INSA Toulouse

WEB SEMANTIQUE

5ISS

Rapport des Travaux Pratiques

Auteurs:
Aymen Boukezzata
Thomas Berton

January 23, 2022

Contents

List of Figures

1 Introduction

Le Web sémantique permet de fournir aux programmes logiciels des méta-données interprétables par les machines pour toute information et donnée publiée, en d'autres termes on peut ajouter des descripteurs de données supplémentaires. Par conséquent, les ordinateurs sont capables de faire des interprétations significatives, de la même manière que les humains traitent l'information.

Pour cela, les technologies du web sémantique sont organisées dans une architecture appelée "Semantic Web Stack".Parmi ces technologies, nous nous intéresserons à OWL (Ontology Web Language), RDF (Resource Description Framework) et SPARQL (SPARQL Protocol And RDF Query Language).

2 Création de l'ontologie

Le but de cette première partie est de créer une ontologie pour décrire des phénomènes météorologiques avec le logiciel Protégé

2.1 L'ontologie légère

2.1.1 Conception

Dans un premier temps, nous avons commencé par créer les classes et les sous classes appropriées afin de former une hiérarchie. Par exemple nous pouvons voir ci-dessous que la classe Lieu contient deux sous classes *Pays* et *Villes*.

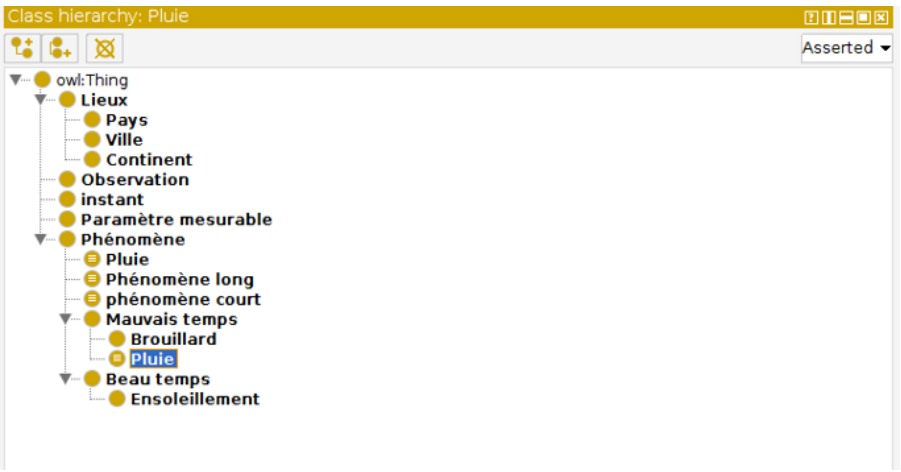

Figure 1: Classes de l'ontologie météo sur Protégé

Par la suite, nous devions ajouter des propriétés de type **object properties** pour les différentes classes crées. Elles permettent de relier des classes entre elles. Nous avons remarqué qu'une propriété d'un objet se transmet également à ses sous-classes.

