The People_Example Ontology

Version 1.0

10/7/2021

Michael DeBellis[1]

1. Introduction

This document is a supplementary addition to an article in process of being published that discusses the essential Semantic Web technologies. In that article we include an example ontology called the People_Example to illustrate some OWL, SWRL, and SPARQL. concepts. This document is for those who wish to follow the examples in the article as well as some additional examples that exceeded the scope of the article. For those who haven't read the article this still may be useful. For a true hands-on tutorial, I recommend you first do the tutorial at: https://www.michaeldebellis.com/post/new-protege-pizza-tutorial However, there are additional examples in this document that may be useful to supplement the Pizza tutorial. If you have done the Pizza tutorial, you can probably skip section 2 as you should be familiar with how to configure the Protégé environment from the Pizza tutorial.

2. Installing the Protégé Ontology Editor and Plugins

In this document, any instructions that involve interacting with the UI will be highlighted in yellow. Note that not every UI action is required so don't just automatically execute every action highlighted.

To begin install Protégé. Go to the home page for Protégé: https://protege.stanford.edu/ and select the large red Download Now button. That should take you to the appropriate page for whatever OS you are using (Mac OS, Windows, or Linux). All our examples were done in the version for Windows. However, Protégé is not specific to the OS and the examples and instructions should work for all operating systems.

After you have loaded Protégé start the editor. You should get an icon on your desktop that you can click on just as with any other installed application. However, there can occasionally be a bug with Windows where double clicking the icon doesn't work. If that is the case open the folder where Protégé is saved (it should be called Protégé-5.5.0-win) and double-click on the run.bat file.

2.1 Loading Plugins

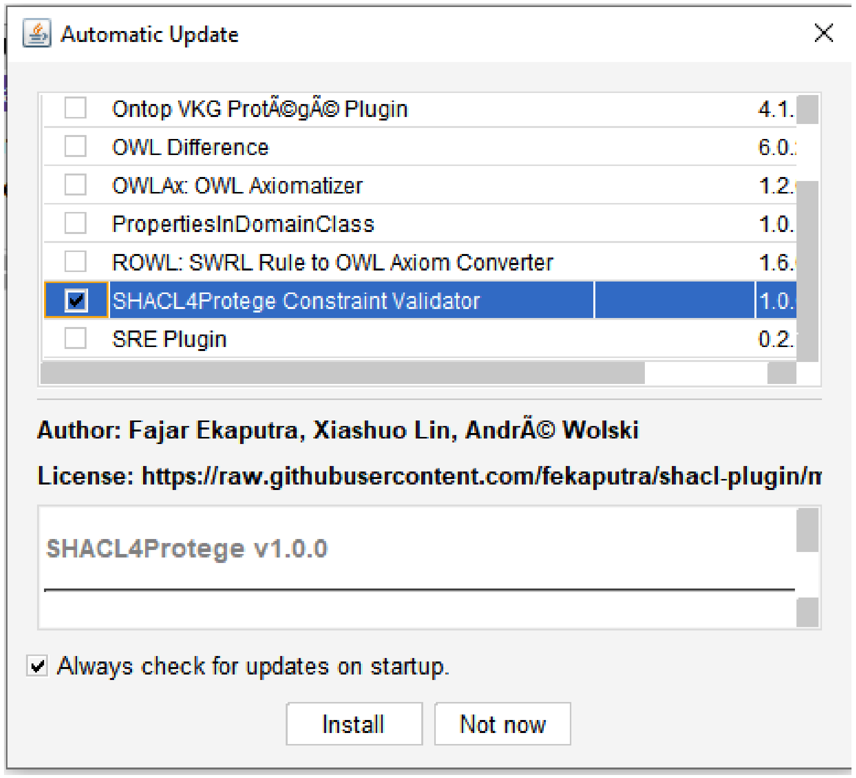
The Protégé community is very active. For those who plan to utilize Protégé we encourage you to sign up for the user support for Protégé mailing list. Go to: https://protege.stanford.edu/support.php and click on Subscribe for the Protégé User Support mailing list.

In addition to offering support to new users via the list many developers have created very useful tools that integrate seamlessly with Protégé called Plugins. When you first start Protégé you should be presented with a pop-up window with all the plugins that are available (see figure 1). You should see this

[1] mdebellissf@gmail.com michaeldebellis.com

window the first time you open Protégé every day. Also, you can see this window by using the menu option: File>Check for plugins. You will need the following plugins to utilize these examples:

- The Pellet Reasoner
- The SWRLTab plugin
- The Snap SPARQL plugin



☐	Ontop VKG ProtÃ©gÃ© Plugin	4.1.
☐	OWL Difference	6.0.
☐	OWLAx: OWL Axiomatizer	1.2.
☐	PropertiesInDomainClass	1.0.
☐	ROWL: SWRL Rule to OWL Axiom Converter	1.6.
☑	SHACL4Protege Constraint Validator	1.0.
☐	SRE Plugin	0.2.

Author: Fajar Ekaputra, Xiashuo Lin, AndrÃ© Wolski

License: https://raw.githubusercontent.com/fekaputra/shacl-plugin/m

SHACL4Protege v1.0.0

☑ Always check for updates on startup.

Fig. 1 The Check for Plugins Window with the SHACL4Progege Plugin Selected

Click the check box next to each of these plugins on the plugin window and then select the Install button at the bottom of that window. Be patient, this may take a minute or two as you are loading several different plugins. You should shortly get a message that says, "Plugins will be available the next time you start Protégé". Quit Protégé and restart it so that the plugins are all available.

2.2 Configuring the Protégé User Interface

The Protégé UI is highly customizable. It consists of a set of tabs and views. A tab is a preconfigured set of views (also called panes in some UI systems) that are relevant to certain kinds of ontology editing. E.g., the classes tab for editing or creating new classes. There are several available tabs and most of them are not displayed by default.

To add (or remove) a tab you use the Window>Tabs option. If you select that option, you will see that several tabs have a check next to them. Those are all the tabs that are currently visible in your version of Protégé. Now that you have loaded the plugins you will want to add some new tabs. Use Window>Tabs and select the following tabs:

- SWRLTab. This will allow you to view the two example SWRL rules in the section on SWRL.
- DL Query. This will allow you to do Description Logic (DL) queries. It will also serve as a place where you can add the Snap SPARQL view described below.
- Individuals by Class. This will allow you to view various individuals. You will also want to customize it to more easily see individuals which we'll describe below.
- Classes. This tab should be available by default but if it isn't select it. This allows you to view the classes and their necessary and sufficient conditions defined in DL.
- Object Properties. This tab may already be available but if not select it. This allows you to view the object property hierarchy and the various options to define properties as functional, inverses, etc.

Now that you have all the tabs that you need, you will need to add some views to some of the tabs. Some of the plugins have their own tabs but others just have views which can be added to existing tabs. To start select the DL Query tab. This is a good place to add the Snap SPARQL view (although you can put it on any tab).

Select Window>Views>Query views>Snap SPARQL Query

This will give you a blue outline for the new view that you can add to the existing DL Query Tab. As you move the outline around it will change depending on where you put it within the tab. We recommend you position it so it will be a sub-tab to the DL Query pane. I.e., in the DL Query tab there is just one view when you start that is used for writing and viewing DL Queries which is called "DL Query". When you have added the Snap SPARQL query view there will be two tabs in this window. Position the blue outline so that it matches the space of the DL Query pane. See figure 2.

