



## *The Ontology Agent*

*Operational guide*

*The AgentService Team*

*Copyright @ 2007 – l.i.d.o. – DIST University of Genoa*

## **Abstract**

In this guide the functioning of the *Ontology Agent* is described. In particular, the implementation (and interpretation) of the *FIPA Ontology Service* is shown. We introduce the two ontologies supported by the ontology agent and we show how an *AgentService* agent can exploit the ontology service.

# Index of Contents

Abstract .....	2
Index of Contents.....	3
1. Introduction .....	5
1.1. The features of the OA.....	5
1.2. The Ontology Agent lifecycle .....	6
1.3. OntologyAgentOntology and OntologyAgentOntologyEx.....	6
2. The Ontology Agent service.....	7
2.1. Introduction.....	7
2.2. The agent template .....	7
2.3. The configuration file .....	8
2.4. OAKnowledge .....	8
2.5. OAConfigurationKnowledge.....	9
2.6. The OAInitialize behaviour .....	9
2.7. The MainBehaviour.....	9
3. OntologyAgentOntology .....	10
3.1. Introduction.....	10
3.2. Predicates.....	11
3.3. Actions.....	14
4. OntologyAgentOntologyEx.....	14
4.1. Predicates.....	15
4.2. Actions.....	16
5. A simple example .....	17
5.1. Introduction.....	17

- 5.2. References to the Ontology Agent Service ..... 18
- 5.3. Finding the OA..... 18
- 5.4. Conversating with the OA..... 19
- 5.5. Sending a message and waiting for a reply ..... 19
- 6. Notes ..... 20
  - 6.1. Please be patient ..... 20

# 1. Introduction

The AgentService platform provides a set of default services which implement the FIPA specification regarding a standard agent architecture. AgentService furnishes a *yellow page service* (Directory Facilitator), a *white page service* (Agent Management System), and a *communication module*. At now the AgentService agents are encouraged to promote their services and interact with their peers by using the features of the *Ontology Agent*.

FIPA provides a specification for the Ontology Agent with some interesting features. Honestly, some of them are instead not much understandable or implementable; other interesting features are not mentioned, so we propose a pragmatic implementation of the FIPA Ontology Service, plus an extension of the service with (in our opinion) interesting and useful features.

In these sections we will show the AgentService Ontology Agent service: what the Ontology Agent does, what an agent can ask to the Ontology Agent (OA), and how the OA replies.

## 1.1. *The features of the OA*

Currently, the Ontology Agent running on AgentService supports a series of functions inherent the standard FIPA Ontology Service and a set of extensions.

In the following list we illustrate the supported FIPA Ontology Service features:

- The OA checks if a schema is an instance of a meta-class (intended as predicate, concept, primitive etc.).
- The OA checks if a given entity is a schema.
- The OA checks if an entity is a primitive.
- The OA verifies if an entity is a slot of a schema.
- In the same way it checks if an entity is a slot of a given schema.
- It verifies if a schema is a super schema of another schema.
- In the same manner it checks if a schema is a sub schema of a given schema.

Regarding the extensions introduced by the extended service:

- The OA offers the possibility to download or upload an ontology assembly to the ontology repository.
- The OA allows client agents to get the ontologies containing a given schema or slot.
- The client agent can retrieve the ontology type descriptor (See paragraph 4.2.5) for a given ontology.
- The OA checks if an ontology exists or not.

Several services ignored in the standard FIPA Ontology Service are implicitly supported by the ontology objects and instances of schemas.

### **1.2. *The Ontology Agent lifecycle***

As a brief abstract of the OA functioning, we just remark that the OA is a common AgentService agent which must be explicitly launched in an AgentService application.

The OA exposes a configuration file where the user can set several parameters as you can see in paragraph 2.3. The OA denotes two knowledge objects: one for the configuration parameters and one handling the collected ontology list. These two knowledge objects are managed by the two behaviours of the OA: the behaviour for the agent initialization and the behaviour which supports the conversation with a client agent (in paragraph 2.7).

In essence, the OA collects ontology assemblies, shares them within the agent community, reasons about them in order to answer to client questions.

