSIG applications on IBM WebSphere Application Server (WAS) 6.0 or 6.1, you must configure the server to use the application's class loader before the container or system class loaders; this will ensure that the WSDL4J classes that are in /WEB-INF/lib directory will be loaded before those in $WAS_HOME/lib.

If you are developing and deploying your application through IBM Rational Application Developer (RAD), you can make the required changes by setting the class loader mode to PARENT_LAST.

If you are installing your application through the WAS admin console, select Enterprise Applications > Your Application > Class loading and update detection. You should then check the boxes labeled as follows:

- Classes loaded with application class loader first
- Single class loader for application

Making these changes should not affect your other applications.

4.4 Security

WSIG supports HTTP authentication, SSL certificates and WS-Security UsernameToken and Timestamp types.

4.4.1 HTTP authentication

The following steps are necessary to enable HTTP authentication:

1) Add to the WEB-INF/web.xml file the sections `security-constraint`, `login-config` and `security-role`, as follows:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>WSIG</web-resource-name>
    <url-pattern>/ws/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>wsig-role</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>WSIG authentication</realm-name>
</login-config>
<security-role>
  <description>The role that is required to login to WSIG</description>
  <role-name>wsig-role</role-name>
</security-role>
```

The authentication scope can be extended to the whole WSIG application (administration console an SOAP services) or just applied to the SOAP services: this can be done modifying the `url-pattern` parameter. Possible values are:

- `<url-pattern>/*</url-pattern>`: to the whole WSIG application
- `<url-pattern>/ws/*</url-pattern>`: only to the SOAP services

2) Add to the user configuration section of the servlet container the relation between the `wsig-role` and the authentication credentials. For Apache Tomcat, in example, that can be done, adding to the `tomcat-users.xml` file the following row:
`<user username="my-username" password="my-password" roles="wsig-role"/>.`

4.4.2 SSL certificate

The following steps are necessary to enable SSL certificate authentication:

1) Add to the `WEB-INF/web.xml` file the `security-constraint section`, as follows:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>WSIG</web-resource-name>
    <url-pattern>/ws/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

The authentication scope can be extended to the whole WSIG application (administration console an SOAP services) or just applied to the SOAP services: this can be done modifying the `url-pattern` parameter. Possible values are:

- `<url-pattern>/*</url-pattern>`: to the whole WSIG application
- `<url-pattern>/ws/*</url-pattern>`: only to the SOAP services

2) Properly configure and add the keystore to the servlet container: for Apache Tomcat, in example, adding or modifying the following row:

```
<Connector port="8443" ....  keystoreFile="<keystore-path>"
keystorePass="store-password"/>
```

in the `server.xml` file.

A new keystore can be generated, exploiting the specific tool provided by the java JDK by means of the following command:

```
keytool -genkey -alias wsig -keypass <store-password> -
keystore wsig.bin -storepass <store-password>
```

This command generates a new keystore, named `wsig.bin`, protected by the `store-password`.

4.4.3 WS-Security

The WS-Security can be enabled, properly configuring the wss section in the WSIG configuration file `conf/wsig.prop`. The WS-Security is applied just to the SOAP services.

The supported criteria are:

- `UsernameToken` of PasswordText type that can be configured as it follows

  ```
  wss.username=<my-wss-username>
  wss.password=<my-wss-password>
  ```

- `Timestamp` that can be configured as it follows:

  ```
  wss.timeToLive=<my-wss-timeToLive>
  ```
  (value in seconds)

5 ADMINISTRATION GUI

The WSIG add-on comes with a simple administration Web GUI by means of which it is possible to check the exposed web services. If the standard installation described in section 4 is followed the WSIG administration GUI can be reached at the http://localhost:8080/wsig URL.

As depicted in `Figure 4` the main page shows the list of exposed web services and the configuration parameters.

Figure 4. The WSIG Administration GUI

By clicking on a service in the services list the relevant information about that service will be shown. These include the agent providing the service, the referenced ontology, the mapper and prefix if any, the list of available operations and a link to the WSDL (see Figure 5).

Figure 5. WSIG Administration GUI: Service details

The WSIG GUI also includes a test page by means of which it is possible to trigger SOAP requests towards an exposed web service. This is done by pasting an XML request message in the SOAP Request area and clicking on the Send button. The response message will appear in the SOAP Response area (see Figure 6). A number of XML request messages referring to the MathAgent example are included in the examples/xml directory of the WSIG distribution.

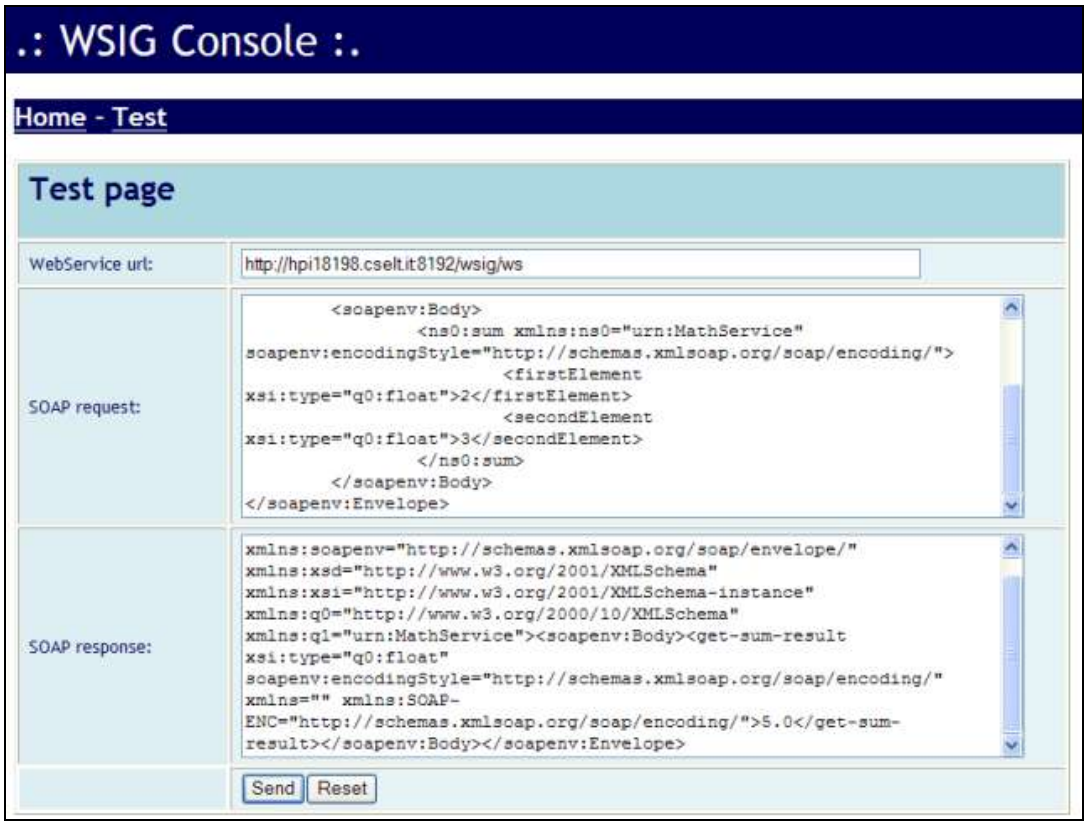

Figure 6. WSIG Administration GUI: Test page

6 ADMINISTRATION API

The next chapter will present the reference to the REST API of the Web Service Integration Gateway (WSIG) add-on. This allows the user to manage the platform using simple HTTP methods. In addition, every resource will be presented with its respective methods, description and possible representations.

6.1 RESOURCES:

6.1.1 /admin/platform

Method: GET
URL: http://localhost:8080/wsig/admin/platform
Description: retrieves the status of the WSIG platform as a String. ACTIVE if it is running normally, DOWN if it has been shut down properly, or UNKNOW in any other case.
Available response representations: • **200 OK**

6.1.2 admin/platform/{status}

Method: PUT		
URL: http://localhost:8080/wsig/admin/platform/{status}		
Description: The template parameter is used to turn on or shutdown the WSIG platform passing “connect” or “disconnect “ respectively.		
Template Parameter	description	
status	connect/disconnect	
Available response representations: • **200 OK**		

6.1.3 admin/configuration

Method: GET