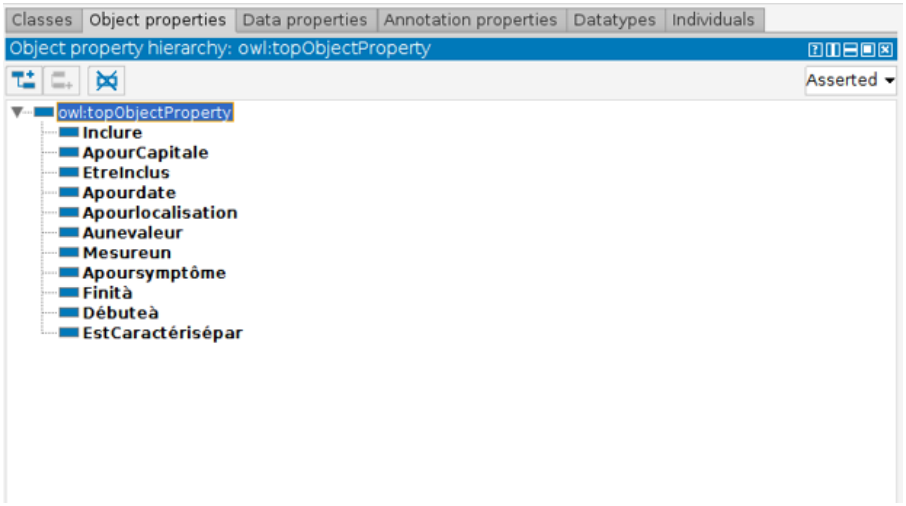

Figure 2: Propriétés des objets de l'ontologie météo sur Protégé

Enfin il était demandé à certains moments de rajouter des attributs de type **data properties** à certaines classes. Par exemple pour representer qu'un instant a une date, nous avons ajouté à la classe un *Instant* une propriété *timestamp* de type *xsd:dateTimeStamp*.

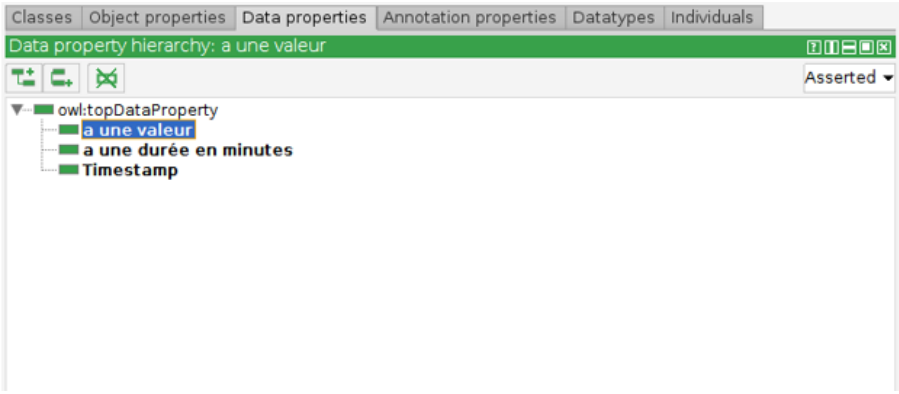

Figure 3: Propriétés de type data de l'ontologie météo sur Protégé

2.1.2 Peuplement

Une fois que nous avions créé nos classes ainsi que leurs différentes propriétés, nous avons ajouter des individus afin de peupler l'ontologie. Nous pouvons relier ces individus avec les différentes propriétés que nous avons créées précédemment.

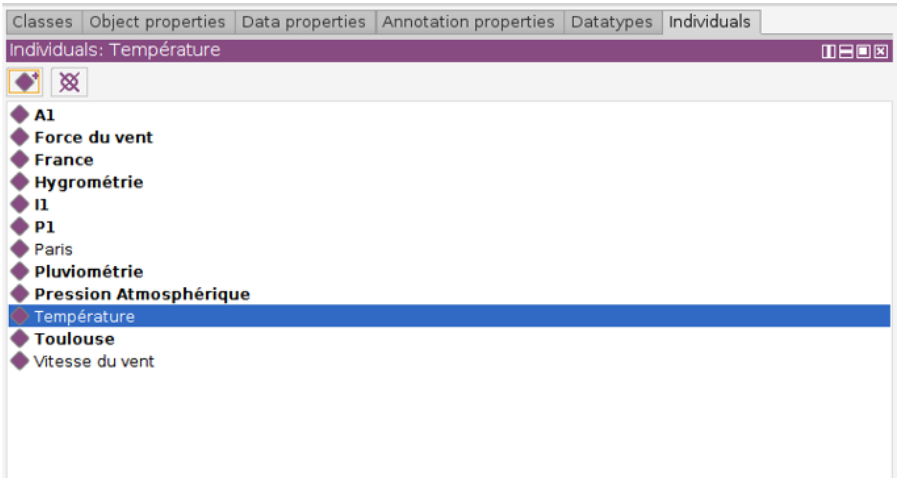

Figure 4: Peuplement de l'ontologie légère

2.2 L'ontologie lourde

Afin d'exprimer des relations plus complexes, des outils plus avancés doivent être utilisés.