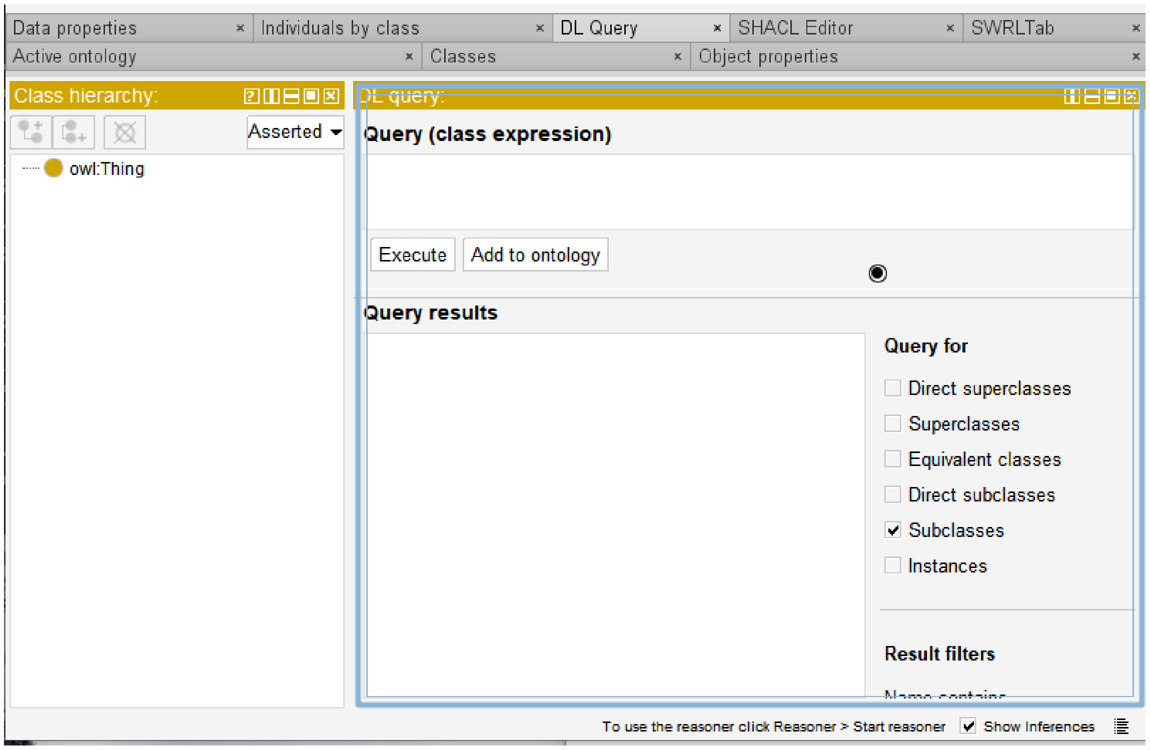


Fig. 2. Adding the Snap SPARQL View to the DL Query Tab

When you have the outline looking like figure 2 click the mouse. Your DL Query tab should now look like figure 3. It may take you a couple of tries to get it right. It can be a bit confusing the first time. If you add the view in the wrong place, you can just delete it and add it again. Any view in a Protégé tab can be deleted by clicking the "X" in the far right corner of the view. Also, if you ever accidentally delete a view, you can easily get it back by doing:

Window>Reset selected tab to default state

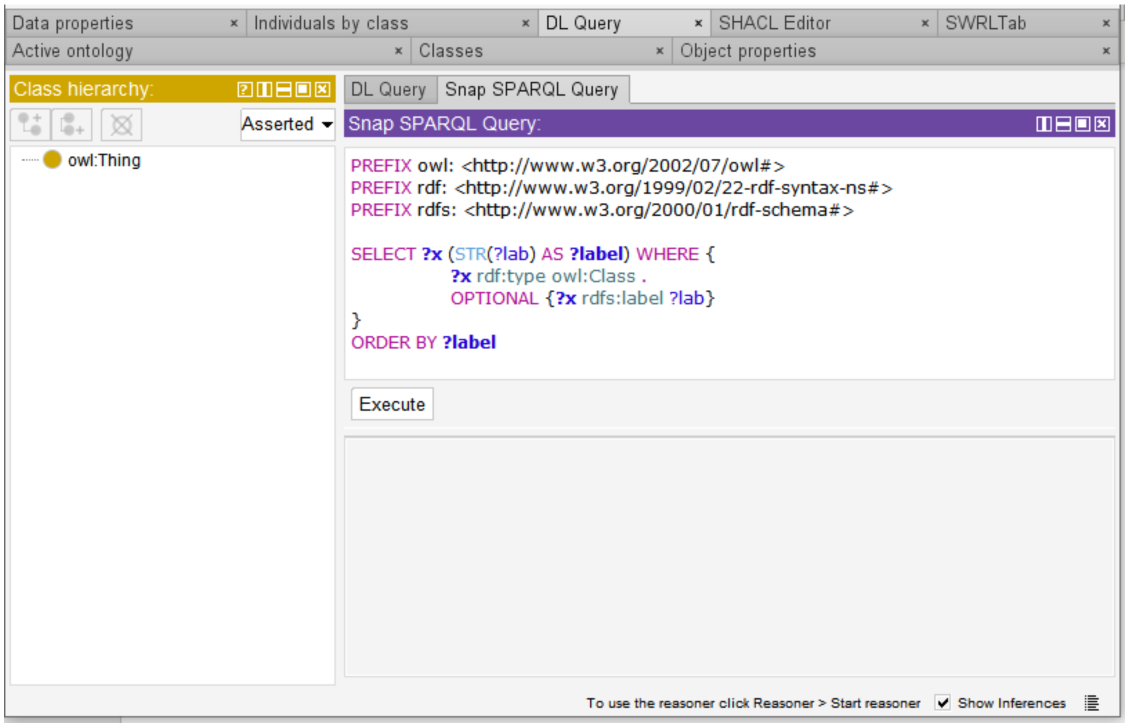
Don't do this for the DL Query tab if you have it the way you want it though because then you will have to add the Snap SPARQL view back again.

Another customization to make is to the Individuals by Class tab. Select this tab. There should be 2 sub-tabs called Annotations and Usage in the upper right quadrant of this tab next to the Classes view. Use:

Window>Views>Individual Views>Individuals by Type (Inferred)

This will give you another view to add. Do as you did with the Snap SPARQL view and put it as a sub-tab (the blue rectangle should take up the same space as the Annotations and Usage views). This will show you all the individuals associated with the lowest class (or classes) in the hierarchy. I.e., For the People_Example ontology you shouldn't see the class owl:Thing show up in this view even though *every* individual is an instance of owl:Thing because that would be redundant. Each individual is an instance of a subclass of owl:Thing so the individuals are only shown for the lowest class in the hierarchy, not their super-classes. Another useful view to add is

Window>Views>Individual Views>Individuals



```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?x (STR(?lab) AS ?label) WHERE {
            ?x rdf:type owl:Class .
            OPTIONAL {?x rdfs:label ?lab}
}
ORDER BY ?label
```

Fig. 3. The DL Query tab with the Snap SPARQL View Added

The first view shows the individuals associated with the class or classes that they are instances of (both defined and inferred by the reasoner). The second view just shows all the individuals in alphabetic order. For larger ontologies this isn't all that useful which is why we recommend the Individuals by Type (Inferred) view but for small ontologies such as the example in this chapter the alphabetic view can be useful. If you add both of these views your Individuals by Class tab should now look similar to figure 4.[2]

Note that John_Doe is selected in the Individuals by type (inferred) view and hence the property values for him are shown in the Property Assertions view and the classes that he is an instance of are shown in the Description view to the left of the Property Assertions view. This is a good way to navigate the various individuals and see the values they have both asserted by the user (in black) and inferred by the reasoner (highlighted in yellow).

