The WSDL2Agent Tool

László Zsolt Varga, Ákos Hajnal and Zsolt Werner

Abstract. The WSDL2Agent tool is used to help the integration of existing web services into agent based systems. The input to the WSDL2Agent tool is the WSDL file of a web service and the tool provides two types of output. The WSDL2Jade part of the tool generates code for a proxy agent that makes the web service available in multi-agent environment. The WSDL2Protégé part of the tool generates project file for the Protégé ontology engineering tool in which the ontology of the web service can be semantically enriched, visualized, or exported to various formats. In this paper we present the technical details of the code generators and the application scenario of the tool.

1. Introduction

The creation and popularity of web services are growing rapidly, and recently, web service interface is more and more often provided for internet services in addition to the traditional web interface. Interest in agent technology is also increasing both in the field of research (grid, semantic web, AI) and industry (ontology modelling, service integration). This motivated us to create the WSDL2Agent tool, which can be used in the integration of existing web services into agent based systems. This way we help to bridge the gap between existing non-agent systems and agent systems. With the help of this tool existing web services can be integrated into agent systems and agents will have access to a wide range of existing services. Moreover agent system developers may concentrate on adding intelligent functions to these services by combining and extending them. The idea of using agents to enable advanced operational modes of web services is advocated by several researchers [1][2] and also by the Web Services Architecture specification of the W3C [3].

The WSDL2Jade tool is applicable to create proxy agents for arbitrary web services. The proxy agent is able to accept client agent requests to invoke the web service, call the web service, and send the web service results back to the client agents (illustrated in Figure 3). The proxy agent thus wraps the web service invocation code into an agent

shell and operates like a FIPA agent: it can be deployed on an agent platform, communicates with other agents using FIPA standard Agent Communication Language (ACL) [4], and has a public ontology that defines the messages it understands.

The WSDL2Protégé tool is applicable to create Protégé [5] project file from the WSDL file of an arbitrary web service. This project file can then be opened in the Protégé knowledge-base modelling environment, where the ontology of the original WSDL document can be prepared and edited (as shown in Figure 2). In addition, Protégé is able to export the ontology in various formats, such as RDF, OIL, DAML+OIL, or OWL. Using the Ontology Bean Generator plug-in [6] of Protégé an agent skeleton can be created from the Protégé project file.

The main contribution of this paper is the detailed description of a technology to integrate web services into agent systems, the implementation of the technology in the WSDL2Agent tool, applying the tool to the Google web service as well as demonstrating in a simple application how an agent can integrate existing web services and add new functionality to them by combining them. As we will see in the related work section, these contributions are novel.

In section 2 we shortly sum up the background information needed to use the WSDL2Agent tool. In section 3 we describe how to use the WSDL2Agent tool. Based on this information you can generate your own code on the web site of the tool. Section 4, 5 and 6 describe in detail how the WSDL2Agent tool generates code. You will need this information to understand, possibly modify and deploy the generated code. Section 4 describes how the WSDL2Agent tool reads and interprets WSDL files. Section 5 describes how the WSDL2Jade tool generates the code, while section 6 details how the WSDL2Protégé tool generates the project file. Section 7 shows a demo agent system integrating web services. Section 8 discusses related work and section 9 concludes the paper.

2. Background

In this section we are going to introduce web services, agents, the JADE agent platform and the Protégé knowledge engineering environment, because the WSDL2Agent tool builds on them.

2.1 Web Services and the WSDL

The purpose of web services [3] is to provide some document or procedure-oriented functionality over the network. The web service is an abstract notion of certain functionality, the concrete piece of software implementing the web service is called provider agent, while other systems that wish to make use of the web services are called requester agents. Requester agents can interact with the provider agent via message exchange using SOAP (Simple Object Access Protocol) messages typically conveyed over HTTP. The interface of the web service is described in machine-readable format by a WSDL (Web Services Description Language) document. It defines the service as a set

of abstract network end-points called ports and a set of operations representing abstract actions available at these ports. For each operation the possible input and output messages are defined and decomposed into parts. The data types of parts are described using the XSD (XML Schema Definition) type system, which may be either built-in XML data types or custom, even nested structures: complex types or arrays. The abstract definitions of ports, operations, and messages are then bound to their concrete network deployment via extensibility elements: internet location, transport protocol, and transport serialization formats.

The definition of the web service interface in the WSDL is encapsulated by a single <definitions> element in which the <service> element contains the internet address of the web service in the location attribute of the <soap:address> element. The <portType> element consists of the list of the operations (<operation> elements) provided by the web service, for which input and output message types are associated via the <input> and <output> elements. Message types are described by individual <message> elements; each containing a list of named parts with either a built-in XML type or a complex type. Complex types must be defined in the <types> section of the WSDL that may form structures or arrays. Finally, the <binding> element of the WSDL is applicable to specify the encoding style and namespace for each operation input and output message.

2.2 Agents

The notion of agent emerged from many different fields including economics, game theory, philosophy, logic, ecology, social sciences, computer science, artificial intelligence and later distributed artificial intelligence. In all these fields an agent is an active component that behaves intelligently in a complex environment to achieve some kind of goal. Experts from the different fields tend to agree that the most important characteristics of agents are those which are defined by Wooldridge and Jennings [7]. First of all an agent is a computer system situated in some environment. The agent is reactive, which means that it is capable of sensing its environment and acting on it. The agent can autonomously act in its environment and make decisions itself. The agent has design objectives and can decide itself how to achieve them. While taking the decisions the agent is not just passive, but can take initiatives towards its goals. The agent has social abilities and can interact with the actors in its environment.

The need for interoperable agent communication created the Foundation for Intelligent Physical Agents (FIPA) [8] standardization body. The aim of FIPA is to develop software standards for heterogeneous and interoperating agents and agent systems, in order to enable the interworking of agents and agent systems operating on platforms of different vendors in industrial and commercial environments. As a result of the FIPA standardization activity, many research labs and industrial organizations started to develop competing agent platforms independently all over the world. FIPA standard agent platforms provide an environment where agents can be deployed and with the help of the agent platform services they can interact with other agents on any

FIPA standard agent platform in a FIPA conformant way achieving agent communication level interoperability.

Agents are starting to become highly relevant to bodies such as the World Wide Web Consortium (W3C) and the Global Grid Forum (GGF). The convergence between grid and agents is of interests, because agent systems require robust infrastructure and Grid systems require autonomous, flexible behaviours [9]. Developments such as web services and semantic web services [10][11] also investigate many of the issues which have already been addressed by agent technologies.

2.3 JADE

JADE (Java Agent DEvelopment Framework, http://jade.tilab.com/) is a software framework that complies with the FIPA specifications and provides a middleware to help in the implementation of multi-agent systems. The agent platform is fully implemented in Java. Software agents also written in Java can be deployed and run in JADE. The agent platform can be distributed across machines and the configuration can be controlled via a remote GUI. In addition to JADE's core functionality, a set of graphical tools are available which support the debugging and deployment phases. For agent developers JADE provides libraries through which agents can easily communicate with other agents due to the built-in message handling, composition and decoding functionalities.

