AGENTCITIES TECHNICAL RECOMMENDATION

Integrating Web Services into Agentcities Recommendation

Agentcities Technical Recommendation Document
actf-rec-00006, *28 November, 2003*

Authors (alphabetically):
Jonathan Dale, Fujitsu
Akos Hajnal, SZTAKI
Martin Kernland, Whitestein Technologies AG
Laszlo Zsolt Varga, SZTAKI

Status

Draft

This version: http://www.agentcities.org/rec/00006/actf-rec-00006a.html
Latest version: http://www.agentcities.org/rec/00006/

Abstract

This document describes how to make Web Services available to agents in an Agentcities environment and how to make agent-based services available to Web Service servers in a Web Services environment.

Contents

1 Introduction

This document serves to demonstrate how FIPA agent-based service can be integrated into a Web Services environment and vice versa. The document covers the following aspects:

- Accessing Web Services through an agent-based service gateway, and,
- Accessing agent-based services through a Web Services-based service gateway.

1.1 Background and Motivation

Agentcities is an ideal test bed for developing and deploying agent services within, where real world situations can be experienced. In Agentcities, agent platforms that conform to the FIPA standard for software agents [DALE01] to communicate can cooperate and share agent services worldwide.

However, the majority of existing services and information technology (IT) systems are not represented in the Agentcities Network, as they do not base their interoperability model on FIPA. For example, one of the emergent service architectures which is being rapidly deployed across IT systems is Web Services. The syntactic and semantic differences between Web Services and FIPA agent-based services prevents their seamless interoperation.

The following describes the advantages that both agent-based services and Web Services-based services could enjoy if they could interoperate:

- Accessing Web Services through an agent service (*enhanced functionality*)
  If agents inside the Agentcities Network could access and use Web Services, then agent developers would profit from this new functionality. Agent applications could combine these Web Services and offer them to other agents as extended services.
- Accessing agent services through a Web Service (*enhanced credibility*)
  If Web Service clients and servers could access and use agent services, then agent developers would be able to offer the benefits of agent services to a Web Service environment. This could demonstrate the first generation of intelligent Web Services and would open the Agentcities Network and its potential to Web Services developers.

1.2 Acknowledgements

The authors would like to thank all of the contributors to the *Integrating Web Services into Agentcities Working Group*, especially Margaret Lyell.

2 Reference Model

2.1 Components

The reference model for allowing Web Services and agent-based services to interact is given in Figure 1.

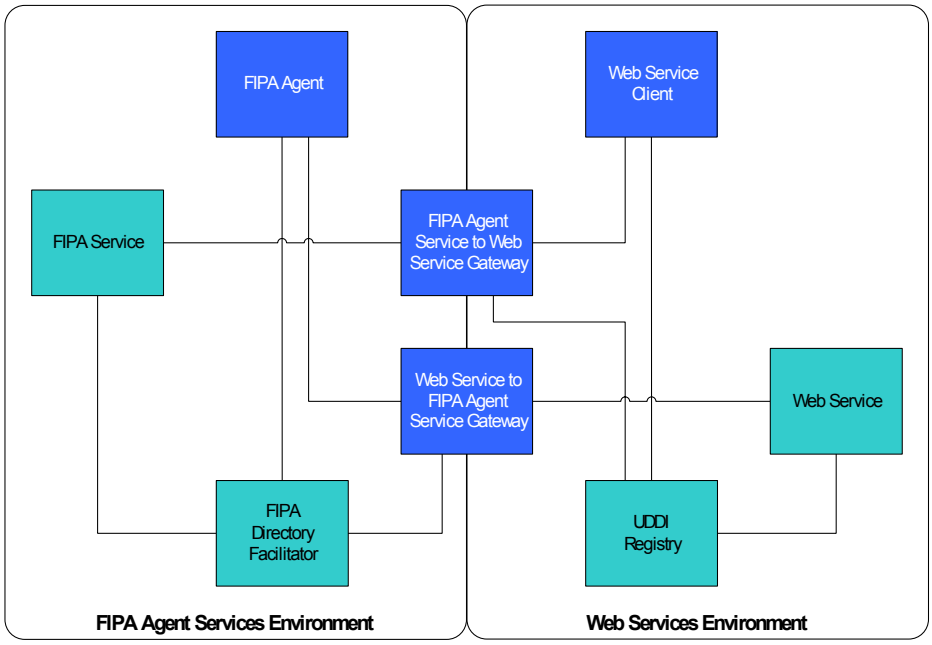

Figure 1: Web Services and Agent Services Reference Model

In this diagram, we assume two environments:

- A FIPA agent service environment
  This environment comprises agents which are compliant to the FIPA 2000 standard for interoperating software agents [DALE01] [FIPA2000].

