Web Semantic and Internet of Things

December 5, 2018

1 Introduction

1.1 Objectif

Les Travaux pratiques s'inscrivent dans le cadre du cours de Traitement des données sémantiques et ont pour but de nous faire manipuler la notion d'ontologie. Nous avons abordé les aspects principaux, développer les différentes étapes de l'ontologie puis étudier les déductions faites par le raisonneur.

1.2 Contexte

Nous avons pour cela pris l'exemple d'une application météorologique intelligente que nous voulons développer. Afin de décrire les données mesurées par les stations météorologiques de la façon la plus précise possible, nous devons définir tous les termes relatifs à la météo, ainsi qu'aux stations.

1.3 Déroulement des Travaux Pratiques

Afin d'aborder la notion d'ontologie dans son ensemble, nous allons procéder en deux temps.

Dans un premier temps, grâce à l'outil Protégé, nous devons définir une ontologie de la météo contenant différents aspects tels que les capteurs, les lieux, les phénomènes, etc. Puis nous allons enrichir cette ontologie à l'aide de relation et de contraintes, puis tester les déductions de raisonneur Hermit sur des instances simples de notre exemple lors du peuplement de notre ontologie.

Dans un deuxième temps, nous allons décrire sémantiquement des données issues d'un open data. Une première étape de conversion sera nécessaire afin de repérer puis définir la notion de capteur défectueux. Finalement, nous pourrons automatiquement identifiés les capteurs défectueux.

2 Création de l'ontologie météorologique

Nous devons définir un vocabulaire technique dans le but de décrire sémantiquement les données relevées par les capteurs.

Concrètement, cette ontologie doit représenter les phénomènes météorologiques, comme la pluie, ou le brouillard, en termes de lieux, dates, et de paramètres mesurables qui les caractérisent, par exemple la température ou la pluviométrie.

2.1 L'ontologie légère

Nous créons différentes classes et sous-classes pour représenter les différents éléments relatifs à notre ontologie.

2.1.1 Conception de l'ontologie météorologique

Voici les classes que nous avons créées pour notre ontologie :

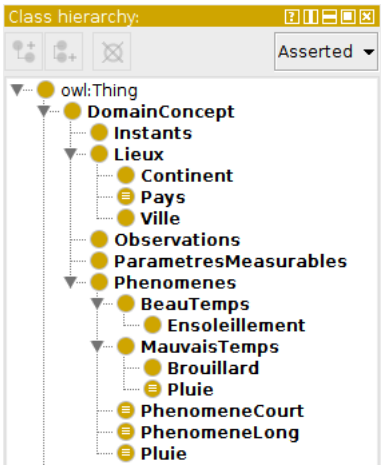

Figure 1: Ensemble des classes de l'ontologie météorologique

Nous avons créé une classe *DomaineConcept* afin de regrouper toutes les classes spécifiques à notre ontologie.

Nous avons veillé à respecter les connaissances de l'ontologie en créant les classes appropriées, ce qui permet de définir les phénomènes météorologiques de façon précise, détaillée et spécifique.

Ainsi, nous avons créé une classe *Phenomenes* afin de regrouper les classes *BeauTemps* et *MauvaisTemps*. Au sein de la classe *MauvaisTemps*, nous avons définit les classes *Brouillard* et *Pluie*, de même au sein de la classe *BeauTemps*, nous avons inclut la classe *Ensoleillement*. Par cette hiérarchie, toutes les instances des classes filles vont héritées des attributs des classes mères.

Ensuite, nous avons créé une classe *Lieux* pour définir les lieux où se produisent les phénomènes météorologiques. Au sein de cette classe, nous avons créé les classes : *Ville*, *Pays* et *Continent*.

De plus, pour définir des phénomènes météorologiques, nous avons créé les classes : *Instants*, *Observations* et *ParametresMeasurables*.

Une fois les classes définies, nous pouvons créer des propriétés pour définir les contraintes que chacune doit respecter, ainsi que les relations qui existent entre elles. Pour ce qui est de la définition des propriétés des classes, nous avons donc créé les relations suivantes :

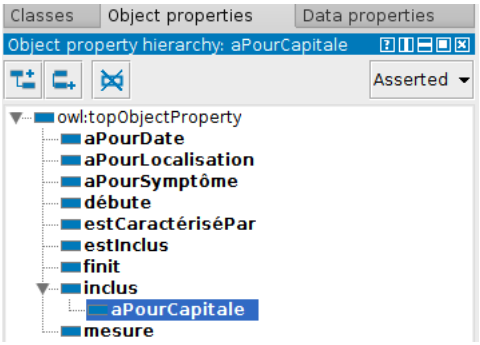

Figure 2: Propriétés des classes définies

Chaque propriété a pour rôle de définir la relation entre deux classes, ainsi nous pouvons voir que comme une *observation* météo a pour date un *instant*, nous devons créer la propriété **aPourDate** qui met en relation les deux classes. Il en est de même pour toutes les autres relations.