We chose JADE as the target platform for the proxy agent in the WSDL2Agent tool because of the above features, and also because this agent platform is the most widely used, as shown by the statistics of the Agentcities worldwide testbed. Chosing JADE as the target platform does not put much restriction on the deployment of the generated agents. Although the generated proxy agent (described in section 5) is required to run in JADE, client agents wishing to communicate with the proxy agent can still run on any FIPA compliant platform.

2.4 Protégé

Protégé (http://protege.stanford.edu/) is an ontology engineering tool developed at Stanford Medical Informatics, and though it has historically been used in the area of biomedical applications, nowadays it is widely used as knowledge-base modelling environment in different application areas as well. Since Protégé's internal model is based on a meta model similar to the object-oriented and frame-based systems, it is basically applicable to design and represent ontologies consisting of classes (frames), properties (slots), property characteristics (facets and constraints), and instances. With Protégé, designers can build, edit, maintain, and visualize ontologies representing the formal model of the knowledge domain on a graphical user interface, which tasks would require large amount of manual work otherwise, especially in the case of real-world applications. In addition, due to the extensibility of the platform, Protégé can import

existing ontologies from, and export a Protégé ontology to various formats (such as CLIPS, XML, UML, OIL, DAML+OIL, RDF, OWL) via plug-ins.

3. Using WSDL2Agent

In this section we are going to describe the engineering process of the integration of web services into agent system and then how you can use the WSDL2Agent tool to generate the code for this process.

3.1 Engineering Process

The architecture of web service integration into agent systems is shown in Figure 1. Each Web Service is represented by a Proxy Agent. The Proxy Agent is able to receive web service invocation requests in FIPA Agent Communication Language (ACL), invoke the web service using the Simple Object Access Protocol (SOAP) and then return the result in FIPA ACL. These proxy agents basically have the same functionality as the web service, but they are able to communicate in FIPA ACL. In order to add autonomous and proactive features to web services, we imagine that an Agent with Agent Logic is also developed. The Agent with Agent Logic is the client agent of one or more Proxy Agents and provides intelligent services to the Client Agent which might be either a user interface agent or any other agent in the agent system.

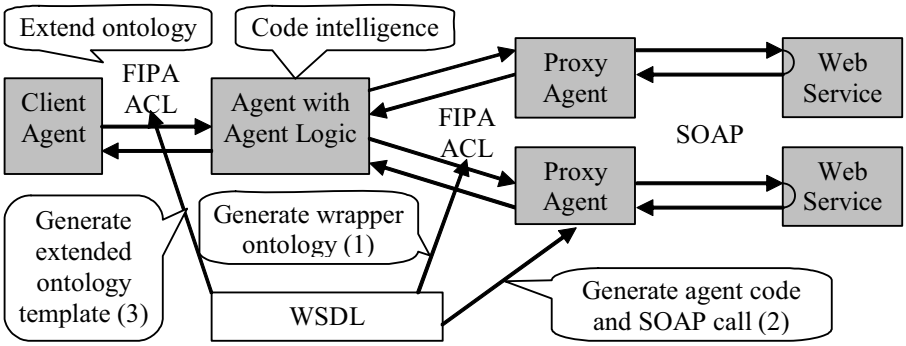

Fig. 1. Web services integrated into agent systems

The two most important software components in our agent system implementation are the ontology code and the agent code. The ontology code implements the way the agents encode, decode and interpret the content of ACL messages. The agent code implements the actions taken by the agent. In the case of the Proxy Agents there are direct mappings from web service definitions to ontology code and web service invocation to agent code, because the Proxy Agent converts the web service protocol to ACL messages and the only type of action it takes is the web service invocation. These

mappings are described in section 5. The WSDL2Agent tool generates the code for the ontology between the Proxy Agent and the Agent with Agent Logic (indicated with No. 1 in Figure 1), as well as the code of the Proxy Agent (indicated with No. 2 in Figure 1).

While communicating with its clients, the Agent with Agent Logic will use an ontology similar to the wrapper ontology possibly extended with a few other concepts, actions and elements, because the Agent with Agent Logic also uses the web service results. The WSDL2Protégé tool generates a template for the ontology between the Agent with Agent Logic and its clients (indicated with No. 3 in Figure1). The Ontology Bean Generator can be used to generate the code template of the Agent with Agent Logic. The agent logic has to be added by the developer both to the extended ontology and the Agent with Agent Logic.

3.2 Generating Code with WSDL2Agent

The WSDL2Agent tool is available on-line at the internet location http://sas.ilab.sztaki.hu:8080/wsdl2agent. You can upload your WSDL file to the tool in two different ways: either by specifying the URL of the file, or directly uploading the file itself. Then select the output you need: either WSDL2Jade to generate the code for the Proxy Agent and its ontology, or WSDL2Protégé to generate the Protégé project files for the template of the extended ontology.

3.2.1 WSDL2Jade Output

The WSDL2Jade tool web application returns the generated files in a zipped archive. Section 5 will describe how these files are generated. Here we give you a summary of the files in order to give you an overview. You may want to come back here and read again this description after reading section 5.

The *webserviceontology* directory in the archive contains the Java files belonging to the ontology, which is further decomposed into *service/port/operation* directory hierarchy in order to separate AgentAction, Predicate and Concept classes corresponding to different operations (and to avoid file name conflict). In the case of complex types a separate *service/port/operation/soap* directory contains the Java classes needed by the SOAP-based communication. And finally, the *webserviceontology/service* directory contains the WebServiceOntology.java file that registers the ontology of the web service, and the common ErrorPredicate.java file.

The code of the proxy agent is placed in the *webserviceagent* directory, and two batch files are provided in the root directory to help in compiling (compile.bat) and running (run.bat) the proxy agent. The WSDL2Jade tool automatically replaces the (potential) Java keywords in the WSDL (by appending the "_value" suffix to such strings) resulting in syntactically correct Java sources. The compilation also creates a jar archive of the ontology which can be published to client agents wishing to contact the proxy agent.

The web page of the WSDL2Agent tool describes how to write a test client agent to invoke the services of the wrapper agent. You can directly copy and paste the code of a test client agent from the documentation section of the web site.

3.2.2 WSDL2Protégé Output

The WSDL2Protégé tool generates a Protégé project that consists of three files called: webservice.pprj, webservice.pont, webservice.pins. To load the project into Protégé, open the file webservice.pprj. The project contains ontology elements, classes (a.k.a. frames) describing the communication with the web service. In order to be able to use the Ontology Bean Generator plug-in, the generated web services ontology complies with the predefined ontology hierarchy of the Ontology Bean Generator. Figure 2 shows an ontology generated by WSDL2Protégé loaded into Protégé. After editing this ontology you can generate an agent code template using the Ontology Bean Generator plug-in.

