Knowledge grows

Developing an Ontology for a Graph Database in Agriculture

Technical Guidelines

By

Marie-Angélique Laporte[1], Siju Idicula[2], Rafael Davoglio Tolotti[2], Dharani Dhar Burra[2], Carlos Quiros[1], Elizabeth Arnaud[1]

The Guidelines were developed with the Varda platform development team and the Alliance Bioversity-CIAT team.

[1]Alliance Bioversity-CIAT

- Marie-Angélique Laporte, Associate Scientist, Ontology and Graph Expert, Digital Solution Team
- Elizabeth Arnaud, scientist, Digital Solutions Team leader, Ontologies CoP Leader

[2]Varda Team

Siju Idicula, Rafael Davoglio Tolotti, Anahita Nafissi, Dharani Dhar Burra

Citation :

Laporte M.A., Idicula S., Davoglio Tolotti R., Dhar Burra D., Quiros C., Arnaud E. - Developing an Ontology for a Graph Database in Agriculture: Technical Guidelines. January 2022. Alliance Bioversity-CIAT, Yara International, Varda.

License CC-BY 4.0

Contents

Contents .... *ii*

Acronyms .... *iv*

Acknowledgements .... *v*

Introduction *1*

Basics about ontology modeling *1*

Before starting: What should be modeled in the ontology (as classes) vs. what can be modelled at the instances level? **1**

Selecting the ontology model for representing measurements **2**

Making decision on axioms – what inference do I need? **3**

Modeling geolocation information **3**

Geolocation modeling **4**

Solution 1: Using a reverse geocoding service 4

Solution 2: using Geonames 4

Solution 3: Using any geodatabase 4

Existing Ontologies: Geonames vocabulary and Geosparql 5

Geolocation modeling Schema 7

Example of a query calling an external service to get values for a given location 7

Using the location to merge datasets 7

Measured vs. predicted values **10**

Soil Depth information **11**

Management practice **13**

Provenance Metadata **14**

Reusing existing standard – the example of the unit ontology **16**

The Unit Ontology (UO) 17

Unit conversion 19

Creating a Unit Ontology based on existing standards 20

Creation of a new unit ontology in protégé, based on existing standards 20

Developing a Soil Ontology in Protégé *22*

Labels were created for all classes: **22**

Checking the Model with Data *32*

Steps taken for loading the ontology and data **33**

Creation of RDF triples **34**

Visualization of the results in the ontology *37*

Adding a 'DC:source' **38**

Creating a new ttl file after running the script. **39**

Sorting out Measurements and Predictions with a reasoner **40**

Importing data into NEO4J after conversion into RDF ***43***

Two Plug-ins to install **44**

Creation of the NOE4J Database **44**

Create the database and configure the graph 44
Run the import: 45
Use of Cypher language with example of queries based on nodes and relationships. 46

Examples of Queries **46**

Query #1 46
Query #2 46
Query #3 47
Query #4 48

Acronyms

AgrO	Agronomy Ontology
API	Application Programming Interface
BFO	Basic Formal Ontology
DC	Dublin Core (Metadata Schema)
OBOE	Extensible Observation Ontology
OM	Observation & Measurement
OWL	Web Ontology Language
ProVO	Provenance Ontology
RDF	Resource Description Framework
SHACL	Shape Constraint Language
SPARQL	SPARQL Protocol and RDF Query Language
ttl	Turtle File (RDF serialization)
UO	Unit Ontology
URI	Uniform Resource Identifier.
W3C	World Wide Web Consortium

Acknowledgements

We thank Mathavan Arugalaimuthu for supporting the collaboration between the Alliance and Varda teams, Anahita Nafissi, Ontology expert of the Varda team, for reviewing the document, Yana Kovalska for facilitating the organization of the consultancy, and Vincent Johnson, Science writer, Alliance Bioversity International-CIAT, for the English editing.

Introduction

These guidelines are aimed at users already familiar with an Ontology[1] structure, and its definitions of classes, properties and axioms. This document guides the reader through the steps of an ontology modeling process to be integrated into a multi-source graph database such as NEO4J[2]. The development of a Soil Ontology is used as an example. It covers the principles of modeling to include soil measurements, geocoordinates and units, so that the ontology can support data transformation and queries.

These guidelines result from collaboration between the Varda platform development team and the CGIAR Ontology Team.

Basics about ontology modeling

This section guides the user through key questions and steps when developing an ontology to capture specific dataset semantics.

Before starting: What should be modeled in the ontology (as classes) vs. what can be modelled at the instances level?

1. What do you want to model?
2. What kind of inferences do you need?

   Note: 'under-modeling' is always better than 'over-modeling' as the later could lead to wrong inferences

   - Locations will have geocoordinates. They can be regions or countries, which could be represented either as points or polygons.
   - Region can have soil, a type of soil, etc.
   - Soil will have properties
3. Who are going to be the main users?
   a. domain experts or not? how many stakeholders?
4. How many data sources do you want to integrate?
5. What already exists in terms of ontologies or controlled vocabularies?

Here we will use the example of soil ontology that will support the description of soil datasets, with the following key elements: **Soil, Soil properties, Geolocation**

[1] An ontology is a trait dictionary, e.g. The Crop Ontology (CO) - a digital breeding tool providing access to validated lists of defined crop traits and variables- (www.cropontology.org).
[2] https://neo4j.com/

Selecting the ontology model for representing measurements

Domain knowledge can be modelled in different ways. There is no “wrong” answer, just different ways at looking at a given domain. For instance, several models already exist that represent observations and the related measurements (Observation and Measurements, OBOE, Basic Formal Ontology, …).

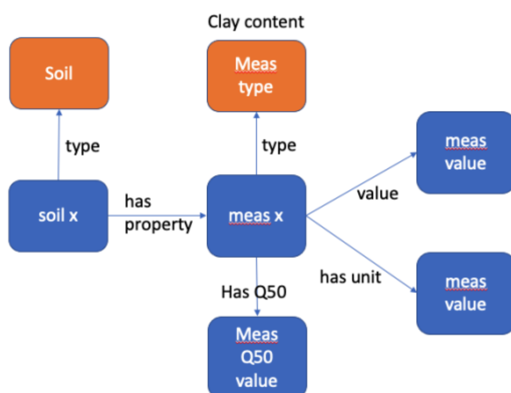
In this section, two models are proposed that are inspired from existing standards. The objective is to get something simple enough to implement it in a graph database (such as NEO4J) that doesn’t have a strong inference engine.

An ontology model is composed of statements (axioms) that describe a domain of interest by defining classes and properties as a domain vocabulary. In the examples below:

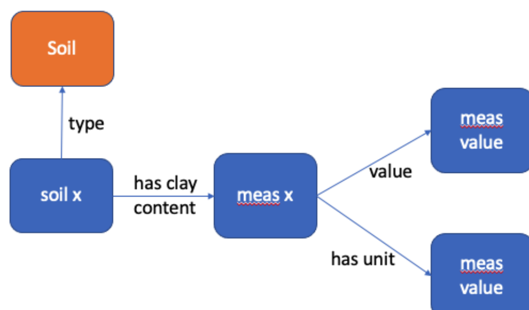
- schema axioms describe the domain → TBox (Orange)
- data axioms express facts → Abox (Blue)

Basically, there are two ways for modeling information. The modeling can be done at the class level (model 1) or at the property level (model 2).

Model 1—All the types of parameters are created as classes in the ontology. Then an instance corresponding to a given measurement in a dataset is linked to a measurement type, which is a class in the ontology. In the example below, the ‘meas X’ (= measurement) is exemplified here as a type ‘Clay content’.



Model 2 — Each measurement is modelled as an object property. My instance of soil then *'has a clay content*' - object property - the ‘Clay content’ value that is measured



Both models will need all the measurement types to be created as Classes or as Properties. The decision to choose between the two models will be based on how you will need to query the data, but also on the need to attach more complex information to the class. For instance, in the case of modeling measurements as properties, apart from the basic metadata that you can attach to the property, you won't be able to say that measurement x is somehow related to or affects measurement y. To represent this type of relation, measurements need to be modelled as classes. But in some cases, it is acceptable to represent things as properties.

Recommendation - The domain should not be modelled exactly after the data. In a dataset, measurements likely appear as properties of some things (location, soil, plant, …) but it might be better to model them in the ontology as classes, to define more precisely what they are.

If you add new data sources, the domain (i.e. the classes in the ontology) won't change, e.g. soil will always have same properties. When new data are described using different vocabularies then you may be led to remove or add axioms (e.g. adding a synonym, linking to a new term). You can add axioms in the domain (extend it) without modifying it too much. The content will change but not the modeling principles.

Making decision on axioms – what inference do I need?

When looking at a knowledge domain, it is easy to get carried away, and to want to represent everything as precisely as possible within the ontology.

To decide what kind of axioms are needed, or what terms need to be included, it is good practice to look at what semantic standards exist, and look at the data that need to be annotated with the ontology. It is also important to look at what different stakeholders might want to harvest from the data.

If we look at the Soil Physics ontology (https://archive.researchdata.leeds.ac.uk/42/), which is an Ontology of Soil Properties and Processes, they define a relationship as *'has impact on'*, which is used to present things, such as 'SoilClayMineral *'has an impact on'* SoilbufferingCapacity', or 'SoilTemperature' *has impact on* and *is influenced by* 'soil cover'. Being able to derive these types of inferences from the data might be useful at some point in the process, but might not be an immediate need in terms of integrating several soil datasets together. It is important to ask if the influence of the climate or soil cover on soil temperature will be useful for the project, or if you will have data on this. Also, modeling complex things in a web ontology language (OWL) implies that the database storing the data will be able to exploit OWL. For instance, the graph system called DGraph does not have an OWL reasoner. In such a situation, other ways to model complex information, if needed, will need to be implemented.

Modeling geolocation information

'**Geocoordinates'** are important pieces of information that need to be captured. They can be represented as a property of a location. At class level, you can indicate 'Soil' has a 'Location'

and the 'Location' has Geocoordinates. Then, you can use another database to get the location names (e.g. Geonames)

Proposal - Create a separate ontology for the Geocoordinates and Location, so if new locations imply changes, then you just modify the separate location ontology and you refresh the import into the Soil ontology.

Recommendation: When modeling you should think about the entire graph and not focus on the triple. Looking at the whole graph will provide vision of the links between e.g. a clay content and a measurement, a soil type.

Geolocation modeling

We are looking at modeling the two following use cases.

Use Cases:

- For a given geocoordinate, get region/country information
- For a given geocoordinate, get other points that are in the vicinity of it (even if the decimals are not the same)

Different approaches can be applied as follows.

Solution 1: Using a reverse geocoding service

e.g. Google map Geocoding API and Python libraries exist to do this. It means that you send geocoordinates and you get the admin levels in return.

Check guidelines: https://developers.google.com/maps/documentation/geocoding/overview

Solution 2: using Geonames

- Solution 2
  - Using Geonames: download and store the entire database
    - For each lat/long, get admin levels.
    - Ask for the nearest neighbor to a point to get the nearest feature (can be any admin level)
    - http://download.geonames.org/export/dump/
    - Geoname ontology: for their properties: https://www.geonames.org/ontology/documentation.html

Solution 3: Using any geodatabase

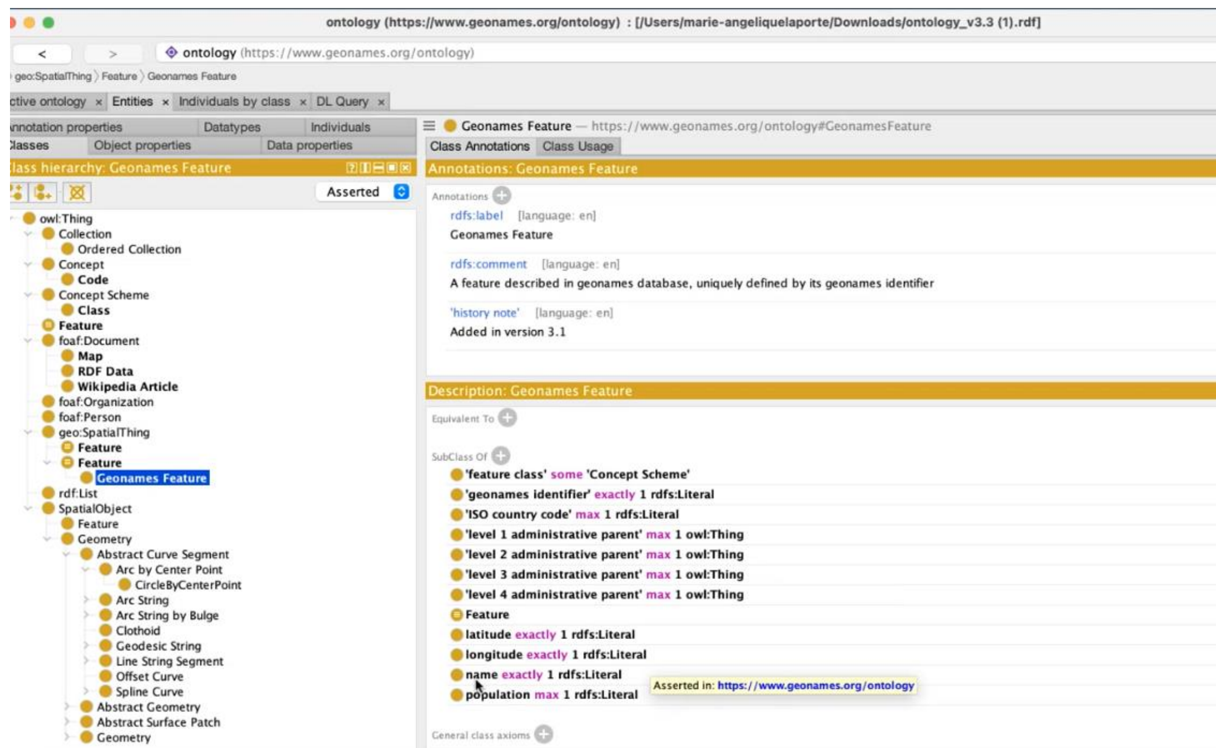
- Solution 3
  - Use any geodatabase with a GIS (ArcGIS, postGIS)
  - e.g. GADM https://gadm.org/) GADM is a database of the location of the world's administrative areas (boundaries).

Graph databases have limitations when dealing with geocoordinates and particularly when these have different formats. Apparently, there is no satisfactory solution in NEO4J to handle coordinates with different decimals.

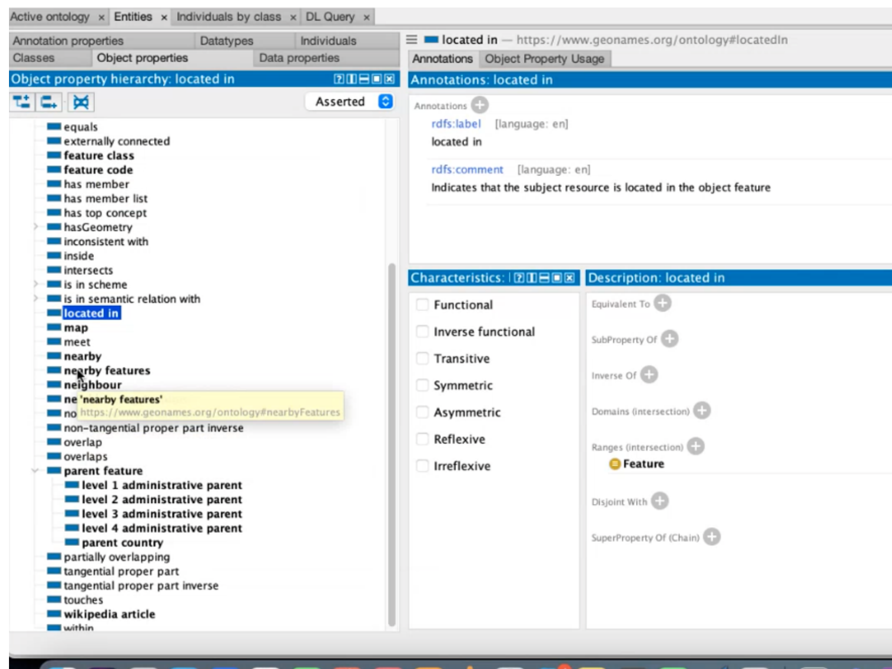
Existing Ontologies: Geonames vocabulary and Geosparql

- Defining properties and types:
  - Geonames: https://www.geonames.org/ontology/documentation.html
  - Geosparql: http://schemas.opengis.net/geosparql/1.0/geosparql_vocab_all.rdf

A Geoname Ontology can be opened in Protégé (https://protege.stanford.edu/, open-source tool to edit ontologies). It provides a good model.



and properties:



A polygon requires calculations to generate the points of a polygon and this must be embedded into the database. The issue affects the querying, not the storing.

Check the documentation on GEOSPARQL: https://Jena.apache.org/documentation/geosparql

Jena is a java library integrated in APACHE. Useful to programmatically develop ontologies.

The labelling of collections is as follows:

Collection	Geometry
MultiPoint	Point
MultiCurve	LineString, Curve
MultiSurface	Polygon, Surface
MultiGeometry	Point, LineString, Curve, Polygon, Surface

The Query presented hereunder enables returning to a polygon and, with the ‘contain’ property, users should indicate: give me all points contained in this ‘box’. ‘Location’ is a shape, and this shape has information. If you want to get all points coming from different data sources but are located in a given polygon, then you will need a specific SPARQL query.

