Semantic Web of Things lab report

January 18, 2020

Lucie Bechtet, Vincent Erb, Isabella Mi Hyun Kim - 5ISS A

1 Introduction

This document reports the concepts and methods implemented during the two labs of the course **Semantic Web of Things**.
The main point was to familiarize ourselves with the concept of **ontology**. An ontology is a specification of concepts, and the relations between these concepts. During the labs, we created an ontology and tried to showcase the most important applications it can offer :

- During the first lab, we used the software *Protege* to help us build a light ontology, then a heavy ontology, to finally reason on instances of the ontology.
- In the second lab, we developed a program to convert CSV data from an existing dataset to more exploitable 5-Star data using the ontology of the first lab.

2 Creating the ontology

2.1 Light ontology

2.1.1 Conception

In this first part, we started by creating an ontology to describe meteorological data by defining classes and sub classes. The goal of constructing this ontology is to construct a vocabulary that can describe semantically this data. So at the end we're able to construct a chain of connected data.
We defined 5 main classes to meteorology: Phenomenon, Place, Observation, MeasurableParameter and Instant. And for some of these classes we indicated some subclasses: "BadWeather" and "GoodWeather" belonging to Phenomenon and "City", "Continent" and "Country" belonging to City. Theses relationships can be seen in Figure 1.

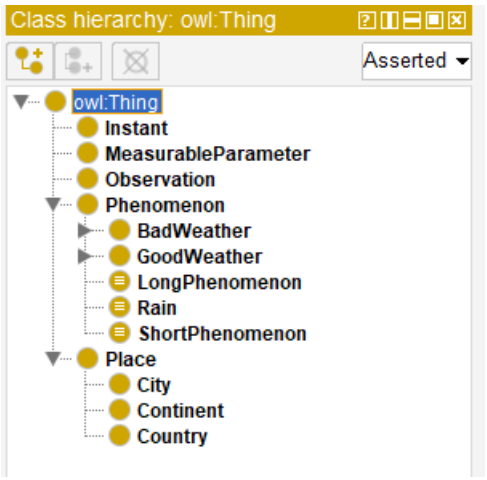

Figure 1: Classes and sub classes of the ontology.

After that, to define the relationships between the classes and sub classes we added properties (and even sub properties) as an instance of the built-in OWL class owl:ObjectProperty. These object properties make the link between individuals. By adding this to our ontology we can make semantic connections such as: Observation *measures* MeasurableParameter. In this case, our object property (measures) asserts a range "MeasurableParameter" to the domain "Observation". All the object properties can be seen in Figure 2.

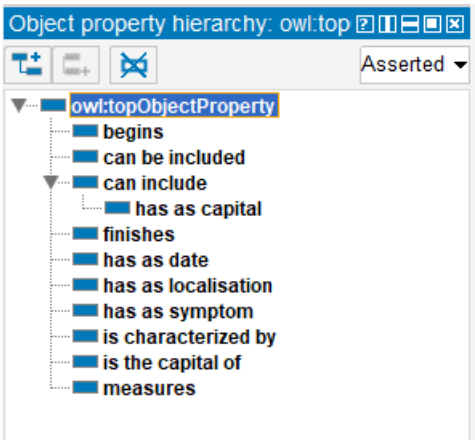

Figure 2: Properties of the ontology.

Apart from defining domains and ranges to the object properties, we can even add a parameter to say that one property is the inverse of another. In the case of our ontology we established that "can include" is the inverse of

the "can be included". That means that if 2 individuals present one of these properties, the other property cannot be true (because it represents the opposite semantically).

We used the Datatype properties to attribute data to our ontology and link individuals to data values. In Figure 3, we show the data created to our ontology. With that we can define relations such as: an Observation *has a duration* in minutes. To create a data property, we have to attribute a domain to the data and a range, which in this case is the unit (int, unit less, float, etc) to the value.

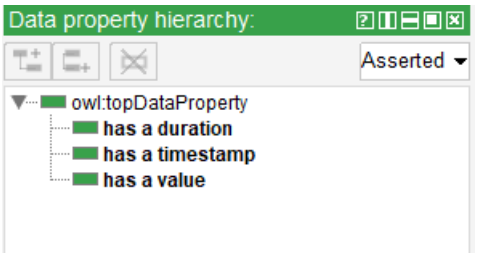

Figure 3: Data properties of the ontology.