2.2.1 Conception

Pour concevoir une ontologie lourde, nous devons utiliser la syntaxe Manchester qui nous permet d'écrire des relations plus avancées sur mes différentes classes de notre ontologie. Par exemple pour exprimer le fait qu'un phénomène court est un phénomène dont la durée est de moins de 15 minutes", nous avons écrit :

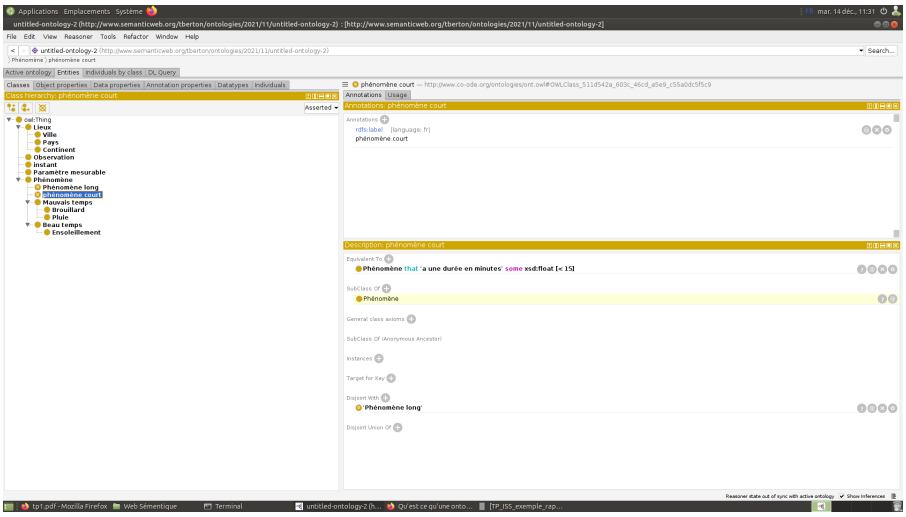

Figure 5: Propriété avancée pour phénomène court avec le langage Manchester

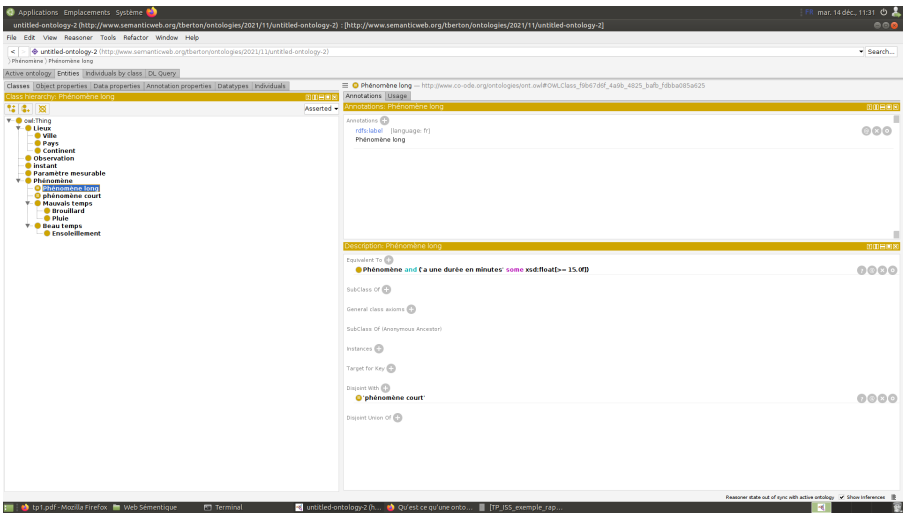

Figure 6: Propriété avancée pour phénomène long avec le langage Manchester

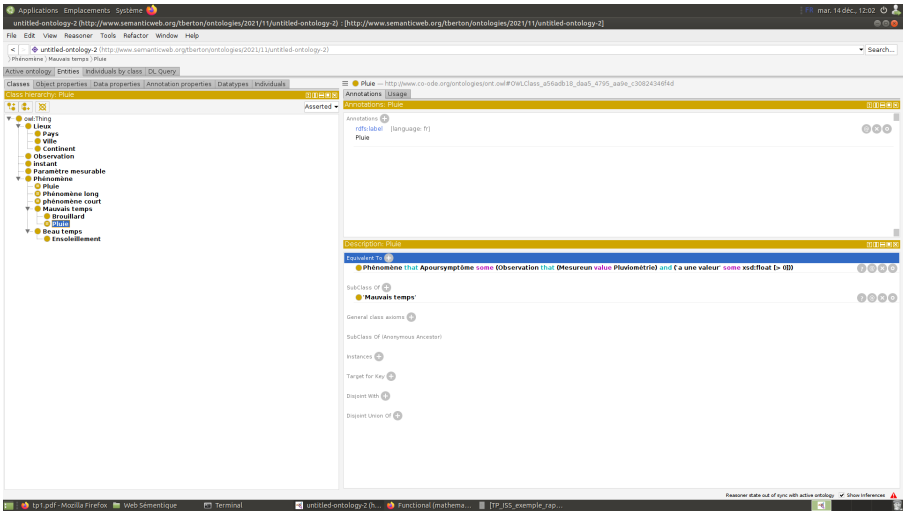

Figure 7: Propriété avancée pour pluie avec le langage Manchester