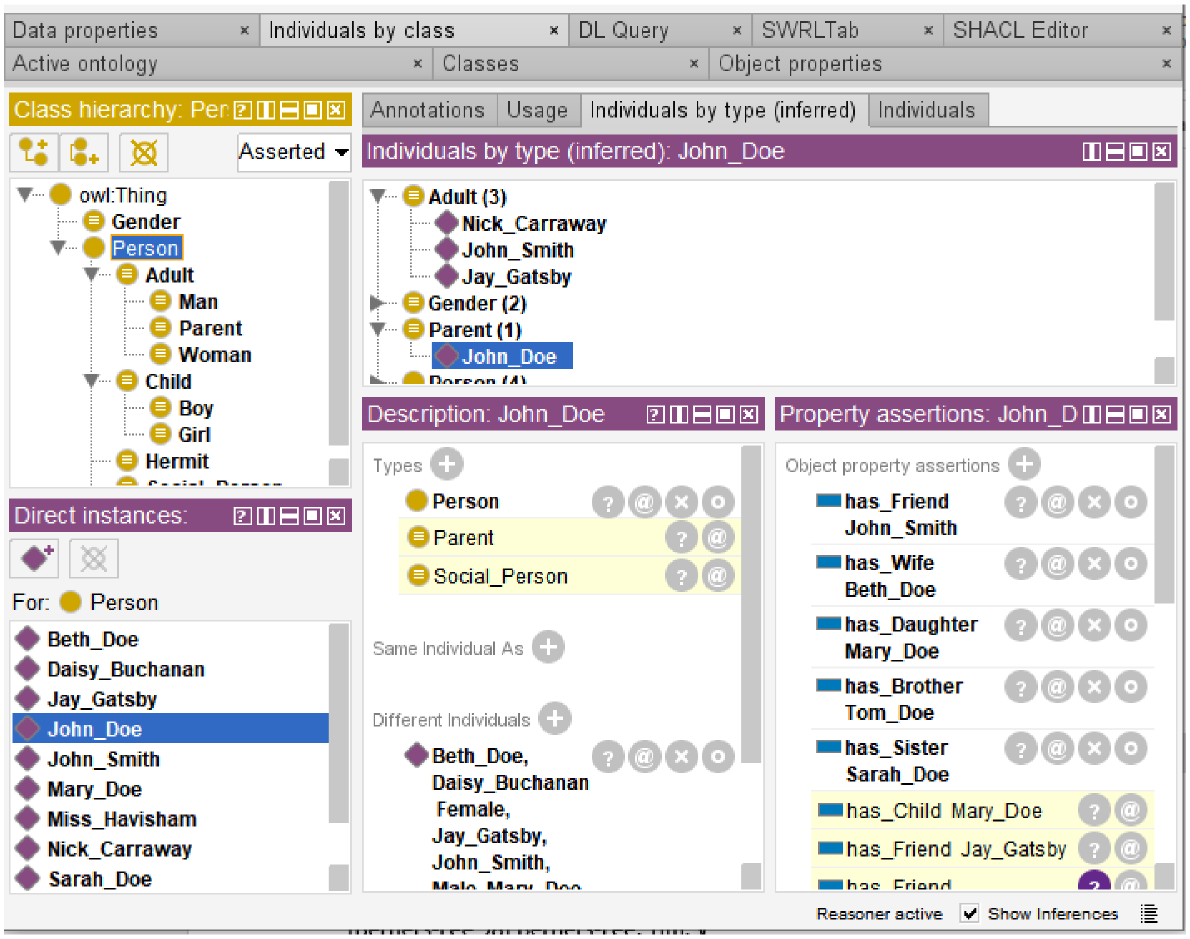


Fig. 4. The Configured Individuals by Class Tab

2.3 Loading Ontologies and Starting the Reasoner

You are almost there. Now load the People_Example ontology that can be downloaded at: https://tinyurl.com/PillarsExamples Download the sample ontology to your hard drive and then use File>Open.

[2] It won't look identical yet because you haven't loaded the ontology or run the reasoner.

The reason you loaded the Pellet reasoner plugin is that this example uses SWRL rules and Pellet is the best reasoner in Protégé to use with SWRL. When you click on the Reasoner menu option there should be several reasoners listed at the top. Pellet should now be included in that list. Select Reasoner>Pellet.

Next you want to configure the reasoner. By default, several of the inferences that the reasoner is capable of are not enabled because for large ontologies some of these inferences can take some time. However, for this simple ontology and for many ontologies that is not an issue and in order to understand the full power of the OWL reasoner you will want to enable all possible inferences. To do so select: Reasoner>Configure. This should bring up a window with two tabs. The default tab: "Displayed Inferences" is the one you want. In that tab select *every* box. You will get warnings that a couple of the selections may take some time but again that is not an issue except for large complex ontologies. After you have checked every box select OK.

Then select the Reasoner menu option again. There should be a black dot to the left of Pellet indicating it is the selected reasoner. The first menu option should be Reasoner>Start reasoner. Use that and start the reasoner. After a second or so the message in the very far right lower corner should say "Reasoner Active" followed by a checked box and "Show Inferences". If you navigate to the Individuals by class tab and select John_Doe, your UI should now look like figure 4.

Whenever you load an ontology or after you make any changes the message will change to "Reasoner state out of sync with active ontology". That means you need to run the reasoner again. To do so select Reasoner>Synchronize reasoner. That should get you back to the message that just says "Reasoner Active" which means the ontology is in synch with the reasoner.

3. The Web Ontology Language (OWL)

OWL provides the capability for a powerful semantic language that implements many of the concepts from set theory and logic that in the past were confined to research systems that could not scale to the enterprise level. An OWL model is called an ontology. An OWL ontology consists of a set of logical axioms. These axioms all map to triples in an RDF graph. However, it is most intuitive to think of them as logical axioms.

3.1 OWL Classes

OWL classes are built on top of and add additional semantics to RDFS classes. Whereas classes in most other languages only have heuristic definitions OWL classes have a rigorous formal definition. An OWL class is a set. A superclass is a superset, i.e., a set that is more general and contains more elements than its subset. Individuals in OWL are elements of sets. When an individual is an instance of a class then that individual is an element of the set represented by that class. The properties that describe sets in set theory also apply to OWL classes and can be asserted by various axioms.

There are 3 types of OWL classes:

1. Primitive classes. These are classes that have axioms that provide *necessary* but not sufficient conditions for an individual to be an instance of the class. The reasoner can infer information about an instance based on the fact that it is declared as an instance of a primitive class, but it cannot infer that an individual is an instance of a primitive class.
2. Defined classes. These are classes that have both *necessary* and *sufficient* conditions. The reasoner can both infer new information about an individual defined to be an instance of a

defined class and it can also infer that an individual is an instance of a defined class even if the user has not explicitly asserted that it is. We will see examples of this in our simple example ontology below.

3. Anonymous classes. Anonymous classes are created by the reasoner based on various other axioms. E.g., if the definition for the domain or range of a property is that it must be an instance of Dog or an instance of Cat the reasoner will create an anonymous class that is the union of the classes Dog and Cat.

3.2 OWL Properties

Relations between individuals are described by OWL properties. There are various axioms from set theory that can be used to enhance the semantics of OWL properties:

- Functional properties. Functional properties must have at most one value for each individual. E.g., the has_Age property would be functional. A person can have only one value for their age.
- Symmetric properties. Symmetric properties point in both directions. E.g., has_Sibling or has_Spouse are symmetric properties. If X has_Sibling Y, then Y has_Sibling X. See figure 5.
- Inverse properties. Inverse properties go in opposite directions. E.g., has_Parent and has_Child are inverse properties. If X has_Parent Y, then Y has_Child X. See figure 6.

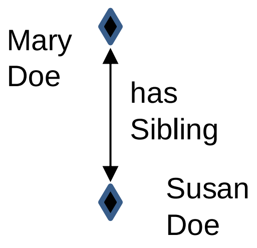


Fig. 5 Symmetric properties

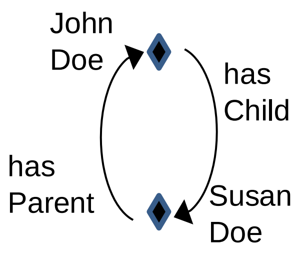


Fig. 6 Inverse properties

- Transitive properties. Transitive properties propagate relations from one object to another. The > or < relations are example of transitive properties. If X > Y and Y > Z, then X > Z. See figure 7.
- Reflexive properties. Reflexive properties are properties that always apply from an individual back to that same individual. See figure 8. Equality is an example of a reflexive property. Any individual is always equal to itself. Owl provides a property called owl:sameAs that is reflexive. This is a way that OWL is different from traditional OOP. Entities are not assumed to be distinct from each other. Reflexive properties always have owl:Thing as their domain and range and should be used with care.
- Super and sub properties. A super property is more general than its sub-property. E.g., has_Parent is a super-property to has_Father. If X has_Father Y, then X has_Parent Y. However, the inverse is not necessarily true. Just because X has_Parent Y does not require that X has_Father Y (Y could be the mother of X).