URL: http://localhost:8080/wsig/admin/configuration
Description: retrieves the current configuration parameters of the WSIG platform
Available response representations: • **200 OK + (XML/JSON)** Configuration

Method: PUT
URL: **http://localhost:8080/wsig/admin/configuration**
Description: updates the current configuration parameters of the WSIG platform whit the information passed in the request body.
Request Body: **(XML/JSON)** Configuration
Available response representations: • **200 OK**

6.1.4 /admin/services

Method: GET
URL: **http://localhost:8080/wsig/admin/services**
Description: returns the list of services running in the WSIG platform
Available response representations: • **200 OK + (XML/JSON)** services

6.1.5 /admin/services/{serviceName}

Method: GET
URL: **http://localhost:8080/wsig/admin/services/{serviceName}**
Description: returns the information of the service which name has been specified in “serviceName”
Available response representations: • **200 OK + (XML/JSON)** service

6.1.6 /admin/services/{serviceName}/wsdl

Method: GET
URL: **http://localhost:8080/wsig/admin/services/{serviceName}/wsdl**
Description: returns the *.wsdl file that represents the service which name has been specified in “serviceName”
Available response representations: • **200 OK + WSDL file**

6.2 XML AND JSON CODE EXAMPLES

6.2.1 Configuration

XML Format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration>
    <jadeMainHost>localhost</jadeMainHost>
    <jadeMainPort>1099</jadeMainPort>
    <jadeContainerName>WSIG-Container</jadeContainerName>
    <containerLocalPort>1200</containerLocalPort>
    <WSIGAgentClass>com.tilab.wsig.agent.WSIGAgent</WSIGAgentClass>
    <WSIGVersion>2.9 - revision 1955 of 2013/02/26
15:46:51</WSIGVersion>
    <WSIGServicesURL>http://localhost:8080/wsig/ws</WSIGServicesURL>
    <WSIGAdminURL>http://localhost:8080/wsig</WSIGAdminURL>
    <WSIGTimeOut>30000</WSIGTimeOut>
    <WSIGJavaTypePreservation />
    <WSDLHierarchicalComplexType>false</WSDLHierarchicalComplexType>
    <WSDLLocalNameSpace>impl</WSDLLocalNameSpace>
    <WSDLStyle>document</WSDLStyle>
    <WSDLWriteEnable>false</WSDLWriteEnable>

<WSDLWritePath>C:\Users\Administrator\wtpwebapps\wsig\wsdl</WSDLWritePat
h>
    <uddiEnable>false</uddiEnable>
    <uddiQueryManager />
    <uddiLifeCycleManager />
    <uddiBusinessKey />
    <uddiUserName />
    <uddiPassword />
    <uddiTModel />
</configuration>
```

