Semantic Web of Things laboratory Report

Xu Andy, Zennaro Thomas, ISS-Groupe B1, Binôme 3

December 8, 2020

Contents

Introduction

The aim of both labs is to manipulate the notion of ontology, to discover the main aspects associated with it, and to be able to use a reasoner to observe the steps of deduction when developing our ontology on a weather example. The idea is to develop a small smart application of meteorology. The work consists of two main steps:

1. a first session on designing a weather ontology with the Protégé software in which the Hermit grape grower is used.
2. a second session on converting data from a given dataset in CSV format into 5-star interpretable data, using our weather ontology.

1 Designing an ontology and using the reasoner in Protégé

1.1 Light Ontology

1.1.1 Conception

First of all, we expressed the given knowledge by creating the most relevant classes and sub-classes. We defined five classes to meteorology as *Phenomenom*, *MeasurableParameters*, *Instants*, *Observations* and *Lieu*. Furthermore, we added two sub-classes for *Phenomenom* (*GoodWeather* and *BadWeather*) and three sub-classes for *Lieu* (*Continent*, *Pays* and *Ville*). You can see the hierarchy of the classes in Figure 1.
Then, we need to define the links between classes and sub classes. For that, it is necessary to add properties and sub properties included in the owl:topObjectProperty. These object properties will allow the relationships between individuals. For example, for the assertion "*un phénomène a pour symptôme une observation*", we defined '*a pour symptôme*' as a property, '*phénomène*' as a domain and '*observation* as a range. All the object properties can be seen in Figure 2. Moreover the example is given in Figure 3.
We can add other characteristics for one property. For instance, if we take the '*includes*' property we can say that it is the inverse of '*is included in*'. Indeed, it means that if an individual present the '*includes*' property, the '*is included in*' property cannot be true because at the semantic level, they can only be opposed.
We used the data properties to attribute types of data to our ontology. They are essential when we need to put in relation an instance of a class with a data value. The data properties for our ontology are shown in Figure 4. With that we can define relations such as the assertion n°4: "*Un instant a un timestamp*". To create this data property, we have to attribute '*instants*' as a domain and '*xsd:dateTimeStamp*' as a range.

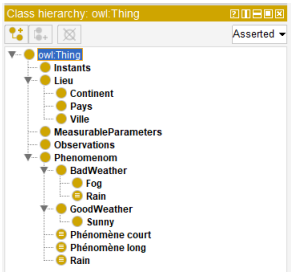

Figure 1: Classes and sub-classes of the ontology

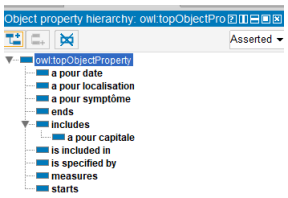

Figure 2: Properties and sub-properties of the ontology

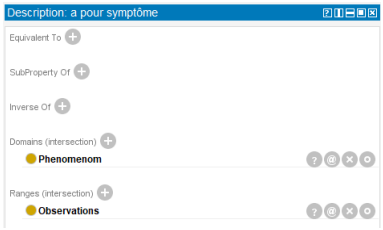

Figure 3: Example for the '*a pour symptôme*' property description

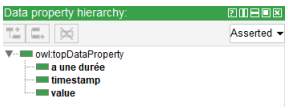

Figure 4: Data properties of the ontology

1.1.2 Populating

The next step is to create individuals based on our classes in order to populate our ontology. You can found the individuals in Figure 5.

After the creation of the individuals, we could start the Hermit reasoner to deduce information from our current ontology. For instance, if we take the assertion n°4 "*Toulouse est située en France*", the reasoner succeed to deduce that Toulouse and France are individuals which belong to the *Lieu* class and used the '*is included in*' property. Moreover, Hermit deduced that France contains Toulouse by using the '*includes*' property (i.e. the inverse property of '*is included in*'.If we take a second example, for the assertion n°6 "*La France a pour capitale Paris*", the reasoner concludes that France '*includes*' and '*a pour capitale*' Paris, that Paris '*is included in*' France, that France is a '*Pays*' and that Paris is a '*Ville*' (see Figure 6 and Figure 7).

