Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

University of Padua

Department of Information Engineering

Master Degree in
Computer Engineering

Design and Development of an Ontology for Amyotrophic Lateral Sclerosis

Supervisor:
Prof. Nicola Ferro

Master Candidate:
Nicola Marchetti
1219363

Discussant:
Prof. Gianmaria Silvello

Academic Year 2021/2022
April 4, 2022

Abstract

Amyotrophic Lateral Sclerosis (ALS) is a degenerative disease that affects the nervous system, causing a progressive deterioration in the quality of life of affected patients. The European project Brainteaser leverages the value of Big Data, including health, lifestyle and environmental data, and Artificial Intelligence tools in order to deliver algorithms capable of predicting the progression of such disease. Since Brainteaser adopts an open-science approach and considering the trend in this field to use ontologies, i.e. models of formal representation of knowledge, the need to develop an ontology for Amyotrophic Lateral Sclerosis emerged. Based on this need, in this thesis we will present the design and development of an ontology to model clinical data for Amyotrophic Lateral Sclerosis. In addition, we will also present the development of a Data Mapper: a software that aims to map clinical data on Amyotrophic Lateral Sclerosis in an RDF (Resource Description Framework) dataset according to the ontology developed.

Sommario

La Sclerosi Laterale Amiotrofica (SLA) è una malattia degenerativa che colpisce il sistema nervoso, causando un deterioramento progressivo della qualità della vita dei pazienti colpiti. Il progetto europeo Brainteaser sfrutta il valore dei Big Data, compresi quelli relativi alla salute, alle abitudini di vita e all'ambiente, e gli strumenti dell'Intelligenza Artificiale al fine di fornire algoritmi in grado di prevedere la progressione di questa malattia. Dal momento che Brainteaser adotta un approccio open-science e considerata la tendenza in questo ambito di utilizzare le ontologie, ovvero modelli di rappresentazione formale della conoscenza, è emerso il bisogno di sviluppare un'ontologia per la Sclerosi Laterale Amiotrofica. Sulla base di questa necessità, in questa tesi presenteremo la progettazione e lo sviluppo di un'ontologia che si occupa di modellare i dati clinici della Sclerosi Laterale Amiotrofica. Inoltre, presenteremo anche lo sviluppo di un Data Mapper: un software che ha lo scopo di mappare i dati clinici sulla Sclerosi Laterale Amiotrofica in un dataset RDF (Resource Description Framework) in accordo all'ontologia sviluppata.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

ALS	Amyotrophic Lateral Sclerosis.
BO	Brainteaser Ontology.
CSV	Comma Separated Values.
EOSC	European Open Science Cloud.
HTTP	HyperText Transfer Protocol.
IRI	International Resource Identifier.
JSON	JavaScript Object Notation.
LOD	Linked Open Data.
OWL	Web Ontology Language.
RDF	Resource Description Framework.
RDFS	Resource Description Framework Schema.
TSV	Tab Separated Values.
URI	Uniform Resource Identifier.
W3C	World Wide Web Consortium.
XML	Extensible Markup Language.
XSD	XML Schema Definition.

Chapter 1

Introduction

1.1 The Brainteaser project

Brainteaser is a data science project that seeks to exploit the value of Big Data, including those related to health, lifestyle habits, and environment, to support patients with Amyotrophic Lateral Sclerosis and Multiple Sclerosis, their family and clinicians[1]. Amyotrophic Lateral Sclerosis (ALS) and Multiple Sclerosis (MS) are two chronic diseases characterized by a progressive or alternating impairment of neurological functions (motor, sensory, visual, cognitive). The common feature is that both these chronic diseases affect the nervous system and progressively modify the quality of life of patients and their families in a significant way. The Brainteaser project will integrate large clinical datasets about ALS and MS with new personal and environmental data collected using low-cost sensors and applications. The collected data will allow the development of Artificial Intelligence (AI) tools able to address the current needs of precision medicine, enabling early risk prediction of disease fast progression and adverse events. The main objectives of Brainteaser can be summarized as follows:

- Help promote predictive health care approaches and support clinicians, so patients with ALS and MS live healthier, more fulfilling lives;
- Leverage Artificial Intelligence to enhance clinical care and help design personalized health and care pathways;
- Adopt an open science paradigm that makes the results of scientific research accessible to all levels of society, at the same time respecting the privacy and ownership of patient data.

