



## *The Ontology Service*

*An introduction*

*The AgentService Team*

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

## Abstract

The aim of this guide is to allow users to exploit the *Ontology Service* of AgentService. We describe the whole life-cycle of an ontology, from the design by using Protégé, to the deployment in an AgentService platform.

An overview on the object model and the solution of the Ontology Service is done, commenting the main features of the described classes and namespaces.

# Index of Contents

Abstract .....	2
Index of Contents .....	3
1. Introduction .....	4
2. The ontological meta-classes .....	4
2.1. Attributes and facets .....	5
2.2. Schemas .....	5
2.3. The Ontology class .....	6
2.4. Facets design .....	8
2.5. Abstract descriptors and introspector .....	8
3. Designing an ontology .....	9
3.1. BasicOntology .....	10
3.2. Adding schemas and slots .....	10
4. Designing ontologies with Protégé .....	12
4.1. How to design an ontology .....	12
4.2. IRE .....	13
4.3. Ontology .....	13
4.4. Protege2AgentService .....	14
5. SL .....	14
6. Ontological conversations .....	15
6.1. How to open a conversation .....	16
6.2. Incoming conversations .....	16

## 1. Introduction

AgentService furnishes a support for ontologies which allows the ontology design and management, to represent knowledge bases and especially message contents.

An ontology is a exhaustive set of elements and represents a discourse domain. A discourse domain is composed by all that about which we speak and reason, considering a precise topic.

Ontologies for AgentService provide the elements necessary to manage a conversation among agents. Practically an ontology provides classes to instantiate and to use as message content and provides a series of rules and hints for a correct use of the ontology tool.

Through the ontology, the semantic and syntactic representation is strict and shared by agents, avoiding errors about interpretation and inconsistencies.

Moreover, the Ontology Service provides the validation of the message content, namely it is checked in order to test the conformity with the ontology rules (for example: the compulsoriness of a certain object property, the cardinality, various facets, etc...).

## 2. The ontological meta-classes

Every ontology furnishes a set of elements that belong to generic categories (*meta-classes*). An ontology contains:

- *Concepts*: they represent abstract or concrete objects and they are, practically, the subject or a complement of a sentence. Typical examples of concept are: person, dog, planet, job...
- *Predicates*: they are assertions that can be *true* or *false*. They involve concepts. An example of predicate is: (*owner: dog, person*), that is an assertion (true or false) which states that the given *person* is the *owner* of the given *dog*. Obviously a predicate must be instantiated creating instances of *person* and *dog* too.
- *Actions (AgentActions)*: they are particular concepts expressing commands conveyed by an agent to another agent. As an example: (*sell: book, person*) where an agent asks the *seller* agent to sell a *book* to a *person*.

- *IRE*: they are queries and represent questions. They holds a predicate containing a *variable object*. E.g: (*query: padrone, dogVariable*), which means the request to return the list of *dogs* with a given *owner*.
- *Variables*: they are used into IREs. They are undetermined objects which represent for example instances of concepts or aggregates of concepts.
- *Primitives*: they are primitive types as *string, float, boolean*, etc.
- *Aggregates*: they represent a list, array or whatever set of elements.

These entities are named *schema types*. When you design an ontology, it must render schemas which derive from the aforementioned schema types.

For example, an ontology can be equipped with the concept *person*, of which there is a schema derived from the schema type *Concept*; there will be the action *sell*, that is a schema of type *AgentAction*, etc...

## 2.1. Attributes and facets

Every schema can be equipped with *attributes* (also named *slots*). Attributes are similar to the properties or data members of C# classes. A schema attribute, unlike the properties of a C# class, is endowed with *facets* and *compulsoriness*. With the compulsoriness we can state if an attribute is *optional* or *mandatory*. As an example, the schema *person* could denote a mandatory attribute *name* and an optional attribute *age*. For every attribute we can also specify the *maximum* and *minimum cardinality* regarding the number of values of a slot.