Alternatively, utilising more shapes, relations and spatial reference systems can be achieved by converting the dataset to the GeoSPARQL structure.

```
?subj geo:hasGeometry ?geom .
?geom geo:hasSerialization ?geomLit .
#Coordinate order is Lon/Lat without stated SRS URI.
BIND("POLYGON((...))"^^<http://www.opengis.net/ont/geosparql#wktLiteral> AS ?box) .
FILTER(geof:sfContains(?box, ?geomLit))
```

Filtering functions:

Function Name	Description
?spatialObject1 **spatial:equals** *?spatialObject2*	Find *spatialObjects* (i.e. *features* or *geometries*) that are spatially equal.
?feature **spatial:intersectBox**(*?latMin ?lonMin ?latMax ?lonMax* [*?limit*])	Find *features* that intersect the provided box, up to the *limit*.
?feature **spatial:intersectBoxGeom**(*?geomLit1 ?geomLit2* [*?limit*])	Find *features* that intersect the provided box, up to the *limit*.
?feature **spatial:withinBox**(*?latMin ?lonMin ?latMax ?lonMax* [*?limit*])	Find *features* that intersect the provided box, up to the *limit*.
?feature **spatial:withinBoxGeom**(*?geomLit1 ?geomLit2* [*?limit*])	Find *features* that are within the provided box, up to the *limit*.
?feature **spatial:nearby**(*?lat ?lon ?radius* [*?unitsURI* [*?limit*]])	Find *features* that are within *radius* of the *distance* units, up to the *limit*.
?feature **spatial:nearbyGeom**(*?geomLit ?radius* [*?unitsURI* [*?limit*]])	Find *features* that are within *radius* of the *distance* units, up to the *limit*.
?feature **spatial:withinCircle**(*?lat ?lon ?radius* [*?unitsURI* [*?limit*]])	Find *features* that are within *radius* of the *distance* units, up to the *limit*.
?feature **spatial:withinCircleGeom**(*?geomLit ?radius* [*?unitsURI* [*?limit*]])	Find *features* that are within *radius* of the *distance* units, up to the *limit*.

Geolocation modeling Schema

During the data import into NEO4J, one solution would be to call an external geo-service to send the points you have and get reverse geocoding results. Geo-services are spatially referenced web services which make geodata accessible in structured form. Then, you can add the admin level information to the data (city, region, country) and store the information.

The query can then be done in NEO4J on the desired admin level for example. Give me all the points located in country X.

Conclusions

Modeling latitude and longitude (lat & long) in the ontology will be useful for using a geo model in the graph or with an external service.

Modeling the Polygon (boundaries) in the ontology is necessary.

Example of a query calling an external service to get values for a given location

```
In [126]:
1  def build_query(long, lat):
2      ##subclass of plant product
3      sparql_query = ("PREFIX gn: <http://www.geonames.org/ontology#> "
4                      "PREFIX spatial: <http://jena.apache.org/spatial#> "
5                      "PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> "
6                      "SELECT ?link ?name ?region ?region_name "
7                      "WHERE { "
8                      "?link spatial:nearby("+long+" "+lat+" 1 'km') .    "
9                      "?link gn:name ?name . "
10                     "?link gn:parentADM1 ?region . "
11                     "?region gn:name ?region_name
12                     "}" )
13     return sparql_query
```

Query is: “I send Lat and Long to get the name of the region (ADM1)”.

Note: Translation of region names: for example, Location names can be provided in various languages. GEONAMES provide some **translation** facility.

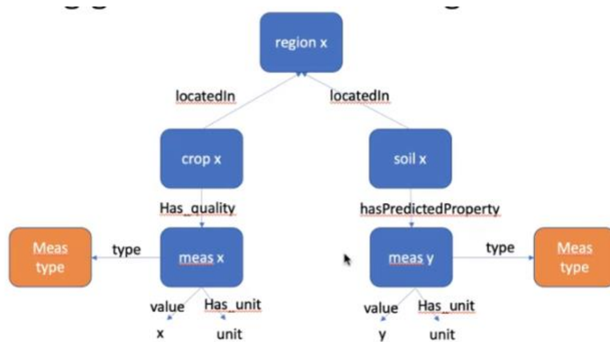
Query in the graph: Give me data located in Bavaria.

```
8
9  SELECT distinct ?uri ?label ?value ?unit
10 WHERE {
11     ?uri rdfs:label ?label .
12     ?uri rdf:type soil:CationExchangeCapacity .
13     ?uri rdf:value ?value .
14     ?uri soil:has_unit ?unit .
15     ?soil location:locatedIn location:Bavaria .
16     ?soil soil:HasPredictedProperty ?uri
17 }
18 '
```

Using the location to merge datasets

In this section, we will illustrate how two datasets from different domains can be merged. For that, we will use the soil ontology to annotate soil data, and a crop ontology to annotate crop data (crop measurement, growth stage, species/varieties). We recommend having a **generic**

ontology aside from crop and soil ontologies, that can be used to merge datasets that are described, and harmonize the generic types to facilitate querying.



With this model (illustrated above), 'Cropx' and 'Soilx' are linked by the location. For a cropx located in a given region, we can retrieve information on the crop and also on the soil because the two datasets are linked by their common geo-information.

The Soil and Crop ontologies were both loaded together in the script, along with instances and merged into one file to enable query.

```
In [151]:
1  %%script bash
2  arq --results CSV --data /Users/marie-angeliquelaporte/Documents/YARA/merged_soil_crop.owl
3  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
5  PREFIX owl: <http://www.w3.org/2002/07/owl#>
6  PREFIX soil: <http://www.semanticweb.org/ontologies/ODX_Soil#>
7  PREFIX location: <http://www.semanticweb.org/ontologies/ODX_Location#>
8
9  SELECT distinct ?uri ?label ?type ?value ?unit
10 WHERE {
11     ?uri rdf:type ?type .
12     ?type rdfs:label ?label .
13     ?uri rdf:value ?value .
14     ?uri soil:has_unit ?unit .
15
16 }
17

lkDensity,145,http://www.semanticweb.org/ontologies/ODX_Unit/cg/cm³
http://www.semanticweb.org/ontologies/ODX_Soil#odx_381,bulk density,http://www.semanticweb.org/ontologies/ODX_Soil#Bu
lkDensity,144,http://www.semanticweb.org/ontologies/ODX_Unit/cg/cm³
http://www.semanticweb.org/ontologies/ODX_Soil#odx_509,bulk density,http://www.semanticweb.org/ontologies/ODX_Soil#Bu
lkDensity,144,http://www.semanticweb.org/ontologies/ODX_Unit/cg/cm³
http://www.semanticweb.org/ontologies/ODX_Soil#odx_1,bulk density,http://www.semanticweb.org/ontologies/ODX_Soil#Bulk
Density,126,http://www.semanticweb.org/ontologies/ODX_Unit/cg/cm³
http://www.semanticweb.org/ontologies/ODX_Soil#odx_5,bulk density,http://www.semanticweb.org/ontologies/ODX_Soil#Bulk
Density,138,http://www.semanticweb.org/ontologies/ODX_Unit/cg/cm³
```

Get all the data, types and unit. You can get data for both the crop and the soil

```
http://www.semanticweb.org/ontologies/ODX_Soil#odx_415,organic carbon density,http://www.semanticweb.org/ontologies/O
DX_Soil#OrganicCarbonDensity,78,http://www.semanticweb.org/ontologies/ODX_Unit/g/dm³
http://www.semanticweb.org/ontologies/ODX_Soil#odx_474,organic carbon density,http://www.semanticweb.org/ontologies/O
DX_Soil#OrganicCarbonDensity,205,http://www.semanticweb.org/ontologies/ODX_Unit/g/dm³
```

and from the crop:

```
http://www.semanticweb.org/ontologies/ODX_Crop#odx_58,no3,http://www.semanticweb.org/ontologies/ODX_Crop#odx_19,0.0,h
ttp://www.semanticweb.org/ontologies/ODX_Unit/Co
http://www.semanticweb.org/ontologies/ODX_Crop#odx_57,co,http://www.semanticweb.org/ontologies/ODX_Crop#odx_8,0.0,ht
tp://www.semanticweb.org/ontologies/ODX_Unit/Unit
```

Get all data located in '**Bayern**'.

```
0 SELECT distinct ?location ?type ?value ?unit
1 WHERE {
2
3     ?location location:locatedIn location:Bayern .
4
5     {?location soil:HasPredictedProperty ?uri .
6      ?uri rdf:value ?value .
7      ?uri soil:has_unit ?unit .
8      ?uri rdf:type ?t . ?t rdfs:label ?type
9     } UNION
```

Relying on a generic ontology, which is shared between the two domains, is powerful as it makes it possible to query any class in the graph (crop, soil, etc.) that is linked to the same geocoordinates. For example, we may need to get the weather stations located in a specific area.

Get all data from ‘**somewhere**’ in Germany:

```
10 SELECT distinct ?location ?type ?value ?unit
11 WHERE {
12
13     ?location location:locatedIn ?somewhere .
14     ?somewhere rdfs:subClassOf location:Germany
15
16     {?location soil:HasPredictedProperty ?uri .
17      ?uri rdf:value ?value .
18      ?uri soil:has_unit ?unit .
19      ?uri rdf:type ?t . ?t rdfs:label ?type
20     } UNION
21     {
22         ?location crop:has_quality ?uri .
23         ?uri rdf:value ?value .
24         ?uri rdf:type ?t . ?t rdfs:label ?type
25     }
```

Class country

The ontology resulting from merging the Soil and Crop Ontologies:

To link location and geocoordinates in the Soil ontology, it is necessary to use a SPARQL (SPARQL Protocol and RDF Query Language) endpoint, and an Application Programming

Interface (API) to get information on the geocoordinates. Use an object property to make the link:

```
#print(data)
location = URIRef(NS+"odx_"+str(cpt))
cpt += 1
location_name = data["geometry"]["coordinates"]
long = str(location_name[1])
lat = str(location_name[0])
g.add((location, RDFS.label, Literal(long+","+lat)))
##get region
sparql_query = build_query(long, lat)
r = get_agrovoc_data(sparql_query)

if r.status_code == 200:
    results = r.json()["results"]["bindings"]
    for entry in results:
        region_name = entry["region_name"]["value"].replace(" ", "_")
        g.add((location, URIRef(NS_geo+"locatedIn"), URIRef(NS_geo+region_name)))
        break
```

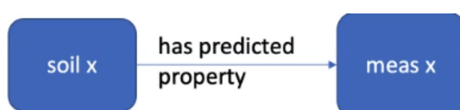
Measured vs. predicted values

Users need to be able to differentiate between measurements that have been measured in the field and those that derive from models or other predictive tools. The approach described below illustrates two different ways of modeling this type of information in the ontology. Inference engines or specific queries can be built on the knowledge represented in the ontology in order to derive the expected results: in this case, measured vs predicted values. A similar approach can be extended to different type of expected inferences, in different contexts.

There are two different ways of modeling knowledge to enable differentiating between measured and predicted values.

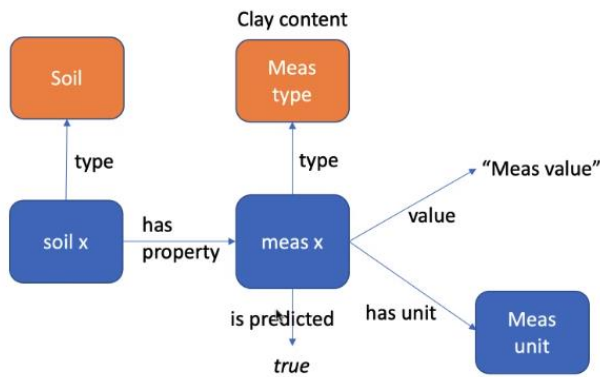
Solution #1 - Have a dedicated property and pointing to prediction and measurement

Use different properties for the measured vs. predicted values. Instead of using a 'Has property' to link a soil instance to a measurement instance, use a dedicated one that will have specific range 'has predicted property' since the range of this property will be defined as a Prediction, so everything linked will be a prediction.



Solution #2 - Use a data type property with a Boolean, in addition to all information.

We add a data type property 'is predicted', where all predictions would have the property 'has a prediction' and all measurements would not have this property.



The solution #1 is simple where 'is predicted' property is defined as 'object property'. For a given 'soil x location', several types of measurements and predictions can be done so the user should be able to separate the predictive value. The way data is collected or produced is different. Therefore, 'Predictions' need more information that can be enforced like:

- the model that generated the data
- Accuracy of the data
- Calibration used

Note: **Definition of a prediction in Crop Ontology:** A predictive value is a result of a model or a spectrum before it is directly observed so information about the model used is important. Also noted that recording the calibration will be important.

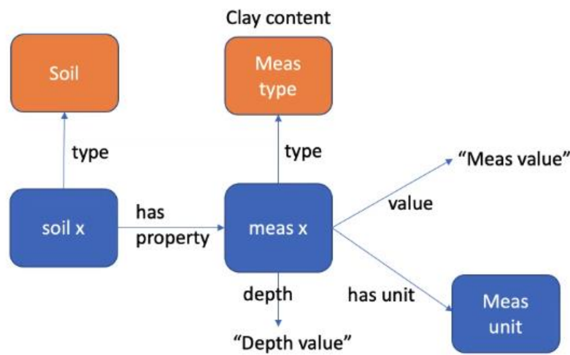
In RDF, we can build a shape in a Shape Constraint Language (SHACL; https://www.w3.org/TR/shacl/), which is the language for describing and validating RDF graphs. It will validate the fact that predicted data that has attached the necessary information. All World Wide Web Consortium (W3C) standards have a data validation language built on top of RDF to validate data, define constraints. It is possible then to check that the information required for a prediction really is in the data you hold, and not in accompanying metadata for example.

Soil Depth information

Another common need is to attach contextual information to a given measurement. For example, soil measurements are made at a certain soil depth, or within a specific soil depth range. Again, several modeling approaches are possible to fulfill the goal. Two of those are described below.

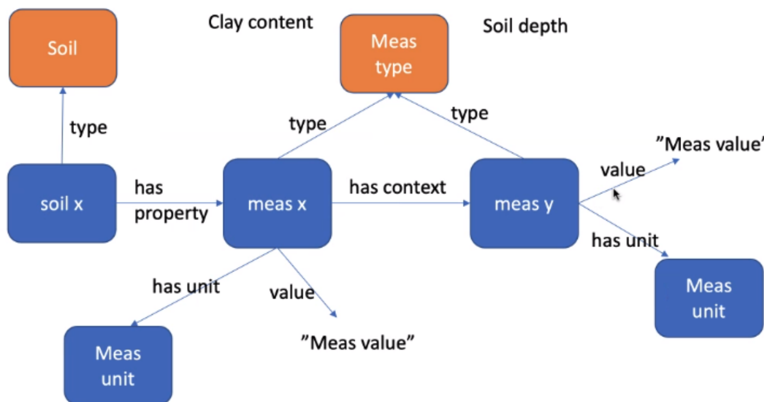
Model #1 - Easier solution: Create a data property to attach to the 'soil measurement'

The measurement has 'Depth' and we attach the value '**Depth value'** as a label – a limitation of this model is that **'Depth value' is literal (0-5 cm)** so the graph cannot be extended further. If the label is 0-5cm then it means that the unit is already in the label name and this won't accommodate a data provision from multiple sources as it would assume that all measurements are expressed in cm which won't always be the case. Having the information modelled as a literal element will therefore be less efficient for querying the data, as the literal element would need to be parsed before being used to filter the data for instance.



Model #2 - Emphasizing the flexibility for query: To create a second and contextual measurement for Soil depth.

⇒ Preferred solution to have **flexibility** in the way you query the data



⇒ Soil has 'property measurement'
⇒ Measurement X has 'context'
⇒ Context measurement will be 'Soil Depth' and has its own value and unit

This solution will lead to a larger and more complex model but powerful for querying the data.

Model #1 – This model is based on 3 major assumptions:

a) the way the measurements are done is standardized e.g. always cm
b) so if the data comes in differing formats, a transformation of the data will be necessary prior to upload into NEO4J (or any other database)
c) can't directly attach the unit unless you create a new property 'Depth unit'. It will add a lot of information on the node 'measurement x'

Model #2 - Soil Depth as second measurement

a) It assumes that datasets are/can be very different (units, prediction vs measurements, etc.).
b) In NEO4J create 2 nodes, each having their own properties.

c) The example of Soil Depth is a contextual measurement, but the same modeling could apply to any sort of contextual measurement

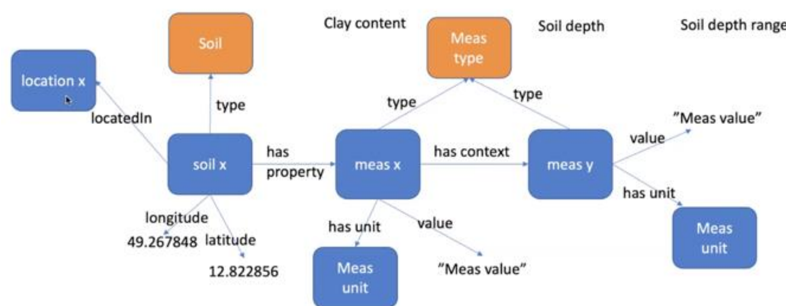
For example, if your soil-depth data ranges from 0 to 5cm, then this range could be represented at the class level in the ontology. **Add a class 'soil-depth range'.**

However, data sources will bring different scales like 0-10. Data Transformation won't always be possible (e.g. 0 - 10 vs. 0 - 5). The ontology should provide as much information as possible to support a description of data as precisely as possible, letting the scientist decide how to manage data with differing scales.