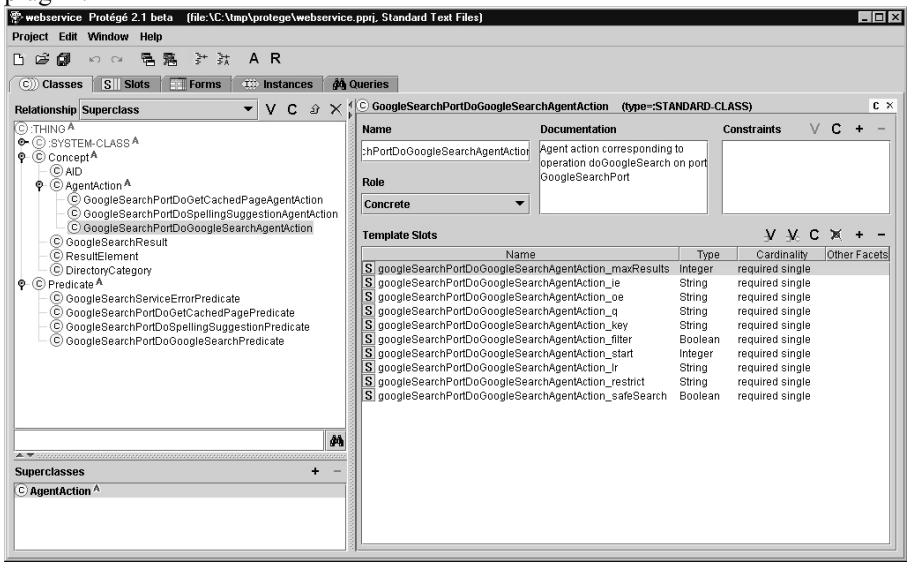

Fig. 2. The model in Protégé visual development environment

4. WSDL Parsing

In this section we describe how the WSDL2Agent tool builds an internal model of the web service description in the WSDL file. The WSDL2Jade and WSDL2Protégé tools use this internal model to generate code. In order to be able to process any WSDL file, the most critical is the parsing of complex and array types.

All parsing takes place in org.sztaki.wsdl2jade.WsdlParser, which is supported by some helper classes mostly for storing data. The processing starts with a call to one of parseWsdlFile(), parseWsdlURI() or parseWsdlInputStream(), depending on the physical source of the WSDL. All the three methods return an org.sztaki.wsdl2jade.Wsdl instance, after reading in the WSDL definition

(javax.wsdl.Definition) from the source and calling processDefinition() in the same class.

The processDefinition() processes the <definitions> element of the WSDL, and builds the WSDL model in the memory, which is then returned as result. In the outermost loop, it iterates through the services javax.wsdl.Definition. Inside of the loop, there is another loop iterating over the ports. It reads the <soap:address location=""> extensibility element from the port body; the binding; sets transport and style of the port; gets the operations of the newly-read binding. As there can be more than one operations associated with a binding, it iterates over them. It sets the soap action for the operation, sets soap:body encodingStyle and namespace of the port and processes the input and output parts by calling processInputAndOutputParts().

The latter two are of extreme importance because of the complex type system a WSDL can have. The processInputAndOutputParts() processes all <part>'s of the passed <input> / <output> tags. Uses a boolean to decide between input and output. The method is not recursive because Part objects can not reference (contain) other Part objects, unlike with the case of ComplexType objects (and <complexType> tags). This is where we decide whether the part in question is a complex type or an array (or the combination of both). Depending on this, we give a further call to processComplexType().

The processComplexType() is passed a newly created ComplexType instance and a String denoting the "name" attribute of a <complexType> in the WSDL file. We search for this <complexType> in the entire XML document. Upon finding it, we decide whether it's an array declaration or a standard <complexType> with <elements> (see section 5.4 on array handling). In both cases, the enclosed type information is read and if any <element> (or, in the case of an array declaration, the <attribute> tag) references another <complexType>, another instance of ComplexType is created. After that, the method is recursively called back.

Please note that we do exactly the same as in processInputAndOutputParts() before starting recursion over with the slight difference that we have a Hashtable instead of a pair of Strings to be set because parts and complex types are inherently the same.

5. The WSDL2Jade Tool

The WSDL2Jade tool is applicable to create a proxy agent, which is able to accept client agent requests, call a web service, and send the web service results back to the client agent. The proxy agent, wrapping the web service invocation code into an agent shell, works like an ordinary agent: it can be deployed in an agent platform and it communicates with other agents using FIPA ACL. The code generated by the WSDL2Jade tool uses the FIPA SL content language [12] supported by JADE. In multi-agent systems web services can be accessed easily by simply communicating with the proxy agent. Agent programmers can thereafter focus on the agent logic, which results in faster and more reliable development, since they do not have to implement occasionally complicated SOAP web service invocation code each time an agent wishes to use web services.

The WSDL2Jade tool is available on the internet, and developers only need to submit the WSDL file of the web service (or its URL) to get the complete Java sources composing the proxy agent. After compilation, the proxy agent can be deployed in the JADE agent platform. The ontology of the proxy agent can be published for potential client agents, who are then able to communicate with the proxy agent: compose requests and interpret reply messages, that is, to use web services. Figure 3 illustrates how the proxy agent interacts with client agents and the web service.

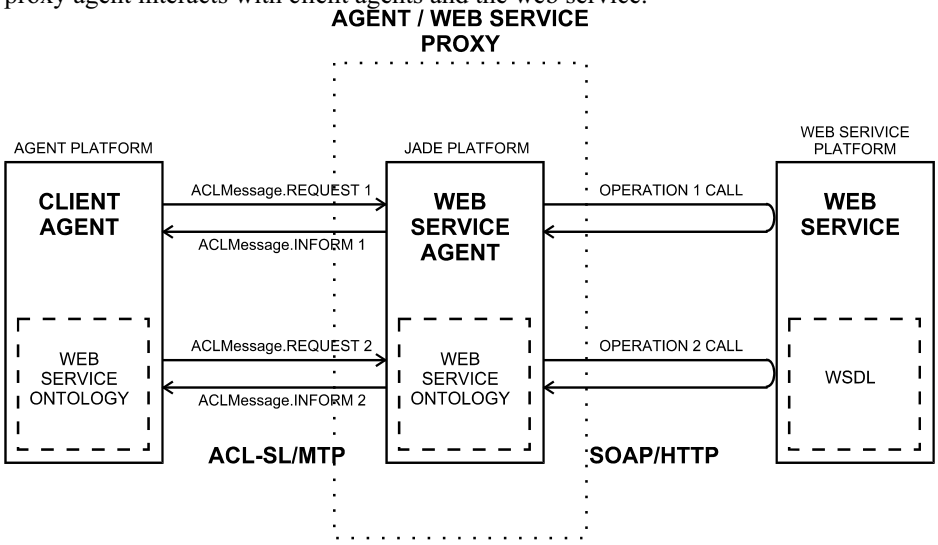

Fig. 3. The interaction of the proxy agent, the client agent, and the web service

The web service data types defined in the WSDL are mapped to ontology data types. The concrete mapping is discussed in the next sections. The web service invocation is mapped to an ACL message of FIPA REQUEST type corresponding to an agent action. The returned web service result is mapped to an ACL message of FIPA INFORM type corresponding to a predicate in the ontology. The proxy agent receives the FIPA REQUEST, interprets it according to the ontology, invokes the web service on the port defined in the WSDL, encodes the result of the web service invocation into a FIPA INFORM of the ontology and returns the result.