Nous avons également joué sur les caractéristiques des propriétés sur les relations que nous avions créé précédemment. Il y a par exemple la propriété de transitivité,la symétrie et la réflexivité. Dans cet exemple, nous n'avons utilisé que la propriété Inverse OF et la transitivité

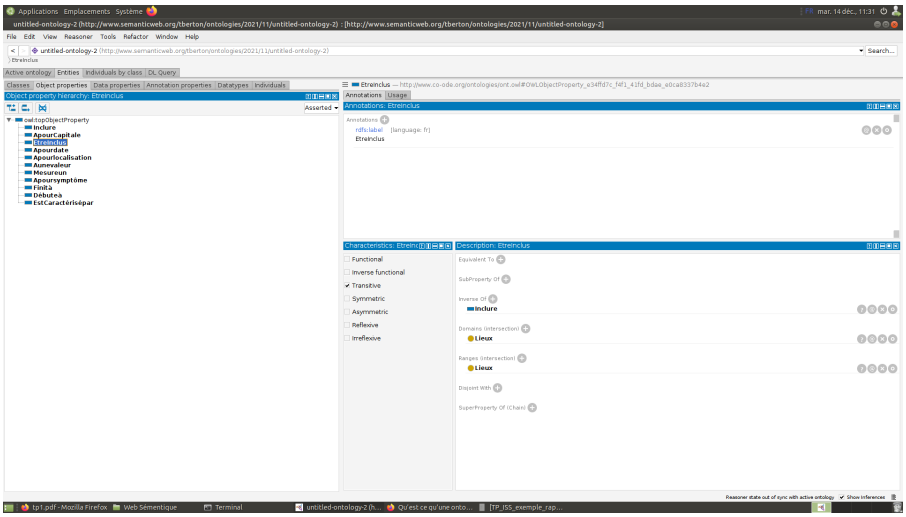

Figure 8: Propriété inverse of

2.2.2 Peuplement

Le raisonneur peut ensuite déduire des informations en se basant sur les règles définies dans l'ontologie. Dans l'exemple précédent, une déduction possible est "A1" est de classe *Phénomène* et "Paris" est de classe *Ville*. Après avoir exprimé les relations "Paris est la capitale de la France" et "la Ville Lumière est la capitale de la France", ainsi que "à tout pays correspond une et une seule capitale", le raisonneur a pu en déduire que les individus "Paris" et "La Ville Lumière" sont les mêmes.