### **1.3. *OntologyAgentOntology and OntologyAgentOntologyEx***

The OA supports two ontologies:

- *OntologyAgentOntology*: it contains the schemas describing the predicates and concepts defined and suggested by the FIPA specification.
- *OntologyAgentOntologyEx*: it contains the schemas describing the extensions which complete the AgentService Ontology Service.

A client agent can use both the ontologies because *OntologyAgentOntologyEx* is a specialization of *OntologyAgentOntology* or it can use only the basic service supported by *OntologyAgentOntology*. The OA adapts the conversation on the basis of the ontology supported by the client.

## 2. The Ontology Agent service

### 2.1. Introduction

In this section we will show the details about the Ontology Agent service: how to configure it, which knowledge objects are used and which behaviours are involved.

Essentially the OA is composed by these entities:

- *OntologyAgent* template: it is the agent template which manages the agent instantiation creating knowledge objects and behaviours and which handle incoming conversations.
- *oa.conf* file: the configuration file for the OA.
- *OAKnowledge*: it contains the list of collected ontologies.
- *OAConfigurationKnowledge*: it stores the current configuration of the agent.
- *OAINitialize*: a *one shot* behaviour which initializes the OA.
- *MainBehaviour*: the behaviour which manages a single client conversation.

There are also two worker classes:

- *Configuration*: it contains the data members describing the configuration parameters.
- *OAHelper*: it is a static class containing useful methods for the OA.

### 2.2. The agent template

Let's see the OA template, in order to understand the general functioning of the agent. The class is stored in *OntologyAgent.cs*. The *Configure* method overrides the one of the *AgentTemplate* class (the parent class of *OntologyAgent*).

This method reads the configuration file and sets the parameters in the collection *this.settings* (which can host every user defined parameter).

Note that the path of the configuration file is suggested in the *controller.Services.CreateAgent* method called in your application batch class (paragraph 5.2). The objects collected in *this.settings* are then read during the

initialization of the *OAConfigurationKnowledge* knowledge object (paragraph 2.5).

The *Initiate* method creates the two knowledge objects and creates (and runs) the two behaviours.

Strategic is the *HandleConversation* method which is in charge of accepting or refusing the incoming conversations and running the behaviours.

*HandleConversation* first checks if the peer agent supports the *OntologyAgentOntology* or *OntologyAgentOntologyEx* ontology. Only in this case the conversation is accepted. Then, a new *MainBehaviour* is created and launched, so for each conversation there is a single specific behaviour.

### 2.3. The configuration file

In order to run the Ontology Agent, the AgentService user must configure the configuration file. As reported below, there are three parameters to set.

```
<settings>
  <ontologyRepository>C:\AgentServiceOA\Repository</ontologyRepository>
  <debug>False</debug>
  <timeout>20000</timeout><!-- in milliseconds -->
</settings>
```

The parameter *ontologyRepository* contains the path of the folder which hosts the collected ontology assemblies. The parameter *debug* states the debug mode, namely if the agent has to print on the console execution information.

The attribute *timeout* sets the milliseconds the OA must wait before to close an inactive conversation.

### 2.4. OAKnowledge

The *OAKnowledge* (in *OntologyAgentKnowledge.cs*) maintains the list of known ontologies. Every ontology is described by an object named *Ontology\_Type\_Descriptor* (a concept described in the *OntologyAgentOntology*, see paragraph 4.2.5). *Ontology\_Type\_Description* has four properties:

- *Assembly*: it contains the path of the ontology assembly.
- *Name*: the name of the ontology.
- *TypeName*: the full type name of the ontology (that is the namespace plus the ontology class name).
- *References*: currently is unused, but it should contain the references to the current ontology.

## 2.5. *OAConfigurationKnowledge*

This knowledge object contains the configuration parameters of the Ontology Agent. It handles a *Configuration* object with the same members described in the configuration file (see paragraph 2.3).

## 2.6. *OAInitialize behaviour*

*OAInitialize* has two main goals: to subscribe the ontology agent to the Directory Facilitator and to construct the list of known ontologies.

Regarding the first task, it registers its service (*Ontology-Agent*), declaring the supported language (*FIPA-SL*), the supported ontologies (*OntologyAgentOntology* and *OntologyAgentOntologyEx*).

Then, it parses the folder containing the ontology repository, processes every existing assembly searching for an ontology. For each found ontology, an *Ontology\_Type\_Descriptor* is created and then it is added to the list.