Nous devons différencier les propriétés qui définissent les relations entre deux classes, et les propriétés qui définissent le type de la classe. Ainsi, nous devons déclarer quel est le type de données que contiennent les classes *Instants*, *Observations* et *Phenomenes*. Vous pouvez voir ci-dessous les propriétés que nous avons définit:

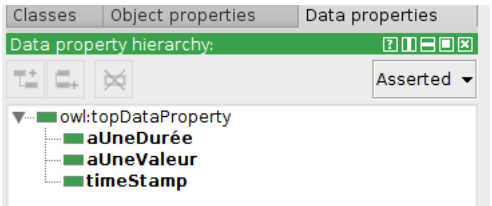

Figure 3: Propriétés des données des classes définies

2.1.2 Peuplement de l'ontologie légère

Après avoir représenté les connaissances de l'ontologie, nous pouvons décrire des faits simples relatifs à la météo. Voici les instances des classes que nous avons créées :

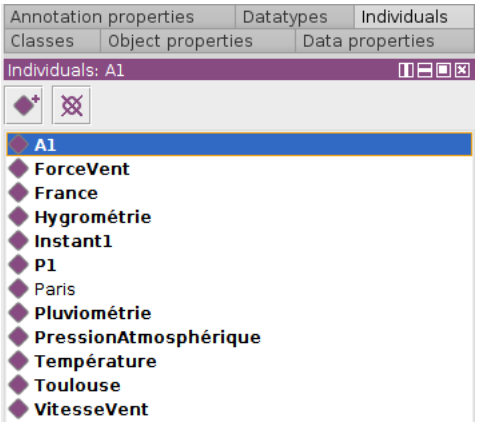

Figure 4: Peuplement de l'ontologie légère

Pour chacun des faits décris ci-dessus, nous avons lancé le raisonneur *Hermit*. Nous avons obtenu les résultats suivants :

- Nous avons déclaré les instances *Toulouse* et *France* comme des *Lieux*, nous n'avons pas typé ces instances. Lorsque nous avons précisé que *Toulouse* est inclus en *France*, le raisonneur *Hermit* en a déduit que la *France* inclus aussi *Toulouse*, comme les deux relations sont inverses.

- Nous avons déclaré que *A1* a pour symptôme *P1* qui est une observation. Or la relation *aPourSymptôme* lie des observations à des phénomènes, ainsi le raisonneur *Hermit* en a déduit que *A1* est un phénomène.

- Nous avons précisé que *Paris* est la capitale de la *France*, et nous n'avons pas typé *Paris*. Le raisonneur *Hermit* a donc déduit que *Paris* était une *ville* et que la *France* était un *pays*.

2.2 L'ontologie lourde

2.2.1 Conception

Nous avons utilisé la syntaxe ***Manchester*** afin de décrire les connaissances que nous savons sur l'ontologie.

Nous avons ainsi précisé, par exemple, qu'une *Ville* ne peut pas être a priori un *Pays*, ainsi ces deux classes sont disjointes.

Nous pouvons aussi modéliser des classes définies, qui ont pour particularité d'avoir des valeurs contraintes (supérieure à 0 pour la pluviométrie par exemple). Ainsi, nous pouvons aussi diviser les *phénomènes* en deux parties : les courts (< 15 min) et les longs (> 15 min). Ces derniers sont donc deux phénomènes disjoints.

Dans la question 3.5 on utilise la relation d'inverse. Il est ainsi possible de définir deux propriétés comme étant l'inverse l'une de l'autre. Dans notre cas *inclus* et *estInclus* sont deux propriétés inverses car si un lieu A est inclus dans un autre lieu B, cela signifie donc que B inclut A.

La question 3.6 illustre la caractéristique de transitivité qui veut que si A est inclus dans B et B est inclus dans C alors forcément A est inclus dans C. Par exemple, si Paris est en France et la France est en Europe, alors Paris est en Europe.

A la question 3.7, on cherche à exprimer l'unicité. On utilise la syntaxe ***Manchester*** *exactly* pour spécifier par exemple qu'un pays ne peut avoir qu'une seule ville : "aPourCapitale exactly 1 Ville". Une autre façon de faire est d'utiliser la caractéristique "functional" de *aPourCapitale*.