The codes presented in the next sections are generated by the WSDL2Jade tool for the web services of Google. The interface of the services is defined by the WSDL description located at http://api.google.com/GoogleSearch.wsdl. The Google Web API consists of three operations called doGoogleSearch, doGetCachedPage, and doSpellingSuggestion. The doGoogleSearch operation is applicable to search in Google's index of web pages by submitting a query string (and search parameters) for which Google sends the set of search results back. The doGetCachedPage operation returns the cached contents of a specified URL when Google's crawlers last visited the page; and finally, the doSpellingSuggestion operation can be used to obtain spell

correction for the submitted expression. For more information about Google web services refer to [14].

5.1 The Agent Code

The source code of the proxy agent can be divided into two main sections: agent setup code and ontology code described in the following section. After parsing the submitted WSDL file and building an internal model, the WSDL2Jade tool is capable of generating all the sources automatically, by adapting a previously constructed internal code skeleton using the actual information in the WSDL. The source code complies with the guidelines of the JADE [13] Programmer's Guide.

The agent code is contained by the generated Java class called *WebServiceName*Agent (where *WebServiceName* is the name attribute of the service element in the WSDL), which is responsible to start the proxy agent in the agent platform. In the *setup* method, the proxy agent registers the language (the FIPA standard SL *codec*), registers the ontology, and specifies the *behaviour* of the agent (addBehaviour), defining its life-cycle (e.g. OneShotBehaviour, CyclicBehaviour) and functionality. The corresponding part of the source code of the proxy agent generated for the Google web services [14] is illustrated below:

```
class GoogleSearchServiceAgent extends Agent {
  ContentManager manager = getContentManager();
  void setup() {
    manager.registerLanguage(new SLCodec());

manager.registerOntology(WebServiceOntology.getInstance
());
    addBehaviour(new HandleRequestBehaviour(this));
  }
```

The behaviour of the agent is determined by the inner class called HandleRequestBehaviour, which extends JADE's CyclicBehaviour so that the agent accepts messages in infinite loop. In the *action* method of the HandleRequestBehaviour class the agent handles the incoming messages. In accordance with standards used in agent technology and considering the protocol of web service message exchange, a client request message has to be mapped to the FIPA REQUEST type message, and the proxy agent's answer message conforms with the FIPA INFORM type message, respectively. In JADE, REQUEST type messages contain a single Java object implementing JADE's AgentAction interface, and INFORM messages contain an object implementing the Predicate interface. Since a web service may contain several operations (these are the actual functions that can be invoked) the WSDL2Jade tool assigns a unique *OperationName*AgentAction - *OperationName*Predicate pair of classes to each operation (where *OperationName* string is the name attribute of the operation element in the WSDL).

When a client request message arrives (and the *action* method is invoked), the proxy agent can thus decide which web service operation has to be called by simply matching

the incoming AgentAction class instance (which is the *ContentElement* object extracted from the message) against potential AgentAction classes assigned to different operations:

```
void action() {
  ACLMessage msg =

receive(MessageTemplate.MatchPerformative(ACLMessage.RE
QUEST));
  ContentElement ce =
manager.extractContent(msg).getAction();
  if (ce instanceof DoGoogleSearchAgentAction)...
  if (ce instanceof DoGetCachedPageAgentAction)...
  if (ce instanceof DoSpellingSuggestionAgentAction)...
```

The proxy agent can then create a call object (in the appropriate if branch), set the matching operation name (the name attribute of the related operation element in the WSDL), and from the AgentAction class it can read and add the required input parameters to the call object (*params.add*). This sequence corresponds to a standard web service call using SOAP. Note that since field types in the AgentAction class sent in the ACL message may be incompatible with the types required by Apache Axis SOAP implementation (used by the proxy agent), instead of the ordinary getter methods the generated _SOAP postfixed getter methods are applied here, which return data with type conforming to Axis (detailed in the next section). The operation can be invoked after setting the internet address of the web service (*setTargetEndpointAddress*), which can be found in the location attribute of the soap:address element:

```
org.apache.axis.client.Call call =
service.createCall();
call.setOperationName(new QName("urn:GoogleSearch",
"doGoogleSearch"));
params.add(ce.getKey_SOAP());
params.add(ce.getQ_SOAP());
...
call.setTargetEndpointAddress(
    new URL("http://api.google.com/search/beta2"));
...
resp = call.invoke(paramsObject);
```

If the web service call is successful (no exception is thrown), a new Predicate object is created representing the related operation output into which the web service result (*resp* object) is written. Finally, an INFORM message is constructed with the Predicate object (*fillContent*), and sent back to the client agent (*createReply*). The proxy agent, like in the case of reading the input parameters, uses the generated _SOAP postfixed setter method to write the received SOAP-type data into the ACL-compatible Predicate class.

```
DoGoogleSearchPredicate result = new
DoGoogleSearchPredicate();
result.setReturn_value_SOAP(resp);
ACLMessage answerMsg = msg.createReply();
answerMsg.setPerformative(ACLMessage.INFORM);
manager.fillContent(answerMsg, result);
send(answerMsg);
```

If the web service call fails for some reason, an error message, containing a dedicated ErrorPredicate class (with the description of the *AxisFault*) is returned.

5.2 XML Data Types in ACL Messages

The data types and data structures used by web service inputs and outputs are described by the WSDL file using XML Schema [15] language. We chose the Apache Axis implementation of SOAP to invoke and pass parameters to (or receive from) the web service in the proxy agent. Accordingly, Java sources (classes and field types) representing the XML types in the WSDL have to conform to the JAX-RPC specification [16] defining the XML-Java bindings. For example, in the case of a 'string' XML type, Axis assumes a java.lang.String object representation of the data used in the interaction with web service. Note that these bindings apply to the communication between the proxy agent and the web service.

On the agent side, however, data types used in the ACL messages have to be matched against the data types available in JADE, which supports five (symbolic) types called: STRING, INTEGER, FLOAT, BOOLEAN and DATE. As with the XML-Java bindings in Axis, JADE also prescribes how these symbolic data types have to be represented in the underlying Java code: STRING corresponds to String, INTEGER can be implemented by either Integer or Long, FLOAT can be either Float or Double, BOOLEAN corresponds to Boolean, and, finally, DATE must be implemented by java.util.Date class.

When creating the WSDL2Jade tool we had to decide how to represent XML types in JADE messages; and how to translate SOAP to ACL messages, and vice versa. In Table 1 we summarized the XML-JADE type mapping used by the WSDL2Jade tool.

Note that these associations may result in imprecise translation in some cases (e.g. integer, decimal) due to the limited number of data types available in JADE. (The encoding of such data types in strings would change the original semantic that we tried to avoid.) In practice, however, this mapping is still applicable to convert SOAP and ACL messages satisfactorily.