All properties can have super and sub properties. Only object and data properties can be functional. The rest of the characteristics can only apply to object properties.

.

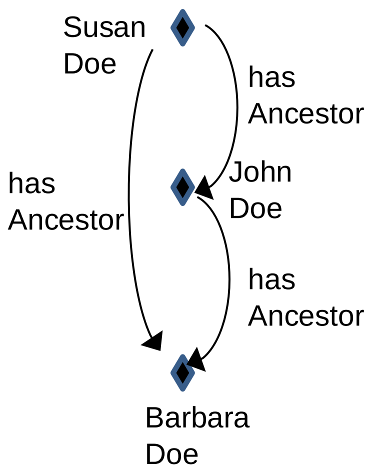


Fig. 7 Transitive properties

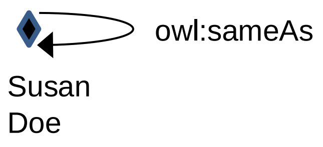


Fig. 8 Reflexive property

Figure 9 shows the object properties for the example ontology displayed in the Protégé ontology editor. Note that has_Father is a sub-property of has_Parent which is a sub-property of has_Social_Relation_With.

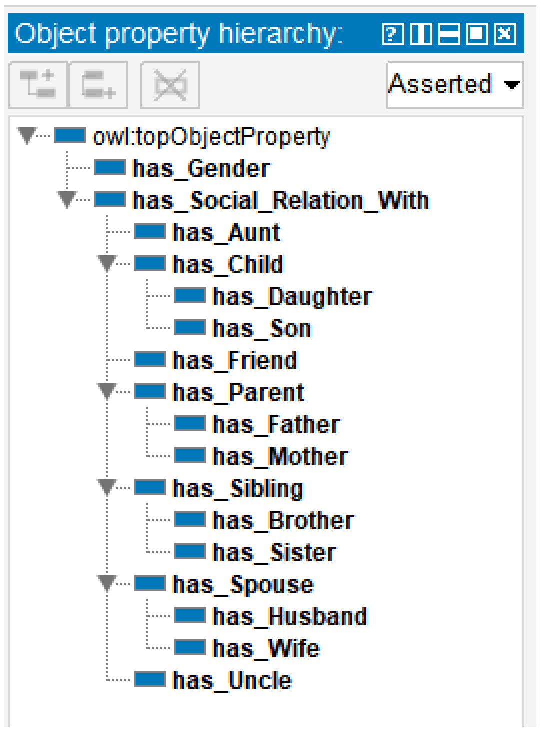
Figure 10 shows the description for the individual John_Doe. In Protégé the information displayed in a normal black font is information that has been asserted by the user. Information that is highlighted in yellow is information that has been inferred by the reasoner. For John_Doe, there are 5 axioms asserted about him:

- has_Friend John_Smith
- has_Wife Beth_Doe
- has_Daughter Mary_Doe
- has_Brother Tom_Doe
- has_Sister Sarah_Doe

All the additional axioms have been inferred by the reasoner based on the definitions of the various properties. For example:

- Because has_Daughter is a sub-property of has_Child and John_Doe has_Daughter Mary_Doe he also has_Child Mary_Doe.
- Because has_Wife is a sub-property of has_Spouse and John_Doe has_Wife Beth_Doe, he also has_Spouse Beth_Doe.
- Because has_Spouse and has_Child are sub-properties of has_Social_Relationship, John_Doe also has_Social_Relationship with his wife and child.

- The reasoner has inferred that he also has_Friend Jay_Gatsby. This is because on Jay_Gatsby there is a user assertion that he has_Friend John_Doe and has_Friend is a symmetric property.

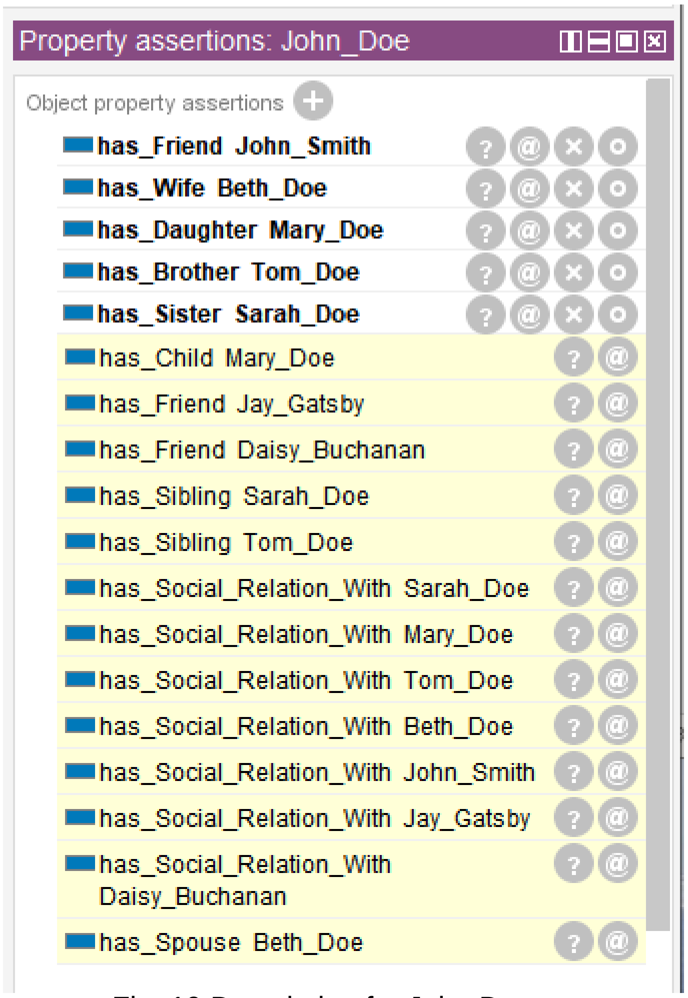


Object property hierarchy:

Asserted

- owl:topObjectProperty
  - has_Gender
  - has_Social_Relation_With
    - has_Aunt
    - has_Child
      - has_Daughter
      - has_Son
    - has_Friend
    - has_Parent
      - has_Father
      - has_Mother
    - has_Sibling
      - has_Brother
      - has_Sister
    - has_Spouse
      - has_Husband
      - has_Wife
    - has_Uncle

Fig. 9 Object Properties in Protégé



Property assertions: John_Doe

Object property assertions

has_Friend John_Smith
has_Wife Beth_Doe
has_Daughter Mary_Doe
has_Brother Tom_Doe
has_Sister Sarah_Doe
has_Child Mary_Doe
has_Friend Jay_Gatsby
has_Friend Daisy_Buchanan
has_Sibling Sarah_Doe
has_Sibling Tom_Doe
has_Social_Relation_With Sarah_Doe
has_Social_Relation_With Mary_Doe
has_Social_Relation_With Tom_Doe
has_Social_Relation_With Beth_Doe
has_Social_Relation_With John_Smith
has_Social_Relation_With Jay_Gatsby
has_Social_Relation_With Daisy_Buchanan
has_Spouse Beth_Doe

Fig. 10 Description for John Doe

3.3 OWL Axioms

One of the most powerful features of OWL is the ability to provide formal definitions of classes using the Description Logic (DL) language. These axioms are typically asserted on property values for the class. There are three kinds of axioms that can be asserted about OWL classes:

1. Quantifier restrictions. These describe that a property must have some or all values that are of a particular class or datatype.
2. Cardinality restrictions. These describe the number of individuals that must be related to a class by a specific property.
3. Value restrictions. These describe specific values that a property must have.

Recall that First Order Logic (FOL) has two types of quantifiers: existential and universal quantification. Existential quantification means that there exists at least one value that matches a variable in a formula and makes it true. Universal quantification means that all possible values that match a variable in a formula will make it true. OWL implements existential quantification via the Description Logic (DL) operator *some* and universal quantification via the DL operator *only*.