Enfin, pour la question 3.8, on utilise la notion de sous-propriété. Cela permet de lier des propriétés et d'établir des relations entre elles. Ici, si *aPourCapitale* est sous-propriété de *inclus*, alors tout pays qui a pour capitale une ville, inclus forcément celle-ci.

2.2.2 Peuplement

Nous pouvons compléter le peuplement que nous avons déjà réalisé précédemment, et nous obtenons donc les instances suivantes :

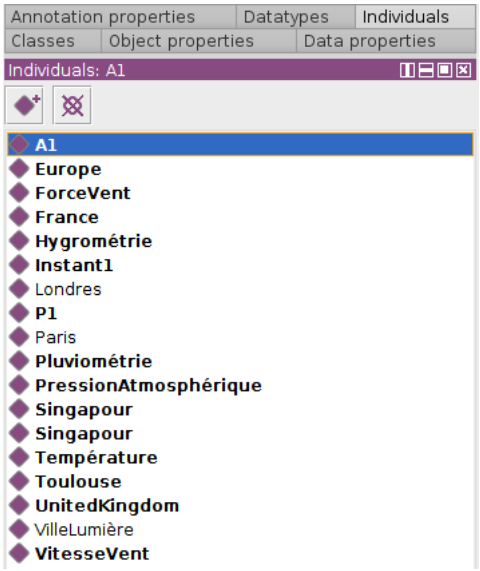

Figure 5: Peuplement de l'ontologie lourde

Grâce à l'attribut de transitivité, le raisonneur déduit que Paris et Toulouse sont inclus dans l'Europe, puisque ces deux villes sont inclues en France et que la France est inclue dans l'Europe.

Quand on définit à la fois *Paris* et *La ville lumière* comme capitales de la France, le raisonneur déduit que ces deux villes sont confondus "same individual as", ce qui signifie donc que *Paris* est *La ville lumière*.

Si on essaie de définir Singapour comme étant à la fois une ville et un pays, une erreur apparaît, puisque nous avons spécifié qu'une ville ne peut être un pays (Q 3.1). En revanche, il nous est possible de définir deux instances Singapour différentes, une étant une ville et l'autre étant un pays.

Le raisonneur déduit de A1 qu'il est de type *phénomène*. Nous avons également mis la valeur de la durée de A1 à 45 et le raisonneur change alors le type de A1 comme étant un *phénomène long*.

3 Exploitation de l'ontologie

Le but de cette partie est d'écrire un programme en Java, à l'aide d'une API, afin d'enrichir et d'annoter un dataset sémantiquement. Cette version enrichie sera ensuite exploiter dans Protégé.

Le dataset utilisé correspond à un open data ouvert par la ville d'Aarhus au Danemark. Ce dernier recense des observations faites par des capteurs météorologiques au sein de cette ville.

Par rapport aux données brutes issues du dataset, qui nous sont fournies au format CSV, l'ontologie joue le rôle de convertisseur donnant en sortie des données 5 étoiles, ce qui signifie qu'elles sont liées entres elles.

Comme pour la manipulation via Protégé, l'API fait une distinction entre le label et l'URI. Pour illustrer cela, prenons l'exemple de la fonction *createPlace(String name)*, qui permet la création d'une instance de type *Lieu*. Cette dernière va alors faire appel à la fonction *createInstance(String label, String type)* afin de créer l'instance. Les arguments de cette fonction sont :

- *label :* qui représente le nom de l'instance.
- *type :* qui correspond à l'URI du type (ici *Lieu*) de l'instance.

Ainsi, cette exemple montre bien que le label et l'URI sont deux choses distinctes. Pour créer notre fonction *createPlace(String name)* il nous a donc fallu déterminer l'URI du type *Lieu*. Pour cela, nous avons utilisé la fonction *getEntityURI(String label)* qui permet d'obtenir l'URI correspondant au nom de la classe entrée en paramètre (ici *Lieu*. Voici le code de la fonction *createPlace(String name)* :

```
@Override
public String createPlace(String name) {
    // TODO Auto-generated method stub

    // public String createInstance(String label, String type)
    return model.createInstance(name,model.getEntityURI("Lieu").get(0));
}
```

Figure 6: Fonction createPlace

De la même manière, nous avons créé la fonction *createInstant(TimestampsEntity instant)* qui permet la création d'une instance de type *Instant*. Cependant, ici nous devions nous assuré de l'unicité de cette instance. Comme nous pouvons le voir dans le code de la fonction ci-dessous, une fois cette unicité vérifiée, nous créons l'instance, puis nous lui attribuons la propriété *"a pour timestamp"*, représentant ce timestamp.