As an example, part of the XML definition of the input message of the *doGoogleSearch* operation is shown below:

```
<message name="doGoogleSearch">
  <part name="key" type="xsd:string"/>
  <part name="start" type="xsd:int"/>
  <part name="filter" type="xsd:boolean"/>
  ...
</message>
```

Table 1. XML-JADE type mapping

XML (WSDL)	JAVA REPRESENTATION(AXIS)	JADE ONTOLOGY	JAVA REPRESENTATION (JADE)
string	java.lang.String	STRING	java.lang.String
integer	java.math.BigInteger	INTEGER	java.lang.Long
int	java.lang.Integer	INTEGER	java.lang.Integer
long	java.lang.Long	INTEGER	java.lang.Long
short	java.lang.Short	INTEGER	java.lang.Integer
decimal	java.math.BigDecimal	INTEGER	java.lang.Long
float	java.lang.Float	FLOAT	java.lang.Float
double	java.lang.Double	FLOAT	java.lang.Double
boolean	java.lang.Boolean	BOOLEAN	java.lang.Boolean
byte	java.lang.Byte	INTEGER	java.lang.Integer
dateTime	java.util. GregorianCalendar	DATE	java.util.Date
base64Binary	byte[]	STRING	java.lang.String
hexBinary	byte[]	STRING	java.lang.String
unsignedInt	java.lang.Long	INTEGER	java.lang.Long
unsignedShort	java.lang.Integer	INTEGER	java.lang.Integer
unsignedByte	java.lang.Short	INTEGER	java.lang.Integer
time	java.util. GregorianCalendar	DATE	java.util.Date
date	java.util. GregorianCalendar	DATE	java.util.Date
anySimpleType	java.lang.String	STRING	java.lang.String

The WSDL2Jade tool generates the following AgentAction class wrapping the above input parameters:

```
class DoGoogleSearchAgentAction implements AgentAction{
  java.lang.String key = null;
  void setKey (String param) { key = param; }
  String getKey () { return key; }
  String getKey_SOAP () {
    String jadeSlot = this.key;
    String soapSlot = jadeSlot;
    return soapSlot;
  }
  java.lang.Integer start = null;
  ...
  java.lang.Boolean filter = null;
  ...
```

As it is seen, in the AgentAction class (used in the incoming ACL messages) fields are created with names corresponding to the name attribute of the part element and with type corresponding to the JADE Java type associated to the XML type in the WSDL. In addition to the simple getter and setter methods (used by JADE) for each field a _SOAP postfixed getter method is generated through which the proxy agent can access the data contained in the field in the Java type required by Axis (although in this example JADE and XML Java types coincide). In the case of an incoming message the proxy agent can thus easily pass web service input parameters to Axis using the _SOAP getter methods.

Similarly, in the Predicate classes (used in the answer ACL messages) _SOAP setter methods are generated by the WSDL2Jade tool, which expect SOAP Java type input parameters and set the related JADE type field. The proxy agent in this way can directly fill JADE type fields with SOAP Java type web service results.

Note that *base64binary* and *hexBinary* XML types, are represented by strings in the ACL messages, which contain the base64 encoded form of the original byte arrays (by convention). The encoding/decoding is performed automatically by the proxy agent. This is illustrated in the doGetChachedPagePredicate class:

```
class DoGetCachedPagePredicate implements Predicate {
  java.lang.String return_value;
  void setReturn_value (String param) { return_value =
param; }
  String getReturn_value () { return return_value; }
  void setReturn_value_SOAP (byte[] soapSlot){
    String jadeSlot = new String
            (starlight.util.Base64.encode(soapSlot));
    this.return_value = jadeSlot;
  }
}
```

5.3 Complex Type Concepts

In the case of simple web services, operation inputs and outputs can be wrapped completely in the fields of the related AgentAction or Predicate classes. In the <types> section of the WSDL file, however, web service providers may define custom structures composed of built-in XML types or further complex types (nested to arbitrary depth). Since such compound data structures cannot be represented by a single, primitive Java type, separate Java classes are created by the tool for each complex type. The Java type of the field corresponding to an input or an output parameter in the AgentAction/Predicate classes can be a basic Java type (as listed in Table 1), or a reference to a Java class representing the compound structure, depending on whether it is a built-in XML type or a complex type message part.

For example, in the WSDL file of Google web services we can find the following complex type definitions for ResultElement and DirectoryCategory:

```
<types>
  <xsd:complexType name="ResultElement">
    <xsd:element name="summary" type="xsd:string"/>
    <xsd:element name="URL" type="xsd:string"/>
    <xsd:element name="directoryCategory"
          type="typens:DirectoryCategory"/>
    ...
  </xsd:complexType>
  <xsd:complexType name="DirectoryCategory">
    <xsd:element name="fullViewableName"
                       type="xsd:string"/>
    ...
  </xsd:complexType>
  ...
</types>
```

The WSDL2Jade tool generates the following two classes (getter and setter methods are omitted here):

```
class ResultElement implements Concept {
  java.lang.String summary = null;
  java.lang.String uRL = null;
  DirectoryCategory directoryCategory = null;
 ...
}
class DirectoryCategory implements Concept {
  java.lang.String fullViewableName = null;
  ...
}
```

Note that, Java classes belonging to complex types have to implement JADE's Concept interface (instead of AgentAction or Predicate interfaces), but fields are generated in the same way as was described at the AgentAction and Predicate classes.

Since web services can interpret structures containing SOAP Java types only, when a data structure needs to be forwarded to the web service (or structured web service results need to be sent back to the client agent), the proxy agent has to translate whole structures. For this reason the WSDL2Jade tool actually generates two classes for each complex type: one corresponding to the structure that can be used in ACL messages (shown above) and one corresponding to the structure that can be used in SOAP messages containing SOAP Java types only. To avoid confusion, these latter classes are named with a _SOAP postfix. Fields in these classes are named in the same way as in the corresponding JADE classes, but contain SOAP Java type fields (according to the associations in Table 1).

To make the conversion of whole objects easier for the proxy agent, in JADE classes two additional methods are generated: *getSOAPClone* and *set_SOAP*. Through them, a complete SOAP type clone object can be obtained from the set of JADE type fields, and all the JADE type fields can be filled from a SOAP type counterpart object. Both of these methods execute a sequence of elementary type conversions steps for class fields (as described before). Proxy agents can thus read and write structured data as easily as obtaining and setting primitive JADE and SOAP Java type fields.

The source code of the _SOAP classes belonging to *ResultElement* and *DirectoryCategory* complex types are shown below (getter and setter methods are omitted here)

```
class ResultElement_SOAP {
  java.lang.String summary = null;
  java.lang.String uRL = null;
  DirectoryCategory_SOAP directoryCategory = null;
  ...
}
class DirectoryCategory_SOAP {
  java.lang.String fullViewableName = null;
  ...
}
```

The *getSOAPClone*, *set_SOAP* methods of the ResultElement class is illustrated below (ommitting the previously listed fields here):

```
class ResultElement implements jade.content.Concept {
  ResultElement_SOAP getSOAPClone () {
    ResultElement_SOAP clone=new ResultElement_SOAP();
    clone.summary = this.getSummary_SOAP();
    clone.uRL = this.getURL_SOAP();
    clone.directoryCategory =
this.getDirectoryCategory_SOAP();
    ...
    return clone;
  }

  void set_SOAP(ResultElement_SOAP param) {
    setSummary_SOAP(param.summary);
    setURL_SOAP(param.uRL);
    setDirectoryCategory_SOAP(param.directoryCategory);
    ...
  }
  ...
}
```

5.4 XML Arrays and ACL Lists

Web service operations may require or return arrays of data as input or output parameters. Such structures of the web service are also declared in the `<types>` section of the WSDL file, where either arrays of built-in XML types or arrays of complex types can be declared. They can even be multi-dimensional.