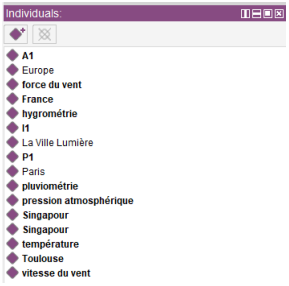

Figure 5: Individuals for our ontology

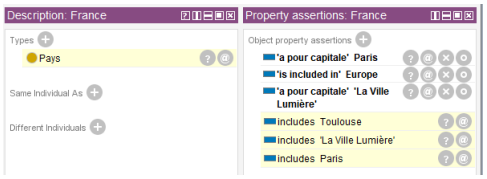

Figure 6: Description of the individual **France**

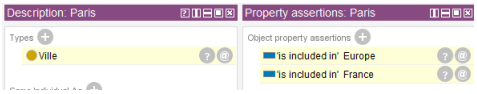

Figure 7: Description of the individual **Paris**

1.2 Heavy ontology

1.2.1 Conception

After creating the light ontology, we can conceive the heavy ontology. To illustrate it, you can found the explanation of examples. We need to set up several relations and attributes. For the knowledge from 1 to 4, we need to create the sub-classes "*Phenomenon court*" and "*Phenomenon long*" (see Figure 8). Indeed, for them, we need to characterize these classes by using the Manchester syntax. Indeed for the "*Phenomenon court*" sub-class, we have: "**Phenomenom and 'a une durée' some xsd:integer[< 15]**. It means that a short phenomenon is equivalent to a phenomenon lasting less than 15 min. The 'a une durée' property is not followed by an object but by a data value. There is the same case for the "*Phenomenon long*".
For the knowledge n°5, we establish the inverse property, i.e if A is included in B, A cannot include B.
For the knowledge n°6, we establish the transitivity characteristic, i.e if A is included in B, and B is included in C, therefore A is included in C. For instance, a city is included in a country which is part of a continent (see Figure 9), so a city is included in a continent.
For the knowledge n°8, we establish the sub property notion : if a city is defined as a capital, this city is necessarily part of the country. So, when we affirm that Paris is the capital of France, it is obvious that Paris is part of France (see Figure 10).
For the knowledge n°9, we can say that rain is equivalent to a phenomenon that shows an observation, measured by a rain gauge, and that the observation value is greater than 0 . The Manchester syntax is given as follows : "**Phenomenon and ('a pour symptôme' some (Observation and (measures value pluviométrie) and (value some xsd:float[> 0.0f])))**" To express that, we created the "a pour symptôme" object property.

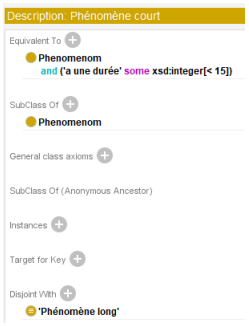

Figure 8: Short phenomenon

Figure 9: Inverse and transitivity properties for 'includes'

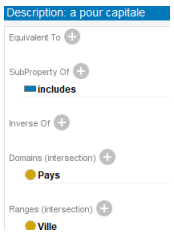

Figure 10: 'a pour capitale' property

1.2.2 Populating

We created some individuals in order to populate our ontology. These individuals can get some specifications. We create both "*Paris*" and "*La ville lumière*" as instances. We assign that "*France*" "*a pour capitale*" (object property) "*Paris*" (see Figure 11). When we used the reasoner we can see that it put both instances to the same category ("*same individual as*"). It is normal because we specified that a capital is unique. Then if we create two individuals with the same name (*Singapour* as a country and *Singapour* as the capital), the reasoner manages to distinguish between the two Singapore and affects the good properties and links between both instances (see Figure 12 and Figure 13).