Once these two tasks are accomplished, the behaviour terminates.

## 2.7. *The MainBehaviour*

*MainBehaviour* is the most important behaviour of the OA, because it implements all the intelligence of the agent.

We will analyze in details the supported features in the next section. In this paragraph we concentrate only on the behaviour functioning.

Essentially, *MainBehaviour* is a loop which terminates only if the conversation ends, or if the timeout expires (namely if the client agent doesn't interact for a certain time).

Inside the loop, the behaviour accepts incoming messages and try to reply (how it does, we will see later). For each conversation is instantiated a new behaviour, so every client agent has its private communication channel with the OA. Considering this, it is fundamental to regulate the concurrent access to the knowledge objects of the OA, as you can see in the body of the behaviour.

Each single service managed by the Ontology Agent is implemented in a private method. For each received message the OA replies with the outcome of the iteration (see sections 3 and 4 for details).

## 3. OntologyAgentOntology

### 3.1. Introduction

In this section we show the Ontology which should implement the FIPA Ontology Service. For every entity described by the ontology and supported by the AgentService OA the possible responses are reported.

Fig. 1 illustrates the ontology developed in Protégé (and shown using Jambalaya). Every entity described in the FIPA Service is implemented in the Ontology. If a particular action or predicate is not supported by the OA, the reply contains the information that the feature is not supported. In case of unsupported actions, the OA answers with the *Done* operator containing the *Not\_Understood* predicate if the conversation supports only the *OntologyAgentOntology* ontology, or *Not\_Implemented* if the conversation supports *OntologyAgentOntologyEx*. In case of unsupported predicate, the reply is one of aforementioned predicates (*Not\_Implemented* or *Not\_Understood*).

Table 3.1: schematic representation of the management of unsupported commands.

Action or predicate	Answer	Extendend ontology
Unsupported action	<i>Done</i> with <i>Not_Understood</i>	
	<i>Done</i> with <i>Not_Implemented</i>	✓
Unsupported predicate	<i>Not_Understood</i>	
	<i>Not_Implemented</i>	✓

The Table 3.1 schematically reports how the OA manages unsupported commands.

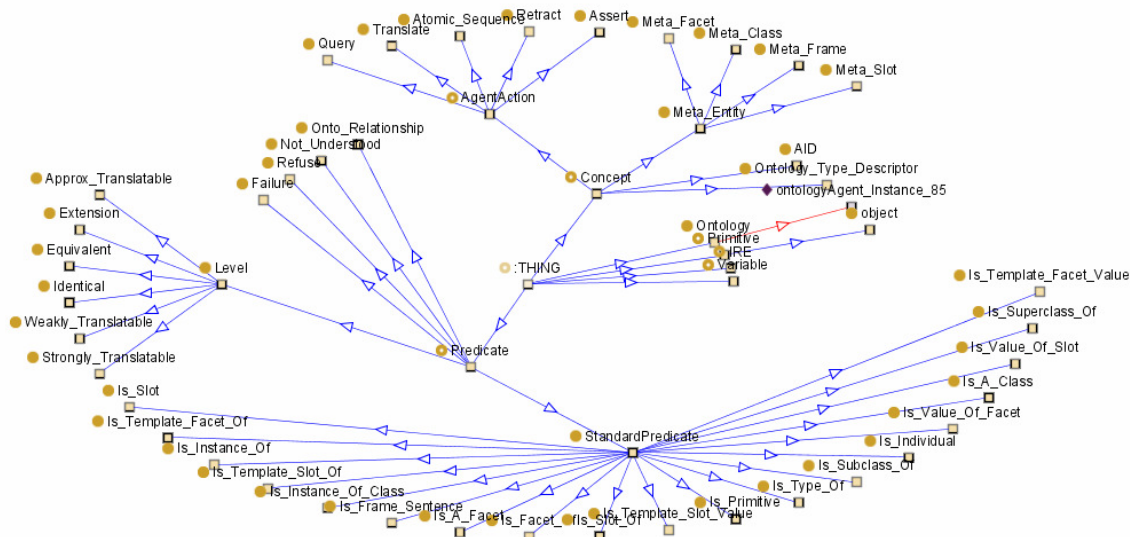


Fig. 1: the OntologyAgentOntology.

