OWL

Read

- Foundations of Semantic Web Technologies: chapter 4, 5.

Supplementary reading:

- OWL Pizzas: Practical Experience of Teaching OWL-DL:Common Errors & Common Patterns

Now we will take our family ontology one step further by adding more semantics using OWL. First, for a soft start and to get into Protégé, ontology editing and OWL, we will start by looking at an existing tutorial ontology, the pizza ontology. Parts of this exercise will be a revisit of first week's exercise.

1 The Pizza ontology

The pizza ontology is a well-known ontology in the semantic web community. It is developed for educational purposes by the University of Manchester, which is a leading university in the development of semantic technologies.

The pizza ontology and a tutorial that uses it is found at

- `http://protegewiki.stanford.edu/wiki/Protege4Pizzas10Minutes`
- `http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/protg-owl-tutorial/`

The tutorial is primarily for learning how to use Protégé 4. Use it to get help on how to use Protégé in the coming exercises.

1.1 Exercise

Open the pizza ontology in Protégé. Take some time to browse the class hierarchy, the property hierarchies and the individuals and note how the ontology describes the domain of pizzas.

1.2 Exercise

Find Margherita and see how it is defined as a pizza with only cheese and tomato topping. Look at the definition of VegetarianPizza. Is a Margherita pizza a vegetarian pizza? Why / why not?

1.3 Exercise

Find `hasIngredient`. What is the domain and range of this property? What are the subproperties of `hasIngredient`? What is the inverse property of `hasIngredient`? What property characteristics does `hasIngredient` have?

1.4 Exercise

Classify the ontology by choosing a reasoner and then "classify" in the reasoner menu. In the "Inferred class hierarchy" two classes show up as subclasses of `owl:Nothing`. Answer the following questions:

- In general, what is the difference between the asserted class hierarchy and the inferred class hierarchy?
- What does it mean for a class to be a subclass of `owl:Nothing`?
- Explain why these two classes appear as subclasses of `owl:Nothing`.
- Find Margherita in the inferred class hierarchy and see which classes are inferred as superclasses of Margherita.

1.5 Exercise

Add a new class Grandiosa as a subclass of NamedPizza. Define "Grandiosa" as something which

- `hasTopping some HamTopping`,
- `hasTopping some TomatoTopping` and
- `hasTopping some CheeseTopping`.

Classify the ontology. What superclasses are inferred as superclass of Grandiosa? Explain why.

1.6 Exercise

State in the ontology that a Grandiosa pizza comes from Norway, and that Norway is different from the other countries already present in the pizza ontology. Apply reasoning and explain the results.

2 Family relations in OWL

So far we have only been allowed to use RDFS vocabulary to describe family relations. Now we will extend our description using OWL constructs. OWL is more expressive than RDFS and allows us to express many more restrictions on properties and class membership than RDFS does.

In this exercise we will only use OWL (1) DL vocabulary (and not OWL 2, which will be next week's exercises). This language is explained in W3C's OWL Web Ontology Language Reference, which may be a valuable resource for these exercises. OWL Web Ontology Language Overview contains a list of the constructs available in RDFS and the different dialects of OWL 1: OWL lite, OWL DL and OWL Full. See also W3C's "portal" on OWL.

You may use Protégé as your editor, but you are also welcome to use a plain text editor to the exercises. Note that there are different OWL languages and that different editors have different tastes. If you are using Protégé as editor, consult the Protégé pizza tutorial. If your using a plain text editor, use the OWL validator and try also regularly to open your file in Protégé. If you have problems using Protégé, consult the Protégé OWL Tutorial.

The OWL vocabulary we will use is listed below. The list is a slightly compacted version of the one found on OWL Web Ontology Language Overview. Almost all items in the list will be put to use in these exercises.