In ACL SL messages, however, we cannot use arrays (there is no such structure), therefore we have to represent arrays used in SOAP messages by lists in JADE. In the corresponding Java classes the WSDL2Jade tool therefore assigns list type fields (implemented by jade.util.leap.ArrayList) to array type elements in the WSDL. The type of the objects wrapped by the list is the JADE Java type corresponding to the built-in

XML type (or the Java class created for the complex type) of which the array is composed. For example, in the case of int[] array in the WSDL, the corresponding list is allowed to contain java.lang.Integer objects.

In contrast to the ordinary getter and setter methods (operating simply on ArrayLists) the get_SOAP/set_SOAP methods expect and return arrays of the related SOAP Java type objects (corresponding to the built-in XML type). These are used in the communication with the web service. The latter methods, on the one hand, create arrays from lists, or lists from arrays, respectively; and, on the other hand, perform the elementary type conversions for each object (SOAP-JADE) contained in the array or list depending on the direction of the transformation. Note that in the case of an array of complex types the elementary type conversion uses the get_SOAPClone/set_SOAP methods of classes assigned to complex types. The proxy agent can thus access array type objects as simple as in the scalar case, produce web service required arrays from lists, and create lists from the web service result arrays.

For example, the following array of complex type declaration can be found in GoogleSearchResult complex type:

```
<xsd:complexType name="GoogleSearchResult">
 <xsd:element name="resultElements"
type="typens:ResultElementArray"/>
 ...
</xsd:complexType>

<xsd:complexType name="ResultElementArray">
 <xsd:complexContent>
  <xsd:restriction base="soapenc:Array">
   <xsd:attribute ref="arrayType"
arrayType="typens:ResultElement[]"/>
   <xsd:attribute ref="soapenc:arrayType"
         wsdl:arrayType="typens:ResultElement[]"/>
 ...
</xsd:complexType>

<xsd:complexType name="ResultElement">
 ...
</xsd:complexType>
```

The generated source code implementing WSDL arrays as lists is shown below:

```
class GoogleSearchResult implements Concept {
  ArrayList resultElements = null;
  void setResultElements (ArrayList param)
    { this.resultElements = param; }
  ArrayList getResultElements () { return
this.resultElements; }
  void setResultElements_SOAP (ResultElement_SOAP[]
param) {...}
  ResultElement_SOAP[] getResultElements_SOAP () {...}
```

The WSDL2Jade tool is also capable of processing multi-dimensional array declarations in the WSDL, although they occur rarely in practice. Multi-dimensional

arrays are represented by lists of lists in the JADE, where each list contains lists corresponding to the representation of one lower dimensional array. For example, the two-dimensional array int[][] is represented by a list of objects each containing a list of java.lang.Integer objects. Since *ArrayList*s cannot be nested directly in JADE (it is not possible to register such structures in the ontology), each list has to contain dedicated objects which may then wrap further lists inside. The two-dimensional integer array is thus represented by a list of *Integer1DArray* objects, where an *Integer1DArray* object contains a single list type field containing java.lang.Integer objects. These intermediate array-wrapper classes (such as *Integer1DArray*) are also created by the tool automatically. As described in the case of one-dimensional arrays, the related _SOAP getter and setter methods are generated in the multi-dimensional case as well. They expect and return multi-dimensional arrays of SOAP Java types in this case.

5.5 Agent Ontology and Registration

The set of Java classes corresponding to web service inputs (AgentActions), outputs (Predicates), and complex types used in web service messages (Concepts) is called the ontology of the proxy agent (and also the ontology of the web service).

When the proxy agent starts up, it has to register its ontology first to be able to serialize/deserialize objects in agent messages by JADE. In the ontology registration code each class has to be added to JADE's knowledge base by submitting the class name and schema type (among of AgentActionSchema, PredicateSchema, ConceptSchema) information, and each field has to be specified in the registered classes by submitting the field name, symbolic field type (such as STRING, INTEGER, or the referenced schema name) and cardinality data (in the case of lists; optional otherwise). Note that cardinality attribute is also used to indicate optional elements in the WSDL (`minOccurs="0"`).

A part of the ontology registration code (WebServiceOntology.java) generated by the WSDL2Jade for Google web services is shown below:

```
class WebServiceOntology extends Ontology {
  WebServiceOntology (Ontology base) {
    add(new AgentActionSchema
       ("GoogleSearchPortDoGoogleSearchAgentAction"),
       DoGoogleSearchAgentAction.class);
    as=(AgentActionSchema) getSchema
       ("GoogleSearchPortDoGoogleSearchAgentAction");
    as.add("key", (PrimitiveSchema)
getSchema(BasicOntology.STRING),
          ObjectSchema.OPTIONAL);
    ...
}
```

6. WSDL2Protégé

The WSDL2Protégé tool can be used to create the model of the web service ontology for the submitted WSDL file in the form of a Protégé project file, which can then be opened in Protégé knowledge engineering environment. The model can be visualized and edited on demand, furthermore, JADE agent code can be generated using the Ontology Bean Generator plug-in. Protégé is also applicable to export the web service ontology to various formats, such as RDF, OIL, DAML+OIL, or OWL.

The WSDL2Protégé tool utilises the same WSDL parser as the WSDL2Jade tool, but generates a different output from the WSDL description. In this section we will focus on the Protégé file generation part, therefore the description will be shorter to reduce space.

6.1 The Model

The generated Protégé ontology follows the pattern of the ontology generated by the WSDL2Jade tool. For input and output messages the conversion assigns Protégé frames (corresponding to classes) with slots (corresponding to fields) representing message parts. Frames assigned to operation input messages are called AgentActions, and frames created for output messages are called Predicates in accordance with the conventions used in agent technology and the WSDL2Jade tool. Since frame and slot names must be globally unique in the knowledge domain, frame names are composed of the port name and operation name strings (in the WSDL) which is followed by either the AgentAction or Predicate strings. Slot names start with the container frame name and this is followed by the part name as specified in the WSDL file. Web service invocation faults are represented by the ErrorPredicate frame.

6.2 XML Data Types and Protégé slots

Protégé supports four basic types for slots: BOOLEAN, FLOAT, INTEGER, STRING (and a type called ANY, which can hold a value with any basic type). This implies that message parts and complex type elements in the WSDL have to be represented by such slot types.

In Table 2 we summarized the designed Protégé slot representations of the built-in XML types (using ANY where the XML type cannot be represented correctly in Protégé) used by the WSDL2Protégé tool.

6.3 Complex Type Frames

Complex types defined in the WSDL are represented by frames with slots corresponding to the contained elements. Since complex type names are unique in the WSDL, frame names get the name attribute of the complex type simply, but slot names must still be prefixed with the container frame name. To message parts and complex

type elements referring to another complex type in the WSDL the WSDL2Protégé tool assigns slots with a special type called CLASS and restricts the slot's Allowed Parents facet to the frame corresponding to the referenced complex type. Other built-in XML type elements in complex type definitions are represented simply by slots with type associated according to Table 2.