## 3.2. Predicates

The AgentService Ontology for the OA mainly supports a set of predicates used by agent to query the OA about entities, instances and references among them. In order to send a predicate as a question, the client agent must set a parameter during the message sending. This parameter informs the OA that the received predicate is a question. The OA answers by sending the boolean value of the received predicate.

```
Meta_Class c1 = new Meta_Class();
c1.Name = AgentService.Ontology.Schema.ConceptSchema.base_name;
Is_Instance_Of_Class instance = new Is_Instance_Of_Class();
instance.Class_Name = "Meta_Class";
instance.The_Class = c1;
conv.SendPredicate(instance, Performative.QueryIf);
```

The above code snippet shows how to manage the sending of a predicate as a question, by using *Performative.QueryIf*. If you want to send simply an assertion you can ignore this parameter, or use *Performative.Inform*.

### 3.2.1 Is\_Instance\_Of\_Class

This predicate is used in case the client agent wants to know if an entity is an instance of one of the ontological classes (predicate, concept...). The client must set the name of the investigated entity and it will receive in reply the *TrueProposition* if the entity is an instance of a (meta) class in whatever ontology stored in the OA repository.

Table 3.2: Is\_Instance\_Of\_Class.

<i>Is_Instance_Of_Class</i>	<i>TrueProposition</i>	If the entity is an instance of a class.
	<i>FalseProposition</i>	If the entity is not an instance of a class.

### 3.2.2 Is\_A\_Class

This predicate asserts that an entity is a class. In this case class is intended as schema contained in whatever ontology. The client agent must indicate the name of the presumed class and the OA replies with *TrueProposition* or *FalseProposition*.

Table 3.3: Is\_A\_Class.

<i>Is_A_Class</i>	<i>TrueProposition</i>	If the entity is a class.
	<i>FalseProposition</i>	If the entity is not a class.

### 3.2.3 Is\_Individual

*Is\_Individual* is not supported, but if an agent want to know if whichever object is an individual, then it can check the type of the object. Every object-individual must implement one of the interfaces contained in *AgentService.Ontology.Interfaces.AgentAction*, so it is easy to establish the kind of the object. See the code below as an example:

```

if (isIndividual.Argument is
AgentService.Ontology.Interfaces.AgentAction ||
isIndividual.Argument is AgentService.Ontology.Interfaces.Concept ||
isIndividual.Argument is
    AgentService.Ontology.Interfaces.ContentElement ||
isIndividual.Argument is
    AgentService.Ontology.Interfaces.ContentElementList ||
isIndividual.Argument is AgentService.Ontology.Interfaces.IRE ||
isIndividual.Argument is AgentService.Ontology.Interfaces.Predicate ||
isIndividual.Argument is AgentService.Ontology.Interfaces.Term ||
isIndividual.Argument is AgentService.Ontology.Interfaces.Variable)
{
    ...
}

```

### 3.2.4 Is\_Primitive

Another simple predicate which asserts that an entity is a primitive schema.

Table 3.4: Is\_Primitive.

<i>Is_Primitive</i>	<i>TrueProposition</i>	If the entity is a primitive.
	<i>FalseProposition</i>	If the entity is not a primitive.

### 3.2.5 Is\_Slot

An agent can interrogate the OA in order to check there exists a schema with a given slot. That agent have just to set the name of the hypothetical slot.

Table 3.5: Is\_Slot

<i>Is_Slot</i>	<i>TrueProposition</i>	If the entity is a slot of a schema
	<i>FalseProposition</i>	If the entity is not a slot of a schema.

### 3.2.6 Is\_Slot\_Of

It states that an entity is a slot of a given schema. The client agent must set the name of the slot and the name of schema.

Table 3.6: Is\_Slot\_Of

<i>Is_Slot_Of</i>	<i>TrueProposition</i>	If the entity is a slot of the given schema.
	<i>FalseProposition</i>	If the entity is not a slot of the given schema.

### 3.2.7 Is\_Subclass\_Of and Is\_Superclass\_Of

*Is\_Subclass\_Of* asserts that a given class is a specialization of another given class. On the other hand, *Is\_Superclass\_Of* states that a class is a generalization of another class. In both cases the reply can be *TrueProposition* or *FalseProposition*.