When data is not present (e.g., no information about the depth of the measurement), the ontology will not identify that data does not exist. This information could be in the metadata of the data set. It is then possible to add a 'Data type property' for '**soil-depth information missing'**

Management practice

Management practice

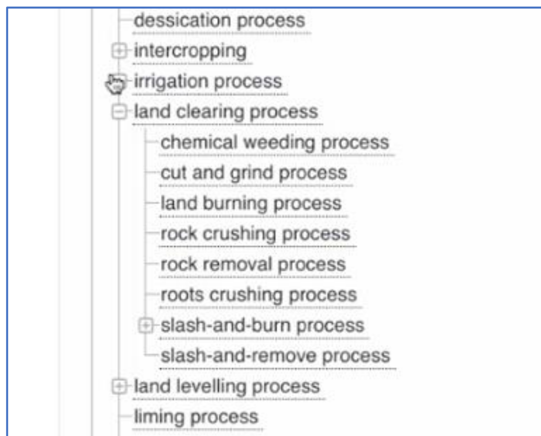


Field management practices are operations applied to the soil that will influence crop yields. Users need to answer queries like: "get a management practice linked to a soil at a precise location", or "give me all the points where tillage was done".

Management practice

- Management practices should be created as classes (as it was done in Soil V4). Definition should be provided
  - AgrO has a list of management practices listed under agricultural process. https://www.ebi.ac.uk/ols/ontologies/agro
- Instances of management practice:
  - Should be typed a given management process
  - Should have a time information ?
  - Should have information about machinery?

Management practices are created as classes with definition. For an example look at the Agronomy Ontology (AgrO) (https://bigdata.cgiar.org/resources/agronomy-ontology/).



Instances of a management process:

- A management process is done at a specific date or has start and end dates. Add the 'Timestamp' in the ontology. In AgrO, which follows the BFO rules (Basic Formal Ontology rules - https://obofoundry.org/principles/fp-000-summary.html), a treatment is a process so by definition is time-bound.
- A management process has machinery that can be added.

Provenance Metadata

Describes the '**Source**' of the data: versions, creation date, etc.

Existing standards allow modeling provenance metadata.

1. The Dublin Core (DC) Metadata Schema: In DC, the provenance is modelled simply, answering the basic questions Who, When How (https://www.dublincore.org/specifications/dublin-core/dcmi-terms/)
2. The Provenance Ontology (ProVO) provides a real modeling to provenance information - Entity generated by an Activity and has an Agent, with an associated date. Enables complex queries and allows to describe complex type of information.

DC Terms for provenance

DC Term	Relation	PROV Term	Rationale
dct:created	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the time of creation of a resource (i.e., the time of its generation). We map it as a subproperty of prov:generatedAtTime because "creation" is one of the many activities that generate an entity (for example, generation includes modification, issue, acceptance, etc.).
dct:creator	rdfs:subPropertyOf	prov:wasAttributedTo	A creator is one of the agents who participated in the creation of a resource. They have the attribution for the outcome of that activity.
dct:contributor	rdfs:subPropertyOf	prov:wasAttributedTo	A contributor is associated with either the creation activity or the updating of the resource. Therefore, he/she has attribution over the outcome of those activities.
dct:dateAccepted	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the date when the resource was accepted. dct:dateAccepted is mapped as a subproperty of prov:generatedAtTime because the accepted resource was generated by an "Accept" activity which may have changed it from its previous state.
dct:dateCopyrighted	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the date when the resource was copyrighted. dct:dateCopyrighted is mapped as a subproperty of prov:generatedAtTime because the copyrighted resource was generated by a "CopyRight" activity which may have changed it from its previous state.
dct:dateSubmitted	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the date when the resource was submitted. dct:dateSubmitted is mapped as a subproperty of prov:generatedAtTime because the submitted resource was generated by a "Submit" activity which may have changed it from its previous state.
dct:hasFormat	rdfs:subPropertyOf	prov:alternateOf	See rationale for dct:isFormatOf (as prov:alternateOf).
dct:isFormatOf	rdfs:subPropertyOf	prov:alternateOf	dct:isFormatOf refers to another resource which is the same but in another format. Thus, the term is mapped to prov:alternateOf.
dct:isFormatOf	rdfs:subPropertyOf	prov:wasDerivedFrom	dct:isFormatOf refers to another "pre-existing" resource which is the same but in another format (according to cdt:hasFormat), implying that the new resource is based on the former.
dct:issued	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the date when the resource was issued. dct:issued is mapped as a subproperty of prov:generatedAtTime because the issued resource is an entity itself, which has been generated at a certain time.
dct:modified	rdfs:subPropertyOf	prov:generatedAtTime	Property used to describe the date when the resource was modified. dct:modified is mapped as a subproperty of prov:generatedAtTime because the modified resource was generated by a "Modify" activity that changed it from its previous state.
dct:publisher	rdfs:subPropertyOf	prov:wasAttributedTo	A publisher has the attribution of the published resource after participating in the publishing activity that generated it.
dct:references	rdfs:subPropertyOf	prov:wasDerivedFrom	In PROV, a derivation is defined as "a transformation of an entity into another, an update of an entity resulting in a new one, or the construction of a new entity based on a pre-existing entity". If a resource n1 references another resource o1 then the construction of n1 is based on o1, even if o1 does not influence n1 significantly. Removing the reference to o1 in n1 would lead to the construction of another resource n1', different from n1.
dct:rightsHolder	rdfs:subPropertyOf	prov:wasAttributedTo	The rights holder has the attribution of the license associated to a resource. Thus, we can say that the resource is attributed in part to the rights holder.
dct:source	rdfs:subPropertyOf	prov:wasDerivedFrom	dct:source is defined as a "related resource from which the described resource is derived", which matches the notion of derivation in PROV-DM ("a transformation of an entity in another"). However, prov:wasDerivedFrom also covers broader derivations such as "an update of an entity resulting in a new one" which is not covered by dct:source.

Recommendation

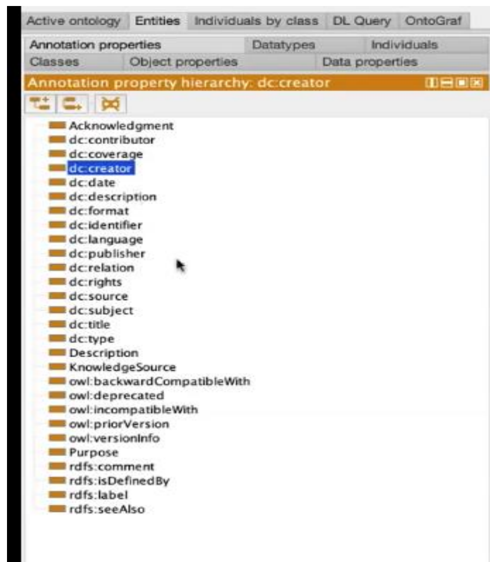
This information could be stored in a resource description framework (RDF) graph which will avoid getting a large graph that links each measurement to the same source. Better to create a new node having all provenance information, make it part of the graph and create an object property ‘has metadata’ that will lead to the record. It will be possible to query it.

For each data source all information of this metadata may not be present. So, it is recommended to define the properties in the ontology as ‘annotation’ so you can use it when importing the data. At each data node, you should have the known elements like the source. Having the info at the node level will be useful for a data scientist in accessing information about the source, for example:

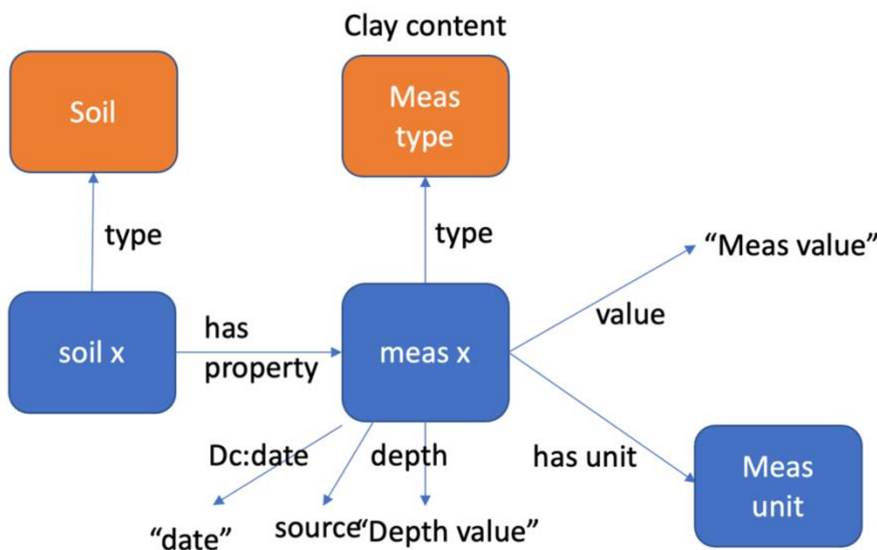
Best Practice

If you want to use an easy Dublin Core (DC) attribute, define it as ‘Annotation properties’ and use it at the data import step. At the level of the measurement, you can have 'dc.date' and have it linked to the data value.

List of DC terms in the Annotation Properties tab of Protégé:



Adding the information at the Node level:

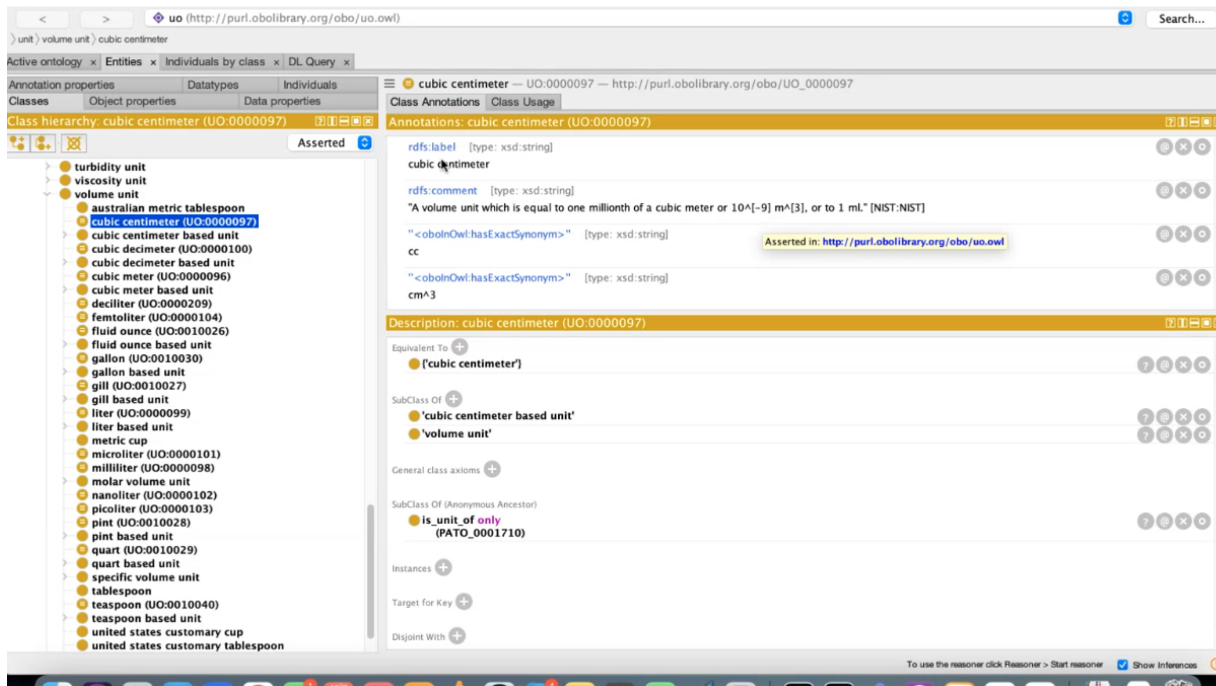


In the ontology you define the information type but not the value of the information. Each of the measurements should have only one source so one value of provenance information.

Reusing existing standard – the example of the unit ontology

Looking at the literature we identified two Unit Ontologies (UO) that could be reused in our soil ontology. The two ontologies are UO and OM. Will look at how they are modelled, and how they can be extended and used in our soil ontology in Protégé. An important need is to be able to use the ontology to guide unit conversion.

The Unit Ontology (UO)



UO provides some conversion rules and synonyms, but it is not easy to use as conversion rules for the data. It contains also some restriction like, for example, cubic meter unit can only be used for volume. The Agronomy Ontology (AgrO) uses UO but many gaps were identified about units that are widely used in Agriculture. Requests were made to ask for these units but it was not yet included.

Ontology of Units of Measure (WUR)

Units used in agriculture may not all be present in this one either. UM is more complex to support real life use cases.

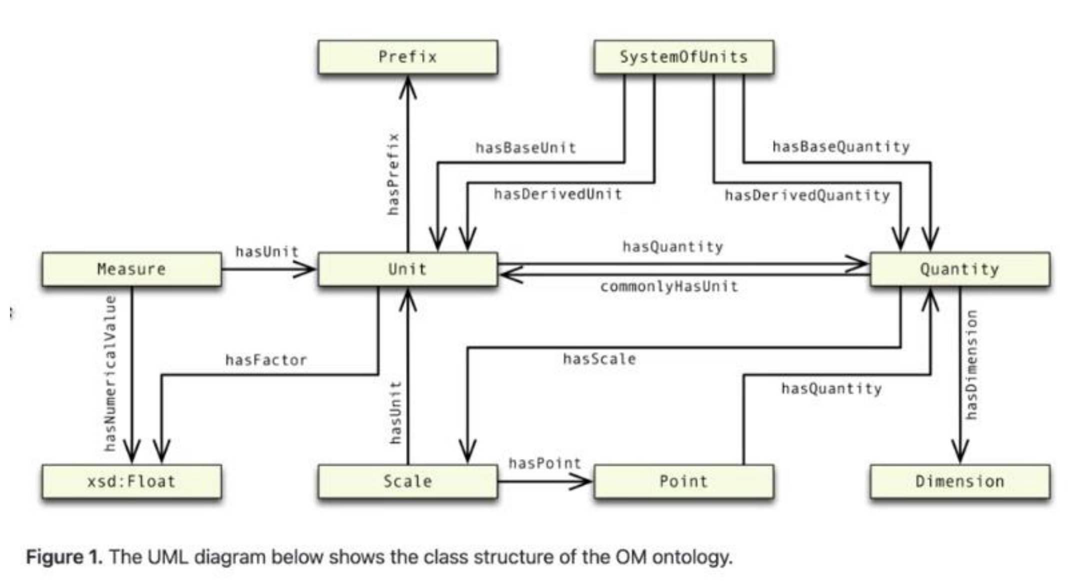


Figure 1. The UML diagram below shows the class structure of the OM ontology.

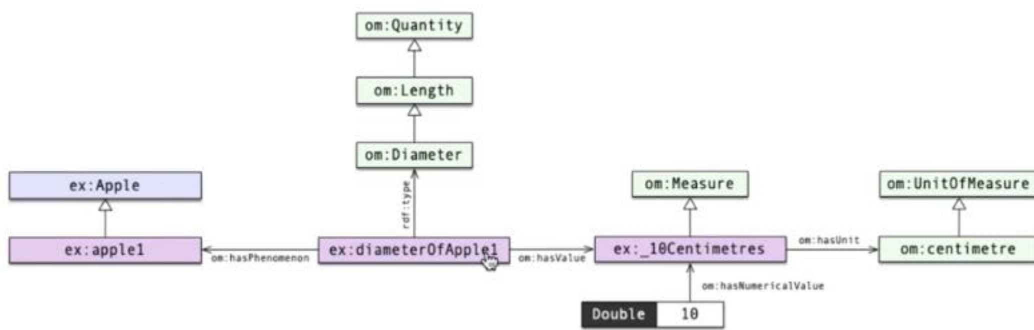
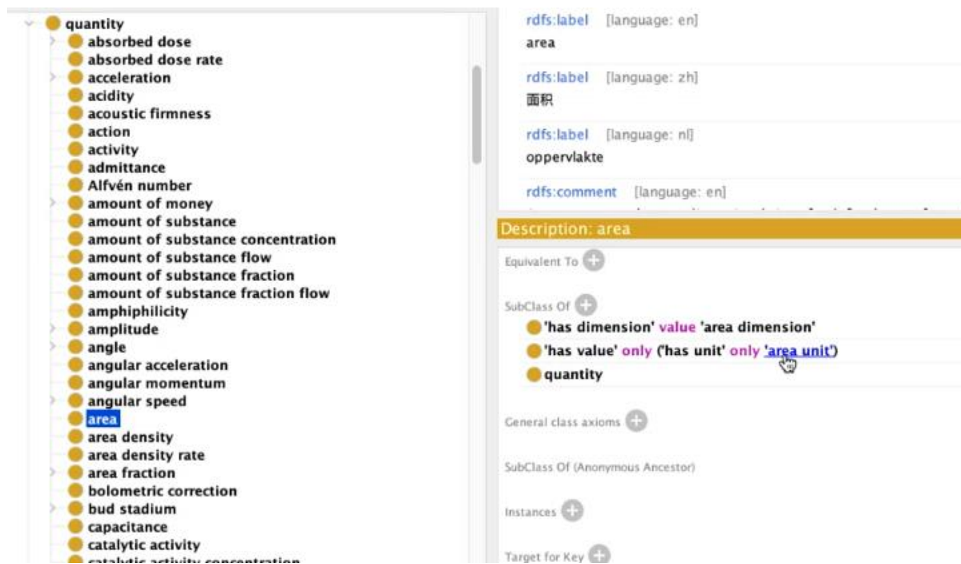
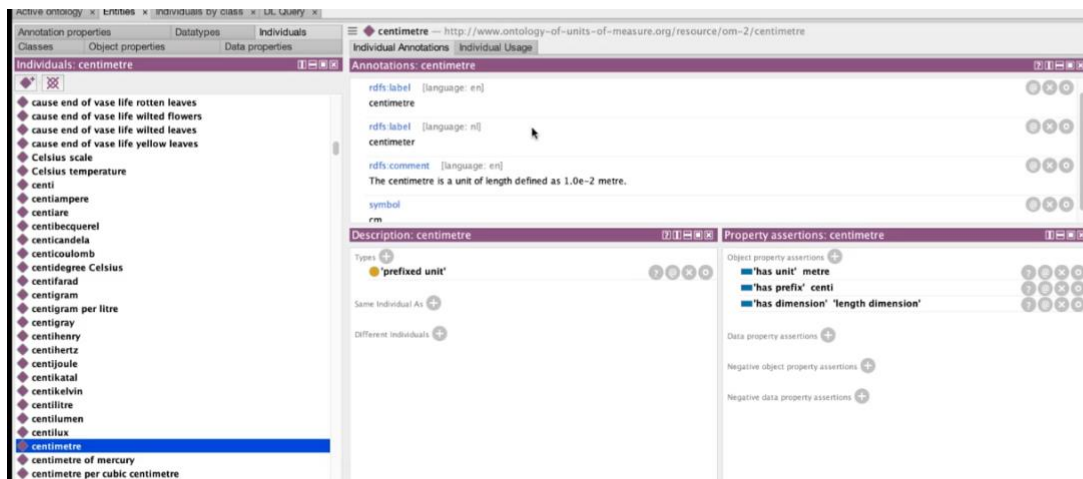


Figure 2. An RDF diagram representing the size of an apple as 10 cm.