2.1.2 Populating

With the defined ontology we could create individuals from different classes to start populating our ontology. The individuals created can be seen in Figure 4.

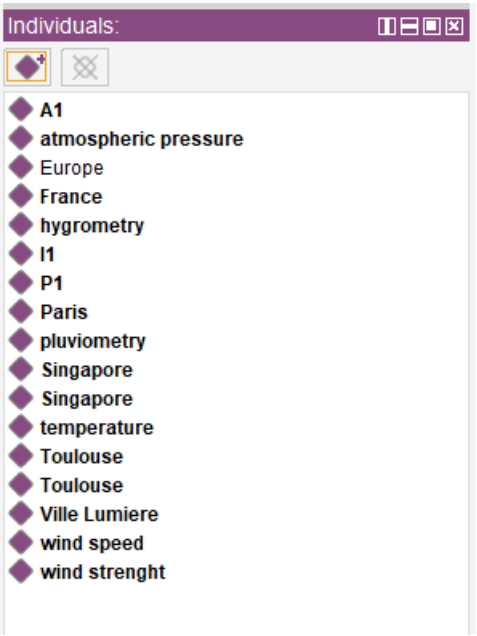

Figure 4: Individuals of the ontology.

After the individuals created, we could apply the Hermit reasoner to deduce information from our current ontology.

From the affirmation 4 (*"Toulouse is located in France"*), the reasoner deduced that Toulouse and France are individuals of type Place by using the property "is included in". And also, that France contains Toulouse by using the property inverse of "is included in".

From the affirmation 6 (*"France has as capital Paris"*) the reasoner could deduce that France in a country and that Paris is a city. It also deduces that France includes/has as capital Paris and that Paris is included in/is capital of France.

2.2 Heavy ontology

2.2.1 Conception

Now the light ontology is created we can move to the "heavy" one. Following are example of what we can do in a heavy ontology. Several relations and attributes are defined in this part.
For the question 2.2.3.8, we establish that if a city is capital of a country , it is therefore part of it. This link is made through the notion of sub property. In this case, after Paris is declared capital of France, it is also declared part of France.
We are also able to state that some properties are inverse, for example when

A is included in B, A cannot include B. Therefore the properties "can include" and "can be included are inverse.
We also defined some parameters, here for the temperature. Using the Manchester syntax, we can define a short phenomenon like this : Phenomenon that 'has a duration' some xsd:float [< 15]. We are using for this data properties and not object properties. Indeed, has a duration is not followed by an object but by a data of the form (here) float. We can declare the same way a long phenomenon. Short phenomenon and long phenomenon are two sub-classes of the class phenomenon. The place to specify the length parameter is in "equivalent to" that specifies that a long phenomenon is equivalent to a phenomenon lasting more than 15 min.
With the same logic, we could specify that rain is equivalent to a phenomenon that shows an observation, measured by the pluviometer, and that the value of this observation is greater than 0 . The code is the following : "Phenomenon and ('has as symptom' some (Observation and (measures value pluviometry) and ('has a value' some xsd:float[> 0.0f]))))." We can clearly see that before writing this, we had to create an object property called "has a symptom", that qualifies that a phenomenon has a symptom which is an observation, as in Figure 5.