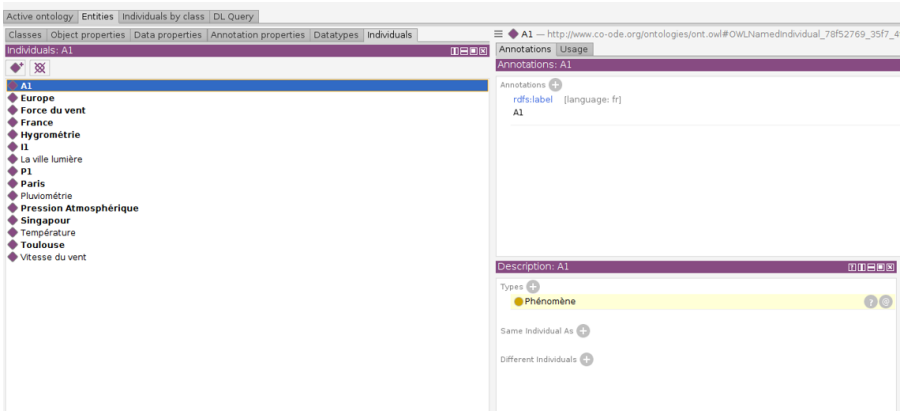

Figure 9: Déduction de la classe du raisonneur pour l'individu A1

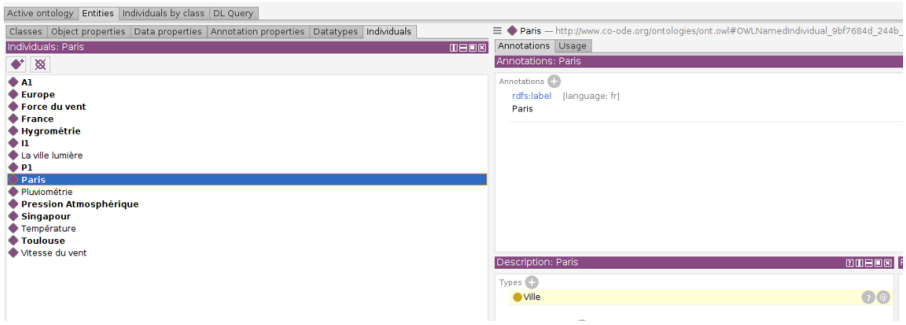

Figure 10: Déduction de la classe du raisonneur pour l'individu Paris

Maintenant si on ajoute Toulouse comme capitale de la France, alors on obtient que Toulouse est équivalent à Paris et Ville-Lumière. Et si on rajoute une condition sur le fait qu'un Pays ne peut contenir qu'une seule Capitale alors le fait que Toulouse et Paris n'appartiennent pas à la même région aurait déclenché un conflit pour le raisonneur.

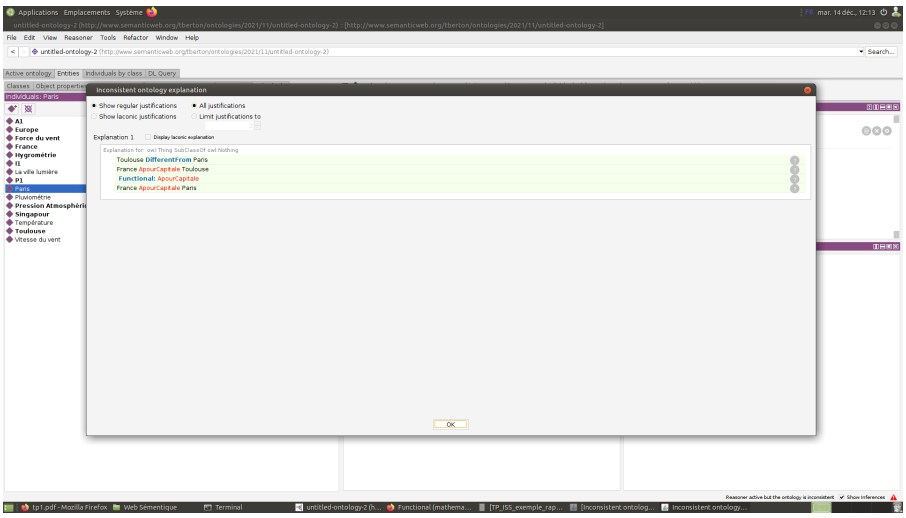

Figure 11: Réaction du raisonneur si Toulouse est déclarée comme la capitale de la France

3 Exploitation de l'ontologie

Cette seconde partie a pour but de se servir de l'ontologie créée précédemment en interagissant avec le jeu de données. Les méthodes ci-dessous permettent de créer ou récupérer des données issues de ce dataset (consulté avec l'éditeur Protégé). Chaque ressource est identifiée par une URI.