Moreover, the user can define his facets by implementing the *Facet* interface (see paragraph 2.4).

## 2.2. Schemas

Each element belonging the discourse domain is then represented by a schema. Every schema derives from the *ObjectSchema* class. Into the namespace *AgentService.Agent.Ontology.Schema* there are the schema types relative to the concept, predicate, IRE, etc. When you want to add the *person* concept, it must suggest that *person* is a *ConceptSchema*; analogously for other schema types.

*ObjectSchema* and then the other schema types denote the following methods:

- `public static ObjectSchema GetBaseSchema()`: it returns a generic *ObjectSchema*. Every schema will return a generic object of its type.

- `public abstract void Add(string name, ObjectSchema slotSchema, int optionality)`: it adds a slot to the schema, passing the name, the schema type and optionality.
- `public abstract void Add(string name, ObjectSchema slotSchema)`: as the previous one, but in this case it is not given the optionality.
- `public abstract void Add(string name, ObjectSchema elementsSchema, int cardMin, int cardMax)`: it adds to the schema a slot endowed with maximum and minimum cardinality.
- `public abstract void Add(string name, ObjectSchema elementsSchema, int cardMin, int cardMax, string aggType)`: it adds a slot to the schema indicating the aggregate type (it is a sort of facet).
- `public abstract void AddSuperSchema(ObjectSchema superSchema)`: it adds to the schema a superschema, from which it inherits every slot.
- `public abstract void AddFacet(string slotName, Facet f)`: it links a facet to a given slot.
- `public abstract string GetTypeNames()`: it returns the schema name.
- `public abstract string[] GetNames()`: it returns the name of the slots that belong to the schema.
- `public abstract void Validate(AbsObject abs, Ontology onto)`: it validates an abstract descriptor (see paragraph 2.5) which represents an object, by following the ontology rules.
- `public abstract ObjectSchema GetSchema(string name)`: it returns the schema associated to a slot name.
- `public abstract bool ContainsSlot(string name)`: it checks if the schema contains the given slot.
- `public abstract bool IsMandatory(string name)`: it checks if the given slot is mandatory.
- `public abstract bool IsCompatibleWith(ObjectSchema s)`: it checks if the current schema is compatible with a given schema type.
- `public abstract bool DescendsFrom(ObjectSchema s)`: it checks if the schema descends from the given schema.
- `public abstract Facet[] GetFacets(string slotName)`: it returns all the facets for a given slot.
- `public abstract AbsObject NewInstance()`: it provides an instance of an abstract descriptor relative to the current schema.

### **2.3. The Ontology class**

Every ontology must derive from a class named *Ontology*, hosted in the namespace *AgentService.Agent.Ontology*.

The *Ontology* class stores the ontological schemas and links them to the C# classes that they represent (and that the programmers must implement when they create the message content).

The ontology class manages a list of *basic ontologies* from which the current ontology inherits the schemas. A basic ontology is useful because it allows the ontology developer to reuse schemas defined in other ontologies.

The *Ontology* class has the following constructors:

- `public Ontology(string name, Ontology Base, Introspector introspector)`: it instantiates an ontology with a given name, a given basic ontology, and a certain introspector (see paragraph 2.5).
- `public Ontology(string name, Ontology Base)`: it is similar to the previous one, but without indications about the introspector which, in this case, is the default introspector.
- `public Ontology(string name, Ontology []Base, Introspector introspector)`: like the first constructor, but in this case is given a list of basic ontologies.
- `public Ontology(string name, Ontology []Base)`: in this case the introspector is the default one.
- `public Ontology(string name, Introspector introspector)`: no basic ontologies are defined.
- `public Ontology(string name)`: it simply constructs an ontology with a name.