- A Web Service environment
  This environment comprises Web Service clients and servers which comply with the specifications for SOAP [SOAP00], WSDL [WSDL01] and UDDI [UDDI02].

And the following logical components

- A FIPA Agent which can take advantage of a FIPA Service[1] by querying a FIPA Directory Facilitator.

- A Web Service Client which can take advantage of a Web Service by querying with a UDDI Registry Server and then invoking the appropriate SOAP method.

- A FIPA Service to Web Service Gateway which allows a FIPA Agent to access Web Services seamlessly.

- A Web Service to FIPA Service Gateway which allows a Web Service Client or Server to access a FIPA Service seamlessly.

[1] The distinction between a FIPA Agent and a FIPA Service is purely for illustrative purposes to distinguish between a service producer and a service consumer; in reality, both are provided by agents, but the service should not be confused with the identity of the agent.

2.2 Mapping Web Services into Agent Services

In this instance, the Web Service to FIPA Service Gateway (WStFSG) acts as a bridge between FIPA Agents that wish to access Web Services in a transparent fashion. The WStFSG has the following logical responsibilities (refer to Figure 1):

- (*Optional*) Query known local UDDI registry services and produce `df-agent-description` mappings which the WStFSG automatically registers with known FIPA Directory Facilitators. This allows FIPA Agents to discover Web Services inside its own environment.

- (*Required*) Intercept `REQUEST` messages from FIPA Agents which are intended for Web Service servers and perform the necessary translation of the data from FIPA ACL into a SOAP method invocation. The WStFSG performs the SOAP method invocation on behalf of the FIPA Agent.

- (*Required*) Intercept the return of SOAP method invocation and perform the necessary translation of the data into FIPA ACL inside an `INFORM` message. The WStFSG then forwards this message to the calling FIPA Agent as if it were a response from a FIPA Service.

Obviously, the WStFSG can only support one-shot Web Services; in the future, more complicated service invocations may not be useable by the WStFSG.

2.3 Mapping Agent Services into Web Services

In this instance, the FIPA Service to Web Service Gateway (FStWSG) acts as a bridge between Web Service Clients that wish to access FIPA Services in a transparent fashion. The FStWSG has the following logical responsibilities (refer to Figure 1):

- (*Optional*) Query known local FIPA Directory Facilitators and produce UDDI registry descriptions which the FStWSG automatically registers with known UDDI registry servers.

- (*Required*) Intercept SOAP method invocations from Web Service Clients which are intended for FIPA Services and perform the necessary translation of the data into FIPA ACL inside a `REQUEST` message. The FStWSG then forwards this message to the FIPA Service as if it were a request from a FIPA Agent.

- (*Required*) Intercept `INFORM` messages from FIPA Services which are intended for Web Service Clients and perform the necessary translation of the data from FIPA ACL into a SOAP method return. The FStWSG performs the SOAP method return on behalf of the FIPA Service.

Obviously, the FStWSG can only support one-shot Web Services; in the future, more complicated service invocations may not be useable by the FStWSG.

2.4 Example Scenario

As an example scenario, consider a FIPA Service which acts as a simple currency converter. It is able to handle a `REQUEST` message with two currencies as input and the result is an `INFORM` message with the exchange rate between the two currencies.

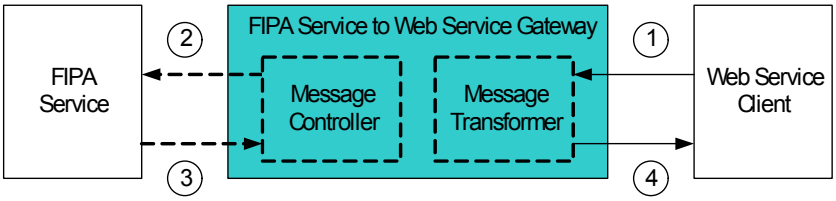

Figure 2: Gateway Interactions[2]