[1] `https://brainteaser.health`

Since the Brainteaser project adopts an open-science approach, it requires, among other things, the development of an ontology to represent all its knowledge in a standardized way. But what precisely is an ontology?
An ontology is a formal description of knowledge as a set of concepts within a domain and the relationships that hold between them[2]. Since in literature and in reality there are no ontologies that model in a unified way both diseases, ALS and MS, the request of the Brainteaser project is precisely to design and develop a new ontology, called Brainteaser Ontology (see Section 1.2), which meets this requirement.

1.2 The Brainteaser Ontology

The Brainteaser Ontology (BO) has the purpose to jointly model both Amyotrophic Lateral Sclerosis (ALS) and Multiple Sclerosis (MS) data. Specifically, the BO will serve multiple purposes:

- to provide a unified and comprehensive conceptual view of ALS and MS, which are typically treated separately, allowing us to coherently integrate data coming from the different medical partners in the project;
- to seamlessly represent both retrospective and prospective data, produced during the lifetime of Brainteaser;
- to enable sharing and reuse of Brainteaser datasets according to Open Science and FAIR principles.

BO is an innovative ontology based on a few basic concepts: Patient, Clinical Trial, Disease and Event. These concepts allow us to jointly model ALS and MS and capture the temporal dimension involved in the progression of these diseases. Indeed, the core idea is that a Patient participates in a Clinical Trial, suffers from some Disease, and experiences Events. These Events are different in nature and cover a wide range of cases, such as onset, symptom, trauma, diagnostic procedure (such as evoked potentials or ALSFRS-R tests), therapeutic procedure (such as mechanical ventilation for ALS or disease-modifying therapy for MS), relapse, and more. Overall, this event-based approach allows us to model ALS and MS in a unified way, sharing concepts between these two diseases, and to track what happens during their progression.

[2] `https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/`

The BO plays an important role in the overall architecture of Brainteaser, shown in Figure 1.1. In detail, it will inform the implementation of the Brainteaser Semantic Data Cloud, since the data contained here will be represented according to the BO, i.e., it will be an instance of the BO. As can be seen in the Figure 1.1, all data will be anonymized before being represented in the BO. This holds for both the retrospective data, i.e., data already held by the clinical partners (on the right side of the figure), and the prospective data, i.e., new data that will be collected during the life of the project (on the left side). The data contained in the Brainteaser Semantic Data Cloud, will also be exported in a suitable format (e.g. CSV) and will then be used to train the AI models needed to predict the progression of ALS and MS. Finally, a subset of the data in the Brainteaser Semantic Data Cloud will be exported to the European Open Science Cloud (EOSC) and will also be shared and exploited for Open Evaluation Challenges.

