Lorine Pose
Théo Rup

Semantic Web of Things report

1 Introduction

Over the past few years, the amount of information available on the internet has increased exponentially. In order to be able to find and use the information efficiently, it is necessary to organise that data and to connect it. That is the purpose of an ontology. An ontology formally represents knowledge as a hierarchy of concepts within a domain, using a shared vocabulary to denotes the types, properties, and interrelationships of those concepts. They are used in many fields such as artificial intelligence, biomedical informatics, semantic web… The semantic web provides a common framework that allows data to be shared and reused across applications. It is a vision of information that can be readily interpreted by machines, so machines can find, share, and combine that information more easily without the human intervention.

In these labs, we had the opportunity to implement an ontology related to meteorology with *Protégé*. We created classes and properties and linked them by forming triples. We also build a semantic-aware application able to interact with the previously implemented ontology. Making it possible to retrieve information from it but also to add individuals to it.

2 Creation of the ontology

2.1 Lightweight ontology

2.1.1 Conception

In order to represent the knowledge of this ontology, we first created the classes which could implement a meteorology application. For instance, we created a Phenomenon class which has two child classes : Nice Weather and Bad Weather. The relationship between the parent and the child classes represents a “is a” relationship. The <child class> is a <parent class>. In this case : Bad Weather is a Phenomenon.

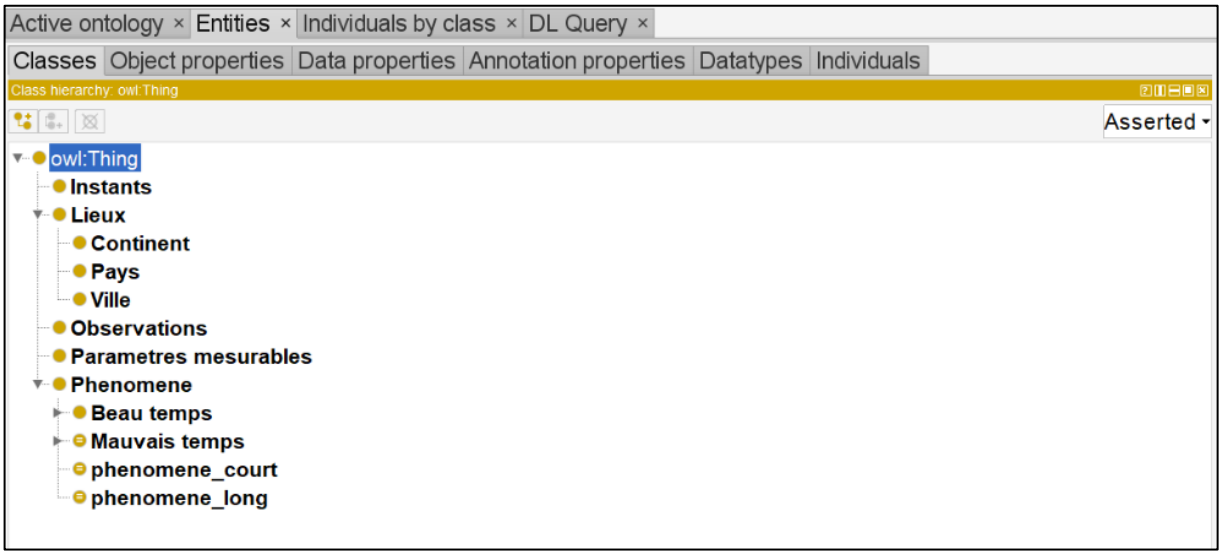

Figure 1 : Our classes tree view

Once all the classes are created, we implemented the properties that will link them. There are two type of properties :

- The object properties such as is characterised by which associates two objects or individuals.
- The data properties such as has a duration which associates an individual to a literal. In this case, to an integer that will represent minutes. To specify that the unit is minutes, we used the comment field.

2.1.2 Filling

In this part, we added individuals to our ontology.

To implement the multilingualism for the température individual, we used the label field with the English language setting and the “temperature” value.

We could also define two individuals as equal as we have done for vitesse du vent and force du vent.

At this point, we created two classes individuals Toulouse and France and linked them with the property is included in. Because this property links has a Place for domain and for range, the Hermit reasoner is able to associate Toulouse and France as Places. When we define that France has for capital Paris, the reasoner can deduct that Paris is a city and France a country. In the same way, A1 which as for symptom P1 which is an Observation class individual will be classified as a Phenomenon.