The ontology designer can use the following methods:

- `public string GetName()`: it returns the name of the ontology.
- `public void Add(ObjectSchema schema)`: it adds a schema to the ontology.
- `public void Add(ObjectSchema schema, Type Class)`: it adds a schema to ontology, also linking a C# class type.
- `public ObjectSchema GetSchema(string name)`: it returns the schema with the given name.
- `public Type GetClassForElement(string name)`: it returns the C# class linked to schema with a given name.
- `public AbsObject FromObject(object obj, Ontology globalOnto)`: it returns an abstract descriptor, for a given object and a given ontology.
- `public AbsObject FromObject(Object obj)`: as the previous one, but with no indications about the ontology.

## 2.4. Facets design

Into the namespace *AgentService.Agent.Ontology.Schema* there is the *Facet* interface the custom facets must implement.

This interface contains only one method:

- `void Validate(AbsObject val, Ontology onto)`: it receives the abstract descriptor representing the object to validate and the ontology containing the rules used to check it.

## 2.5. Abstract descriptors and introspector

An ontology represents in an unambiguous and strict way the rules which describe a given discourse domain. During the validation process it needs to compare the object (namely the instance of the class related to an ontological schema) to validate and the related schema. The validation process is totally hidden to the end-user but is introduced here in order to better explain the features of the introspector.

There is a gap between objects and schemas: the former are instances, the latter are abstract representations. To fill this gap the abstract descriptors are introduced: they are representations that join the features of schemas with the values of objects. Practically, they are schemas which contain slots instantiated with the values of the properties belonging to the objects.

In order to create these abstract descriptors, the validation process uses the method *FromObject*, passing the objects to validate. Regarding slots of primitive types or anyway with maximum cardinality 1, there are no problems to get the values of the object properties. Regarding aggregate slots with particular data structures different from the usual ones (ArrayList, array, etc.) the *FromObject* method needs a tool able to access these unusual data structures. In particular it needs the access methods of the data structure. For example, if is created an ontology with a slot that is a particular type of graph, the *Ontology* class (in particular the *FromObject* method) needs the methods to access the graph, if the data structure is not supported by the *default introspector*.

The default introspector grants the access to the usual data structure. If the user wants to support other data structures, must implements the *Introspector* interface hosted in the *AgentService.Agent.Ontology.Validation* namespace.

To implement this interface, the following method must be defined:

- `AbsObject Externalise(Object obj, ObjectSchema schema, Type Class, Ontology referenceOnto)`: the method receives

the object to convert, the schema related to the object, the class type, and the ontology.

The method is used within the *FromObject* method.

### 3. Designing an ontology

To design a custom ontology the relative ontology class must be derived from the *Ontology* class. For each element belonging the discourse domain and every attribute (slot), a string containing its name must be defined.

The set of these strings (defined as data members of the custom ontology class) originates the *ontology vocabulary*. To refer to a given schema, the correspondent vocabulary string must be used.

```
public class MusicShopOntology: Ontology
{
    public static readonly string ontology_name = "Music-shop-ontology";
    // VOCABULARY
    public static readonly string item = "Item";
    public static readonly string item_serial = "serial_number";
    public static readonly string cd = "CD";
    public static readonly string cd_name = "name";
    public static readonly string cd_tracks = "tracks";
    public static readonly string track = "Track";
    public static readonly string track_title = "title";
    public static readonly string track_duration = "duration";
}
```

These statements represent a fragment of the custom ontology describing a music shop; in particular we can see the derivation from the *Ontology* class, the string containing the name of the ontology and the vocabulary.

In the group of the schemas defined in the ontology, are defined:

- The generic concept *item* endowed with the attribute *serial\_number*.
- The concept CD (derived from *item*) with attributes *title* and *tracks* (an aggregate attribute containing several instances of the concept *track*).
- The concept *track*, with attributes *title* and *duration*.