JSON Format:

```
{
    "jadeMainHost": "localhost",
    "jadeMainPort": "1099",
    "jadeContainerName": "WSIG-Container",
    "containerLocalPort": "1200",
    "WSIGAgentClass": "com.tilab.wsig.agent.WSIGAgent",
    "WSIGVersion": "2.9 - revision 1955 of 2013/02/26 15:46:51",
    "WSIGServicesURL": "http://localhost:8080/wsig/ws",
    "WSIGAdminURL": "http://localhost:8080/wsig",
    "WSIGTimeOut": "30000",
    "WSIGJavaTypePreservation": "",
```

```
        "WSDLHierarchicalComplexType": "false",
        "WSDLLocalNameSpace": "impl",
        "WSDLStyle": "document",
        "WSDLWriteEnable": "false",
        "WSDLWritePath":
"C:\\Users\\Administrator\\wtpwebapps\\wsig\\wsdl",
        "uddiEnable": "false",
        "uddiQueryManager": "",
        "uddiLifeCycleManager": "",
        "uddiBusinessKey": "",
        "uddiUserName": "",
        "uddiPassword": "",
        "uddiTModel": ""
    }
```

6.2.2 Services

XML Format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<services>
  <service>MathFunctions</service>
  <service>BeanMathFunctionsMapper</service>
</services>
```

JSON Format:

```
{
  "service": ["MathFunctions", "BeanMathFunctionsMapper"]
}
```

6.2.3 Service

XML Format:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<service>
    <name>MathFunctions</name>
    <prefix>-</prefix>
    <mapperClass>-</mapperClass>
    <hierarchicalComplexType>false</hierarchicalComplexType>
    <jadeOntology>wsig_math-ontology</jadeOntology>
    <jadeAgent>MathAgent1@WSIGTestPlatform</jadeAgent>
    <uddiServiceKey>-</uddiServiceKey>

<wsdlUrl>http://localhost:8080/wsig/ws/MathFunctions?WSDL</wsdlUrl>
    <operations>
```

```
            <operation>abs</operation>
            <operation>getAgentInfo</operation>
            <operation>multiplication</operation>
            <operation>diff</operation>
            <operation>printTime</operation>
            <operation>getRandom</operation>
            <operation>sum</operation>
            <operation>compareNumbers</operation>
            <operation>convertDate</operation>
            <operation>getComponents</operation>
            <operation>sumComplex</operation>
            <operation>printComplex</operation>
        </operations>
    </service>
```

JSON Format:

```
    {
        "name": "MathFunctions",
        "prefix": "-",
        "mapperClass": "-",
        "hierarchicalComplexType": "false",
        "jadeOntology": "wsig_math-ontology",
        "jadeAgent": "MathAgent1@WSIGTestPlatform",
        "uddiServiceKey": "-",
        "wsdlUrl": "http://localhost:8080/wsig/ws/MathFunctions?WSDL",
        "operations": {
            "operation": ["abs", "getAgentInfo", "multiplication",
                          "diff",  "printTime", "getRandom", "sum",
                          "compareNumbers",               "convertDate",
"getComponents",
                          "sumComplex", "printComplex"] }
    }
```

7 APPENDIX I. DESCRIPTION OF THE EXAMPLES

The WSIG add-on comes with some examples that aim at clarifying how to exploit it to expose agent services as web services. These examples refer to a minimal `MathOntology` defining concepts and actions dealing with simple mathematical operations such as `SUM`, and `MULTIPLICATION` of possibly `COMPLEX` numbers. The bin directory includes `.bat` and `.sh` scripts to start these examples.

In order to try the WSIG examples the following steps should be performed (the same sequence of steps apply to the `.sh` scripts in case you are working on a Linux/Unix machine).

- Launch the `runJade.bat` script to start the JADE Main Container
- Create the WSIG web application including the WSIG examples by means of the `war-examples` target of the ANT build file included in the WSIG distribution.
- Deploy the `webapps/wsig.war` file (produced in the previous step) in Tomcat.
- Start Tomcat
- Look at the JADE administration GUI (RMA): a new container including the WSIG agent should appear.
- Launch the `runMathAgent.bat` script to start (in a new container) an agent registering a service (called `MathService`) referring to the `MathOntology`
- Launch the `runMathAgentPrefix.bat` script to start (in a new container) an agent registering the same `MathService` but specifying a prefix
- Launch the `runMathAgentMapper.bat` script to start (in a new Container) an agent registering another service (called `MathServiceMapper`) and specifying an ontology mapper.
- Use the WSIG test console or the `runSoapClient.bat` script to perform web service invocations on the services exposed by the agents started in the previous steps. The directory `examples/xml` contains several raw SOAP requests to be used to query that services.