Table 3.7: Is\_Subclass\_Of and Is\_Superclass\_Of

<i>Is_Subclass_Of</i>	<i>TrueProposition</i>	If the given class is a subclass of another class.
	<i>FalseProposition</i>	If the given class is not a subclass of another class.
<i>Is_Superclass_Of</i>	<i>TrueProposition</i>	If the given class is a generalization of another class.
	<i>FalseProposition</i>	If the given class is not a generalization of another class.

In order to use the predicates the client agent must set the name of the supposed subclass and superclass.

### 3.3. Actions

The actions described in the ontology are simply managed by the direct sending of the action object, as shown in code below:

```
Download_Ontology download = new Download_Ontology();
Ontology_Type_Descriptor od = new Ontology_Type_Descriptor();

download.Ontology_Name = "DFOntology";
conv.SendAction(download);
```

In reply to an action, the OA can send two SL operators: *Done* and *Result*. *Done* is sent when the action outcome requires only the status of the transaction (a predicate which states the outcome). *Result* is used in case the action requires as a result an object or a set of objects.

At the moment, there are no implemented actions regarding the FIPA *OntologyAgentOntology*.

On the other hand, the *OntologyAgentOntologyEx* ontology supports several actions that extend the capabilities of the ontology agent. These improvements are shown in the next section.

## 4. OntologyAgentOntologyEx

The *OntologyAgentOntologyEx* is designed in order to extend the FIPA service and to fulfill the requirements and features of AgentService. Essentially, this ontology is aimed to satisfy some practical requirements in order to allow agents to discover and use ontologies. Fig. 2 shows the introduction of several actions and concepts.



### 4.1.2 Not\_Implemented

As shown in section 3.1 this predicate asserts that the sent action or predicate is not implemented.

## 4.2. Actions

In *OntologyAgentOntologyEx* several action are defined. They manage essentially the sharing of ontology assemblies.

### 4.2.1 Download\_Ontology

This action allows a client agent to download a given ontology. If the ontology exists, the OA sends the reply with the assembly object representing the ontology.

Table 4.2: Download\_Ontology

<i>Download_Ontology</i>	<i>Result</i> with the assembly	If the ontology exists.
	<i>Result</i> with <i>Failure</i>	If the assembly reflection raises an exception.
	<i>Done</i> with <i>Not_Understood</i>	If the ontology doesn't exist.

### 4.2.2 Upload\_Ontology

*Upload\_Ontology* transports an ontology assembly from a client agent to the OA which stores it in the repository.

Table 4.3: Upload\_Ontology

<i>Upload_Ontology</i>	<i>Done</i> with <i>Not_Understood</i>	If the uploaded assembly doesn't contain any ontology.
	<i>Done</i> with <i>TrueProposition</i>	If the ontology is successfully stored in the repository.
	<i>Done</i> with <i>FalseProposition</i>	If the ontology is already stored in the repository.

### 4.2.3 Get\_Ontologies\_Containing\_Schema

It returns all the ontologies which contain the given schema. These ontology assemblies are stored in an object *Ontology\_Info* (that is a concept of *OntologyAgentOntology*).

Table 4.4: Get\_Ontologies\_Containing\_Schema

<i>Get_Ontologies_Containing_Schema</i>	<i>Result with Ontology_Info</i>	The list of ontologies can be void.
---	----------------------------------	-------------------------------------

### 4.2.4 Get\_Ontologies\_Containing\_Slot

As the previous action, it returns a list of ontology assemblies having concepts with the given slot. Also in this case *Ontology\_Info* is returned and it can be void.

Table 4.5: Get\_Ontologies\_Containing\_Slot

<i>Get_Ontologies_Containing_slot</i>	<i>Result with Ontology_Info</i>	The list of ontologies can be void.
---------------------------------------	----------------------------------	-------------------------------------

### 4.2.5 Get\_Ontology\_Type\_Descriptor

It is an action that returns the type descriptor of an ontology, namely an instance of the concept *Ontology\_Type\_Descriptor* which holds: the assembly name, the ontology name, and the full type name.