- RDFS Features: `Class`, `rdfs:subClassOf`, `rdf:Property`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, `Individual`
- Header Information: `Ontology`, `imports`
- Annotation Properties, `rdfs:label`, `rdfs:comment`, `rdfs:seeAlso`, `rdfs:isDefinedBy`, `AnnotationProperty`, `OntologyProperty`
- Class Axioms: `oneOf`, `dataRange`, `disjointWith`, `unionOf`, `complementOf`, `intersectionOf`
- (In)Equality: `equivalentClass`, `equivalentProperty`, `sameAs`, `differentFrom`, `AllDifferent`, `distinctMembers`
- Property Characteristics: `ObjectProperty`, `DatatypeProperty`, `inverseOf`, `TransitiveProperty`, `SymmetricProperty`, `FunctionalProperty`, `InverseFunctionalProperty`
- Property Restrictions: `Restriction`, `onProperty`, `allValuesFrom`, `someValuesFrom`,
  `minCardinality`, `maxCardinality`, `cardinality`, `hasValue`
- Datatypes: XSD datatypes

For each of the modelling exercises below express the exercise text as a set of description logic (DL) axioms.

2.1 Exercise

Make a new ontology file. Give it the namespace

`http://www.ifi.uio.no/INF3580/v18/family.owl#`

Import the family RDFS file you wrote in last week's exercise.

2.1.1 Tip

Note, as mentioned in an exercise in last week's exercises, not all ontology editors and reasoners interprets manages to handle RDFS as OWL, so you might want to convert your family RDFS file to OWL. Changing all instances of `rdfs:Class` to `owl:Class` and instances of `rdf:Propery` to either `owl:ObjectProperty` or `owl:DatatypeProperty` should take care of most convertion problems.

2.1.2 Solution

I will be using a plain text editor to write the OWL file, so it is easily included it in this document.

```
1  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix owl: <http://www.w3.org/2002/07/owl#> .
4  @prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
5  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6  @prefix fam: <http://www.ifi.uio.no/INF3580/v18/family#> .
7  @prefix :  <http://www.ifi.uio.no/INF3580/v18/family.owl#> .
8  @base <http://www.ifi.uio.no/INF3580/v18/family.owl> .
9
10 <http://www.ifi.uio.no/INF3580/v18/family.owl> rdf:type owl:Ontology ;
11   owl:imports <http://www.ifi.uio.no/INF3580/v18/family> .
```

2.2 Exercise

State that a person has at least one father and one mother.

2.2.1 Tip 1

The exercises are formulated in normal language on purpose. It is up to you to decide how this is best expressed in OWL.

2.2.2 Tip 2

My solution (yours may be different) as a DL axiom:

$$Person \sqsubseteq \exists hasFather.Person \sqcap \exists hasMother.Person$$

2.2.3 Solution

To state this in Protégé add an anonymous superclass to `Person` with the expression

```
hasFather some Person
and hasMother some Person
```

With this we have stated that for every instance of the class Person there must be a hasFather property instance to an instance of Person and a hasMother property instance to a instance of Person. In N3 the same statements look like:

```
12  foaf:Person rdfs:subClassOf [
13    owl:intersectionOf (
14      [ rdf:type owl:Restriction ;
15        owl:onProperty fam:hasFather ;
16        owl:someValuesFrom foaf:Person
17      ]
18      [ rdf:type owl:Restriction ;
19        owl:onProperty fam:hasMother ;
20        owl:someValuesFrom foaf:Person
21      ]
22    )] .
```

Note that there may be many ways to express the same statement. The statement can also been expressed by using minimum cardinality of 1 for Person on both of hasFather and hasMother, in a similar fashion as the solution above. In Protégé

```
hasFather min 1 Thing
hasMother min 1 Thing
```

2.3 Exercise

State that a person can only have one mother and only one father.

2.3.1 Solution

To state this I have used `owl:FunctionalProperty` and stated that both `:hasFather` and `:hasMother` are functional properties. Quoting FunctionalProperty-def:

> A functional property is a property that can have only one (unique) value y for each instance x, i.e. there cannot be two distinct values y1 and y2 such that the pairs (x,y1) and (x,y2) are both instances of this property.

This works since the domain of both properties is Person and the range for `:hasFather` and `:hasMother` are `:Father` and `:Mother` respectively.

In DL it looks like this

$$\geq 2 hasFather.\top \sqsubseteq \bot \qquad \geq 2 hasMother.\top \sqsubseteq \bot$$

which can be translated to "nothing has to hasFather/hasMother relations".