In UM, everything is modeled as instances In Protégé, the example looks like this:





It contains some conversion factors although these are not really useful for our objective.

The conversion will need to be modelled in the Varda soil ontology, which implies maintaining a list of needed units and operations.

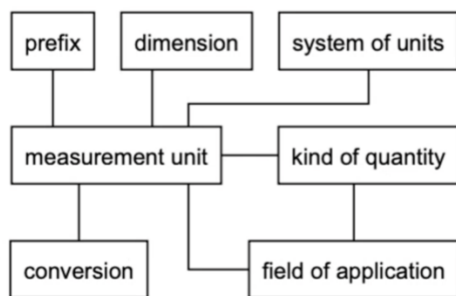
Unit conversion

In the literature, two models have been used to represent unit conversion.

Generic model

The following model can be used to model how unit should be modelled in the ontology with a clear emphasis of the unit conversion part.

Unit Ontology: generic model



From http://www.semantic-web-journal.net/system/files/swj1825.pdf

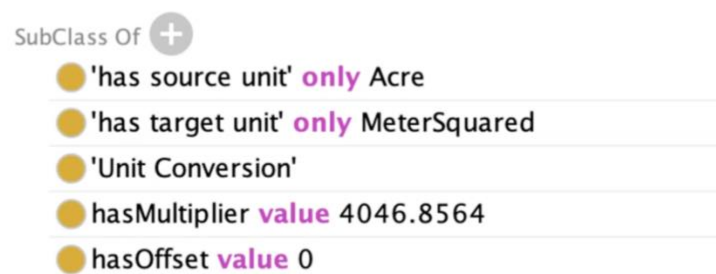
Based on the model, it could be useful to start a new unit ontology for Varda, that will be based on an existing ontology that could be extended with missing units as well as a unit conversion part.

OBOE with a 'unit Conversion' Class

In this example, the Extensible Observation Ontology (OBOE) also models the conversion as 'offset':

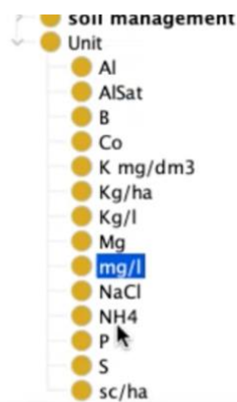
Unit Ontology: Unit of convertion

Example of model coming from the OBOE ontology



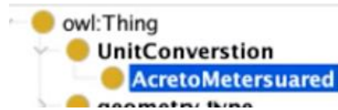
Create a class for '*Acre to square meter*' conversion and a '*multiplier*'. The conversion is at the class level. Create a class ' Unit conversion'

The unit in Protégé is built after some units extracted from the data files. The unit conversion was not modelled into the Varda ontology.



To model a conversion, you can create a class. For example, you want to create an "Acre to MeterSquared conversion" **class** like in the OBE example. So, column 'unitID' is acre and the 'ConvertoUnitID' is meter squared.

The best is to have a 'UnitConversion' class:



and indicate the source unit is 'Acre' and target unit is 'Meter squared' and 'multiplier'

Creating a Unit Ontology based on existing standards

Starting your own ontology: the example of the unit ontology

1- Look at what exists and see if/how it fits your needs
- UO: https://github.com/bio-ontology-research-group/unit-ontology
- OM: https://github.com/HajoRijgersberg/OM

2- Reuse as much as possible
- Select the terms you want in existing ontologies by reusing URI, but also importing all the metainformation linked to that URI
- Place those terms under a "Unit" root that is defined in your namespace

3- Add what is missing
- Create new units as needed

4- Managing version of your ontology
- Define a mechanism for versioning

5- Repeat steps 3-4

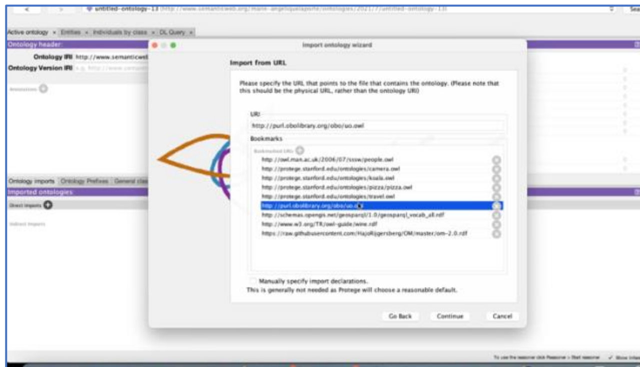
None of the existing ontology was satisfactory, as we found too many gaps. So, one needs to develop one's own ontology. It is still possible to reuse part of the two existing ontologies.

Recommendations:

- keep the same Uniform Resource Identifier (URI) and import all metadata attached to the URI.
- Root term should be in your own namespace to support the querying.
- It is important to have a solution for managing the **versioning** of your ontology and keep metadata attached to the ontology.

Creation of a new unit ontology in protégé, based on existing standards

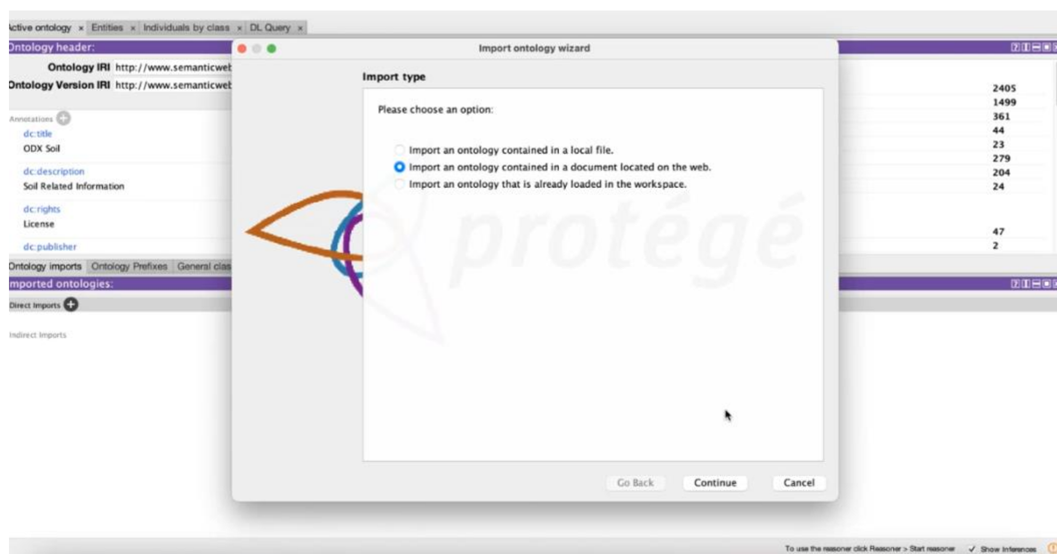
i-Import of UO:



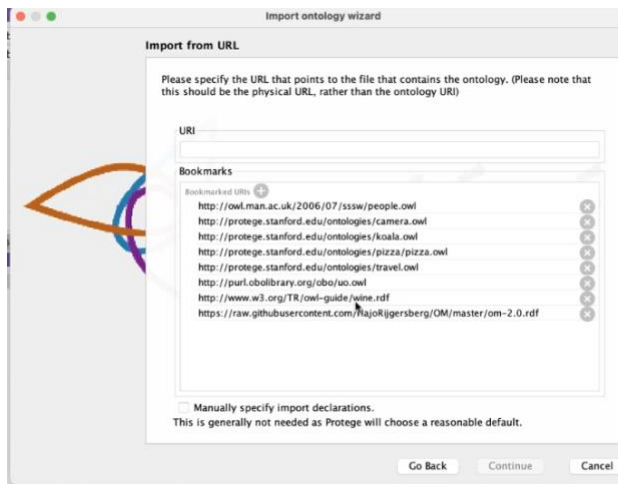
All metadata information is imported (synonyms, definitions, links) and some of the properties.

1. create your own root and namespace
2. clean – remove classes you do not need either manually or programmatically. It is time-consuming at the beginning.
3. Each class keeps its own URI.
4. Add you units with definitions
5. Make a first version of it and start using it.

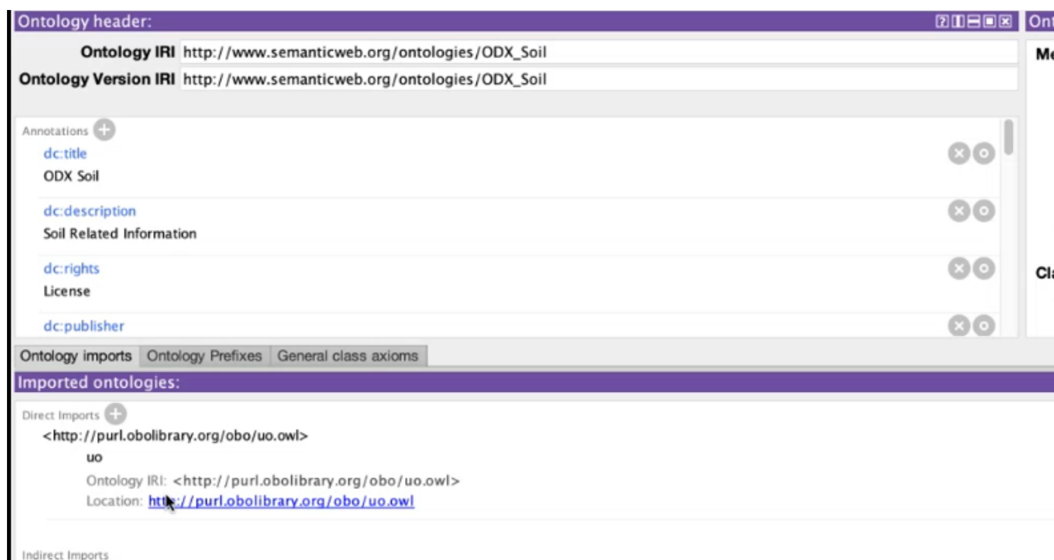
Import an ontology portion:



Select the ontology



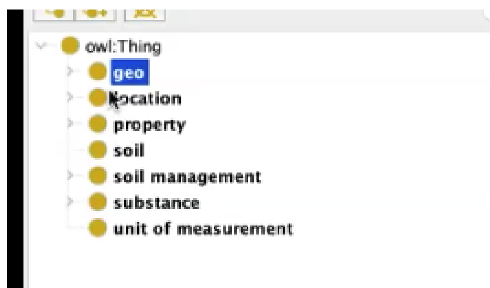
Then the ontology is imported. Example of Unit Ontology:



The above steps show how to build an ontology by importing parts of existing ontologies that you need. Now the ontology can be extended with new terms, and the new classes related to unit conversion.

Developing a Soil Ontology in Protégé

Labels were created for all classes:

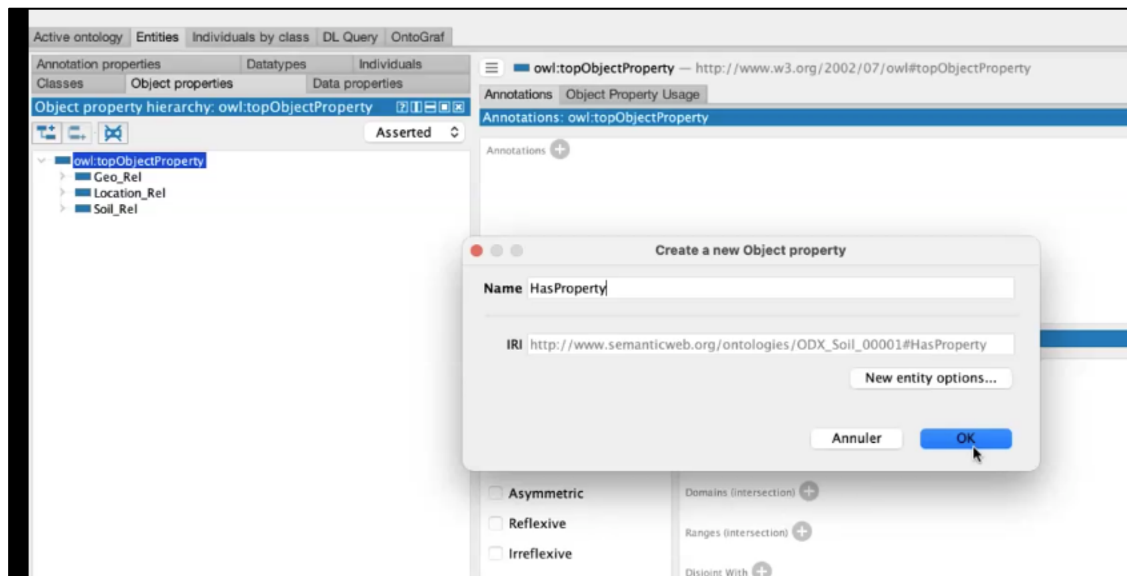


Focus on ‘Properties ‘class

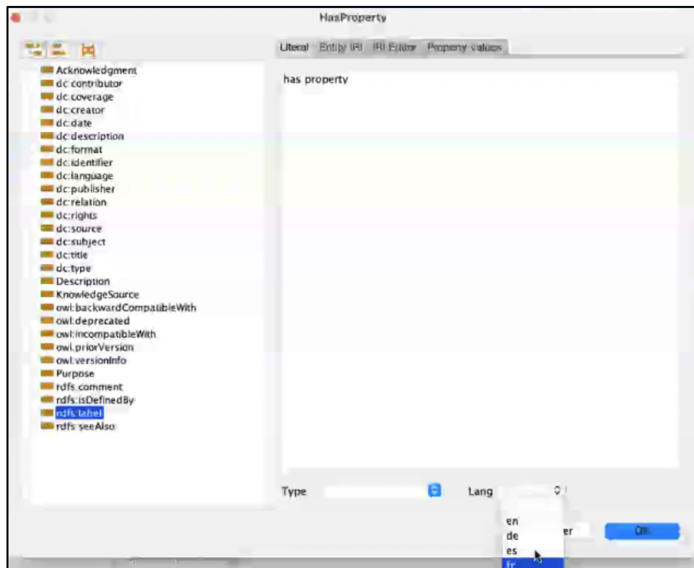
Linked to Properties

- Soil
- Soil properties
- Management also linked to the Soil
- Substances
- Unit of Measurements

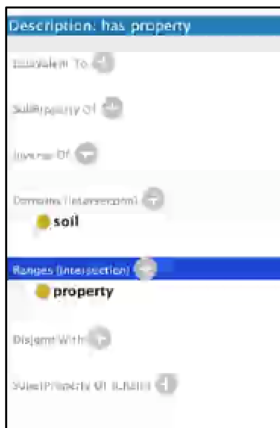
Create a new Object property: **'Has property'**



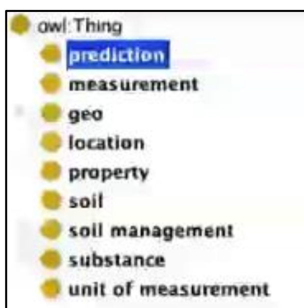
with a label '**has property**'



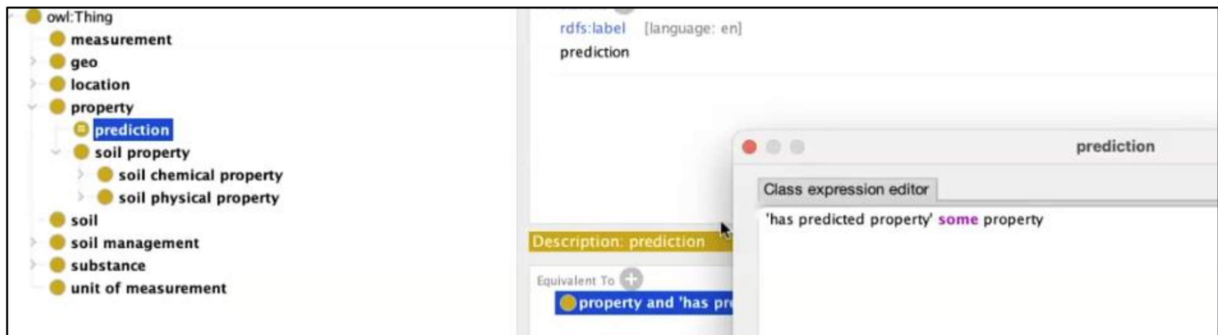
Define the Domain as ‘**Soil’** and Range as ‘**Property**’:



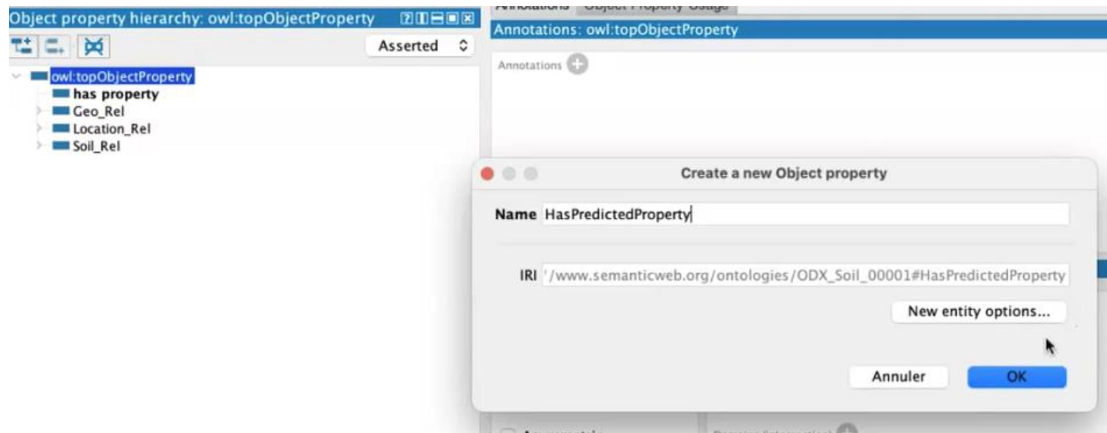
Objective: How to differentiate a real Measurement from a Prediction? A Prediction Class needs to be created and precisely defined in the ontology. The information of what is a prediction versus a measurement is not directly stated as such in the data, but it is a user requirement, as it is important for a scientist using the data to be able to treat them differently. That is a good example where the domain modelling in the ontology goes beyond what is present in the data. A domain ontology should model what is important for users in addition to the direct domain knowledge.