The ontology must be endowed with the property *Instance* which provides the unique instance of the ontology.

```
private static Ontology theInstance = new MusicShopOntology();
public static Ontology Instance
{
```

```

    get
    {
        return theInstance;
    }
}

```

The ontology constructor must be private:

```
private MusicShopOntology(): base(ontology_name, BasicOntology.Instance)
```

As shown in the above statement, the constructor invokes the base class one, passing the ontology name and the instance of the basic ontology.

### 3.1. BasicOntology

In the *AgentService.Agent.Ontology* there is an ontology named *BasicOntology*. This ontology furnishes the schemas for the main primitive types, the widely used concept *AID* (the AgentService agent identifier), the schema for the FIPA-ACL message, and the schemas for the SLO operators (see section 5). The *BasicOntology* is enough for a typical usage of an ontology, but in case of particular requirements the *BasicOntology* can be extended or replaced by a totally new ontology (obviously also the name of this basic ontology can be modified).

### 3.2. Adding schemas and slots

Inside the private constructor of the ontology, the schemas are added and the respective slots are linked to them.

Considering the aforementioned *music shop ontology*, fragments of code are reported in the following:

```

Add(new ConceptSchema(cd), typeof(CD));
Add(new ConceptSchema(item), typeof(Item));
Add(new ConceptSchema(track), typeof(Track));

```

With the *Add* method we include a schema in the ontology. As you can see, three concepts are added, linking the related vocabulary string with the C# class. For example, we add the *ConceptSchema* with the same name contained in the vocabulary string *item*. We link the *Item* class which will be usually used during the implementation of the agent behaviours.

```

[Serializable]
public class Item: Concept
{
    private int serial_number;
    public int Serial_number
    {
        set
        {
            this.serial_number=value;
        }
    }
}

```

```
        get
        {
            return this.serial_number;
        }

        public Item()
        {
            serial_number=0;
        }
    }
}
```

Here is described the *Item* class. It has the *Serial\_Number* that represents the unique slot of the *item* concept. The slot name and the property name must be the same, even if the property name must have the first character in the upper case. The class must be *serializable* because their instances are sent as message content.

Moreover, considering the validation process, the *Item* class must implement the interface (empty) *Concept* (available in *AgentService.Agent.Ontology.Interfaces* with other interfaces describing the other fundamental elements of ontologies: predicates, actions, primitives...). Every class related to an ontological schema must implement the correspondent interface.

To link the slots to the schemas another *Add* method is used:

```
ConceptSchema cs = (ConceptSchema) GetSchema(item);
cs.Add(item_serial, (PrimitiveSchema)GetSchema(BasicOntology.integer),
        ObjectSchema.optional);
```

To the *item* schema, is assigned the *item\_serial* slot which is a *PrimitiveSchema*. This slot represents an integer value. The schema linked to the integer type is a *PrimitiveSchema* defined in the *BasicOntology*. Moreover the slot is made optional, by using the pointer represented by the data member *optional* in *ObjectSchema*.

This is all you need to implement an ontology. We can summarize the steps:

- Create a class for the custom ontology class, which derives from *Ontology*.
- Create a vocabulary containing the terms representing the names for each schema and attribute.
- Define an *Instance* property containing the unique instance of the ontology.

- Create a private constructor which calls the base constructor, passing the name of the ontology, possible basic ontologies, and possible introspector.
- In the body of the constructor:
  - o First, add the schema declarations.
  - o Then, add the slots.
- Create the C# classes representing the schema objects. Please be careful: the property names must be equal to the slots names (with the first character in upper case). These classes must implement the correspondent interfaces contained in *AgentService.Agent.Ontology.Interfaces*.

## 4. Designing ontologies with Protégé

The design of AgentService is a repetitive process that could be easily automatable. In the field of Computer Science ontologies there is a well-known, open-source, and easy to use ontology editor called *Protégé* ([www.protege.stanford.edu](http://www.protege.stanford.edu)).

The designer can develop his AgentService ontology exploiting the comfort of Protégé, starting from a template containing all the elements necessary to model custom ontologies. The user can save the developed ontology in an xml file and then run a small program named *Protege2AgentService* in order to generate C# classes describing the ontology and the assembly containing them.