In Protégé it is expressed by ticking the "Functional" box for each of the properties. In N3 it looks like this:

```
23  fam:hasFather rdf:type owl:FunctionalProperty .
24  fam:hasMother rdf:type owl:FunctionalProperty .
```

We could also have stated this by using maximum cardinality:

```
hasFather max 1 Thing
hasMother max 1 Thing
```

2.4 Exercise

State that a woman can only have female as gender, and a man can only have male as gender.

2.4.1 Solution

One way to express this in OWL is to say that all instances of `Woman` can only have the value `Female` for the property `hasGender`. To do this we must make an anonymous class that only contains the individual `Female`. In Protégé this is done by using the curly brackets. The whole expression is for `Woman`

```
hasGender only {Female}
```

and for `:Man`

```
hasGender only {Male}
```

In DL:

$$Woman \sqsubseteq \forall hasGender.\{Female\} \qquad Man \sqsubseteq \forall hasGender.\{Male\}$$

In N3:

```
25  fam:Woman rdfs:subClassOf
26    [ rdf:type owl:Restriction ;
27      owl:onProperty fam:hasGender ;
28      owl:allValuesFrom
29         [ rdf:type owl:Class ;
```

```
30    owl:oneOf ( fam:Female )
31    ]  ] .
32
33  fam:Man rdfs:subClassOf
34    [ rdf:type owl:Restriction ;
35      owl:onProperty fam:hasGender ;
36      owl:allValuesFrom
37         [ rdf:type owl:Class ;
38    owl:oneOf ( fam:Male )
39    ]  ] .
```

2.5 Exercise

State that nothing can be both male and female.

2.5.1 Solution

This is done by adding `:Male` to the (empty) list of different individuals for `:Female`. In RDF syntax the keyword is `owl:differentFrom`.

In DL one normally operate with a unique name assumption, so if different constants represent different objects, but if not it would simply be:

$$Male \neq Female$$

In N3:

```
40  fam:Female owl:differentFrom fam:Male .
```

2.6 Exercise

Define the gender so that there can only be the genders man and woman.

2.6.1 Solution

State this by making the class `Gender` equivalent to the class `{Female, Male}`. In RDF syntax `owl:oneOf` is used to define a class by listing all the members of the class:

In DL:

$$Gender \equiv \{Male, Female\}$$

In N3:

```
41  fam:Gender owl:equivalentClass
42    [ rdf:type owl:Class ;
43      owl:oneOf ( fam:Male fam:Female )
44    ] .
```

2.7 Exercise

Explain what disjointness is. For all pair of classes in the family ontology, add the correct disjoint axioms.

2.7.1 Solution

Quoting disjointWith-def:

> `owl:disjointWith` is a built-in OWL property with a class description as domain and range. Each `owl:disjointWith` statement asserts that the class extensions of the two class descriptions involved have no individuals in common. Like axioms with `rdfs:subClassOf`, declaring two classes to be disjoint is a partial definition: it imposes a necessary but not sufficient condition on the class.

In our case it is safe to say that all the classes Family, Gender and Person are pairwise disjoint, and that the classes Man and Woman are disjoint.

In DL:

$$Family \sqcap Gender \sqsubseteq \bot \qquad Family \sqcap Person \sqsubseteq \bot$$

$$Gender \sqcap Person \sqsubseteq \bot$$

$$Man \sqcap Woman \sqsubseteq \bot$$

which translates to "nothing is both man and woman".

In N3:

```
45  fam:Family owl:disjointWith fam:Gender, foaf:Person .
46  fam:Gender owl:disjointWith foaf:Person .
47  fam:Man owl:disjointWith fam:Woman .
```

2.8 Exercise

State that a person is either a man or a woman, but not both.

2.8.1 Solution

This translates to "the `Person` is equivalent to the union of `Man` and `Woman`, and `Man` and `Woman` are disjoint".

The classes `Man` and `Woman` are already stated as disjoint, so we need only define that `Person` is equivalent to the union of `Man` and `Woman`.

In DL:

$$Person \equiv Man \sqcup Woman$$

In N3:

```
48  foaf:Person owl:equivalentClass
49    [ rdf:type owl:Class ;
50      owl:unionOf ( fam:Man fam:Woman )
51    ] .
```

2.9 Exercise

Explain what inverse properties are. For all the properties that exist in our ontology, add the correct inverse property axioms. You are not supposed to add new properties, only state that a property is the inverse of an other property if they already exist in the ontology.