Table 2. XML-Protégé type mapping

XML (WSDL)	Protégé slot type
string	STRING
integer	INTEGER
int	INTEGER
long	INTEGER
short	INTEGER
decimal	INTEGER
float	FLOAT
double	FLOAT
boolean	BOOLEAN
byte	INTEGER
dateTime	ANY
base64Binary	ANY
hexBinary	ANY
unsignedInt	INTEGER
unsignedShort	INTEGER
unsignedByte	INTEGER
time	ANY
date	ANY
anySimpleType	ANY

6.4 XML Array and Multiple Cardinality Slots

Array type fields in the WSDL are represented by slots with multiple cardinality (available in Protégé as a slot attribute), as with the list representation of arrays used by the WSDL2Jade tool. In the case of a one-dimensional array a single, multiple cardinality slot is created with type corresponding to the type of the array elements. For n-dimensional arrays, however, intermediate frames have to be created to represent 1, 2, ..., n-1 dimensional arrays where each frame contains a single, multiple cardinality, CLASS type slot referencing to the frame corresponding to the one-lower dimensional array (except for the one-dimensional case). For example, in the case of a two-dimensional integer array, an intermediate frame called Int1DArray is created with an INTEGER type, multiple cardinality slot, and the slot corresponding to the 2-dimensional array is implemented by a multiple cardinality, CLASS type slot with Allowed Parents: Int1DArray.

6.5 The Structure

Frames are organized in a predefined structure required by the Ontology Bean Generator plug-in to be able to generate JADE code. In this structure all the frames of the ontology are the child classes of a special frame called Concept. Frames representing complex types are directly inherited from Concept. Frames corresponding to operation input messages are the subclasses of the dedicated frame called AgentAction (which is a child class of Concept), frames representing operation output messages are the subclasses of the frame called Predicate (also extending Concept). Finally, a so-called AID class instance must be present for the successful agent code generation with the Ontology Bean Generator plug-in.

Figure 2 in section 3 shows the model generated by the WSDL2Protégé tool for Google web services in Protégé visual development environment. As it is seen in Figure 2, three AgentAction frames and three Predicate frames are created for the web service operations of the Google web services. Other frames representing complex types (GoogleSearchResult, ResultElement, DirectoryCategory) are inherited from the abstract Concept frame. Slots within the frames got a unique name and the associated Protégé type in accordance with Table 2. The cardinality of the slots is also set: single in the scalar case, but multiple in the case XML of arrays.

If we export the generated ontology from Protégé into OWL format, then we can publish the ontology. Part of the OWL representation of the *GoogleSearchPortDoGoogleSearchAgentAction* agent action together with the *key* property is the following:

```
<owl:Class
rdf:ID="GoogleSearchPortDoGoogleSearchAgentAction">
 <rdfs:subClassOf>
  <owl:Class rdf:about="#AgentAction"/>
 </rdfs:subClassOf>
</owl:Class>

<owl:FunctionalProperty rdf:ID=
   "googleSearchPortDoGoogleSearchAgentAction_key">
 <rdfs:range

rdf:resource="http://www.w3.org/2001/XMLSchema#string"/
>
 <rdf:type

rdf:resource="http://www.w3.org/2002/07/owl#DatatypePro
perty"/>
 <rdfs:domain

rdf:resource="#GoogleSearchPortDoGoogleSearchAgentActio
n"/>
</owl:FunctionalProperty>
```

The return value of the *GoogleSearchPortDoGoogleSearchAgentAction* agent action is the *GoogleSearchPortDoGoogleSearchPredicate*. Its OWL representation as generated by the WSDL2Agent tool and exported from Protégé is the following:

```
<owl:Class
rdf:ID="GoogleSearchPortDoGoogleSearchPredicate">
 <rdfs:subClassOf rdf:resource="#Predicate"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="googleSearchPort
               DoGoogleSearchPredicate_return_value">
 <rdf:type rdf:resource=
 "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
 <rdfs:domain rdf:resource=
     "#GoogleSearchPortDoGoogleSearchPredicate"/>
 <rdfs:range rdf:resource=
     "http://www.w3.org/2002/07/owl#Class"/>
 <protege:allowedParent
rdf:resource="#GoogleSearchResult"/>
</owl:ObjectProperty>
```

7. Web Service Integration Demo Application

WSID (Web Service Integration Demo) system [17] composed of agents and web-services illustrates the application concept of the WSDL2Agent tool. The Google web service used in the previous sections demonstrates how the proxy agent is generated, while the WSID system shows how an agent can integrate existing web services and add new functionality to them by combining them.

The WSID Agent uses two existing web services: the FinancialFunctions web service and the NumberSpeller web service, both of them publicly available [18]. The FinancialFunctions web service provides several different operations to calculate some finance related numeric information such as deposit future value, deposit present value, etc., according to the submitted input parameters (value, rate, payment, etc.). The NumberSpeller web service simply transforms the submitted integer number to a spell form in the specified language. The Intelligent Financial Agent (IFA) adds to existing financial web services the possibility to make textual reports in the user's language by combining it with the NumberSpeller web service.

The WSID system is shown in Figure 4. The IFA has a web front-end, but client agents can also directly access its services.

Both web services have their own wrapper agent in the agent system. The user can select from a web page the financial operation which he wants to do and the language for the textual report. This information is sent to the central node of the system: the Intelligent Financial Agent. The IFA first invokes the FinancialFunction Wrapper Agent and then, after having received the numeric result, sends it to the NumberSpeller Wrapper Agent. This way the IntelligentFinancial agent is able to provide a financial textual report in the user's language on the web page.

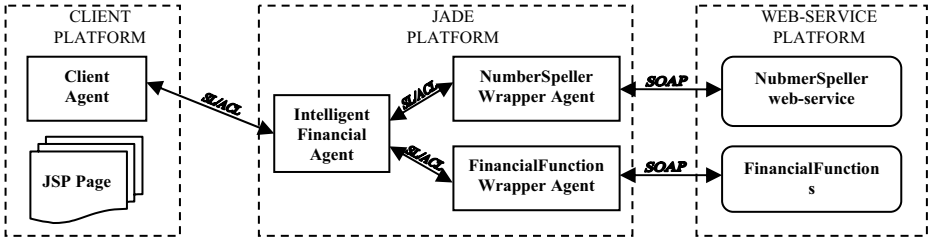

Fig. 4. The web service integration demonstration application architecture

8. Related work

Our work was partly stimulated by similar work within the Integrating Web Services Working Group of the Agentcities project [19] as well as an initiative for a similar interest group of the World Wide Web Consortium. This working group proposed a gateway approach [21][1] for the problem of agent and web services integration, and decomposed the problem into two parts: 1) software agents utilize web services, 2) software agents offer web services. The WSDL2Jade tool was implemented independent of the gateway approach in order to allow the deployment of mass amount of agents in the Agentcities testbed from existing web service applications. Moreover the WSDL2Jade tool is the first and most elaborated implementation to generate proxy agents for the utilisation of web services. It is expected that the JADE system will include similar component in the future.

The WSDL2Protégé part of the tool can be used to generate the basis for the Web Service Modelling Framework (WSMF) elements of semantic web services [11]. The WSDL2Protégé tool helps to translate a WSDL description to a Protégé project file and load in into Protégé where a semantic enrichment can be done easily. Once the semantic enrichment is complete, the goal repository, the ontology of the content language for the communication between proxy agents and mediators, as well as the skeleton of a proxy agent code can be derived [22].