La première interface IModelFonctions est utilisée pour créer des individuals basés sur notre ontologie. L'interface implémentée DoItYourslef utilise le modèle sémantique pour facilement intéragir avec notre ontologie.

La deuxième interface, IControlFonctions a été utilisée pour créer des observations tout en utilisant l'interface IModelFonctions. Le rôle du controller est d'établir un lien entre le dataset et le meta-data. Nous avons donc complété le code manquant pour la première interface afin que notre ontologie puisse intéragir avec notre application Java. Ci-joint, le code :

```java
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) {
        // TODO Auto-generated method stub
        List<String> uri_place=this.model.getEntityURI("Lieu");

        return this.model.createInstance(name,uri_place.get(0));
    }

    @Override
    public String createInstant(TimestampEntity instant) {
        // TODO Auto-generated method stub
        List<String> list_instant=this.model.getInstancesURI(this.model.getEntityURI("Instant").get(0));
        for (int i=0;i<list_instant.size();i++) {
            if(model.hasDataPropertyValue(list_instant.get(i), model.getEntityURI("a pour timestamp").get(0), instant.getTimeStamp())) {
                return null;
            }
            else {
                String new_instance = model.createInstance("Instant", this.model.getEntityURI("Instant").get(0));
                model.addDataPropertyToIndividual(list_instant.get(i), model.getEntityURI("a pour timestamp").get(0), instant.getTimeStamp());
                return new_instance;
            }
        }
        return null;

    }
```

Figure 12: Première partie du code

```java
    @Override
    public String getInstantURI(TimestampEntity instant) {
        // TODO Auto-generated method stub
        List<String> list_instant=this.model.getInstancesURI(this.model.getEntityURI("Instant").get(0));
        for (int i=0;i<list_instant.size();i++) {
            if(model.hasDataPropertyValue(list_instant.get(i), model.getEntityURI("a pour timestamp").get(0), instant.getTimeStamp())) {
                return list_instant.get(i);
            }
            else {
                return null;
            }
        }
        return null;
    }

    @Override
    public String getInstantTimestamp(String instantURI)
    {
        // TODO Auto-generated method stub
        List<List<String>> list=model.listProperties(instantURI);
        for(int i=0;i<list.size();i++) {
            if(list.get(i).get(0).equals(model.getEntityURI("a pour timestamp").get(0))) {
                return list.get(i).get(1);
            }
        }

        return null;
    }
```

Figure 13: Deuxième partie du code

```java

     @Override
     public String createObs(String value, String paramURI, String instantURI) {
         // TODO Auto-generated method stub
         String new_obs=model.createInstance("Observation",model.getEntityURI("Observation").get(0));
         model.addDataPropertyToIndividual(new_obs,model.getEntityURI("a pour timestamp").get(0),instantURI);

         model.addObjectPropertyToIndividual(new_obs,model.getEntityURI("a pour valeur").get(0),value);

         model.addObjectPropertyToIndividual(new_obs,model.getEntityURI("mesure").get(0), paramURI);

         String sensor = model.whichSensorDidIt(this.getInstantTimestamp(instantURI), paramURI);
         model.addObservationToSensor(new_obs, sensor);

         return new_obs;
     }
}

```

Figure 14: Troisième partie du code

4 Question additionnelle

Un *Object property* va lier un objet à un autre objet alors qu'une *Data property* va lier un objet à une valeur.

5 Conclusion

A travers ces deux TP, nous avons pu découvrir les concepts du web sémantique. Dans le premier TP, nous avons utilisé le logiciel afin de créer notre propre ontologie. Le second TP était plus compliqué, nous avons eu plus de mal à manipuler les méthodes du framework Apache Jena. Mais nous avons tout de même réussit à tout implémenter.