<i>Get_Ontology_Type_Descriptor</i>	<i>Result with Ontology_Info</i>	The list of ontologies can be void.
-------------------------------------	----------------------------------	-------------------------------------

## 5. A simple example

### 5.1. Introduction

If you want to exploit the services of the AgentService Ontology Agent, you can take inspiration from this simple example.

We will shortly create a client agent which discover the OA asking the Directory Facilitator and than send some messages to the OA.

In this example we avoid the pedantic description of each step in the development of an AgentService application. We expect that you have already developed an application by using AgentService.

## 5.2. References to the Ontology Agent Service

It is important to include a reference to the ontology agent, both as assembly reference in your *Visual Studio* project and as assembly upload during the batch process that wake up the platform. So, please be sure to add a statement in the batch class for the assembly upload:

```
if (controller.PlatformState == PlatformState.Ready)
{
    controller.Modules.UploadAssembly("OntologyAgent.dll", true);
    IModuleError error =
        controller.Modules.UploadAssembly("AgentApplication.dll", true);
    if (error == null)
    {
```

*OntologyAgent.dll* is the assembly that contains the ontology agent, while *AgentApplication.dll* is your AgentService application.

You must also run the ontology agent, together with your agents:

```
if (controller.PlatformState == PlatformState.Running)
{
    controller.Services.CreateAgent(
        "AgentService.Ontology.OntologyAgent", "OntologyAgent",
        "oa.conf", false);

    controller.Services.CreateAgent("AgentApplication.ClientAgent",
        "Client", "", false);
    controller.Services.ActivateAllAgents();
```

Remember to add and configure the *oa.conf* file.

## 5.3. Finding the OA

Your client agent must track the OA in order to exploit its services. If the client agent doesn't know the name of the OA, it can ask the Directory Facilitator in order to get the name.

```
DFAgentDescription[] agents = this.Runtime.YPageSvc.SearchAgent(new
ServiceDescription("Ontology-Agent", ""));

AID OAAID = null; ;
foreach (DFAgentDescription agent in agents)
{
    OAAID = agent.Aid;
    this.Runtime.Console.AppendMessage("I found an ontology agent! " +
        agent.Aid.Name);
}
```

These are the statements which enables your agent to know the AID of the OA.

### 5.4. *Conversating with the OA*

It's time to establish a conversation between the client agent and the OA. Below you can find the statements which enable your agent to initiate a conversation with the OA.

```
IOntologicalConversation conv = (IOntologicalConversation)
    this.Runtime.MessageSvc.OpenConversation(OAAID,
        OntologyAgentOntologyEx.Instance);

while ((conv.State != ConversationState.Accepted) &&
        (conv.State != ConversationState.Active) &&
        (conv.State != ConversationState.Rejected) &&
        (conv.State != ConversationState.Closed))
{
    System.Threading.Thread.Sleep(10);
}
```

As you can see, it must to manage an ontological conversation, passing to the *OpenConversation* method the AID of the Ontology Agent and the instance of the ontology you are going to use.

### 5.5. *Sending a message and waiting for a reply*

If the OA accepts the conversation the client agent can exploit the services of the Ontology Agent. In the following statements you can take some inspiration for your client agent behaviour.

```
Download_Ontology download = new Download_Ontology();
Ontology_Type_Descriptor od = new Ontology_Type_Descriptor();
download.Ontology_Name = "DFOntology";

if (!conv.SendAction(download))
{
    this.Runtime.Console.AppendMessage("An exception is raised during
        the validation of the message content: " + conv.ValidationError);
}

AgentMessage msg = conv.WaitForMessage();

object content = msg.Body["ContentElement"];
if (content is AgentService.Ontology.SL0.Result)
{
    this.Runtime.Console.AppendMessage("Result");
    // Type here your code statements for the management
    // of the content results.
}
else if (content is Done)
{
    this.Runtime.Console.AppendMessage("Done:" +
        ((Done)content).Condition.GetType().ToString());
}
```

## 6. Notes

### *6.1. Please be patient*

The OA service is currently under development. Perhaps its services are not greeting your requirements, so please be patient. The AgentService staff is open to every suggestion and we are waiting for you on your web forum: <http://agentservice.lido.dist.unige.it/forum/>.

Moreover, even if every service of the OA has been tested, but please consider yourself as... release candidate tester...