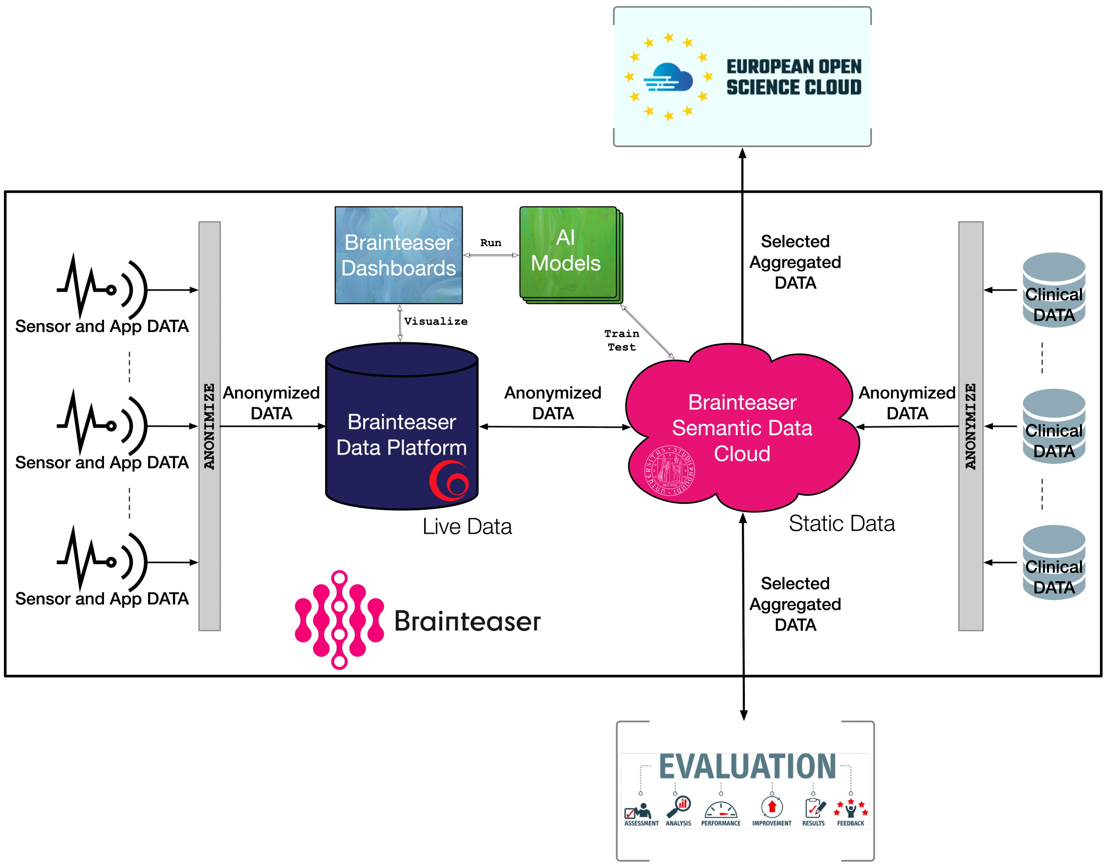


Figure 1.1: The Brainteaser Ontology and its role in the overall Brainteaser architecture.

The Figure 1.2 shows an overview of the entire data flow associated with the BO and the Brainteaser Semantic Data Cloud, also in relation to the Open Evaluation Challenges, the EOSC, and the Linked Open Data Cloud. From this image it can be seen that the Brainteaser Ontology will model the following data sources:

- *"Raw" anonymized data*: these correspond to the retrospective and prospective data anticipated above and is divided as follows:
    - Clinical data;
    - Sensor/App data;
- *Generated data*: once trained on the above data, the AI models can be serialized and become part of the modeled data.
    - Serialized AI models;
- *Evaluation Challenges data*: these are based on three types of data that become part of the modeled data:
    - Evaluation Corpora: these are the training/validation/test sets that are obtained by selecting an appropriate subset of the "raw" data;
    - AI models outputs: these are the results produced by the participating systems;
    - Performance scores and statistical analyses: the results produced by participants will be scored and then statistical analyses will be computed to assess them.

To conclude, the BO will also allow all this data to be linked with the resources already available in the Linked Open Data Cloud.