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

    // TODO Auto-generated method stub
    String name = instant.getTimeStamp();
    String type = model.getEntityURI("Instant").get(0);

    if(!model.getEntityURI(name).isEmpty()) {
        return null;
    }
    else {
        String propertyURI = model.getEntityURI("a pour timestamp").get(0);
        String subjectURI= model.createInstance(name,type);
        //public void addDataPropertyToIndividual(String subjectURI, String propertyURI, String data);
        model.addDataPropertyToIndividual(subjectURI, propertyURI, instant.getTimeStamp());
        return subjectURI;
    }
}
```

Figure 7: Fonction createInstant

Ensuite, nous avons créé une fonction permettant de récupérer l'URI d'une instance de type *Instant* à partir de son timestamp. Cette fonction, *getInstantURI(TimestampEntity)*, va dans un premier temps chercher le label de l'instance de type *Instant* passée en paramètre, puis, si ce dernier existe, renvoyer l'URI de cette instance en se basant sur ce *"name"* :

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

    String name = instant.getTimeStamp();
    if(model.getEntityURI(name).isEmpty()) {
        return null;
    }
    else {
        return model.getEntityURI(name).get(0);
    }
}
```

Figure 8: Fonction getInstantURI

La fonction *getInstantTimestamp(String instantURI)* prend elle en argument l'URI d'une instance de type *Instant* et nous renvoie le timestamp correspondant. Pour cela, nous avons parcourut une liste de couple $< property, object >$, correspondant à l'URI donné en paramètre, afin de trouver celui qui correspond à la propriété *a pour timestamp*. Une fois ce couple trouvé, il nous suffit de retourner la valeur du timestamp qu'il contient.

```
@Override
public String getInstantTimestamp(String instantURI)
{

    int index = 0;
    if(model.listProperties(instantURI).isEmpty()) {
        return null;
    }
    else {
        String propertyURI = model.getEntityURI("a pour timestamp").get(0);
        for(int i = 0; i<model.listProperties(instantURI).size(); i++) {
            if(model.listProperties(instantURI).get(i).get(0).equals(propertyURI)) {
                index = i;
            }
        }
        return model.listProperties(instantURI).get(index).get(1);
    }
}
```

Figure 9: Fonction getInstantTimestamp

Enfin, la dernière fonction implémentée *createObs(String value, String paramURI, String instantURI)* permet de créer une observation et de lui associer une valeur (mesure), un instant et un paramètre. Pour cela, comme pour la fonction *createInstant*, nous avons utilisé la fonction *addObjectPropertyToIndividual* pour attribuer à l'observation un paramètre *"mesure"*, et aussi un instant *"a pour date"*. Enfin, nous lui attribuons une valeur *"value"* grâce à la fonction *addDataPropertyToIndividual*.

```
@Override
public String createObs(String value, String paramURI, String instantURI) {

    String dataValueURI = model.getEntityURI("a pour valeur").get(0);
    String instantPropURI = model.getEntityURI("a pour date").get(0);
    String paramPropURI = model.getEntityURI("mesure").get(0);
    String obsClassURI = model.getEntityURI("Observation").get(0);
    String obsURI = model.createInstance("observationValue", obsClassURI);

    String sensorURI = model.whichSensorDidIt(getInstantTimestamp(instantURI), paramURI);
    model.addObservationToSensor(obsURI, sensorURI);

    model.addObjectPropertyToIndividual(obsURI, instantPropURI, instantURI);
    model.addObjectPropertyToIndividual(obsURI, paramPropURI, paramURI);
    model.addDataPropertyToIndividual(obsURI, dataValueURI, value);


    return obsURI;
}
```

Figure 10: Fonction createObs

Après avoir implémenté toutes ces fonctions, nous les avons utilisé pour effectuer l'enrichissement effectif du dataset. Pour cela, nous utilisons la fonction *controler*. Ainsi, nous obtenons un fichier *export.ttl*. Nous avons alors importé ce fichier dans Protégé. Malheureusement, nous ne sommes pas parvenu à utiliser le raisonneur sur ce modèle.

Enfin, il nous été demandé d'expliquer la différence entre *object property* et *data property* :

- *Object Property :* permet de lier une instance de classe à une autre. Par exemple, on peut lier un lieu à un autre lieu par un lien de type "est inclus dans" ou "a pour capitale".
- *Data Property :* permet de lier une instance de classe à une valeur (data). Par exemple, on peut lier un paramètre mesurable, comme la température, à une valeur avec le lien "a pour valeur". Ces données peuvent ou non avoir une unité.