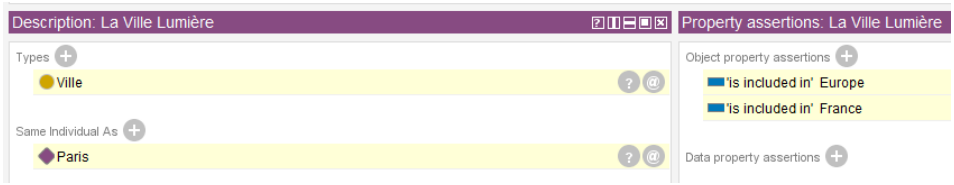

Figure 11: Example with **Paris** and **La ville lumière**

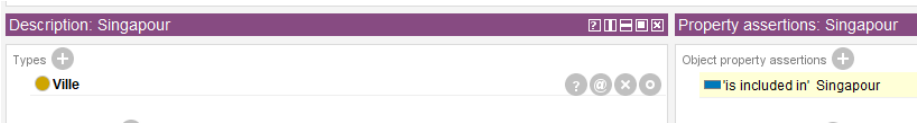

Figure 12: Example with **Singapour** (ville)

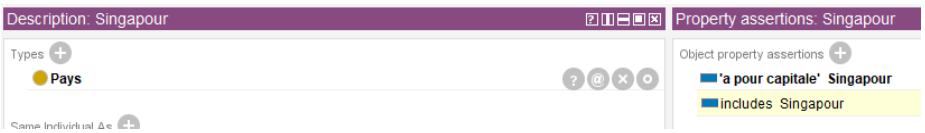

Figure 13: Example with **Singapour** (Pays)

2 Ontology manipulation and dataset annotation using Java

2.1 Interface implementation

We start by implementing the interfaces to convert raw CSV data to manipulable data. This is done by completing the provided Java code that gives us the baseline of our project through the *Apache Jena* and *SPARQL* wrapers.
First is to implement the functions that provide knowledge-base related operations in the *DoItYourselfModel* class, which are quickly described in the *IModelFunctions* class. These functions are used to populate the designed ontology that serves as the database structure for our API. They create the corresponding instances, that is *individuals*, for the designed *entities*, such as *classes* or *properties*. For this lab, we only implement some functions that instantiates mostly the *Observation* class and the *Timestamp* property (see *Appendix: Model*). This code is thereafter validated using the provided instantiating tests which makes sure that our implementation corresponds to the designed ontology.
What can be noticed is that the API distinguishes *labels* and *URI* like Protégé. What is manipulated to construct our database are therefore all *URI* that come from the ontology. *Labels* serve only to provide the human user a readable name and to facilitate *URI* manipulation thanks to the *getEntityURI* or *getInstancesURI* functions. So every link that our API creates, adding the *Timestamp* property to one instance for example, when instantiating the database is done through the *URI*.
Next we implement the *IControlFunctions* class that uses functions from the *DoItYourselfModel* class to enrich the dataset by instantiating the *Observation* class (see *Appendix: Controller*). We test our API using the provided JSON parser to convert the raw CSV data to observations instances, and export our newly populated ontology to exploit it in Protégé. (Here we only use the given data with 300+ observations at path "src/test/temperature.txt" instead of the complete one that was too big and did not compute after one hour of waiting). Also, if we modify the path of the ontology to be loaded from the provided one, which we implemented correspondingly, to the one we created in the first lab, the program fails according to the discrepancies found between them in our case.

2.2 Exploitation in Protégé

We import the generated model in Protégé and confirm that the instances created by the API are correctly done as can be seen in Figure 14 and Figure 15. Each value is an instance of *WeatherObservation* and corresponds to one specific timestamp. It is also linked to a *Output_T* that is associated with/produced by one specific sensor, such as we implemented in the code.