This shows that the reasoner can make deduction on individuals thanks to object and data properties linked to those individuals.

2.2 Heavyweight ontology

2.2.1 Conception

In this part, we wanted to add logical axioms to the ontology. We did it by adding characteristics to the classes and properties.

For instance, we defined that an individual cannot be a city and a country at the same time. To express that, we added an observation ‘disjoint with’ country to the description of the city class.

It is also possible to define a more specific class, for instance the Short Phenomenon is a subclass phenomenon and is equivalent to a phenomenon for which the duration is less than 15 minutes.

We also added logical links to properties using the ‘inverse of’ characteristic which means that if an object A is included in object B, then object B includes object A.

To define the fact that if A includes B, and B includes C then A includes C, we can simply add the transitive characteristic.

The ‘exactly 1’ keyword allows to only define one individual to a certain property. This way a country only has one capital.

The sub-property description makes sure that if a city is a capital of a country then this city in included in the country.

Finally, with the Manchester syntax it is possible to define more complex characteristics as you can see in the following figure defining that Rain is a Phenomenon which as for symptom an Observation measuring Pluviometry and having a value higher than 0.

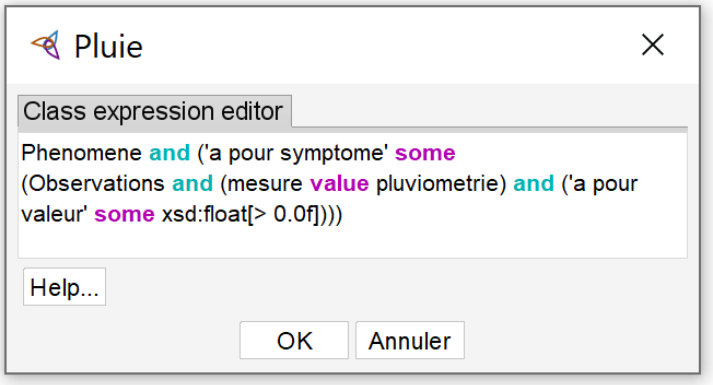

Figure 2 : Expression of the Manchester syntax for Rain

2.2.2 Filling

In the same way as before, we created individuals to fill the ontology. The reasoner was able to make more conclusions : A1 is of class Bad Weather, Paris is included in France, and in Europe. It also reacts by showing an inconsistency since we defined Singapour as both a city and a country because there is a characteristic that does not allow it. The reasoner can prevent from making mistakes. Nonetheless, if we define Toulouse as the capital of France it will not throw an error but only declare that Paris and Toulouse are the same which is an inconsistency that only a human could interpret.

3 Ontology exploitation

In this part we develop a semantic-aware application able to annotate a dataset of meteorological data (temperature and humidity) from the city of Aarhus, in Denmark.

3.1 Implementing the interfaces

3.1.1 Implementing IModelFunctions

The data is stored in CSV files, which is at 3 stars on the Linked Data hierarchy. By using the ontology we previously created, we will make interrelated data that complies to W3C specifications. The database will then be at 5 stars on the Linked Data hierarchy.

We started by implementing IModelFunctions in DoItYourselfModel. These functions provide knowledge-base related operations described as comments above their implementation code.

Just as in the case of *Protégé* manipulation, the API makes a distinction between the label and the URI. It is possible for objects to have the same label, that is why they have a unique URI to differentiate them. This explains that the function getEntityURI(*label*) returns a list of URIs. In our case, since we created the ontology, we know that the labels are unique. Therefore, we can simply get the first element of the list to retrieve the searched URI.

```
//Creates an instance of the class "Place" of our ontology
public String createPlace(String name) {
        String lieuURI = this.model.getEntityURI("Lieu").get(0);
        return this.model.createInstance(name,lieuURI);
}
```