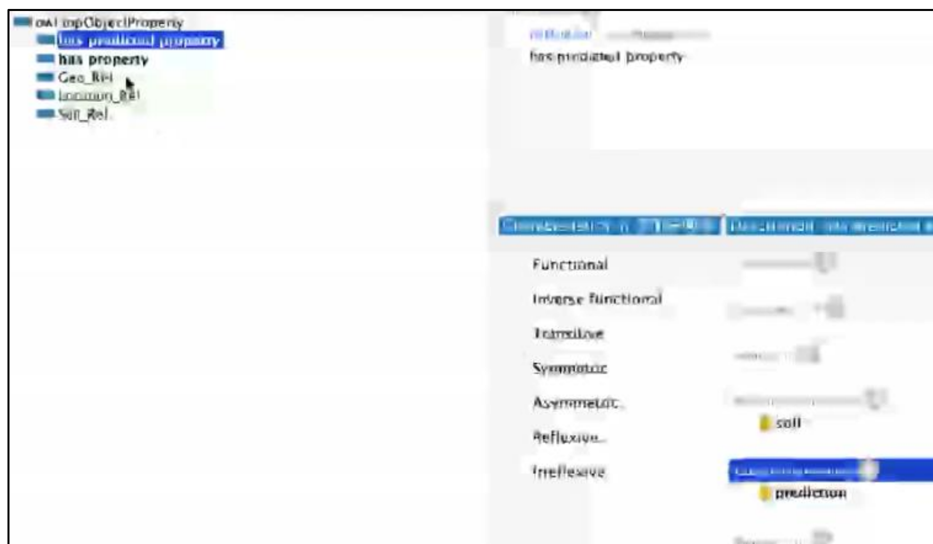
We will create a ‘**Defined class’** so the property ‘**has predictive property**’ will be a **grouping category** to group prediction data. Depending on the relation linked, the system will know if the data is a prediction or a measurement.



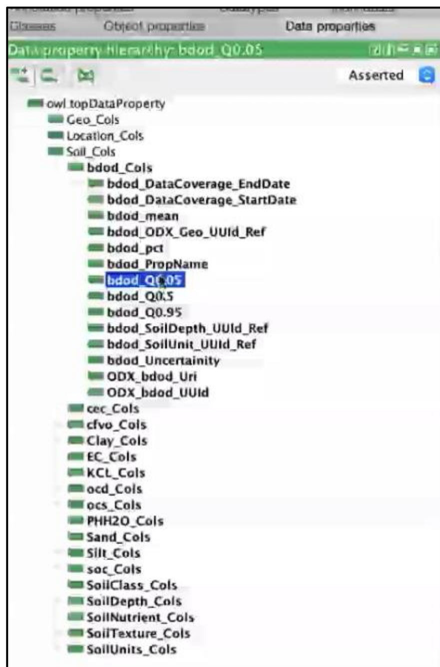
Then create a new Object Property: ‘**HasPredictedProperty**’ with its label



It will go from Domain ‘**Soil**’ to Ranges ‘**Prediction**’:



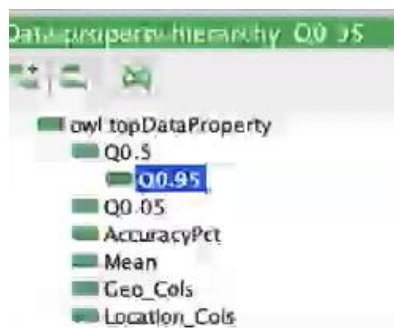
A prediction can have several Data properties:



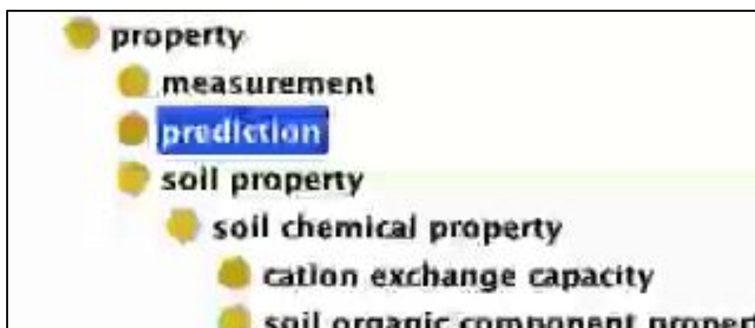
Proposal

use Dublin Core **'Coverage'** that can be spatial or temporal range.

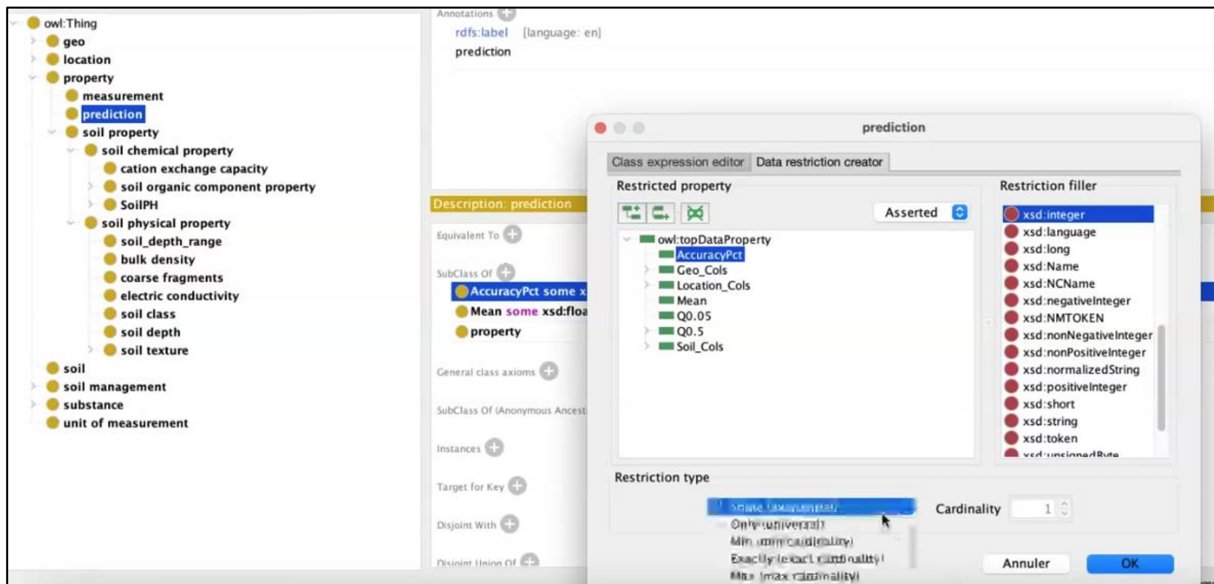
A Prediction' can have a mean, a pct/accuracy so we create **properties**:

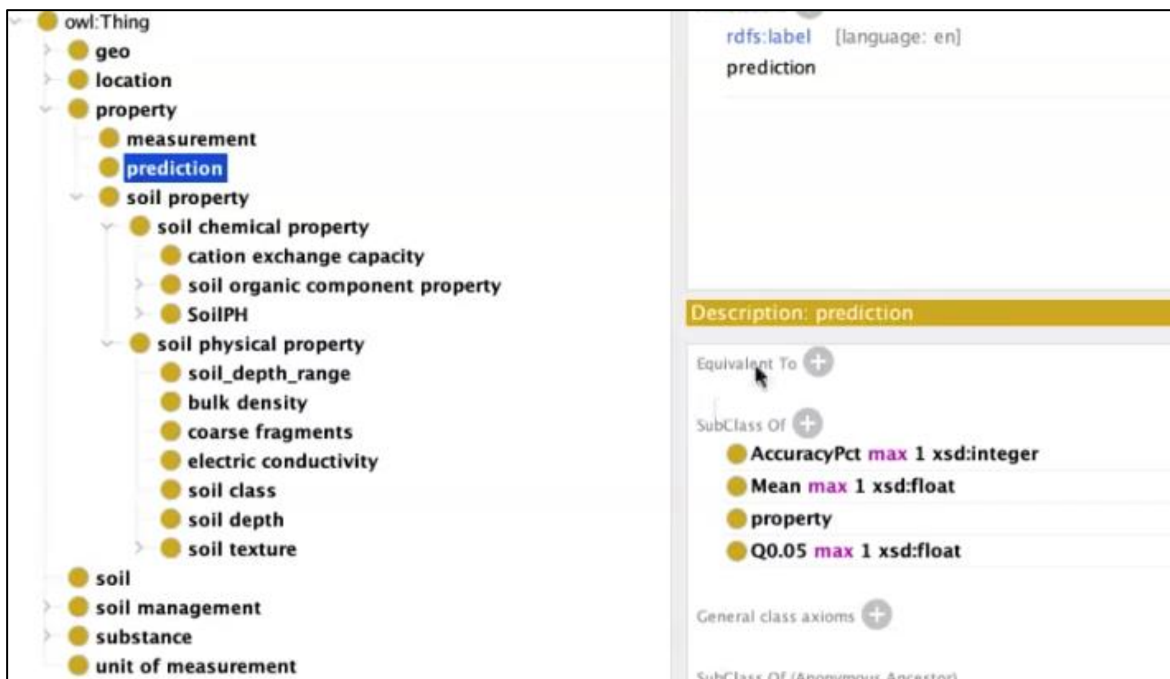


'Prediction' and 'Property' are placed under '**Property**': as grouping categories:



For the 'Prediction' add a 'restriction' like should have an 'AccuracyPct' that should be an integer, should have a mean value:

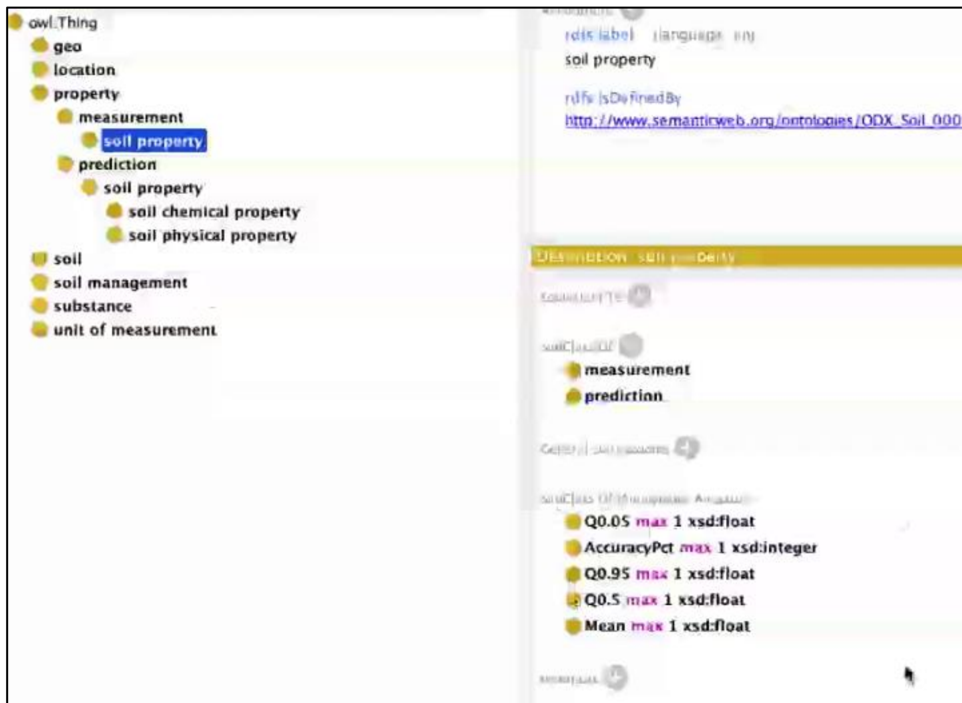




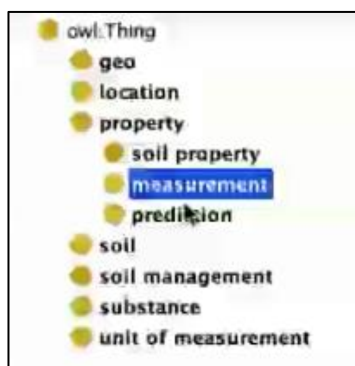
Indicating ‘max 1 xsd:integer’ indicates that it should have one value max.

‘Soil Properties’

It can be placed in both under ‘Prediction’ and ‘Measurement’ classes, but this is not correct. The reason is that the ‘Soil Property’ subclass inherits of the subclasses of both ‘Prediction’ & ‘Measurement’ classes e.g. AccuracyPCT and this is wrong for the measurement:

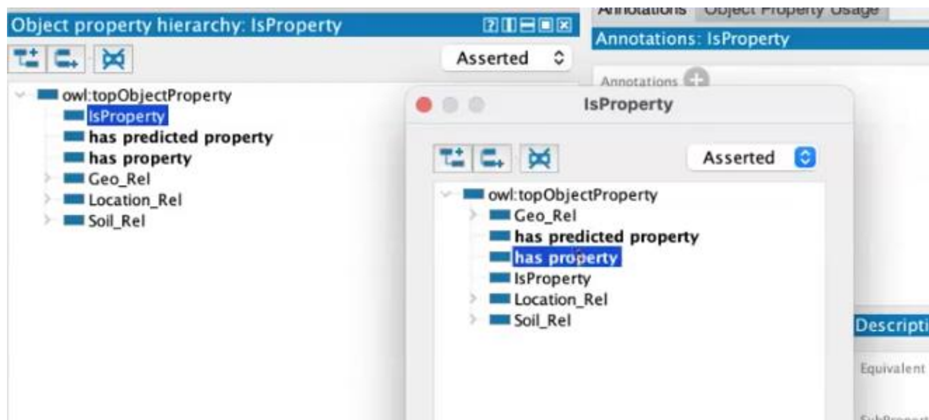


Therefore, ‘Measurement’ and ‘Prediction’ are made subclasses of ‘Property’ aside ‘Soil Property’:

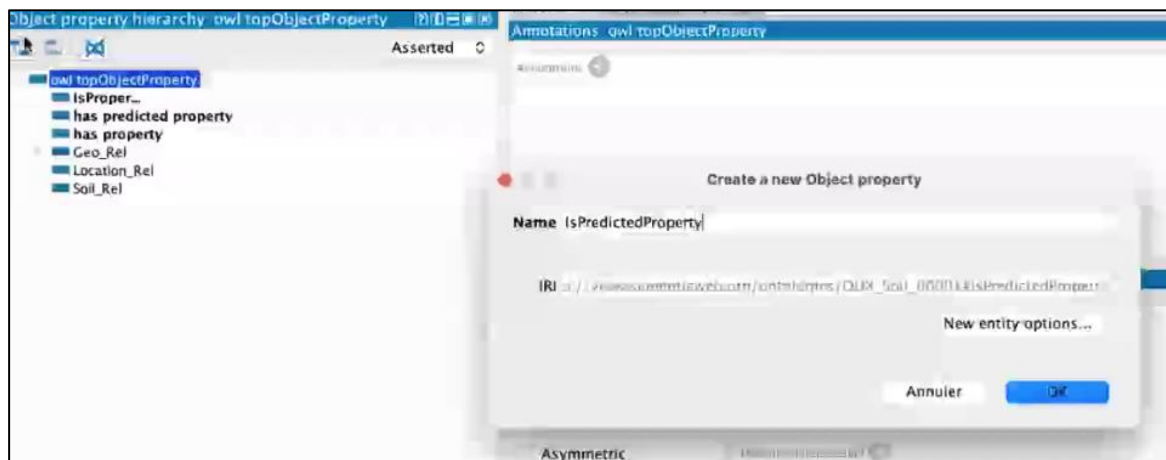


We create the **inverse Property** of **‘Has Property’** to enable the distinction between Prediction and Measurement:

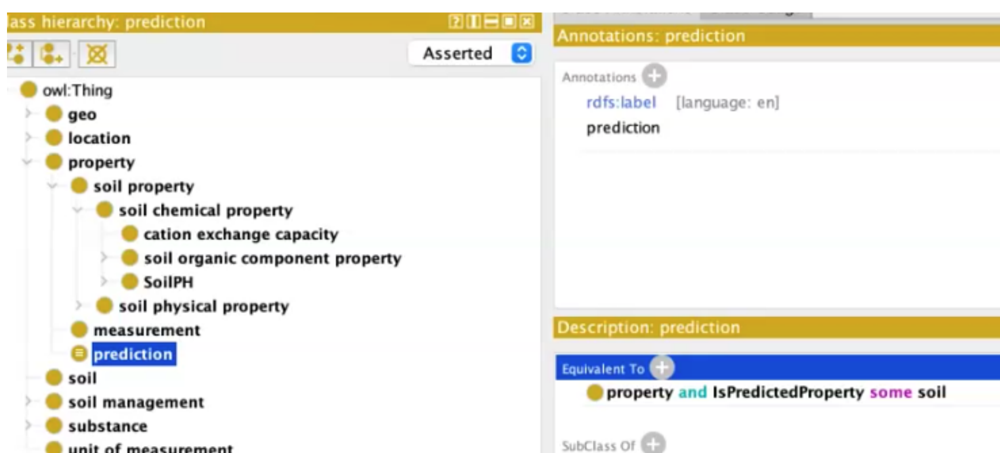
And indicate it is the **reverse** of **Has Property**



And we create the inverse Property 'IsPredictedProperty':



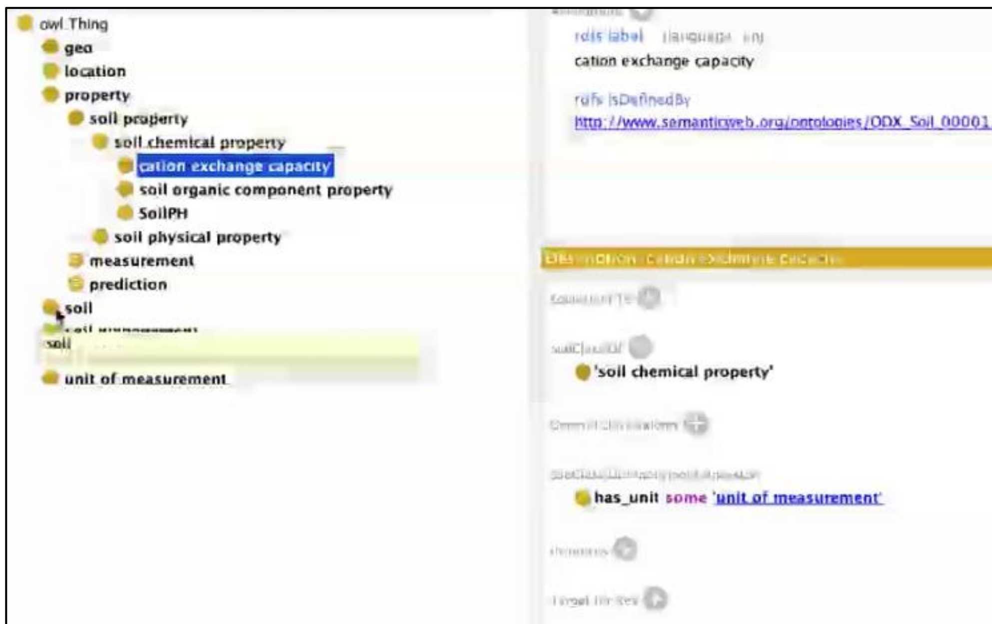
Then we add an **Object Property** to 'Prediction':



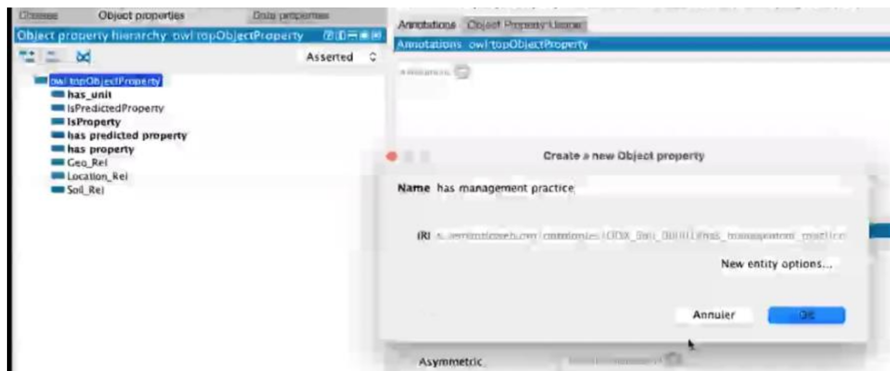
Same for Measurement.

Any Property can be linked to a '**Unit of measurement**':

Now each Soil Property inherited 'Has_Unit of Measurement'

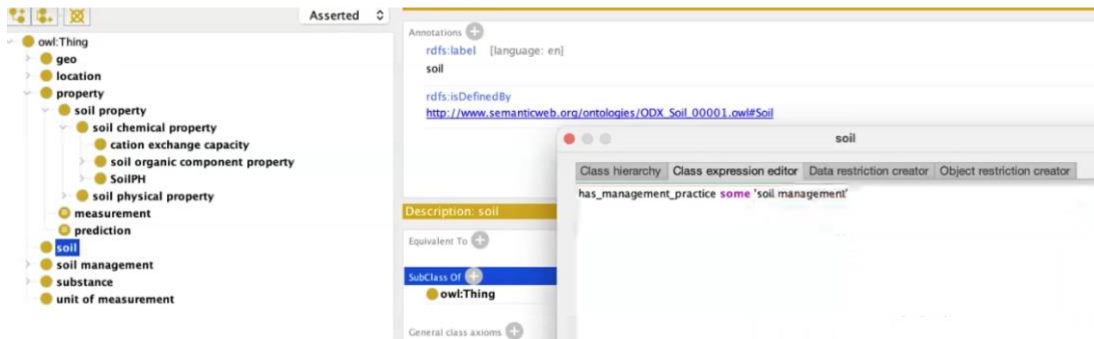


To link 'Soil' to 'Soil Management', we need to create a new **Object Property**:



A 'Soil_depth_range' range property was added under 'Soil physical property'

Then:

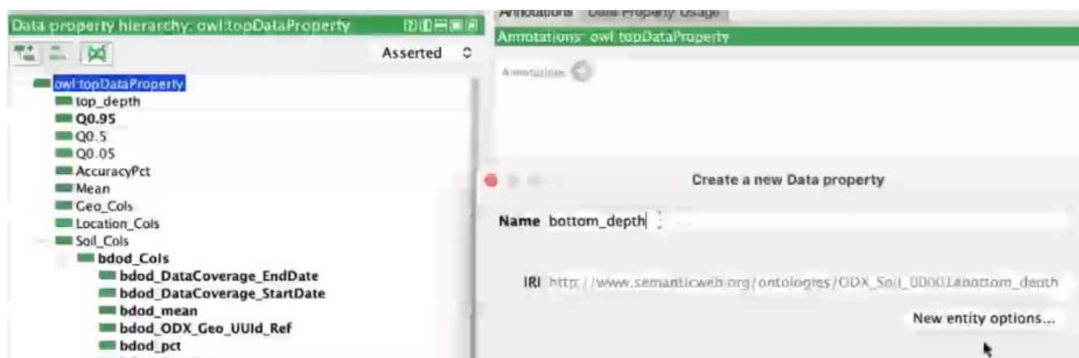


Add property to link '**Soil'** to 'Soil Management Practice'

For '**Soil Depth'** and '**Soil depth range'**

⇒ **Create New data propert**ies:

- Top depth
- Bottom depth



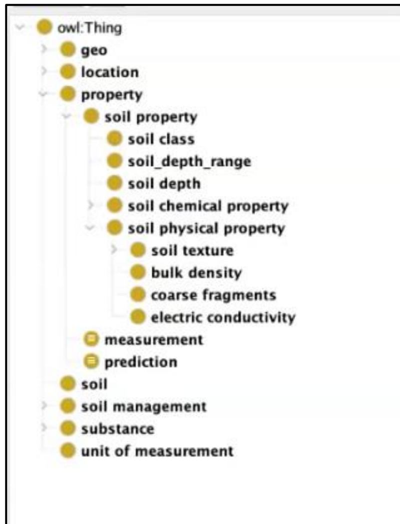
If a Property is a 'Soil Depth range' then its 'Soil_Depth_range' should exactly have a bottom depth and top depth:

Last, we create an object property '**has context'** with

Proposal

'Cation exchange capacity' is a 'Chemical Property' and always comes with a 'Soil Depth' so the proposal is to put 'Soil Depth' directly a sub-class of Property. And Soil Texture is a Physical property'

So the result is:



Note: The Ontology is currently missing all the 'Soil-Class'; import all classes in the ontology


```
],
"query_time_s": 0.32003355026245117,
"wrb_class_name": "Leptosols",
"wrb_class_value": 16,
"wrb_class_probabilit": [
  [
    "Leptosols",
    77
  ],
  [
    "Arenosols",
    7
  ],
  [
    "Regosols",
    6
  ],
  [
    "Solonchaks",
    5
  ],
  [
    "Fluvisols",
    3
  ],
  [
    "Vertisols",
    1
  ],
  [
    "Acrisols",
    0
  ],
  [
    "Albeluvisols",
    0
```


Each time there is data with new soil property, it will be added into the soil Property Class.

Checking the Model with Data

The adopted model provides value, context and unit to the Measurement.

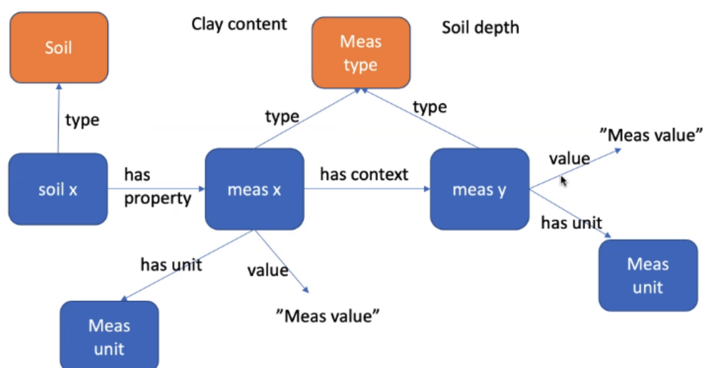


Figure 1: final model

A data import can be performed in Python (https://www.python.org/) for a quick result, loaded the file and built triples based on the ontology. The purpose is to **transform data into RDF** to create the instances to check that the model is consistent, and perform as expected.

jupyter YARA soil Last Checkpoint: 28 minutes ago (autosaved)

Fichier Édition Affichage Insérer Cellule Noyau Widgets Aide

```
In [3]:
1  ## create graph
2  g = Graph()
3  g.parse("/Users/marie-angeliquelaporte/Documents/YARA/ODX_Soil_V4.ttl", format="n3")
4  ## define namespace
5  NS = "http://www.semanticweb.org/ontologies/ODX_Soil#"
6
7  measurements = {"bdod":"BulkDensity","cec":"CationExchangeCapacity","cfvo":"CoarseFragments","clay":"ClayContent",
8                  "nitrogen":"NitrogenContent","ocd":"OrganicCarbonDensity",
9                  "ocs":"OrganicCarbonStocks","phh2o":"PHH2O","sand":"SandContent",
10                 "silt":"SiltContent","soc":"SoilOrganicCarbon"}
11 cpt = 0
```

```
In [99]:
1  with open('/Users/marie-angeliquelaporte/Documents/YARA/soilgrid_json/Krause.json') as f:
2      data = json.load(f)
3      #print(json.dumps(data["properties"]["layers"], indent=2))
4
5      #print(data)
6      location = URIRef(NS+str(cpt))
7      cpt += 1
8      location_name = data["geometry"]["coordinates"]
9      g.add((location, RDFS.label, Literal(location_name)))
10     g.add((location, RDF.type, URIRef(NS+"Soil")))
11     range_0 = 0
12
13     for prop in data["properties"]["layers"]:
14         name = prop["name"]
15         unit = prop["unit_measure"]["mapped_units"]
16
17
18         for d in prop["depths"]:
19             top = d["range"]["top_depth"]
20             bottom = d["range"]["bottom_depth"]
21             unit_depth = d["range"]["unit_depth"]
22             label = d["label"]
```

Steps taken for loading the ontology and data

1. **load the ontology** to get all classes, relationships etc. We created this in Protégé to avoid recreating this in Python.
2. Create a **Data Dictionary** that provides abbreviations with full name
3. Use 'rdflib' library in python
4. Load and parse the **file to create the instances.**

```
1  with open('/Users/marie-angeliquelaporte/Documents/YARA/soilgrid_json/Krause.json') as f:
2      data = json.load(f)
3      #print(json.dumps(data["properties"]["layers"], indent=2))
```

The name in the data will be then converted into a URI. We will create a new ttl file.

5. used the location information from the file to create a Soil instance and that the instance has a name and coordinates:

```
location_name = data["geometry"]["coordinates"]
g.add((location, RDFS.label, Literal(location_name)))
g.add((location, RDF.type, URIRef(NS+"Soil")))
range_0 = 0
```

6. Then parsed the information for each property:

```
for prop in data["properties"]["layers"]:
    name = prop["name"]
    unit = prop["unit_measure"]["mapped_units"]

    for d in prop["depths"]:
        top = d["range"]["top_depth"]
        bottom = d["range"]["bottom_depth"]
        unit_depth = d["range"]["unit_depth"]
        label = d["label"]
        q5 = d["values"]["Q0.05"]
        q50 = d["values"]["Q0.5"]
        q95 = d["values"]["Q0.95"]
        mean = d["values"]["mean"]
        uncertainty = d["values"]["uncertainty"]
```

```
  },
  "properties": {
    "layers": [
      {
        "name": "bdod",
        "unit_measure": {
          "d_factor": 100,
          "mapped_units": "cg/cm³",
          "target_units": "kg/dm³",
          "uncertainty_unit": ""
        },
```

Find a source that provides tools to convert into a given unit

⇒ Check the WUR Unit ontology: https://www.wur.nl/nl/product/Ontology-of-units-of-Measure-OM.htm

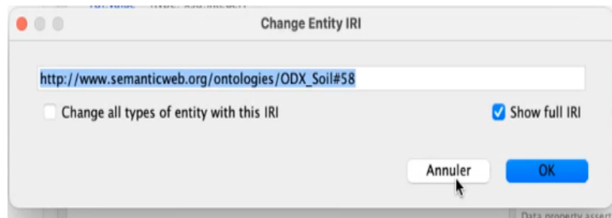
Creation of RDF triples

a. Generate a URI that will be an incremented number (cpt =counter)

```
uri_ip = URIRef(NS+str(cpt))
cpt += 1
```

For example, the URI of 'silt 5-15cm' will be #58:

silt 5-15cm
silt 60-100cm



b. Attached it to the information needed for a specific soil measurement (like in the conceptual model Figure 1: final model)

```
## create triples
g.add((uri_ip, RDF.type, URIRef(NS+measurements[name])))
g.add((uri_ip, RDFS.label, Literal(name+" "+label)))
g.add((URIRef(NS+unit), RDF.type, URIRef(NS+"SoilUnits")))
g.add((uri_ip, URIRef(NS+"has_unit"), URIRef(NS+unit)))
```

c. Attach the unit to the measurement

```
g.add((location, URIRef(NS+"HasPredictedProperty"), uri_ip))
```

d. Link the Soil to the Measurement and it is a Prediction:

```
uri_ip = URIRef(NS+str(cpt))
cpt += 1
## create triples
g.add((uri_ip, RDF.type, URIRef(NS+measurements[name])))
g.add((uri_ip, RDFS.label, Literal(name+" "+label)))
g.add((URIRef(NS+unit), RDF.type, URIRef(NS+"SoilUnits")))
g.add((uri_ip, URIRef(NS+"has_unit"), URIRef(NS+unit)))

g.add((location, URIRef(NS+"HasPredictedProperty"), uri_ip))
```

*A ‘**uri_ip**’ = Instance Property.*

Add a ‘**literal**’ so a value will have a ‘Soil-range’ and the category ‘0-5cm’ will be stable.