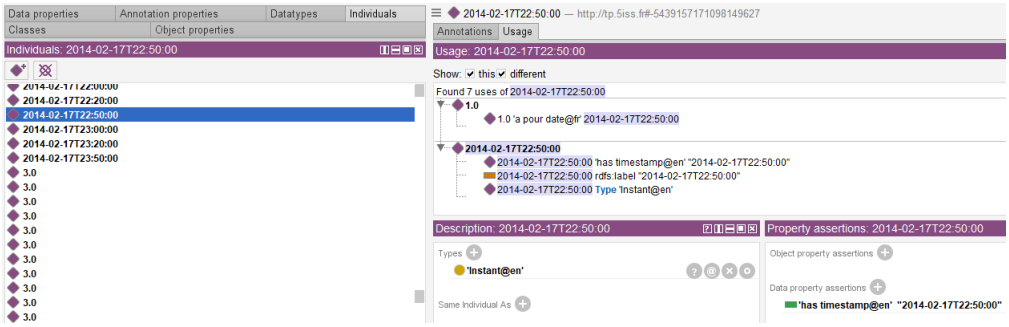

Figure 14: Individual 'has timestamp' property

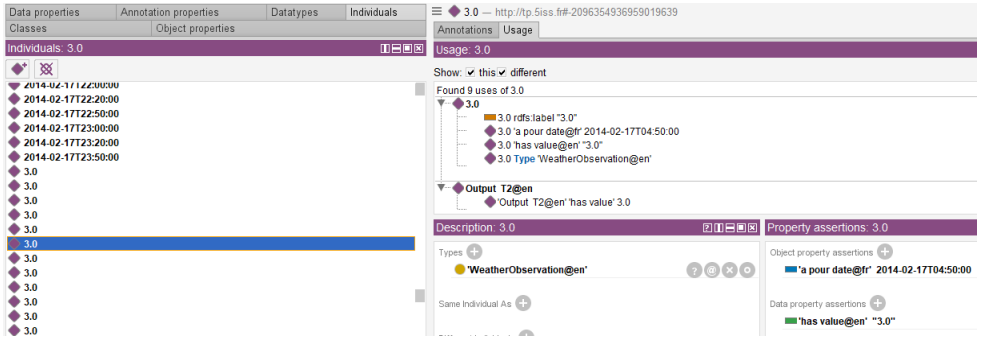

Figure 15: Individual 'has value' property

We then check if the sensors have been chosen wisely as described in the W3C SSN (see Figure 16).

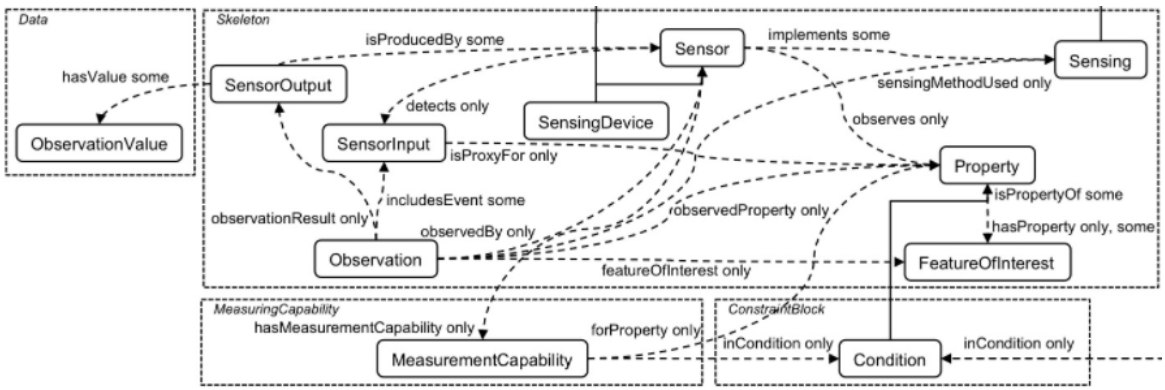

Figure 16: SSN chart as described in the W3C

With what we observed and on Figure 17 and Figure 18, the sensors ontology corresponds to the W3C SSN description.