For example, in the People_Example ontology we define the class Adult as:

```
Person and (has_Age some xsd:integer[> 18])
```

OWL distinguishes between *primitive classes* which have necessary conditions and *defined classes* which have necessary *and* sufficient conditions. In Protégé a defined class has axioms in the `Equivalent To` part of its description and is shown in the UI with three horizontal white stripes in the circle next the name of the class. See figure 11. Because Adult is a defined class, any individual who satisfies its axioms (i.e., is an instance of the Person class and has a value for the has_Age data property that is an integer greater than or equal to 18) will be recognized by the reasoner as instances of that class. In figure 11 the reasoner has inferred that several individuals in the ontology are instances of the Adult class based on their definitions even though the user did not define them as adults.

A defined class in the People_Example ontology that utilizes cardinality restrictions is the Social_Person class. Cardinality restrictions are defined in OWL with the min and max DL keywords. So Social_Person is defined as:

```
Person and has_Social_Relation_With min 5 Person
```

I.e., a Social_Person is an instance of the Person class that has at least 5 values for its has_Social_Relation_With property. The reasoner has inferred that Jay_Gatsby and John_Doe are instances of Social_Person. Note that they have no values explicitly defined for their has_Social_Relation_With properties but these values are inferred as a result of values for sub-properties of this property such as has_Friend and has_Daughter. This illustrates one of the benefits of sub and super-properties. For some types of reasoning, we want to use specific information such as reasoning about daughters and fathers. For other types of reasoning, we wish to work at a higher level of abstraction such as any social relationship.

Another important axiom regarding classes is the Disjoint axiom. Two classes are disjoint if their intersection is the empty set (owl:Nothing). By default, classes are not assumed to be disjoint. However, the reasoner has inferred that the classes `Social_Person` and `Hermit` must be disjoint because the definition for a `Hermit` is

```
Person and has_Social_Relation_With max 0 Person
```

Since any Person who has_Social_Relation_With at least 5 people has a social relation with more than 0 no individual can be both a `Social_Person` and a `Hermit` and the reasoner has inferred that they are disjoint classes.

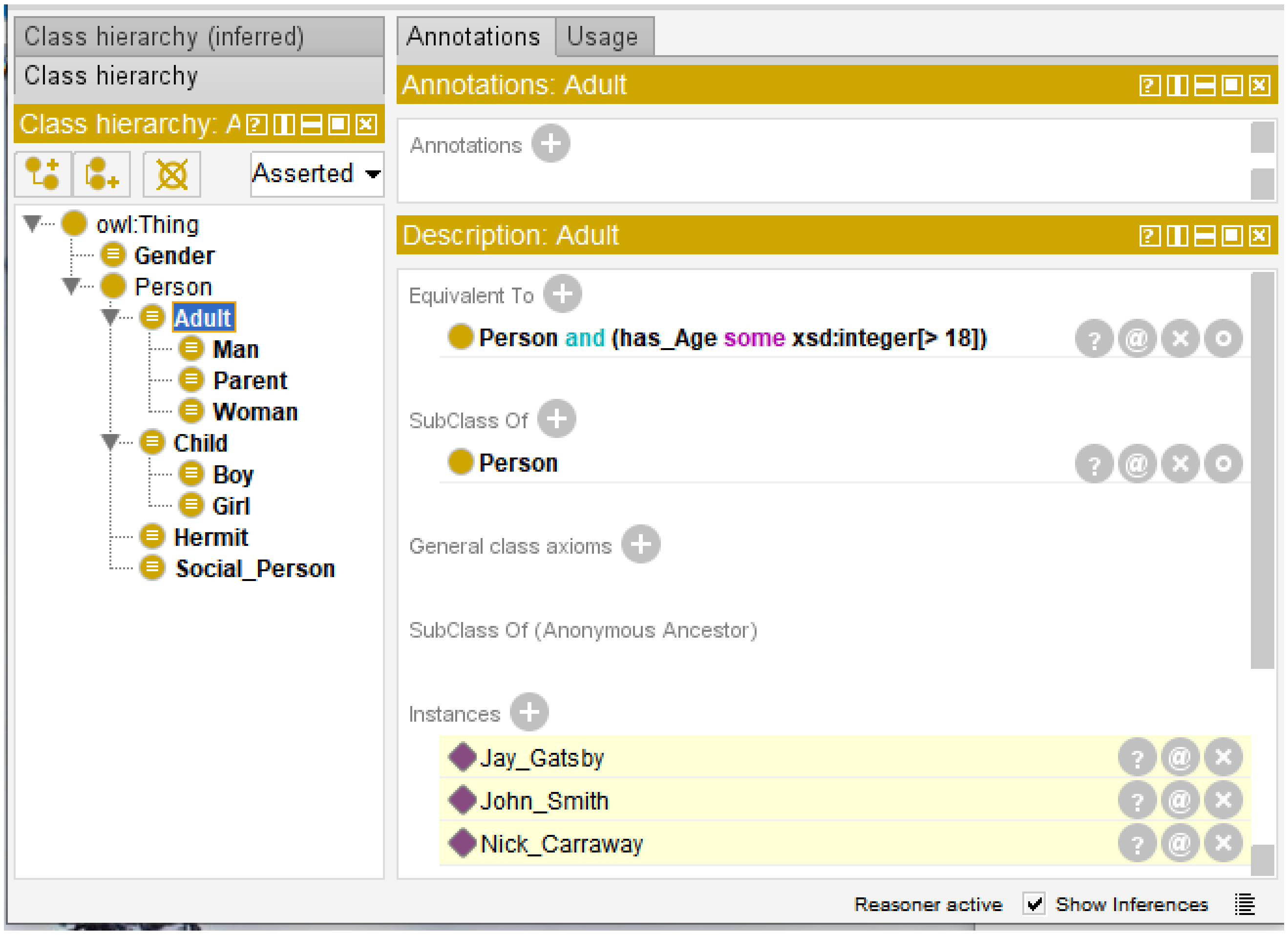
Examples of value restrictions in the People_Example ontology are the defined classes `Man` and `Woman`. `Woman` is defined as:

```
Adult and (has_Gender value Female)
```

The reasoner has inferred that several of the individuals in the ontology are women. It has also inferred that the class `Woman` is disjoint with the class `Man` which has the definition:

```
Adult and (has_Gender value Male)
```

The class `Gender` is an example of a different kind of defined class. All the examples so far have had definitions that are defined via logical axioms that describe conditions on properties. Another way to define a class is to simply list all the possible instances of the class. This is useful for certain types of classes where we know ahead of time that there are only a finite number of possible instances. This type of defined class is known as an enumerated class. Gender is such a class. It has only two possible instances: `Male` and `Female`. In Protégé an enumerated class is defined by a DL formula that lists each



```
owl:Thing
    Gender
    Person
        Adult
            Man
            Parent
            Woman
        Child
            Boy
            Girl
        Hermit
        Social_Person
```

Equivalent To: Person and (has_Age some xsd:integer[> 18])

SubClass Of: Person

Instances: Jay_Gatsby, John_Smith, Nick_Carraway

Fig. 11 Defined Class Adult and Inferred Instances

instance of the class in curly brackets. In the People_Example ontology the definition for Gender is `{Male, Female}`.

3.4 The Open World Assumption (OWA)

Recall that unlike most databases, OWL utilizes the Open World Assumption (OWA) rather than the Closed World Assumption. If some information is not in an OWL ontology the reasoner doesn't infer that the information doesn't exist.