### 4.1. How to design an ontology

First, you can open in Protégé the project file *TemplateOnto.pprj* containing the meta-classes which describe the concept, predicate, action...

The user must specialise these classes creating the schemas belonging the ontology. To create the concept *item*, you must right-click on the meta-class *Concept* and select *Create Class*.

To add a slot to the *item* concept you can click on the listbox *TemplateSlot* and select *Create slot*. Label the slot with a name, select the type, select required if the slot is mandatory and then suggest the maximum and minimum cardinality through *at least* and *at most*.

It is not possible to create two slots (of different schemas) with the same name, unless you want to reuse the slot (you can do this if, instead of selecting

*Create slot*, you select *Add slot*). This procedure can introduce unwanted changes because if you modify a shared slot, the modification becomes true in every schema having the slot.

If you want to name two different slots with the same label, than you must create the two slots with different name, select the Tab Widget *Slots*, select one of the slots, right-click and choice *Change Slot Meta-class*. Instead of the *Standard-slot*, check its specialization *AgentService-slot*. Now, if you open the configuration form of the slot (from the Tab Widget *Slots* or from *Classes*) you can see three new fields. In *AgentService-Name* you can insert the true name of the slot that could be the same of another slot. In this manner the Protégé constrain can be avoided.

Another new field inherited from *AgentService-slot* is *AgentService-TypedAggregate*. You can use this field if the slot has cardinality greater than 1 and the aggregate type is not standard (remember the *introspector* in the paragraph 2.5. If the aggregate is not standard, you must also create a schema describing the aggregate.

If the slot is not a primitive type, but it is a concept defined inside the ontology, in the *Value Type* field select *Class*, then in the new beneath field choice the schema representing the slot.

If the slot is represented by a schema contained in a basic ontology, you must create a fictitious schema (empty, without slots) representing the inherited schema and in the configuration form you must check the field *AgentService-Ignore*, in order to avoid the representation of this dummy concept during the code generation process (in *Protege2AgentService*, see paragraph 4.4).

## **4.2. IRE**

IREs are a tricky solution to compensate the current limitation to SL0 (see section 5). In the next future, when the complete SL language is supported, this cop-out will be abandoned.

In the case of IRE the schema creation represents a particular procedure. First, the IRE schema is specialized with your desired query. Then, you must define a *variable* and a *predicate* containing this variable.

Now you must select the Tab Widget *Instances*, select your IRE and create an instance clicking *Create Instance* in *Instance Browser*. Finally, specify the variable in the *variable* slot and the predicate in the *proposition* slot.

## **4.3. Ontology**

Before the completion of the ontology project, you must create an instance of the *Ontology* class (in the schema tree you can easily find it). Following the

procedure of instantiation illustrated in the previous paragraph (See 4.2) you must enter the name of the ontology you want, the list of basic ontologies, and the introspector.

Finally (and luckily) the project is finished. In order to generate the assembly containing the ontology implementation, export the ontology through *Convert Project to Format* and then *Experimental*.

#### **4.4. *Protege2AgentService***

To build the assembly containing the ontology definition, you must use the simple *Protege2AgentService* software which receives a protégé XML file. The software generates and stores in your desired directory the assembly and the C# files (useful if you want to modify or customize the ontology).

## **5. SL**

SL (*Semantic Language*) is a language created to express contents. By following the FIPA specification is used to represent message content containing ontological instances. (<http://www.fipa.org/specs/fipa00008/>)

In *AgentService* is used the minimal sublanguage *SL0* with some extension. *SL0* provides operators which transport instance of concepts, predicates, actions, IREs. Here you can see the list of supported operators:

- *action*: it has two properties:
  - *Actor*: the *AID* of the agent which orders the action.
  - *Act*: the action to carry out.
- *Done*:
  - *Action*: the given action.
  - *Condition*: the outcome of the action. It is a predicate (*TrueProposition*, *FalseProposition* or whatever predicate).
- *Result*:
  - *Action*: the given action.
  - *Value*: the result of the computation.
- *Equals*: it is the equals (“=”) operator which binds the *Left* object with the *Right* one.