The WSDL2DAMLS tool described in [20] can also be used to migrate web service descriptions to semantic web service descriptions, because it can directly generate a DAML-S description form a WSDL file. The WSDL2DAMLS tool also needs human intervention, because the WSDL file does not contain the needed semantic information. Although the WSDL2DAMLS tool produces a DAML-S file, it does not support the creation of the elements of the WSMF model.

9. Conclusions

The WSDL2Agent tool helps to integrate existing web services into agent systems. It was developed to help the deployment of mass amount of agent systems in the Agentcities worldwide testbed [19]. The WSDL2Jade part of the tool is the first and most elaborated implementation to generate proxy agents for the utilisation of web services. The WSDL2Protégé part is the only tool to translate a WSDL description to a Protégé project in order to support its semantic enrichment. In this paper we gave a detailed technical insight into the operation and usage of the tool.

The WSDL2Agent tool can link agent systems to other technologies as well. We have investigated in [22] how the tool can be used to migrate web services into the semantic web services world [10][11]. The WSDL2Agent tool is useful in grid environments as well. Users have reported to use the WSDL2Agent in grid systems by converting GWSDL files into WSDL descriptions and then applying the WSDL2Agent tool on the WSDL files. This way WSDL2Agent helps the integration of grid applications into agent systems. The WSDL2Agent tool therefore becomes more and more important by supporting the integration of existing systems (either web services or grid applications) into agent applications.

We are planning to investigate how to give better support for semantic web services and grid applications. Currently WSDL2Agent supports WSDL1.1. Support for the recently published WSDL 2.0 specification is also planned.

Acknowledgement

The authors wish to acknowledge the collaboration of the partners in the projects where the development of WSDL2Agent was started. These partners are AITIA Rt., Széchényi National Library, T-Systems Dataware Ltd., and the core partners of the Agentcities project. We also thank Eric Pantera for developing the WSID application.

László Zsolt Varga, Ákos Hajnal, Zsolt Werner
Computer and Automation Research Institute
Kende u. 13-17
1111 Budapest
Hungary
e-mail: {laszlo.varga|ahajnal|werner}@sztaki.hu

References

[1] Lyell, M., Rosen, L., Casagni-Simkins, M., Norris. D.: “On software agents and web services: Usage and design concepts and issues” In Proc. of the 1st International Workshop on Web Services and Agent Based Engineering, Sydney, Australia, July 2003.

[2] Maximilien, E.M., Singh, M.P.: “Agent-based architecture for autonomic web service selection” In Proc. of the 1st International Workshop on Web Services and Agent Based Engineering, Sydney, Australia, July 2003.

[3] Web Services Architecture, W3C Working Group Note 11 February 2004, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[4] Foundation for Intelligent Physical Agents: “FIPA ACL Message Structure Specification”, http://www.fipa.org/specs/fipa00061/, (2002)

[5] Gennari, J., Musen, M., Fergerson, R., Grosso, W., Crubézy, M., Eriksson, H., Noy, N., Tu. S.: “The evolution of Protégé-2000: An environment for knowledge-based systems development” International Journal of Human-Computer Studies, 58(1):89-123, 2003.

[6] van Aart, C.J., Pels, R.F., Giovanni C. and Bergenti F.: “Creating and Using Ontologies in Agent Communication” Workshop on Ontologies in Agent Systems 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, 2002.

[7] Wooldridge, M., Jennings, N.R.: “Intelligent Agents: Theory and Practice” The Knowledge Engineering Review, 10(2), 115-152., 1995

[8] Dale, J., Mamdani, E.: “Open Standards for Interoperating Agent-Based Systems” In: Software Focus, 1(2), Wiley, 2001.

[9] Foster, I., Jennings, N. R., Kesselman, C.: “Brain meets brawn: Why Grid and agents need each other” Proceedings of the 3rd International Conference on Autonomous Agents and Multi-Agent Systems, New York, USA, 8-15., 2004.

[10] OWL-S Coalition “OWL-S 1.1 Release” http://www.daml.org/services/owl-s/1.1/ 2004

[11] Bussler, C., Maedche, A., Fensel, D.: “A Conceptual Architecture for Semantic Web Enabled Web Services” ACM Special Interest Group on Management of Data: Volume 31, Number 4, Dec 2002.

[12] Foundation for Intelligent Physical Agents: “FIPA SL Content Language Specification”, http://www.fipa.org/specs/fipa00008/, (2002)

[13] Bellifemine, F., Poggi, A., Rimassa, G.: "JADE - A FIPA-compliant agent framework", In Proc. of the Fourth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (PAAM'99), London, UK, (1999) pp. 97-108.

[14] Google Web API, http://www.google.com/apis/

[15] W3C Working Draft "XML Schema Part 1: Structures", "XML Schema Part 2: Datatypes", http://www.w3.org/TR/xmlschema-1/, http://www.w3.org/TR/xmlschema-2/

[16] WebServices – Axis, http://ws.apache.org/axis/

[17] Web Service Integration Demo, http://sas.ilab.sztaki.hu/wsid/

[18] Alphabeans web services (See the "Run the demos" section at the URL below.) http://www-106.ibm.com/developerworks/webservices/demos/alphabeans/

[19] Willmott, S.N., Dale, J., Burg, B., Charlton, P., O'brien, P.: "Agentcities: A Worldwide Open Agent Network", Agentlink News 8 (Nov. 2001) 13-15,

http://www.AgentLink.org/newsletter/8/AL-8.pdf

[20] Paolucci, M., Srinivasan, N., Sycara, K., Nishimura, T.: “Toward a Semantic Choreography of Web Services: From WSDL to DAML-S” Proc. of the First International Conference on Web Services (ICWS'03), Las Vegas, Nevada, USA, June 2003, pp 22-26.

[21] Agentcities Task Force. Integrating Web Services into Agentcities Recommendation. http://www.agentcities.org/rec/00006/actf-rec-00006a.pdf, 2003.

[22] Varga, L.Z., Hajnal, A., Werner, Z.: "An Agent Based Approach for Migrating Web Services to Semantic Web Services", Lecture Notes in Computer Science Vol. 3192, C. Bussler, D. Fensel (Eds.), Artificial Intelligence: Methodology, Systems, and Applications 11th International Conference, AIMSA 2004, Varna, Bulgaria, September 2-4, 2004, Proceedings, pp. 371-380., ISBN-3-540-22959-0

Information about Software

Software is available on the Internet as

prototype version

Internet address:

Description of software: The WSDL2Agent tool is available on-line on the Internet. The input to the WSDL2Agent tool is the WSDL file of a web service. The WSDL file can either be uploaded to the web site of the tool or specified with an URL. The tool provides two types of output in a downloadable zip file. The WSDL2Jade part of the tool generates the code of a proxy agent that makes the web service available in multi-agent environment. The WSDL2Protégé part of the tool generates project file for the Protégé ontology engineering tool in which the ontology of the web service can be visualized, edited or exported to various formats.

Online availability address: http://sas.ilab.sztaki.hu:8080/wsdl2agent/

Contact person for question about the software:

email: wsdl2agent@sas.ilab.sztaki.hu