As an example, one of the defined classes is `Hermit` with the definition:

```
Person and has_Social_Relation_With max 0 Person
```

In the ontology there is an individual `Miss_Havisham` (a hermit from the novel Great Expectations). `Miss_Havisham` has no social relations. However, note that while the reasoner has found instances of the defined class `Social_Person` it has found no instances of the class `Hermit` including `Miss_Havisham` who we would think certainly qualifies. This is an example of the OWA. Even though `Miss_Havisham` has no social relations defined in the ontology the reasoner can't infer that there are not additional values that exist but that have not yet been discovered. Although the reasoner would signal an error if someone was defined to be an instance of `Hermit` and had more than 0 social relations it can't recognize instances with no values as hermits. Due to the OWA there are some axioms such as those regarding maximum number of values that are difficult to utilize to recognize instances of defined classes. There are ways around this such as utilizing SPARQL or SHACL.

3.5 Saving Your Work

When you first save an ontology in Protégé you will be given several serialization options to choose from. The two most widely used are RDF/XML and Turtle. RDF/XML is the most portable and least likely to cause problems when going from one tool to another. However, it is rather verbose. Turtle is also very portable and less verbose. If you wish to save changes to this ontology, we recommend using RDF/XML for ultimate portability. This ontology is so small that the difference in file size is trivial. However, saving in Turtle should be fine as well.

4. The Semantic Web Rule Language (SWRL)

OWL is a very powerful language, however, there are certain types of inferences that are difficult or impossible to define with it but are still useful for definition of the semantics of data. In order to extend the semantics of OWL the Semantic Web Rule Language (SWRL) was created.

The easiest way to explain SWRL is with a simple example. One classic example that can't be defined with normal OWL is an aunt or uncle relation. In the People_Example ontology there are two SWRL rules to define these. The SWRL rule that defines the has_Aunt relation is:

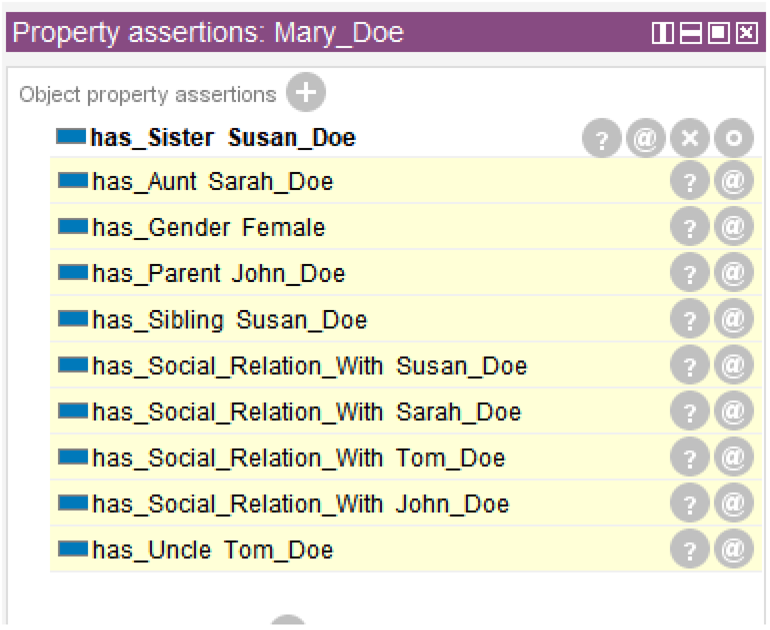
```
has_Child(?p, ?c) ^ has_Sister(?p, ?s) -> has_Aunt(?c, ?s)
```

The "^" symbol is used to specify the AND logical relation. All expressions in a SWRL rule are connected with just this operator meaning that in order for the rule to succeed every expression must succeed for some binding of the variables (the symbols with a "?" before them). To understand SWRL rules it is useful to return to First Order Logic and the concept of quantification. The left hand side of a SWRL rule (all the expressions on the left of the "->") are implicitly universally quantified and the right side is implicitly existentially quantified.

I.e., every variable on the left hand side that begins with a "?" is a wildcard and SWRL will match every possible permutation that satisfies the property value on the left hand side. Similarly, for every "?" value on the right hand side, every time the left hand side is satisfied, SWRL will match an existing appropriate binding if one exists or will create a new property value if the binding does not currently exist.

In this example the reasoner will find all individuals that satisfy the has_Child property. E.g., it will bind ?p to John_Doe and ?c to Mary_Doe. It will then see if any of the bindings for ?p also satisfy the has_Sister property. In this example since ?p is already bound to John_Doe it will check all the individuals that are sisters of John_Doe and bind those to ?s. Thus, ?s will bind to Sarah_Doe. Since the reasoner has found bindings that satisfy all the expressions on the left hand side it will see if there exists a binding that satisfies the right hand side has_Aunt. Since there isn't it will assert a new value for has_Aunt, i.e., that Mary_Doe has_Aunt Sarah_Doe. Figure 12 shows the property assertions for Mary_Doe and we can see that the reasoner has indeed inferred that she has_Aunt Sarah_Doe.

This is also a good time to highlight another advantage of the OWL reasoner: the ability to generate explanations. We can see all the inferences of the reasoner highlighted in yellow, but we can get more information than that as a benefit of the reasoner. Note that every assertion has a "?" next to it. We can click on this for any assertion and the reasoner will generate an explanation for the inference.


Property assertions: Mary_Doe

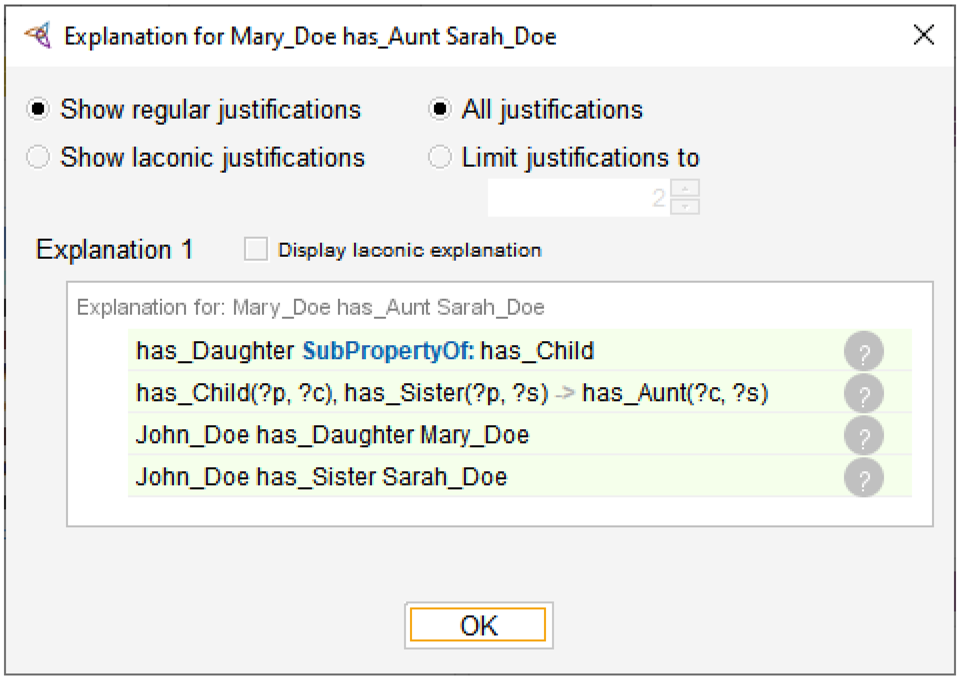
Object property assertions

has_Sister Susan_Doe
has_Aunt Sarah_Doe
has_Gender Female
has_Parent John_Doe
has_Sibling Susan_Doe
has_Social_Relation_With Susan_Doe
has_Social_Relation_With Sarah_Doe
has_Social_Relation_With Tom_Doe
has_Social_Relation_With John_Doe
has_Uncle Tom_Doe

Fig. 12 Property Values for Mary_Doe

Figure 13 shows the window that results when we click on the "?" next to the has_Aunt assertion for Mary_Doe. The system displays the SWRL rule that was utilized, and the assertions: that John_Doe has_Daughter Mary_Doe and has_Sister Sarah_Doe and the fact that has_Sister is a sub-property of has_Child (since the rule is defined in terms of has_Child).