Good practice: Avoid creating a range for every property

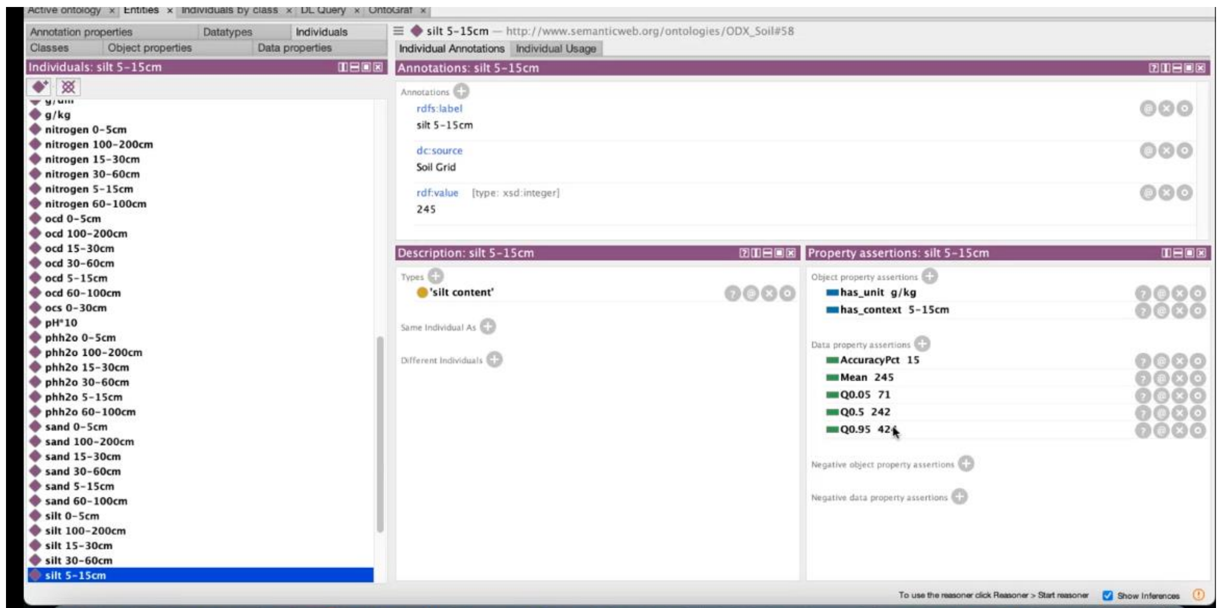
```
if (None, RDFS.label, Literal(label)) in g:
                # context already exists - just link
                for s, p, o in g.triples((None, RDFS.label, Literal(label))):
                    g.add((uri_ip, URIRef(NS+"has_context"), s))
                    break
else:
    context = URIRef(NS+str(cpt))
    cpt += 1
    g.add((uri_ip, URIRef(NS+"has_context"), context))
    g.add((context, RDF.type, URIRef(NS+"soil_depth_range")))
    g.add((context, URIRef(NS+"top_depth"), Literal(top) ))
    g.add((context, URIRef(NS+"bottom_depth"), Literal(bottom) ))
    g.add((context, URIRef(NS+"has_unit"), URIRef(NS+unit_depth) ))
    g.add((URIRef(NS+unit_depth), RDF.type, URIRef(NS+"SoilUnits")))
    g.add((context, RDFS.label, Literal(label) ))
```

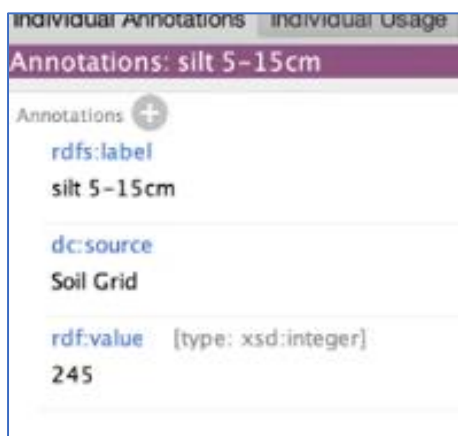
Attaching all uncertainty and mean values to the measurement:

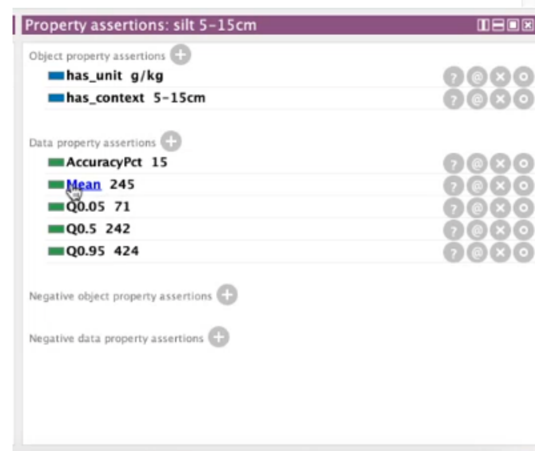
```
g.add((uri_ip, URIRef(NS+"Q0.05"), Literal(q5) ))
g.add((uri_ip, URIRef(NS+"Q0.5"), Literal(q50) ))
g.add((uri_ip, URIRef(NS+"Q0.95"), Literal(q95) ))
g.add((uri_ip, URIRef(NS+"Mean"), Literal(mean) ))
g.add((uri_ip, URIRef(NS+"AccuracyPct"), Literal(uncertainty) ))

g.add((uri_ip, RDF.value, Literal(mean) ))
```

Every Measurement or Prediction should have a value and a unit, so it applies for a query on both a prediction or a measurement. It is important to have a value for each measurement.

The statement RDF.value, Literal (mean) translates like hereunder in the ontology for Silt measurement:

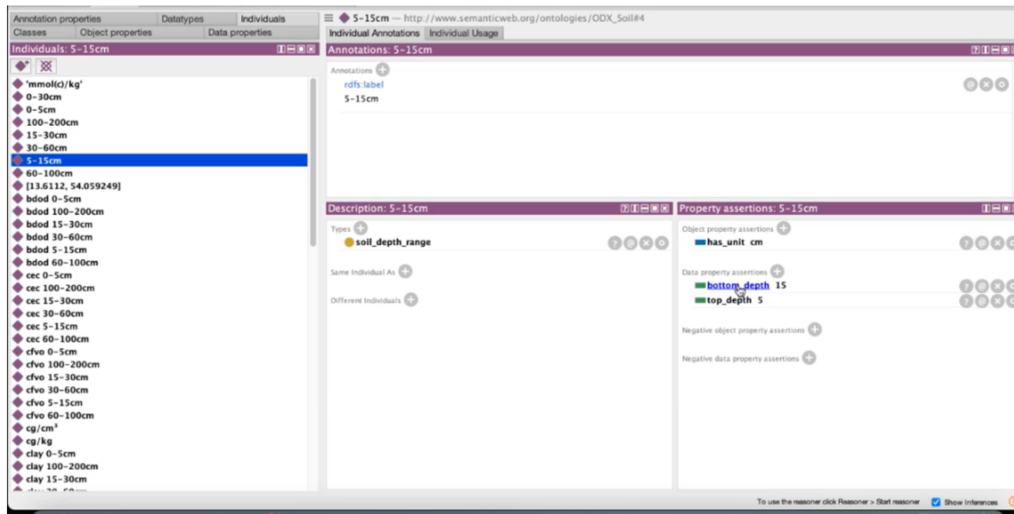






The '**mean**' value '245' appears in both annotations and property assertion, but this is just to attach the real measurement to the property.

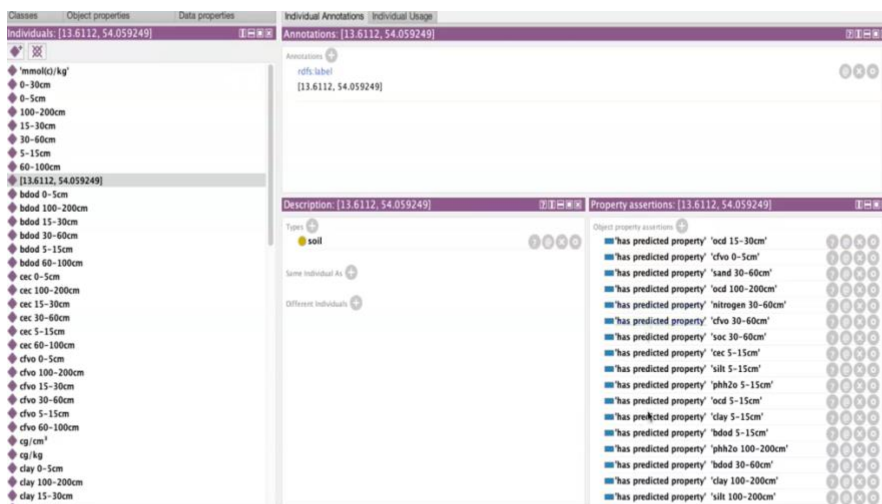
Instance 'Soil Depth', '5-15cm':



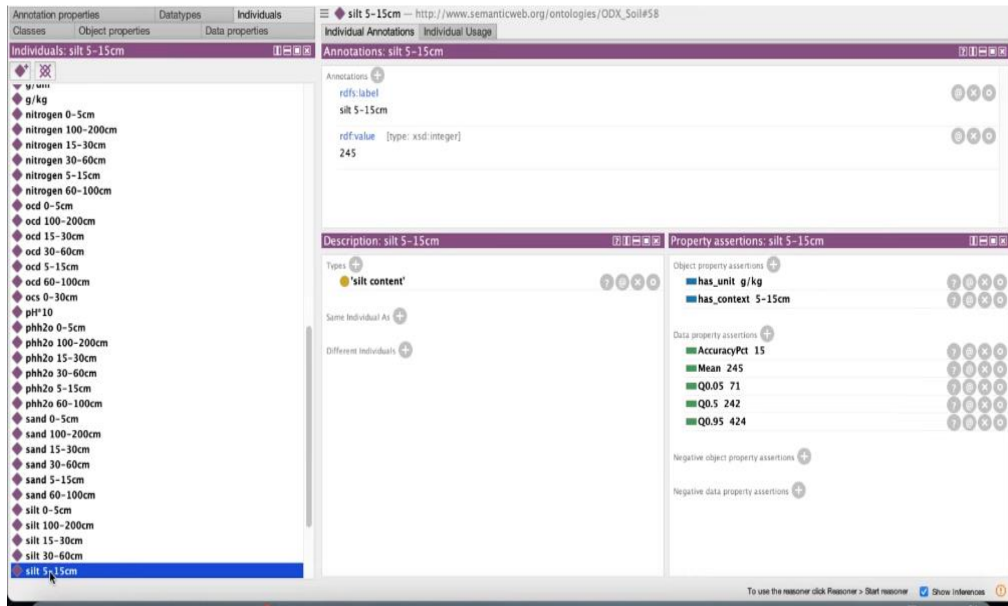
Visualization of the results in the ontology

the script created all the instances in tab **'Individuals'**

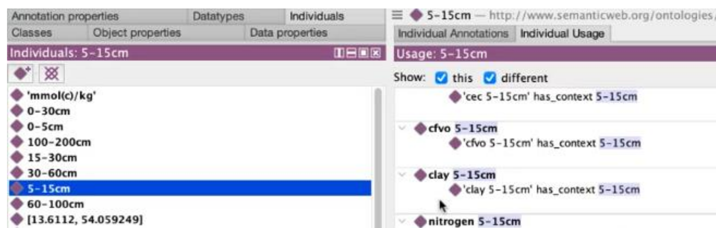
The given Soil is represented by the geo coordinates as label and for this soil, we can see all the measurement obtained from the data in the right lower window.



When selecting the predicted property 'Silt 5-15cm,' we can see the properties in the right window:



All measurements having a 5-15 cm are visible in the window:



Object, annotation, data properties are all linked to the same URI in the file. You can add as many attributes as possible.

The instance of unit was created as we did not have any instance for unit in the ontology, no label, so no information about the unit. But to do transformation from one unit to another one, you need to model the unit and add information to it and link it to the measurement. So units were created as instances.

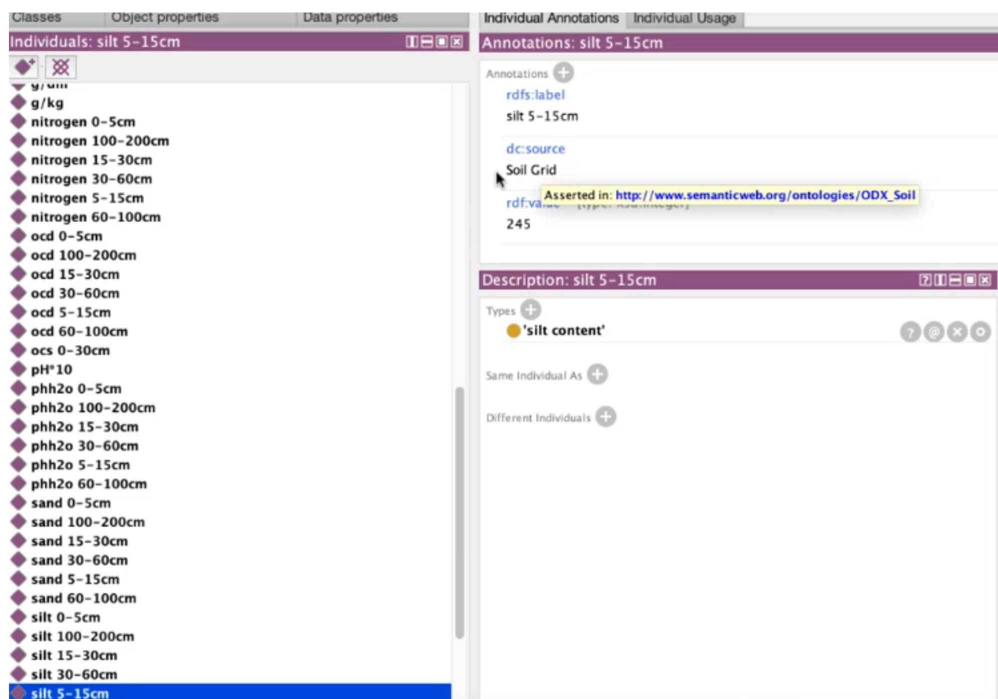
Adding a 'DC:source'

```
uri_ip = URIRef(NS+str(cpt))
cpt += 1
## create triples
g.add((uri_ip, RDF.type, URIRef(NS+measurements[name])))
g.add((uri_ip, RDFS.label, Literal(name+" "+label)))
g.add((URIRef(NS+unit), RDF.type, URIRef(NS+"SoilUnits")))
g.add((uri_ip, URIRef(NS+"has_unit"), URIRef(NS+unit)))
g.add((uri_ip, DC.source, Literal("Soil Grid")))
```

```
g.add((uri_ip, DC.source, Literal("Soil Grid")))
```

The DC.source of every URI of the properties will be e.g. 'Soil Grid'

Run the script and check in the ontology the creation of the source 'Soil Grid'



Note: To add more metadata from Dublin Core, we can add all as shown above or create a new URI for the source and the URI will be linked to another URI that has all metadata needed, instead of making them data properties.

Give a 'type' to the measurement.

Then all knowledge is included in the ontology.

Creating a new ttl file after running the script.

In annotation properties are all the Dublin Core classes

```
g.add((uri_ip, DC.source, Literal("Soil Grid")))
```

Every URI of the measurement properties will have a given source. example in ontology 'Individuals', Silt 0-5 cm has a “DC source:SoilGrid”

Create the RDF triples from your data and check if the ontology is consistent. Load sample data in RDF to check the consistency of the ontology which is a recommended step.

```
1377        ODX_Soil:Mean 72 ;
```

```
1382        ODX_Soil:has_un
1383        rdf:value 72 .
```

For a given soil, you can have both a prediction and a measured value. In the field, you measure the clay content so you will only have the unit and the real measured value. You won’t have these properties which are for predictions:

```
ODX_Soil:AccuracyPct 43 ;
ODX_Soil:Mean 99 ;
ODX_Soil:Q0.05 6 ;
ODX_Soil:Q0.5 65 ;
ODX_Soil:Q0.95 284 ;
```

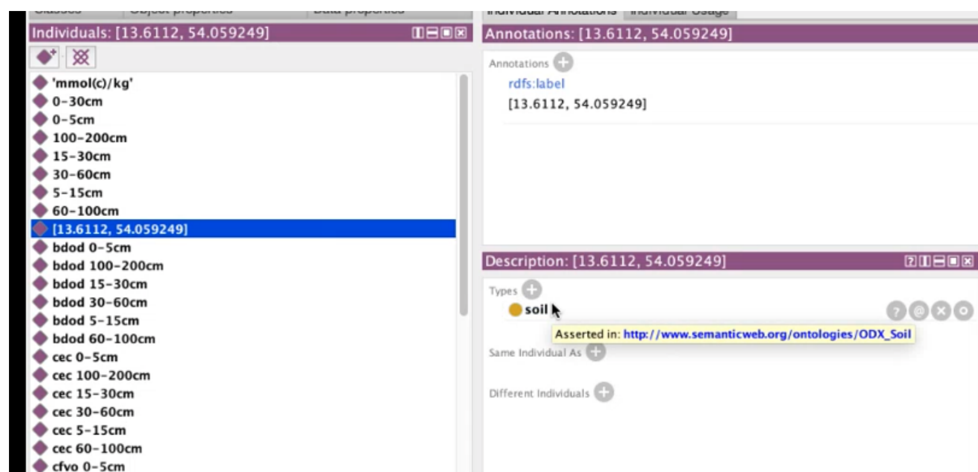
If you search "*give me all 'Clay content' in this location*", then if you have predictions and measurements, without this duplicated property, you will have to search twice, using separately the Mean value and RDF value. With this duplication, you can get all predictions and measurements in one query.

Property **'has context'** (line 1393): it means there is relationship between these nodes and why are they linked. For example, the link between node 25 and node 2 exists. Without the label 'clay 0-5 cm' (line 1386), you would not know at which depth was the measurements done.

A query can be run about getting all measurements made between 0-5 cm in a given location

Example when you get predictive and a measured values in same location. The predictive value has a much higher GPS resolution than a physical measurement - how to get rid of this difference?

In the ontology the soil is currently represented by the geocoordinates:



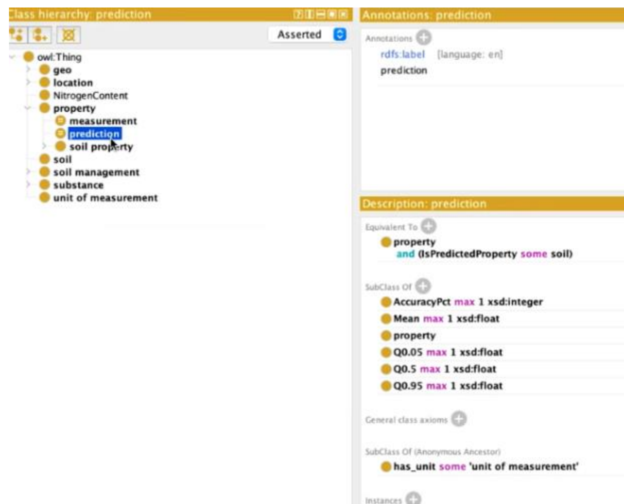
It could have been the 'Location' instead of the 'soil' but if you want to link the management practices, it may be better to keep the soil. In the window the decimals are 4 & 6 which comes from a prediction and any GPS would have between 2 and 4 decimals, so for a given soil location you would have 2 different values.

Display of the RDF file in the graph

When loaded in NEO4J using the plugin neo-semantics, classes and instances are created as nodes, object properties as relationships, datatype properties and annotation properties are created as attributes of a node.

Sorting out Measurements and Predictions with a reasoner

Last time we created two classes: *'Measurement'* and *'Prediction'* so you can run queries to get only predictions or only measurements.