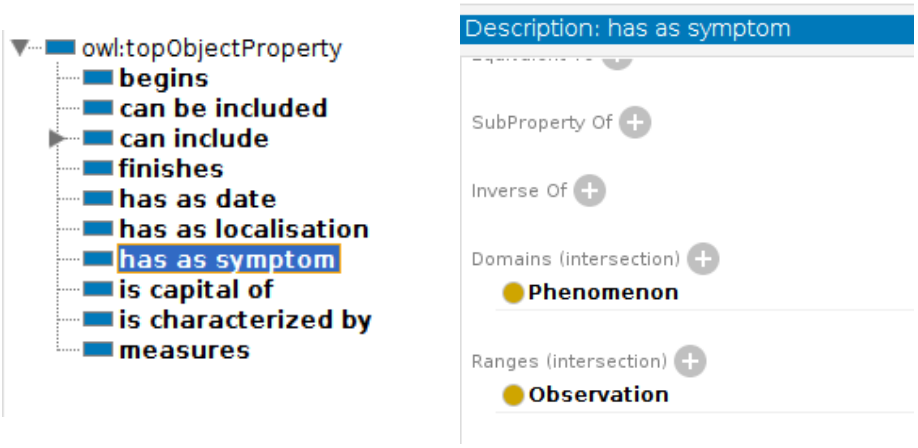

Figure 5: Representation of the object property "has a symptom

2.2.2 Populating

We can create some instances to populate the ontology. These instances can get some assertion and specification.
For example, if we create "Paris" and "Ville des lumières" we can specify that they are the same in the category "same individual as". In assertion, we can for example use a "is capital of" object property created previously to say that paris is the capital of France (another instance).
After giving the information we have, we can use the reasoner. The reasoner

will establish links between instances for example, or dependencies that we did not specify. The reasoner will, in our example, add that the "ville des lumière" is capital of France. But this is to use with precaution, because if we create another instance called "toulouse" and we state that toulouse is capital of france, the reasoner will tell us that toulouse is the "same individual as" paris. To avoid this issue, it is important to specify that they are different individual. In that case , the reasoner raised an issue if we specify that a capital is unique, like in Figure 6.

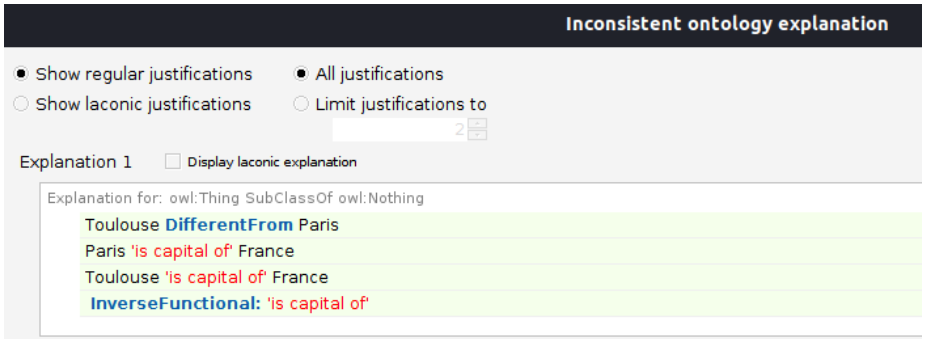

Figure 6: Issue raised by the reasonner

From all the information given, the Figure shows what the reasoner deduces for the instance France.

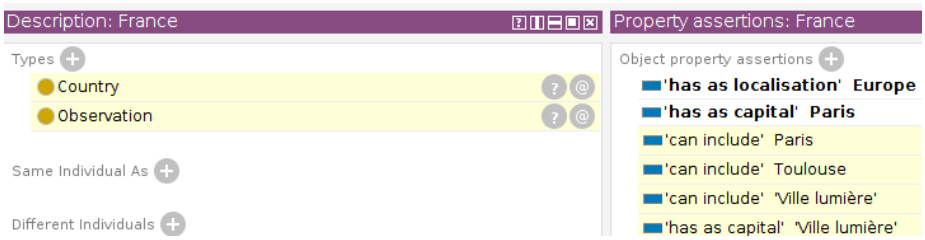

Figure 7: Reasoner deduction for France

3 Exploiting the ontology

In the second lab, we illustrated one of the use cases of an ontology on a real smart city data set. The goal was to take as an input a CSV file (3-star data) and turn it into 5-star data thanks to the ontology we created in the first lab.

We had to create a program that could parse the CSV file and enrich it, creating an RDFS knowledge base that we could then exploit in *Protege.* The program was developed in Java using mainly the Jena library.