2.9.1 Solution

Quoting inverseOf-def:

> An axiom of the form P1 owl:inverseOf P2 asserts that for every pair (x,y) in the property extension of P1, there is a pair (y,x) in the property extension of P2, and vice versa.

To illustrate this let us use an example: If Homer is the husband of Marge, than Marge must be the wife of Homer. If this is true in every possible case, then `hasHusband` is the inverse of `hasWife`, from which it follows that `hasWife` is the inverse of `hasHusband`.

In DL it is common to use $\cdot^-$ to indicate the inverse of a relation:

$$hasChild \equiv hasParent^- \qquad hasHusband \equiv hasWife^-$$

In N3:

```
52  fam:hasChild owl:inverseOf fam:hasParent .
53  fam:hasHusband owl:inverseOf fam:hasWife .
```

2.10 Exercise

Explain what it means for a property to be transitive or symmetric.

For all the properties in our ontology, if it is natural, state that they are transitive and/or symmetric.

There is no standard way of asserting characteristics for properties in DL, so you may skip this part. The more or less *common* way of assering that a property P is asymmetric, symmetric, reflexive, reflexive or transitive in DL literature is $\mathsf{Asym}(P)$, $\mathsf{Sym}(P)$, $\mathsf{Ref}(P)$, $\mathsf{Irr}(P)$ or $\mathsf{Tra}(P)$, respectively.

To say that two properties P_1 and P_2 are disjoint is commonly done in DL literature with $\mathsf{Dis}(P_1, P_2)$.

2.10.1 Solution

A property R is symmetric if, for all a and b, a is related to b by R means that also b is related to a by R. The symmetric properties are in our case `fam:isRelativeOf`, `fam:hasSibling` and `fam:hasSpouse`. If `fam:hasSibling` is symmetric, then if Bart is the sibling of Lisa, then Lisa must be a sibling of Bart—which is reasonable.

A property R is transitive if, for all a, b and c, a is related to b by R and b is related to c by R means that a is related to c by R. The transitive properties are in the family ontology `fam:hasBrother`, `fam:hasSister` and `fam:hasSibling`. If `fam:hasSister` is transitive then if Bart has a sister Lisa and Lisa has a sister Maggie, than Maggie is also the sister of Bart—which is also reasonable.

```
54  fam:isRelativeOf rdf:type owl:SymmetricProperty .
55
56  fam:hasBrother rdf:type owl:TransitiveProperty .
57  fam:hasSister rdf:type owl:TransitiveProperty .
58  fam:hasSibling rdf:type owl:TransitiveProperty .
59  fam:hasSibling rdf:type owl:SymmetricProperty .
60
61  fam:hasSpouse rdf:type owl:SymmetricProperty .
```

2.11 Exercise

Is a subproperty of a transitive property necessarily also transitive? Explain why / why not?

2.12 Exercise

Is a subproperty of a symmetric property necessarily also symmetric? Explain why / why not?

2.13 Exercise

Explain what it means for a property to be inverse functional.

For all properties in our ontology, state that they are inverse functional if you believe that is correct.

2.13.1 Solution

Quoting InverseFunctionalProperty-def:

> If a property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject (some individual). More formally, if we state that P is an

owl:InverseFunctionalProperty, then this asserts that a value y can only be the value of P for a single instance x, i.e. there cannot be two distinct instances x1 and x2 such that both pairs (x1,y) and (x2,y) are instances of P.

Assume `foaf:name` is inverse functional. Than all persons must have a distinct name, which clearly is not correct. This *is* correct for the Simpson instances of the family ontology, but it does not hold in general.

There is no property that is a good candidate for a inverse functional property.

Note also that `foaf:name` is an `owl:DatatypeProperty` and such properties are not allowed as inverse functional in OWL DL.

3 OWL metrics

3.1 Exercise

Make a java program which loads an OWL ontology and lists

- the number of classes,
- the number of object properties,
- the number of datatype properties,
- the number of individuals and
- the DL expressivity of the ontology.