In Figure 2, a FIPA Service is exposed as a Web Service using a FStWSG. A Web Service Client uses this exposed service by sending a SOAP request message to the gateway (step 1). The following is an example representation of this SOAP message (note the method `getRate` with the arguments `USD` and `CHF`):

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getRate soapenv:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
      <arg1 xsi:type="xsd:string">USD</arg1>
      <arg2 xsi:type="xsd:string">CHF</arg2>
    </getRate>
  </soapenv:Body>
</soapenv:Envelope>
```

The FStWSG then transforms this SOAP message into a FIPA ACL message and sends it to the targeted FIPA Service (step 2). Since the ACL message communication is performed in an asynchronous fashion, the FStWSG needs to keep track of this conversation with the FIPA Service and raise a timeout exception in the case that the it does not respond in a timely manner. The following is an example of the `REQUEST` message that the FStWSG sends to the FIPA Service on behalf of the Web Service Client:

```
(REQUEST
  :sender (agent-identifier
    :name Exchange12@VOYAGER2:1099/JADE
    :addresses (sequence http://voyager2:7776/acc))
  :receiver (set (agent-identifier
    :name exchange@voyager2:7774/JADE
    :addresses (sequence http://voyager2:9999/acc)))
  :content "USD-CHF")
```

Once received, the FIPA Service processes the FIPA ACL message and returns the results inside an `INFORM` message (step 3):

```
(INFORM
  :sender (agent-identifier
```

[2] The illustrated components of the gateway (Message Transform and Message Controller) are architectural proposals since both gateways (FStWSG and WStFSG) need to transform messages and to control the sending and receiving of the messages.


```
    :name exchange@voyager2:7774/JADE
    :addresses (sequence http://voyager2:9999 ))
  :receiver (set (agent-identifier
    :name Exchange12@VOYAGER2:1099/JADE
    :addresses (sequence http://voyager2:7776/acc)))
  :content "1.38")
```

The FStWSG receives this message, transforms it to a SOAP message and returns the answer as a SOAP message to the Web Service client (step 4):

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getRateResponse
      soapenv:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
      <getRateReturn xsi:type="xsd:float">1.38</getRateReturn>
    </getRateResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

In the opposite direction, from a FIPA Agent to a Web Service using the WStFSG would be very similar from the view of the messages and their content since both gateways hide the target service by wrapping it with the same interface as the requesting service.

3 References

[DALE01] Dale, J. and Mamdani, E., *Open Standards for Interoperating Agent-Based Systems*. In: Software Focus, 1(12), John Wiley and Sons, 2001. http://www.fipa.org/docs/input/f-in-00023/

[FIPA2000] *FIPA 2000 Specification*. Foundation for Intelligent Physical Agents, 2000. http://www.fipa.org/

[FIPA00037] *FIPA Communicative Acts Specification.* Foundation for Intelligent Physical Agents, 2002. http://www.fipa.org/specs/fipa00037/

[SOAP00] *Simple Object Access Protocol, version 1.1*. World Wide Web Consortium, 2000. http://www.w3.org/TR/SOAP/

[UDDI02] *Universal Description, Discovery and Integration of Web Services*, version 2.0. OASIS, 2002. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2

[WSDL01] *Web Services Description Language, version 1.1*. World Wide Web Consortium, 2001. http://www.w3.org/TR/wsdl

4 Annex A – Technical Details

4.1 Whitestein Technologies Gateway Implementation Details

The Web Services Agent Gateway (WSAG) allows existing agent services to be deployed as Web Services. Any agent running on a FIPA-compliant agent platform that is capable of communicating via FIPA ACL messages can be deployed as a Web Service.

WSAG consists of a gateway that transforms a SOAP call into any kind of agent conversation with FIPA ACL messages. The following diagram illustrates the communication of the gateway:

Figure 3: Whitestein Technologies' Web Services Agent Gateway

When a Web Service invokes a SOAP call on the gateway, the gateway transforms this synchronous call into an asynchronous message communication via FIPA ACL messages to the software agent(s) that provides the service. The gateway waits for the answer of the agent and can, if necessary, produce a timeout on the Web Service side if the agent does not respond within a configurable timeframe. When the answer arrives, the gateway forwards the response to the requesting Web Service client.

As conversations between agents are much more complex and semantically richer than the simple request/response of Web Services, no generic mapping of this communication is possible. However, communication from an agent to a Web Service can be automatically mapped (as described in section 4.2).

The implementation of the WSAG currently allows manual configuration of the problematic parts as a first step, so that an agent service can still be mapped to a Web Service. Future enhancements and semi-automatic mapping procedures can be developed by the community.

4.1.1 Gateway Configuration

Before communication can occur, the WSAG has to be set up and configured. In this section, a simple ping service (as used by all Agentcities agent platforms) will exemplify an agent service that is being deployed as a Web Service. The WSAG is a Web application running within a servlet container and as soon as it is loaded into the servlet engine, it can be configured with a standard Web browser. In order to deploy an agent service as a Web Service, the following information has to be provided:

- The invocation call signature
  A normal Java interface with the method structure needs to written and compiled. Here is the example for the aforementioned ping service:

```
public interface Ping {
  public String sendPing(String content);
}
```

  The next step is to execute the Agent Generator script which produces a set of files based on this Java interface.

- The structure of the conversation with the targeted agent service

Because of the aforementioned semantic gap between the agent communication and the simple request/response paradigm of Web Services, the conversation of the WSAG with the targeted agent has to be implemented manually. The conversation is not limited to one participating agent, so even more complex auctions or contract-net protocols can be implemented. The file with the name composed out of the interface name and the method name with the term `Agent` at the end is the one to alter. In this example it would be the file `PingSendPingAgent.java` and the code to change can be found in the method `sendPing` (same as in the interface).

Here is the changed code:

```
public void sendPing(String content) {
  MessageTemplate responseTemplate =
    MessageTemplate.MatchPerformative(
      ACLMessage.INFORM);
  addBehaviour(new DefaultBehaviour(this, responseTemplate));
  ACLMessage msg = new
  ACLMessage(ACLMessage.QUERYREF);
  msg.setContent(content); //content should be “ping”
  sendMessage(msg);
}
```

Two things are of importance: one, to match the perfomative of the expected answer, and, two, to send the correct performative of the initial message of the conversation. After the code changes, the second Agent Generator script can be executed and a .jar file is generated which should be uploaded to the WSAG.

- The address of targeted agent service
  In order for the WSAG to find the right agent service that should be deployed, the user needs to enter its agent address. The target agent can already be running on a platform and it will not be aware of whether the conversation is with the WSAG or a “normal” agent.

With this information, the gateway is able to do the transition from a synchronous SOAP call to an asynchronous ACL message communication.

4.1.2 Gateway Implementation

The WSAG is a Web application running within a servlet container[3] and is constructed from the following components:

- Gateway Controller
  This manages the whole gateway and feeds Web pages to the user. It consists of several servlets, JSPs and some helper classes.

- Gateway Agents
  These agents perform the actual transition from the SOAP call to the ACL message communication. Their implementation is partially generated by the Agent Generator script and completed manually by the developer.

- Axis
  This is the SOAP engine of the Apache project and is the entry point of all incoming SOAP communication with Web Services.

- JADE Main Container

[3] WSAG was developed and tested for the Apache Tomcat 4.x, but in general it should be able to run on any specification-compliant servlet engine.

The open source platform JADE was chosen for an internal agent platform because it is currently the most commonly used platform in the Agentcities Network. The generated Gateway Agent is deployed in this container.

4.2 SZTAKI Wrapper Implementation Details

Since the existing Web Services and agent systems use different protocols in their communications (typically Web Services use the SOAP protocol over HTTP and agents use ACL messages sent over IIOP or HTTP) client agents cannot access Web Services directly. For this reason some kind of wrapper solution is needed to convert agent requests into the appropriate Web Service operation call and Web Service results into agent communication language.

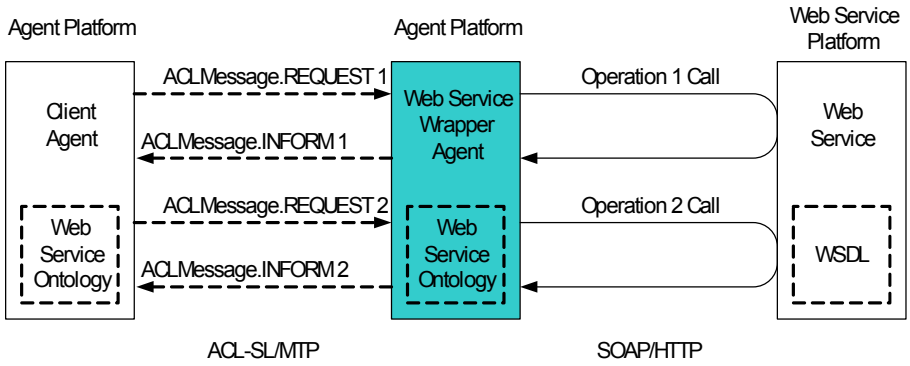

Figure 4: SZTAKI Service Wrapper Agent

The SOAP/HTTP side of the Web Service wrapper agent, that is, the interface of the Web Service call (location, operation names and parameter types) is completely defined by the WSDL file, therefore the code of the actual call is straightforward using the `org.apache.soap` library. On the FIPA ACL-SL/MTP side of the Web Service wrapper agent some important design decisions were made relating to:

- the type of FIPA ACL messages,
- the representation of Web Service operations, and,
- the representation of data types used in the Web Service operation parameters and results.

The WSDL2JADE tool processes the Web Service description file (WSDL) of an existing Web Service and automatically generates an agent ontology as well as agent deployment code implementing the Web Service wrapper agent for JADE platforms.