In SLO are also supported two very used predicates: *TrueProposition* and *FalseProposition*, which express the boolean values *True* and *False*.

In the following, we list some indications about the usage of SLO operators:

- An action instance is transported only by the *Action* operator. The action outcome is conveyed exclusively by *Done* and *Result*. *Done* provides the outcome of an action, in form of predicates that state of the computation. *Result* gives also the result (for example, a concept instance or an aggregate of concept instances).
- Predicates and IREs are not sent as a content of a SLO operator. Responses to predicates are expressed by using *Equals*, putting in the left member the given predicate and in the right member the value of the predicate (*TrueProposition* or *FalsePredicate*). Regarding the IRE responses, you must use the *Equals* operator with left member the IRE, and right member the result of the query (an object or an aggregate of objects).
- Only actions (through *Action*), predicates and IREs can be hosted as message content. Every other schema instance can be hosted only as an attribute of actions and predicates.

The C# classes that implement these operators are stored in the *AgentService.Agent.Ontology.SLO* namespace. These operators are represented as schemas in *BasicOntology*: in this way, during the validation process, their validity is checked. For this reason, you must include *BasicOntology* in your set of basic ontologies.

## 6. Ontological conversations

In this section we will show how you can use the ontology service inside an agent behaviour, exploiting the validation checking functionality for message contents. For this reason you must use a particular kind of conversation: *IOntologicalConversation*. The basic use of *IOntologicalConversation* is similar to the usual *IConversation* of *AgentService*.

Additional properties are shown here:

- `Ontology TheOntology`: it returns an instance of the current ontology.
- `bool Strict`: it is set to true, the validation process is extended to all the `BodyItem` of the message. Otherwise, only the `BodyItem` containing an SL operator (or a predicate or IRE) will be checked. This

- manner allows using in a flexible way the ontological service, also introducing non ontological contents (then not validation-prone).
- `string ValidationError`: it returns the string containing the description of the error that generates the validation exception.

Moreover, *IOntologicalConversation* handles the following methods:

- `bool SendQuery(IRE query)`: it sends a query, avoiding the agent the burden to compose the adequate SL operator. It returns true if the process successfully ends. If false, you can read the `ValidationError` in `IOntologicalConversation`.
- `bool SendPredicate(Predicate predicate)`: analogously, this method sends a predicate.
- `bool SendAction(AgentAction action)`: this method sends an action.
- `bool SendActionDone(AgentAction action, Predicate outcome)`: it send the outcome of an action, then you must provide the action and the outcome (every predicate defined in the ontology).
- `bool SendActionResult(AgentAction action, object result)`: it sends the result of an action.
- `bool SendEqualsPredicate(object left, object right)`: it sends the = operator. Obviously you must indicates the left and right members.

### **6.1. How to open a conversation**

In order to open an ontological conversation, you can call the *OpenConversation* method, passing the *AID* of the agent and the instance of the involved ontology.

Once the conversation is opened, ontological messages can be exchanged, by using the aforementioned methods.

### **6.2. Incoming conversations**

Also the interlocutor agent must manage an ontological conversation, for example in order to run the validation process during the message reception (we are considering hypothetical interferences that could corrupt the message along the line).

This is the code you can use to manage an incoming conversation:

```
IOntologicalConversation conv =  
(IOntologicalConversation) this.IncomingConvs[0];
```

The validation process is automatically executed inside the methods that implement the reception of a message (as *ReceiveMessage*, *WaitForMessage*, etc.). Remember that the *Strict* property in the conversation object it is possible to validate only the effective ontological content or, alternatively, every *BodyItem*.