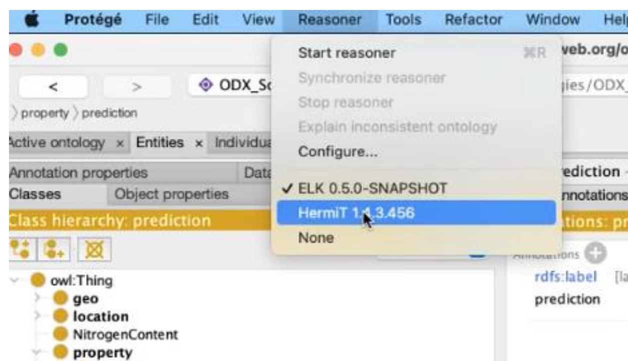


And we created this property:

This with the objective of using a **reasoner** that will create the instances for us.

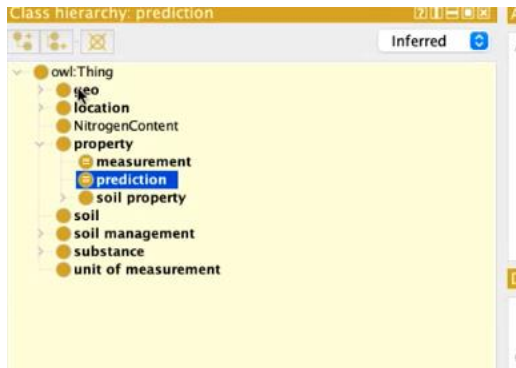
The reasoner will "understand" that the measurement are predictions because we have a prediction range for the property. It is not based on the direct data translation but about knowledge modeling:

We use 'HermiT' that is provided by Protégé:

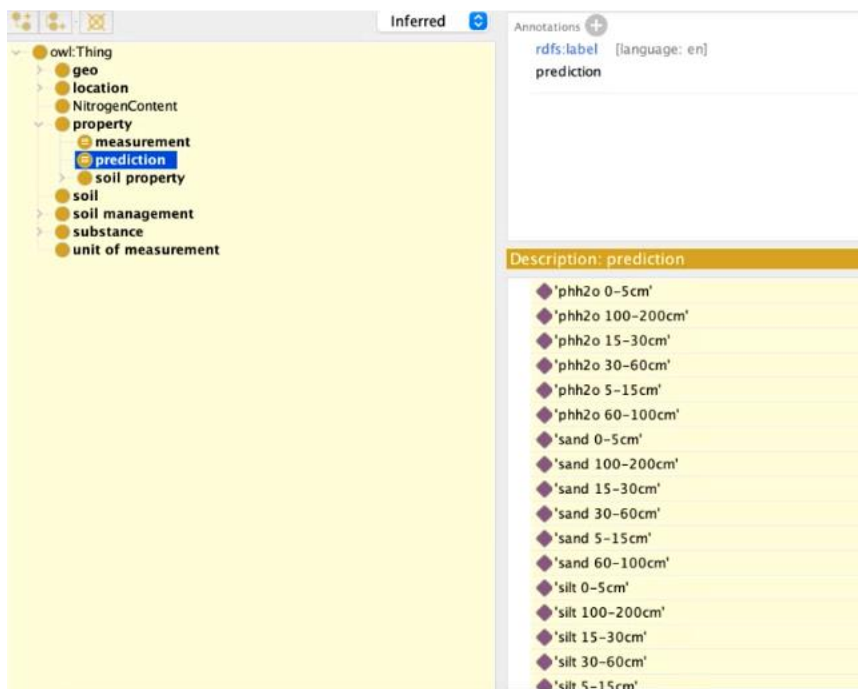


Select '*Start reasoner*'

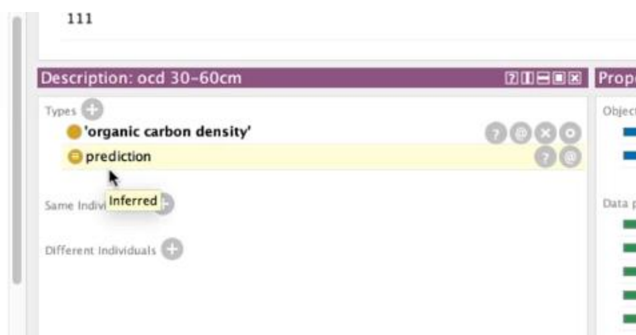
You select *'inferred'* to check that nothing appeared incorrect – would appear in red:



Result: instances are automatically classified under class 'Prediction'.



if we select 'organic carbon density' it is indicated that it is a 'prediction'.



The reasoner used the property 'has prediction' and classified automatically. It is convenient for querying the data and only getting predictive values or measured values. The reasoner interprets that all these values are prediction because the range of the property is *'Prediction'*

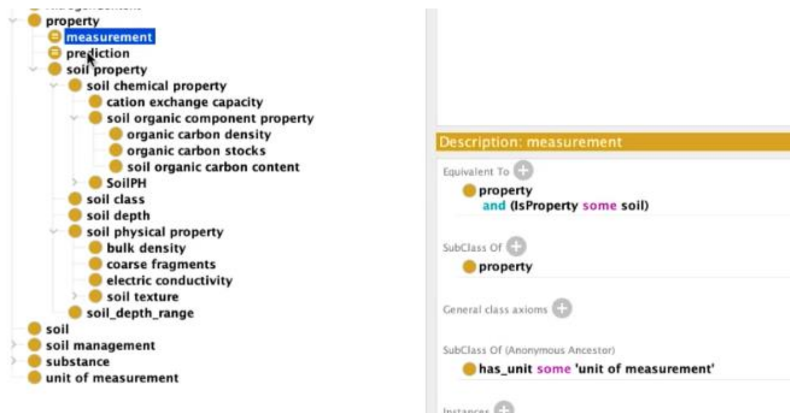
On the instance/class modeling aspect, if your use case calls on a 'smart' question or action (like a unit conversion), then you will need to model things as classes. For example, anything measured can have a unit of measurement, and if it is a weight measurement, you want the system to allow only weight units such as grams, kilograms, etc. as possible units for this measurement. Modeling unit as classes will allow you to do that. This type of knowledge that is added to the ontology can later help for some quality check on the data.

Good practice: for each created class for units, an instance should be created but it multiplies the number of triplets – Ontology punning can be used as an approach to reduce this phenomenon (see: https://www.w3.org/TR/owl2-new-features/#F12%3A_Punning)

One objective is to not load in the ontology what is not needed or required by the data & queries.

A scientific process can be the validation of predictions with measured values and could answer a question like: "Give me all measurements made that confirm or inform a prediction?" As soil measurements are linked to a location, then this type of query can be asked of the system.

Note: A unit ontology can directly be imported.



you need instances to infer. Using the reasoner helps confronting the reality of the data and the knowledge model.

Importing data into NEO4J after conversion into RDF

Importing in neo4j

- load data
  - CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS_AND_NODES'});
  - CREATE CONSTRAINT n10s_unique_uri ON (r:Resource)
  - ASSERT r.uri IS UNIQUE;

All instances and ontology were loaded in RDF and now it will be imported into NEO4J.

When you initialize the database, you need to give a graph configuration.

Use 'HandleRDFTTypes': it creates a NEO4J label and a node for each type. So it will enable to run query like in RDF with SPARQL. Having a representation of the types makes the graph bigger but easier to query.

All resources (instances, classes) must have a unique URI.

Two Plug-ins to install

1. APOC to download the data back into RDF. Useful for implementing versioning information, can clone a graph or nodes only.
2. NEO4Jsemantics plug-in, to load RDF directly.

Creation of the NOE4J Database

Create the database and configure the graph

```
## load data
CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS_AND_NODES'});

CREATE CONSTRAINT n10s_unique_uri ON (r:Resource)
ASSERT r.uri IS UNIQUE;

CALL n10s.rdf.import.fetch("file:///Users/marie-angeliquelaporte/Documents/YARA/merged_soil_crop.ttl","Turtle");

CYPHER queries
## get all the values of cec with unit
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits) return n.rdfs__label, n.rdf__value, u.uri

## get all the values of cec with unit, when soil depth = 20 or soil depth range contains 20
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20))
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label

## get all the values of cec with unit, at a specific location (e.g. bavaria)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
match (s:ns1__Soil)-[:ns3__locatedIn]->(somewhere)
match (s)-[:ns1__HasPredictedProperty|:ns1__HasProperty]->(n)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20)) and (somewhere.uri
contains "Bayern")
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label

## get all the values of cec with unit, in germany
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
match (s:ns1__Soil)-[:ns3__locatedIn]->(somewhere)
match (s)-[:ns1__HasPredictedProperty|:ns1__HasProperty]->(n)
match (somewhere)-[:rdfs__subClassOf*]->(germany)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20)) and (germany.uri
contains "Germany")
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label

## all the measurement
match (location)-[:ns3__locatedIn]->(somewhere)
match (location)-[:ns1__HasPredictedProperty|:ns1__HasProperty|:ns0__has_quality]->(n)
optional match (n)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n)-[:rdf__type]->(type)
```

```
## load data
CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS_AND_NODES'});

CREATE CONSTRAINT n10s_unique_uri ON (r:Resource)
ASSERT r.uri IS UNIQUE;
```

The Graph is configured.

As we have classes and instances: use the RDF import to import all triples we have in the ontology:

```
1 CALL n10s.rdf.import.fetch("file:///Users/marie-angeliquelaporte/Documents/YARA/merged_soil_crop.ttl","Turtle");
2
```

```
neo4j$ CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS AND NODES'});
```

```
neo4j$ CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS_AND_NODES'});
```

	param	value
1	"handleVocabUris"	"SHORTEN"
2	"handleMultival"	"OVERWRITE"
3	"handleRDFTypes"	"LABELS_AND_NODES"
4	"keepLangTag"	false
5	"multivalPropList"	null
6	"keepCustomDataTypes"	false

Started streaming 15 records after 1 ms and completed after 9 ms.

Run the import:

Results: 10,850 triples imported

```
neo4j$ CALL n10s.rdf.import.fetch("file:///Users/marie-angeliquelaporte/Documents/YARA/merged_soil_crop.ttl","T…
```

	terminationStatus	triplesLoaded	triplesParsed	namespaces	extraInfo	callParams
1	"OK"	10850	10850	(see below)	""	{ }


```
{
  "dct": "http://purl.org/dc/terms/",
  "owl": "http://www.w3.org/2002/07/owl#",
  "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
  "skos": "http://www.w3.org/2004/02/skos/core#",
  "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
  "ns0":
"http://www.semanticweb.org/ontologies/ODX_Crop#",
  "ns2":
"http://www.semanticweb.org/ontologies/ODX_Location#",
  "ns1":
"http://www.semanticweb.org/ontologies/ODX_Soil#",
  "dc": "http://purl.org/dc/elements/1.1/",
  "ns3":
"http://www.semanticweb.org/ontologies/ODX_Unit/"
```


Started streaming 1 records in less than 1 ms and completed after 489 ms.

```
neo4j$ CALL n10s.graphconfig.init({handleRDFTypes: 'LABELS_AND_NODES'});
```

param	value

It is better to see the namespaces on the right.

Use of Cypher language with example of queries based on nodes and relationships.

```
CYPHER queries
## get all the values of cec with unit
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits) return n.rdfs__label, n.rdf__value, u.uri

## get all the values of cec with unit, when soil depth = 20 or soil depth range contains 20
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20))
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label

## get all the values of cec with unit, at a specific location (e.g. bavaria)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
match (s:ns1__Soil)-[:ns3__locatedIn]->(somewhere)
match (s)-[:ns1__HasPredictedProperty|:ns1__HasProperty]->(n)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20)) and (somewhere.uri contains "Bayern")
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label
```

Examples of Queries

Query #1

Get the values of all the cation exchange capacity with a soil unit and return the rdfs labels:

```
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]→(u:ns2__SoilUnits) return n.rdfs__label, n.rdf__value, u.uri
```

Result:

n.rdfs__label	n.rdf__value	u.uri
"cec 20 cm"	18.4	"http://www.semanticweb.org/ontologies/ODX_Unit/meq/100g"
"cec 20 cm"	19.3	"http://www.semanticweb.org/ontologies/ODX_Unit/meq/100g"
"cec 20 cm"	17.64	"http://www.semanticweb.org/ontologies/ODX_Unit/meq/100g"
"cec 60-100cm"	221	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"
"cec 100-200cm"	207	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"
"cec 15-30cm"	209	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"

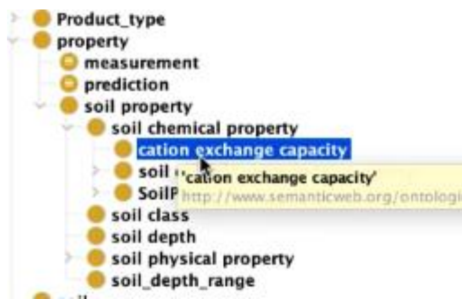
streaming 87 records after 1 ms and completed after 3 ms.

Query #2

Get Value of all ‘cation exchange capacity’ with a ‘Soil unit’ of 3 and a context 'soil depth range’ with a upper value 20 or bottom <20 '

```
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]→(u:ns3__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]→(sd)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20))
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label
```

Namespaces come from the ontology:



At the import, NEO4J maps the ontologies namespace to a ‘nx’

```
@prefix ODX_Geo: <http://www.semanticweb.org/ontologies/ODX_Geo#> .
@prefix ODX_Location: <http://www.semanticweb.org/ontologies/ODX_Location#> .
@prefix ODX_Soil: <http://www.semanticweb.org/ontologies/ODX_Soil#> .
@prefix ODX_Crop: <http://www.semanticweb.org/ontologies/ODX_Crop#> .
```

```
"http://www.semanticweb.org/ontologies/ODX_Crop#",
  "ns2":
"http://www.semanticweb.org/ontologies/ODX_Location#",
  "ns1":
"http://www.semanticweb.org/ontologies/ODX_Soil#",
  "dc": "http://purl.org/dc/elements/1.1/",
  "ns3":
```

- ‘ns1’ is the Soil Ontology and ‘cation exchange’ is a class attached to the Soil Ontology namespace.
- namespace: (n:ns1_CationExchangeCapacity) :
- 'ns1_CationExchangeCapacity' is the label of the node in the ontology. NEO4J does the map of the label to ns1

Result of the query:

n.rdfs__label	n.rdf__value	u.uri	sd.uri	sd.rdfs__label
"cec 15-30cm"	274	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"	"15-30cm"
"cec 15-30cm"	153	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"	"15-30cm"
"cec 15-30cm"	180	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"	"15-30cm"

Query #3

Get values of ‘Cation Exchange Capacity’, measurement with unit or prediction, in 'Germany' which is a sub class of 'somewhere'

SoilGrid Data are tagged with a region with using the Geonames service.

```
## get all the values of cec with unit, in germany
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]->(u:ns2__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]->(sd)
match (s:ns1__Soil)-[:ns3__locatedIn]->(somewhere)
match (s)-[:ns1__HasPredictedProperty|:ns1__HasProperty]->(n)
match (somewhere)-[:rdfs__subClassOf*]->(germany)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and sd.ns1__bottom_depth>20)) and (germany.uri
contains "Germany")
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label
```

```
match (n:ns1__CationExchangeCapacity)-[:ns1__has_unit]→(u:ns3__SoilUnits)
match (n:ns1__CationExchangeCapacity)-[:ns1__has_context]→(sd)
match (s:ns1__Soil)-[:ns3__locatedIn]→(somewhere)
match (s)-[:ns1__HasPredictedProperty|:ns1__HasProperty]→(n)
match (somewhere)-[:rdfs__subClassOf*]→(germany)
where (sd:ns1__soil_depth_range or sd:ns1__SoilDepth) and (sd.rdf__value=20 or (sd.ns1__top_depth<20 and
sd.ns1__bottom_depth>20)) and (germany.uri contains "Germany")
return n.rdfs__label, n.rdf__value, u.uri, sd.uri, sd.rdfs__label
```

Result of the query:

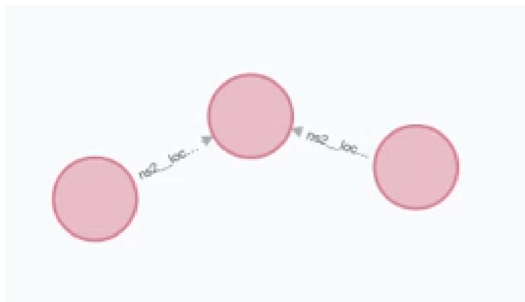
n.rdfs__label	n.rdf__value	u.uri	sd.uri
"cec 15-30cm"	274	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"
"cec 15-30cm"	153	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"
"cec 15-30cm"	180	"http://www.semanticweb.org/ontologies/ODX_Unit/mmol(c)/kg"	"http://www.semanticweb.org/ontologies/ODX_Soil#odx_6"

In this graph 'Somewhere' is the location and not the geocoordinates which are connected to Soil.

```
match (thing)-[:ns2__locatedIn]→(somewhere)
match (c:ns0__Crop)-[:ns0__has_quality]→(n)
match (p:ns0__Product_type)-[:ns0__has_quality]→(n)
match (somewhere)-[:rdfs__subClassOf*]→(country)
where (country.uri contains "Germany") and (location:ns0__Product_type or location:ns0__Crop)
return c, p, somewhere
```

- A Thing, which is a Crop, is located somewhere
- A Crop that has a quality
- Somewhere has to be in Germany
- A Product is located somewhere

The graph shows the crop and the product linked to same location



send the region for a given location.

Query #4

Crop and a Product located somewhere in Germany

```
match (p:ns0__Product_type)-[:ns2__locatedIn]→(somewhere)
match (c:ns0__Crop)-[:ns2__locatedIn]→(somewhere)
match (somewhere)-[:rdfs__subClassOf*]→(country)
where (country.uri contains "Germany")
return c, p, somewhere
```