Due to the support of XML and WSDL processing, the tool is written in Java. The JADE agent platform was chosen to host the wrapper agents because it is widely used and enables Java-based agents in which environment the implementation of the Web Service calls are supported.

The following sections will discuss the decisions made and illustrate them with examples from the WSDL2JADE tool which was developed to automatically generate JADE agent code for the Web Service wrapper agent.

4.2.1 ACL Message Types

The `REQUEST` and `INFORM` communicative acts from [FIPA00037] were chosen when designing agent messages that are passed between client(s) and the wrapper agent since

they describe the most precisely the character of a Web Service call. The `REQUEST` message requests the Web Service wrapper agent to execute an agent action (which in fact is the Web Service operation call) and the `INFORM` message informs the client agent of the result of the Web Service operation.

Both message types in JADE may contain a single object which is a class instance in a serialized encoded form. An object in a request message corresponds to the agent action, an object in an inform message is a predicate. Class instances in `REQUEST` messages must extend the `AgentAction` class of JADE and objects in `INFORM` messages must extend the `Predicate` class.

4.2.2 Representation of Web Service Operations

Client agents can call Web Services by sending a `REQUEST` message to the wrapper agent to execute an agent action. A client `REQUEST` message must contain information about which operation of the Web Service should be called and what are the input parameter values. Therefore, in the ontology between the client agent and the wrapper agent, a unique agent action named `OperationNameAgentAction` must be defined for each operation. The instance type of the Java class representing the agent action will identify the operation and its field values in the object related to the input parameters of the operation call.

Similarly, predicates are defined named `OperationNamePredicate` for the result of the Web Service operation call. The predicates are returned in an `INFORM` message to the client agent.

For example, when a client wants to call a Web Service operation named `add`, it first creates a new `AddAgentAction` class instance, fills its fields according to the input parameters of the operation and sends this object to the wrapper agent in a `REQUEST` message. When the wrapper agent gets this message, it first decides which operation this class instance belongs to (by comparing the received class instance to the known operation agent action classes), extracts the input parameters from the class instance and calls the related Web Service operation. When the Web Service operation call returns, the wrapper agent creates a new `AddPredicate` class instance, fills its field(s) with the Web Service return values and sends this object back to the client agent who can then extract the results and the call operation of the client is completed.

Note that the Web Service invocation between the wrapper agent and the Web Service is synchronous, but the agent communication between the client and the wrapper agent is asynchronous. The client agent can match the received `INFORM` message to the corresponding `REQUEST` message with the help of the `in-reply-to` or the `conversation-id` parameters of the ACL messages.

4.2.3 Representation of Data Types

At the time when the tool was developed there was no standard way for associating XML Schema with Java data types[4], so a proprietary mapping was developed which included mapping such as, `xsd:int -> java.lang.Integer`, `xsd:string -> java.lang.String`, etc. When creating such mapping, Web Service call conventions should also be considered, for example, in the case of the `xsd:int` XML Schema type, `java.lang.Integer` should be used and not `int` (because if the `int` type is passed instead of `Integer`, then the SOAP call will not work).

Some of the XSD-Java bindings used are shown below:

[4] Recently, Sun's JAXB project has helped to deal with this problem.

XSD type	Java type
boolean	java.lang.Boolean
integer	java.lang.Integer
float	java.lang.Float
double	java.lang.Double
date	java.util.Date
hexBinary	byte[]

Since JADE accesses the fields of an object to be sent in a message via its getter/setter methods, when generating such agent Action/Predicate classes, the related getter and setter methods for the fields should be generated.

An example for an agent action class generated from a WSDL file is shown below:

```
<message name="InMessageRequest">
  <part name="numberToConvert" type="xsd:string"/>
  <part name="encodedlocale" type="xsd:string"/>
</message>
```

And the corresponding Java:

```
public class GetSpelledFormAgentAction implements
  jade.content.AgentAction {
  private java.lang.String numberToConvert;
  public void setNumberToConvert (java.lang.String param){
    this.numberToConvert = param; }
  public java.lang.String getNumberToConvert () {
    return this.numberToConvert; }
  private java.lang.String encodedlocale;
  public void setEncodedlocale (java.lang.String param) {
    this.encodedlocale = param; }
  public java.lang.String getEncodedlocale () {
    return this.encodedlocale; }
}
```

In JADE, before an agent starts, it has to register the relevant ontology, that is, in our case the above AgentAction/Predicate pairs of classes. Since JADE cannot explore object fields when a new classes in the ontology is registered, it must be done manually by going through all the fields in the class and registering the field names and the associated types individually. JADE has five built-in data types which may be used for this. When registering a field, JADE assumes that the underlying Java field has the Java type as shown below:

JADE type	Java type
BOOLEAN	java.lang.Boolean
INTEGER	java.lang.Integer
FLOAT	java.lang.Float
DATE	java.util.Date
BYTE_SEQUENCE	byte[]

As a consequence of the above restrictions all basic WSDL data constructs must be mapped to the above five basic JADE data types. Derived WSDL data types must be mapped to JADE data types derived from the basic JADE data types with the help of structures and lists.

Some Java code for registering AgentAction is shown below.

```
add(new AgentActionSchema("GetSpelledFormAgentAction"),
  GetSpelledFormAgentAction.class);
as = (AgentActionSchema) getSchema("GetSpelledFormAgentAction");
as.add ("numberToConvert", (PrimitiveSchema) getSchema
  (BasicOntology.STRING));
```



```
as.add ("encodedlocale", (PrimitiveSchema) getSchema
  (BasicOntology.STRING));
```

4.2.4 Services Provided by the Wrapper Agent to Client Agents

The Web Service ontology created for the wrapper agent shown above must be made available to client agents so that clients and wrapper agents are able to communicate with each other. Client agents send `REQUEST` messages and read `INFORM` messages, while wrapper agents read `REQUEST` messages and send `INFORM` messages. The ontology may be available as a downloadable jar file on the Web page of the wrapper service, for example, or some other solution may be implemented.

The agent code on the JADE platform, that is, the Java classes corresponding to agents, must extend the `Agent` class of JADE. After registering the ontology, the Web Service wrapper agent waits for incoming requests from clients in a blocking cycle. This is implemented as a behavior by extending the `CyclicBehaviour` class of JADE. When a `REQUEST` message is received, then the wrapper agent determines from the type of the request class the Web Service operation to which this `REQUEST` message is related. The input parameters of the operation can be decoded from the `REQUEST` and the Web Service operation can be invoked. The result of the Web Service invocation is encoded to the corresponding `Predicate` class of the ontology and then the predicate is sent to the client as an `INFORM` message.

4.2.5 Automatic Code Generation

To generate the code of the wrapper agent from WSDL, the following steps must be followed:

1. The operations contained in the WSDL file must be explored. For the input and output messages of each operation the related `AgentAction` and `Predicate` classes must be created with fields corresponding to the input and output parameters.

2. The agent must register these classes as the ontology of the wrapper agent

3. Agent code must be generated to be able to receive requests, dispatch the request to the appropriate operation, decode input parameters, call the Web Service, then encode and return the `INFORM` message. Because the above described transformations have a template, the agent code and the ontology for all Web Service invocations can be generated from the WSDL automatically.

5 Change Log

5.1 Version A: 22 September, 200311/28/2003

Page 1: Initial version