Use a Pellet reasoner to do your reasoning.

3.1.1 Tip

You should get the same results from your program as you get when loading an ontology in Protégé.

I used the Pellet API to get hold of the expressivity of the ontology, using the classes `JenaLoader` and `KnowledgeBase`.

3.1.2 Solution

My solution is a program called `OWLMetrics` and extends the program `RDFSMetrics`.

```
1  import org.apache.jena.rdf.model.*;
2  import org.apache.jena.ontology.*;
3  import org.apache.jena.reasoner.*;
4  import org.mindswap.pellet.jena.*;
```

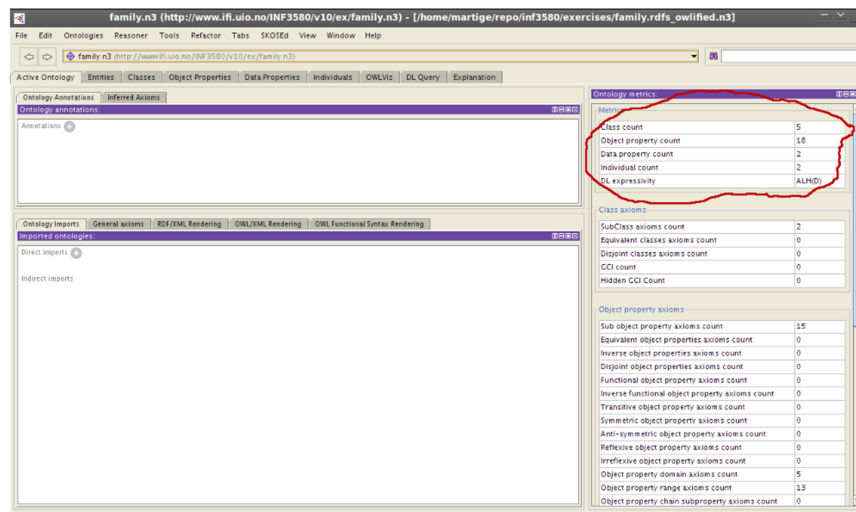

Figure 1: Protégé screenshot with its relevant metrics section marked.

```
5   import org.mindswap.pellet.*;
6   import com.clarkparsia.pellet.expressivity.*;
7
8   public class OWLMetrics extends RDFSMetrics{
9
10    protected static OntModel modelOWL;
11
12    public void readModel(String file){
13      super.readModel(file);
14      Reasoner r = PelletReasonerFactory.theInstance().create();
15      InfModel emptyOWL = ModelFactory.createInfModel(r, ModelFactory.createDef
16      InfModel allOWL = ModelFactory.createInfModel(r, modelRDF);
17      Model diff = allOWL.difference(emptyOWL);
18
19      OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_MEM);
20      modelOWL = ModelFactory.createOntologyModel(spec, diff);
21    }
22
23    public void printMetrics(){
24      System.out.println("Named classes: " +
25          getIteratorSize(modelOWL.listNamedClasses()));
26      System.out.println("Named object properties: " +
27          getIteratorSize(modelOWL.listObjectProperties()));
28      System.out.println("Named datatype properties: " +
29          getIteratorSize(modelOWL.listDatatypeProperties()));
30      System.out.println("Named individuals: " +
```

```
31          getIteratorSize(modelOWL.listIndividuals()));
32    }
33
34    public void printExpressivity(String URI){
35      JenaLoader jl = new JenaLoader();
36      KnowledgeBase kb = jl.createKB(URI);
37      System.out.println("Expressivity: " + kb.getExpressivity());
38    }
39
40    public static void main(String args[]){
41      OWLMetrics dave = new OWLMetrics();
42      dave.readModel(args[0]);
43      dave.printMetrics();
44      dave.printExpressivity(args[0]);
45    }
46  } // end class
```

3.2 Exercise

Test the metrics of your family ontology.

3.2.1 Tip

Note that if your file uses RDFS class or properties, you can have trouble getting the results you expect from Jena, so it is smart to convert the relevant RDFS constructs to OWL. This is easily done manually, as explained in an eariler exercise for this week, or you can open the file in Protégé and save it, which should convert it to OWL.