Figure 3 : Implementation of the createPlace function

```
//Creates an instance of the "Instant" class of your ontology. You'll have to link it to a data property that
//represents the timestamp, serialized as it is in the original data file. Only one instance should be created
//for each actual timestamp.
public String createInstant(TimestampEntity instant) {
        String instantClassURI = this.model.getEntityURI("Instant").get(0);
        String individualURI = this.model.createInstance("instant" + instant.getTimeStamp(), instantClassURI);
        String timestampPropertyURI = this.model.getEntityURI("a pour timestamp").get(0);
        this.model.addDataPropertyToIndividual(individualURI, timestampPropertyURI, instant.getTimeStamp());
        return individualURI;
}
```

Figure 4 : Implementation of the createInstant function

At first, in the getInstantURI function, we simply looked for the URI of an instant labelled "instant" + instant.getTimeStamp() since that is the way the method createInstant() works. Nonetheless, we decided it was a better to use a more generic approach in case we are not the only ones interacting with the ontology. To do so, we decided to check the timestamp of every individual of class Instant until we find the good Instant and we can get its URI.

```
//Return the URI of the representation of the instant, null otherwise.
public String getInstantURI(TimestampEntity instant) {
        boolean find = false;
        int i=0;
        String result = null;
        String instantClassURI = this.model.getEntityURI("Instant").get(0);
        List<String> instantIndividualsURI = this.model.getInstancesURI(instantClassURI);
        String propertyURI = this.model.getEntityURI("a pour timestamp").get(0);
        while (find == false && i<instantIndividualsURI.size()) {
                find = this.model.hasDataPropertyValue(instantIndividualsURI.get(i), propertyURI, instant.getTimeStamp());
                if (find == true) {
                        result = instantIndividualsURI.get(i);
                }
                i++;
        }
        return result;
}
```

Figure 5 : Implementation of the getInstantURI function

```
//Return the value of the timestamp associated to the instant individual, null if the individual doesn't exist
public String getInstantTimestamp(String instantURI) {
        boolean find = false;
        int i=0;
        String result = null;
        List<List<String>> propertiesInstant = this.model.listProperties(instantURI);
        while (find == false && i<propertiesInstant.size()) {
                if ((propertiesInstant.get(i)).get(0) == this.model.getEntityURI("a pour timestamp").get(0)) {
                        result = (propertiesInstant.get(i)).get(1);
                }
                i++;
        }
        return result;
}
```

Figure 6 : Implementation of the getInstantTimestamp function

For the createObs function, creating the observation and adding its different properties was not enough since we had to find from the instantURI given in parameter the sensor that made the observation and to associate the created observation to that sensor. Nonetheless, the sensor part is only appropriate for the JUnitTest since in our ontology we do not have Sensors linked to our observations, so we will have to modify the code a bit if we want to use our ontology.

```
//Creates an Observation of the provided value for the provided parameter
//at the provided time. It uses both object and data properties from the ontology
//to link the observation to its value, instant, and parameter.
public String createObs(String value, String paramURI, String instantURI) {
        String obsClassURI= this.model.getEntityURI("Observation").get(0);
        String obsURI= this.model.createInstance("obs_"+instantURI, obsClassURI);
        String propInstantURI= this.model.getEntityURI("a pour date").get(0);
        String propValueURI= this.model.getEntityURI("a pour valeur").get(0);
        String propMesureURI= this.model.getEntityURI("mesure").get(0);
        String timestamp = getInstantTimestamp(instantURI);
        String sensorURI= this.model.whichSensorDidIt(timestamp, paramURI);
        this.model.addObjectPropertyToIndividual(obsURI,propInstantURI,instantURI);
        this.model.addDataPropertyToIndividual(obsURI, propValueURI, value);
        this.model.addObjectPropertyToIndividual(obsURI, propMesureURI, paramURI);
        this.model.addObservationToSensor(obsURI, sensorURI);
        return obsURI;
}
```