SWRL also has a library of what are called built-in functions. [W3C 2004] These look similar to property expressions except they can often have more than 2 values. These can be used for standard types of mathematical, string, date-time, and other computations and comparisons. This makes SWRL a very powerful rule language similar to forward chaining rules in expert system shells.


Explanation for Mary_Doe has_Aunt Sarah_Doe

[✓ Show regular justifications] Show laconic justifications

[✓ All justifications] Limit justifications to 2

Explanation 1 Display laconic explanation

Explanation for: Mary_Doe has_Aunt Sarah_Doe

has_Daughter SubPropertyOf: has_Child

has_Child(?p, ?c), has_Sister(?p, ?s) -> has_Aunt(?c, ?s)

John_Doe has_Daughter Mary_Doe

John_Doe has_Sister Sarah_Doe

Fig. 13. Explanation for Inference that Mary_Doe has_Aunt Sarah_Doe

5. SPARQL

SPARQL (pronounced “sparkle”) is a recursive acronym. It stands for SPARQL Protocol and RDF Query Language. SPARQL is to RDF as SQL is to relational databases. SPARQL works at the level of RDF, however since OWL is mapped very directly to RDF, SPARQL can also be used with OWL ontologies.

The examples below will be shown in the Snap SPARQL plugin for Protégé. This is not the default Protégé SPARQL tab but a more complete implementation of SPARQL.

The power of SPARQL is its ability to match any or all parts of an RDF triple. There are 3 parts of any SPARQL query:

1. Prefix mappings. These are identical to the mappings discussed in the section on RDF (although the syntax is slightly different). Just as with RDF they allow the developer to access IRIs by typing a prefix followed by a colon rather than the entire IRI.
2. The action clause. This can be one of several keywords: SELECT, CONSTRUCT, INSERT, DELETE, etc. This tells the SPARQL processor what to do with the data that is matched in the WHERE clause. The most straight forward is the SELECT clause which lists parts of the WHERE clause and just displays them.
3. The WHERE clause. This describes one or more triples to match as well as other constructs to further constrain (or expand) the query such as FILTER and OPTIONAL. Each triple can contain 0 to 3 wildcards. A triple with all wildcards will retrieve every resource in the graph.

A simple SPARQL query is:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX : <http://www.semanticweb.org/mdebe/ontologies/example#>
```

```
SELECT  ?p
WHERE {?p rdf:type :Person}
```

The prefixes for the query map the prefix rdf to the IRI for the RDF vocabulary and the empty string to the IRI for the example ontology we have been using. The SELECT clause tells SPARQL to return all the resources that bind to the parameter ?p. The WHERE clause matches all the triples in our ontology that have the type `Person`. I.e., all the instances of the `Person` class. This highlights the way that OWL is built on top of RDF and RDFS. The structure of entities in an OWL ontology is defined as nodes and links in an RDF graph.

To run this query, go to the Snap SPARQL view. Make sure you have run the Pellet reasoner. Delete the example query and prefaces that are already there and enter the prefixes and the rest of the query above. For the following queries we omit the prefixes because they don't change so to run later queries leave the prefixes in place and just delete the previous query and enter the next query. Running this first query will list all the instances of Person in the ontology.

Another simple query would be:

```
SELECT  ?p ?s
WHERE {?p :has_Spouse ?s.}
```

This will list all the individuals in the ontology that have a spouse and their spouses. Since there is only one married couple in our small sample ontology you will just see John_Doe and Beth_Doe. Note that you will see them twice. That is because the query had two different matches. It matched ?p and ?s to John_Doe and Beth_Doe respectively and then it matched ?p and ?s to Beth_Doe and John_Doe. If you view the individuals in the Individuals by Class tab you will see that only John_Doe has a value defined by the user for has_Spouse. However, since has_Spouse is symmetric, the reasoner inferred that Beth_Doe has_Spouse John_Doe and that is reflected in our query results.

Often you want to see every parameter that matched your query. Rather than listing each one in the SELECT clause you can simply use a "*". If we had used a "*" rather than listing the variables in these examples, the results would have been the same.

Suppose you want to see all the people who have any social relation (either by family or friends). You can do that with the following query:

```
SELECT  *
WHERE {?p :has_Social_Relation_With ?r.}
```

Recall that counting in SWRL is difficult due to monotonic reasoning. In SPARQL this is easy. There are built-in expressions such as COUNT, SUM, AVG, etc. that can perform common mathematical computations on the number of triples that match a pattern. For example, if we wanted to count the number of social relations for each person we could do:

```
SELECT  ?p (COUNT(?r) AS ?rcount)
WHERE {?p :has_Social_Relation_With ?r.}
```

```
GROUP BY ?p
```

The GROUP BY expression is used on aggregates in SPARQL. An aggregate can be formed explicitly by adding values to a list or implicitly as here by using expressions such as COUNT. The results of this query would be a list of each individual who has at least one social relation and the number of social relations they have. The AS expression is always used with aggregate expressions such as COUNT. It binds the result to a parameter, in this case ?rcount.

Our next example will illustrate two new expressions: OPTIONAL and FILTER. These are often useful together. OPTIONAL specifies a triple that can be bound but doesn't have to be in order for the query to succeed. Without using OPTIONAL every triple pattern in the WHERE clause of a query must bind to some value for the query to succeed. FILTER can be used to filter out certain values from our result. I.e., there may be some triples that matched but that we want filtered out of the result.

For example, recall that with OWL we couldn't detect that Miss_Havisham was a Hermit due to the OWA. Since SPARQL is not constrained by the OWA we can detect this. The following query will list any people who don't have any social relations:

```
SELECT  ?p

WHERE {?p a :Person.

OPTIONAL {?p :has_Social_Relation_With ?r.}

FILTER (!bound(?r)) }
```

There are several new expressions here. First the triple ?p a :Person. One of the most common triples is to look for instances of a certain class or datatype. The predicate for this is rdf:type. However, since this is such a common predicate there is a shortcut for it in SPARQL where we can just use "a". So, this triple is the same as: ?p rdf:type :Person. The OPTIONAL expression allows us to embed one or more triples that are optional. SPARQL will match them if it can but if it can't the query won't fail as would normally be the case. FILTER on the other hand will filter out any triples that match the query but do not satisfy the expression in the FILTER. This is a common pattern to as Bob DuCharme says in his book on SPARQL [DuCharme 2011] "look for data that isn't there".

In this case the FILTER expression is testing if the variable ?r is bound. The ! means not in SPARQL. So, this query will first bind ?p to all instances of the Person class. It will then bind ?r to each individual that is the value of the has_Social_Relation_With object property. Then the FILTER expression will filter out all the people that have at least one social relation (at least one binding for ?r). That will leave us with only lonely Miss_Havisham.

If we execute this query in the Protégé Snap SPARQL tool the only result will be Miss_Havisham. That is useful but what we really want is to change the ontology so that Miss_Havisham is an instance of the Hermit class. We can do this in the Snap SPARQL tool with the CONSTRUCT expression.[3] The following query will accomplish this:

[3] The more common way to do this would be via the INSERT expression, however neither INSERT nor DELETE are currently supported in the SPARQL tools that are integrated with Protégé. To get the full power of SPARQL it is

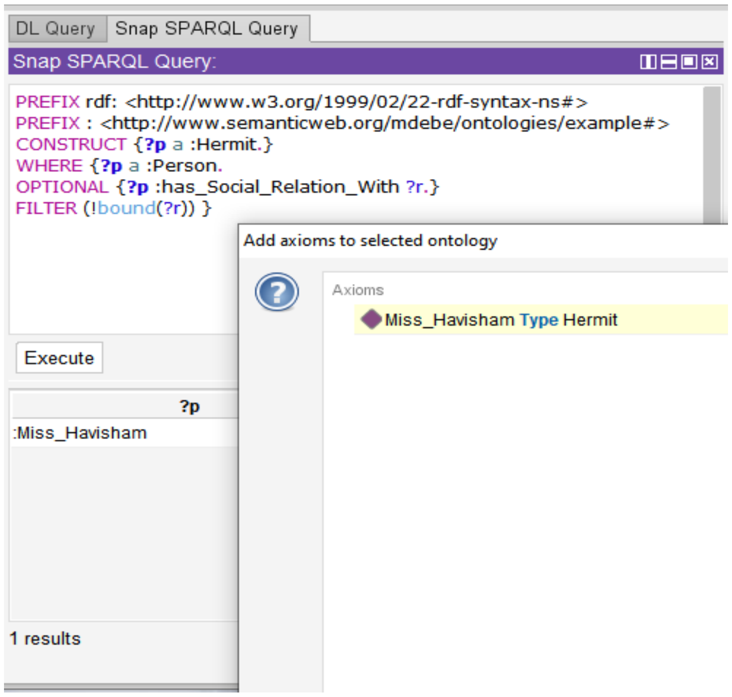
```
CONSTRUCT {?p a :Hermit.}

WHERE {?p a :Person.