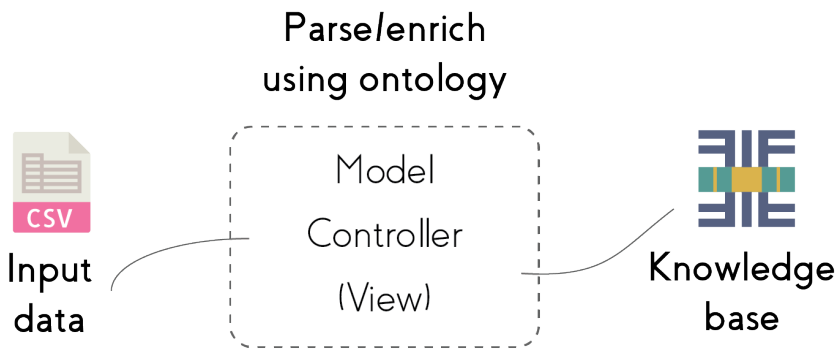

Figure 8: Outline of the second lab

We mainly worked on developing the core model and controller methods. First, we had to implement the functions used to create *instances* of different classes in our knowledge base, taking a String as an input.

The *createPlace* function didn't cause much problems, but we were faced with implementation choices for the *createInstant* function.
Indeed, we had to create an Instant instance from a timestamp input value, making sure that only one Instant was created for every timestamp.

The first issue was deciding what to do with labels, and subsequently iterating over instants. At first, we thought of naming the Instant instances with the string of the timestamp, and then making sure that it was unique by comparing labels. However, this is not ideal because if someone elses contributes to the knowledge base and doesn't follow this convention, the whole system breaks.
Instead, we decided to use a meaningless label, and then no label at all (we created the instance with an empty string), because it did not seem relevant to us to use a label to describe a time Instant. To iterate on Instant instances, we checked the data property of existing instances directly.

```
@Override
public String createInstant(TimestampEntity instant) {

    // Cache for timestamp values

    Boolean exists = false;
    List<List<String>> list = model.listProperties(instantURI);


   // Get list of all instant instances IRI
    List<String> l =  model.getInstancesURI(model.getEntityURI("
    Instant").get(0));

    for(String s : l) {
```

```
        // If one instant has the correct timestamp, return it
        if(model.hasDataPropertyValue(s, model.getEntityURI("a pour
     timestamp").get(0), instant.getTimeStamp())) {
                exists = true;
        }
    }

    if(!exists) {
        String inst = model.createInstance("", model.getEntityURI("
    Instant").get(0));
        model.addDataPropertyToIndividual(inst, model.getEntityURI(
    "a pour timestamp").get(0), instant.getTimeStamp());
        return inst;
    } else {
        return null;
    }
}
```

The following function *getInstantIRI* reuses most of the code from the previous function.

Then, the function *getInstantTimetamp* had us look at the data structure that the function *listProperties* returns. It's a list containing a String list, meaning that each list element contains the property IRI and its value (or the object IRI for an object property).
Once We understood that, we just had to iterate over every property of the Instant instance, compare the IRIs to find the one corresponding to the "a pour timestamp" one and return the corresponding value.
After completing the *createObs* function that simply called the addDataProperty and addObjectPreperty functions that were available, we could go and move on to the controller.

We only had to complete one function, and it involved reusing the previously created functions in the model package :

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

When running the controller, the parsing of the CSV data (more than 8000 Observations) was taking a really long time. The creation of Instants involves sending a request to get all Instant instances in the knowledge base and checking each one. More thant being a n2 complex problem, it creates a choke point on the knowledge base which considerably slows things down. To change that, we

implemented a memoïzation solution : we simply cached the timestamp values when we created them. In doing that, there was no need to check Instant instances anymore.

```
@Override
public String createInstant(TimestampEntity instant) {


    Boolean exists = false;

    // Cache for timestamp values
    for(String s : cache) {
            if(instant.getTimeStamp().equals(s)) {
                    exists = true;
            }
    }

    if(!exists) {
            String inst = model.createInstance("", model.
    getEntityURI("Instant").get(0));
            model.addDataPropertyToIndividual(inst, model.
    getEntityURI("a pour timestamp").get(0), instant.getTimeStamp()
    );
            return inst;
    } else {
            return null;
    }

}
```

In the end, we enriched our CSV data thanks to the ontology we created in the first lab. Unfortunately, we weren't able to export our newly created model to *Protege*, where we could have run queries to analyse the data coming from the smart city.
For example, we could have created Manchester syntax queries to look at sensor data and check if the sensor chosen were relevant to the data that was being collected.