Figure 7 : Implementation of the createObs function

3.1.2 Implementing IControlFunctions

We implemented IControlFunctions in DoItYourselfModel. The controler uses functions from the model and uses them to enrich the dataset.
The function instanciateObservations has been implemented to enrich our dataset. Once done, we can use it to fill our ontology with the dataset of meteorological data (temperature and humidity) from the city of Aarhus and export the ttl file. The code we have implemented is not optimized and could be improved. One of the main drawbacks is that we created instances of our object only in the memory of the program. It means that all the knowledge base will be deleted at the end of execution of the program.

```
//This function parses the list of observations extracted from the dataset, and instanciates them in the knowledge base.
public void instantiateObservations(List<ObservationEntity> obsList, String paramURI) {
        //We keep track of the timestamp URIs created to not create it twice
        Map<String, String> instantsMap = new HashMap<String, String>();
        int i =0;
        //loop for all the observations
        for(ObservationEntity oe : obsList) {
                i++;
                System.out.println("Observation "+i);
                String instantURI;
                //We check that the timestamp has not been previously created
                //if not we create an instant with that timestamp
                if(!instantsMap.containsKey(oe.getTimestamp().getTimeStamp())){
                        instantURI = this.customModel.createInstant(oe.getTimestamp());
                        instantsMap.put(oe.getTimestamp().getTimeStamp(), instantURI);
                }else{
                        instantURI = instantsMap.get(oe.getTimestamp().getTimeStamp());
                }
                //we create the observation with the previously implemented function createObs
                this.customModel.createObs(oe.getValue().toString(), paramURI, instantURI);
        }
}
```

Figure 8 : Implementation of the instanciateObservations function

3.2 Exploitation in *Protégé*

By running Controler.java, we obtained a ttl file containing the Observations created with the data contained in the csv file. In *Protégé*, we can see the created observations (Figure 9).

You can see below a screenshot of the individuals that are now populating the ontology. Each of them has a timestamp and a value. In the figure, you can see that the Observation has been done at 23:50:00 the 7th of March 2014. This is a humidity measurement equal to 70.0.

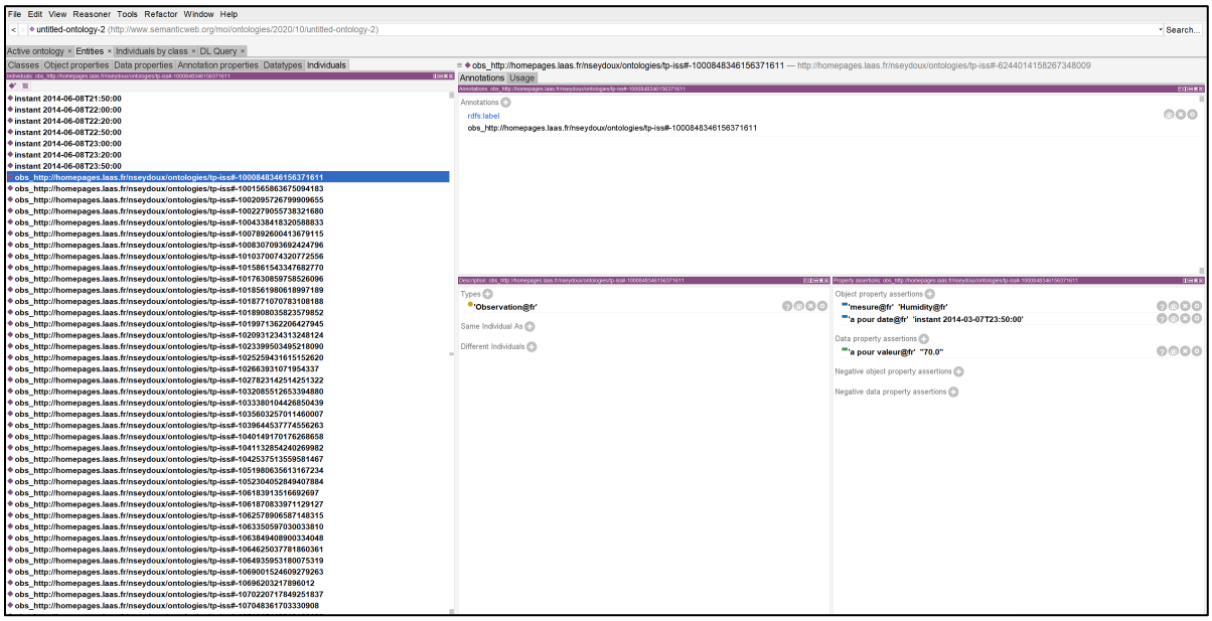

Figure 9 : View of the Individuals that are now populating the ontology

We can notice that the sensor makes measurement in a cycle of 10min/20min/30min. This configuration does not seems very adapted for a humidity/temperature measurement. This analyse could have been automated if we learned the machine that a humidity/temperature measurement is more relevant when it is done in a regular cycle and/or when a big change occurs.