OPTIONAL {?p :has_Social_Relation_With ?r.}

FILTER (!bound(?r)) }
```

Rather than just display the result, this will construct a new triple that asserts that anyone who has no social relations is an instance of the `Hermit` class. If executed in the Snap SPARQL tool (see figure 14) the system presents the user with the option to add the new triple and `Miss_Havisham` will then be an instance of the `Hermit` class.



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://www.semanticweb.org/mdebe/ontologies/example#>
CONSTRUCT {?p a :Hermit.}
WHERE {?p a :Person.
OPTIONAL {?p :has_Social_Relation_With ?r.}
FILTER (!bound(?r)) }
```

?p
:Miss_Havisham

1 results

Miss_Havisham Type Hermit

Fig. 14. Making Miss_Havisham an Instance of the Hermit Class

5.1 The SERVICE keyword and Linked Data

One of the powers of SPARQL is that it can bring in data from many diverse sources such as DBpedia and integrate them as if they were all in the same database. Essentially, it allows your computer to have the entire web as one huge, distributed database right at your fingertips. This is a concept called Linked Data.

One way to access these linked data resources is just to include additional IRIs in your prefixes that give you access to open datasets. Another way is to use the SERVICE keyword. The difference is that using a preface assumes that all the data and actions for that IRI are available to all users. For very large knowledge graphs such as DBpedia this wouldn't be practical since it would allow people to corrupt the graph.

often best to load your ontology into some other tool that supports all of SPARQL such as the AllegroGraph tool from Franz Inc. used below in our complex query that searches multiple knowledge graphs.

Thus, large open knowledge graphs are typically available via SPARQL endpoints. An endpoint requires that you use the SERVICE keyword. SERVICE sends the query to the SPARQL endpoint and leaves it to the server at the endpoint to determine how to reply to the query.

The above queries have been very basic to illustrate basic SPARQL concepts. We will end with a more realistic query that utilizes endpoints and that illustrates the way SPARQL can integrate data from different sources. In order to do that we need to use a different tool than Protégé because the SPARQL implementations currently integrated with Protégé are not complete and are designed primarily for educational purposes. We will use the SPARQL implementation available in the AllegroGraph triplestore from Franz Inc. via a public demo server provided by Franz.

We want to answer the question: "What is the average income for the place where Barack Obama was born?" For this query we first go to DBpedia to find his birthplace, and the corresponding Geonames ID. Then we go to Geonames to find all the other Geonames-ids that are populated places within 10 miles of President Obama's birthplace, with this list of Geonames we then go to the 2000 census and find the average income for each place. Here is the SPARQL query:

```
PREFIX geo: <http://franz.com/ns/allegrograph/3.0/geospatial/>

PREFIX geonames: <http://sws.geonames.org/>

PREFIX dbpedia_rsrc: <http://dbpedia.org/resource/>

PREFIX dbpedia_onto: <http://dbpedia.org/ontology/>

PREFIX dbpedia_prop: <http://dbpedia.org/property/>

PREFIX census: <tag:govshare.info,2005:rdf/census/>

PREFIX census_samp: <tag:govshare.info,2005:rdf/census/details/samp/>

SELECT distinct ?censusplace ?income {

  dbpedia_rsrc:Barack_Obama dbpedia_onto:birthPlace ?birthplace .

  ?birthplace dbpedia_prop:hasGeonamesID ?geonamesresource .

  SERVICE <https://localhost:10000/catalogs/demos/repositories/geonames>

            { ?geonamesresource geonames:isAt5 ?location .

              ?otherplace geo:inCircleMiles (geonames:isAt5 ?location 10) .

              ?otherplace geonames:feature_code "PPL" .

              ?geonamesresource geonames:feature_code "PPL" .

          SERVICE <https://localhost:10000/catalogs/demos/repositories/census>

            { ?censusplace dbpedia_prop:hasGeonamesID ?otherplace .

              ?censusplace census:details ?detail .

              ?detail census_samp:population15YearsAndOverWithIncomeIn1999 ?d .

              ?d census_samp:medianIncomeIn1999 ?income .}}}
```

To see this query in action, do the following:

1. Enter this URL in your browser: https://tinyurl.com/LDExample
2. Click on the demos catalog link. There will be several demo repositories. Select the dbpedia repository.
3. Under "Pre-Defined Queries" click on the link: barack-obama-query-using-service. This will take you to the query listed above displayed in the AllegroGraph SPARQL query window.
4. Click on the Execute button beneath the window where the query is displayed. This will execute the query. There may be some overhead to start the server so be patient it may take a minute to warm up the server. You will see the answer displayed below the query. If you click on execute again, now that the server is warmed up it should be about a second.
5. This returns the average income of all the populated places within ten miles of Obama's birthplace. To see the answer to our question you can edit the first line of the query after the prefixes so that rather `SELECT distinct ?censusplace ?income` it is: `SELECT (AVG(?income) AS ?avgincome)`. This will give the average of the four median incomes returned by the initial query.

This shows the true power of SPARQL. We have utilized data from 3 different large data sources: DBpedia, Geonames, and the 2000 Census. All of these were developed independently with no explicit design for integration.

6. Conclusion

This has been a brief overview of some of the capabilities of OWL, SWRL, and SPARQL. In the future I may expand this document or add additional documents to the page on my blog where this example is located: https://tinyurl.com/PillarsExamples For a more complete tutorial that includes SHACL see: https://www.michaeldebellis.com/post/new-protege-pizza-tutorial If you have questions or comments feel free to email me at: mdebellissf@gmail.com.

7. Bibliography

[Aasman 2011] Aasman, J. Cheatham, K. RDF Browser for Data Discovery and Visual Query Building. Workshop on Visual Interfaces to the Social and Semantic Web (VISSW2011) Co-located with ACM IUI 2011. Palo Alto, US. (2011)

[Berners-Lee 2001] Berners-Lee, Tim with James Hendler and Ora Lassila. The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American. May 2001. Volume 284, Issue 5.

[DeBellis 2018] Semantic Web Rule Language (SWRL) Process Modeling Tutorial. https://www.michaeldebellis.com/post/swrl_tutorial

[DeBellis 2021] A Practical Guide to Building OWL Ontologies Using Protégé 5.5 and Plugins Edition 3.0. https://www.michaeldebellis.com/post/new-protege-pizza-tutorial

[DuCharme 2011] DuCharme, Bob (2011). Learning SPARQL. O'Reilly Media - A. Kindle Edition.

[Engelbart 1968] The Mother of All Demos, presented by Douglas Engelbart (1968). https://www.youtube.com/watch?v=yJDv-zdhzMY Accessed August 23, 2021.

[Noy 2019] Noy, Natasha (2019). Industry-Scale Knowledge Graphs: Lessons and Challenges. With Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, Jamie Taylor. Communications of the ACM. Vol. 62. No. 8. August 2019

[W3C 2004] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21 May 2004. https://www.w3.org/Submission/SWRL/

[W3C 2005] SKOS Core Vocabulary Specification. 2 November 2005. https://www.w3.org/TR/swbp-skos-core-spec/

[W3C 2012] OWL 2 Web Ontology Language New Features and Rationale (Second Edition). W3C Recommendation 11 December 2012. https://www.w3.org/TR/owl2-new-features/