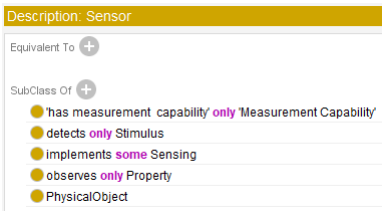

Figure 17: Sensor

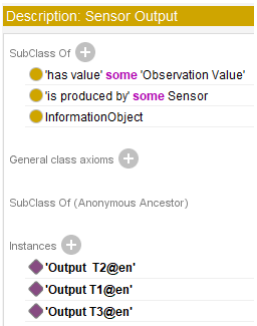

Figure 18: Sensor Output

Finally, *object property* is a property that link two class instances between them whereas *data property* links one class instance to one datatype, which is a data that that instance provides.

Conclusion

In this lab we manipulated an ontology through an example about meteorology and learned about translating dataset structure to its corresponding ontology, the usage and properties of a reasoner that is a inference engine, and finally about automated ontology instantiation by Java API.

Appendix

A Java Code

A.1 Model

```
package semantic.model;

import java.util.List;

public class DoItYourselfModel implements IModelFunctions
{
   IConvenienceInterface model;

   public DoItYourselfModel(IConvenienceInterface m) {
      this.model = m;
   }

   @Override
   public String createPlace(String name) {
      return this.model.createInstance(name, this.model.getEntityURI("Lieu").get(0));
   }

   @Override
   public String createInstant(TimestampEntity instant) {
      List<String> myList =
          this.model.getInstancesURI(this.model.getEntityURI("Instant").get(0));
      for(String s : myList){
         if(this.model.hasDataPropertyValue(s, this.model.getEntityURI("a pour
             timestamp").get(0), instant.getTimeStamp())){
            return null;
         }
      }
      String newInstant = this.model.createInstance(instant.getTimeStamp(),
          this.model.getEntityURI("Instant").get(0));
      this.model.addDataPropertyToIndividual(newInstant, this.model.getEntityURI("a pour
          timestamp").get(0), instant.getTimeStamp());
      return newInstant;
   }

   @Override
   public String getInstantURI(TimestampEntity instant) {
      List<String> myList =
          this.model.getInstancesURI(this.model.getEntityURI("Instant").get(0));
      for(String s : myList){
         if(this.model.hasDataPropertyValue(s, this.model.getEntityURI("a pour
             timestamp").get(0), instant.getTimeStamp())){
            return s;
         }
      }
      return null;
   }

   @Override
   public String getInstantTimestamp(String instantURI)
   {
      List<List<String>> myDoubleList = this.model.listProperties(instantURI);
      for(List<String> ls : myDoubleList){
         if (ls.get(0) == this.model.getEntityURI("a pour timestamp").get(0)){
            return ls.get(1);
         }
      }
      return null;
   }
```

```
    @Override
    public String createObs(String value, String paramURI, String instantURI) {
        String newInstant = this.model.createInstance(value,
            this.model.getEntityURI("Observation").get(0));
        this.model.addDataPropertyToIndividual(newInstant, this.model.getEntityURI("a pour
            valeur").get(0), value);
        String timestamp = this.getInstantTimestamp(instantURI);
        this.model.addObjectPropertyToIndividual(newInstant, this.model.getEntityURI("a
            pour date").get(0), instantURI);
        String sensorURI = this.model.whichSensorDidIt(timestamp, paramURI);
        this.model.addObservationToSensor(newInstant, sensorURI);
        return newInstant;
    }
}
```

A.2 Controller

```
package semantic.controler;

import java.util.List;

import semantic.model.IConvenienceInterface;
import semantic.model.IModelFunctions;
import semantic.model.ObservationEntity;

public class DoItYourselfControl implements IControlFunctions
{
    private IConvenienceInterface model;
    private IModelFunctions customModel;

    public DoItYourselfControl(IConvenienceInterface model, IModelFunctions customModel)
    {
        this.model = model;
        this.customModel = customModel;
    }

    @Override
    public void instantiateObservations(List<ObservationEntity> obsList,String paramURI) {
        for(ObservationEntity obsv : obsList){
            this.customModel.createObs(obsv.getValue().toString(), paramURI,
                this.customModel.createInstant(obsv.getTimestamp()));
        }
    }
}
```

List of Figures