1.3 Scope and organization of the thesis

The purpose of this thesis is to design and develop the part of the Brainteaser Ontology that deals with the management of Amyotrophic Lateral Sclerosis (ALS) retrospective data. Subsequent to this, the other goal is to develop an ALS Data Mapper that allows the mapping of "raw" retrospective data, coming from the ALS partner clinics, into a structured RDF (Resource Description Framework) graph that is compliant with the Brainteaser Ontology. In the following chapters we will present the various steps that were required to achieve the two established goals, in detail:

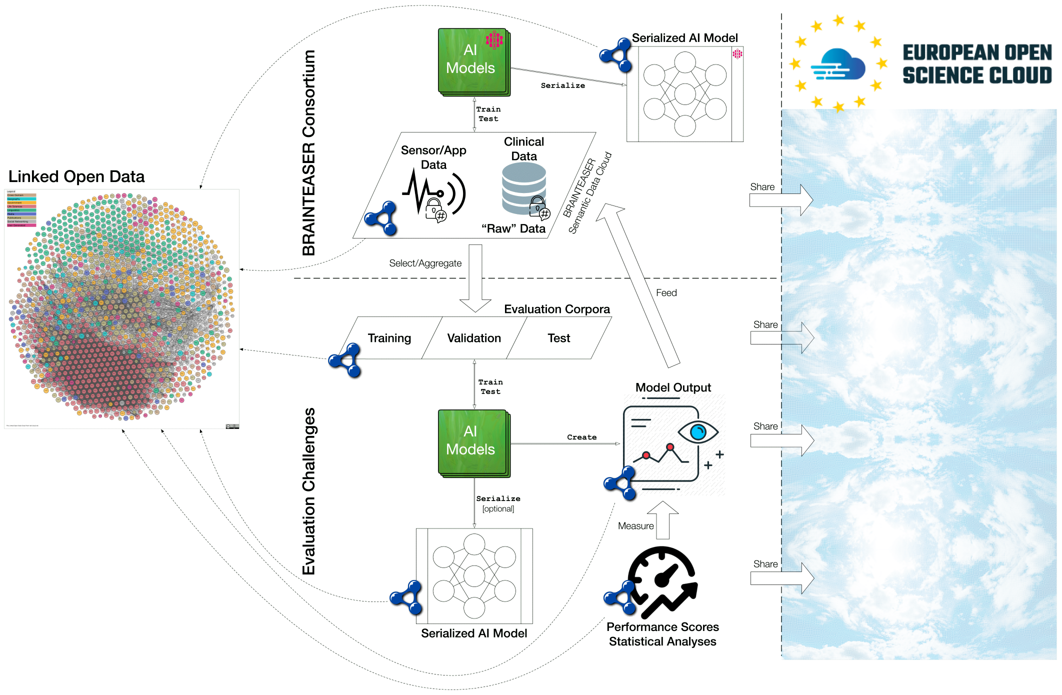


Figure 1.2: The Brainteaser Ontology and the overall data flow.

Chapter 2: State of the Art. In this chapter we present an introduction on Semantic Web and Linked Data and a background on its technologies, namely RDF, OWL, SPARQL. In addition, we briefly introduced the Protégé and GraphDB software, which are respectively two useful tools for ontology development and RDF dataset management. The combination of these tools formed the basis for the development of the BO and the respective ALS Data Mapper.

Chapter 3: Requirements analysis. This chapter briefly summarizes the ontology design approach adopted, which is based on the involvement of domain experts and partners in order to initially obtain requirements, and then to validate the design choices and definitions in the BO.

Chapter 4: Design of an Ontology for Amyotrophic Lateral Sclerosis. In this chapter we initially present the design principles of the BO and the external ontologies that have been imported inside it. After this, we described in detail the various areas that compose the BO, and the classes and properties contained in each of these.

Chapter 5 : Ontology and Data Mapper development. This chapter covers the development of the BO using the Protégé software and also describes the development of the ALS Data Mapper that aims to ingest the retrospective data received from the ALS clinical partners.

Chapter 6: Datasets statistics. In this chapter we briefly introduce statistics regarding the retrospective input datasets provided to us by the clinical partners. In addition to these, we will also present statistics on the RDF dataset obtained by running the ALS Data Mapper.

Chapter 7: Conclusions and Future Work. This chapter presents the overall conclusions of the thesis and the results achieved.

It is important to note that the details about the part of the BO focused on the management of Multiple Sclerosis (MS) retrospective data will never be covered in subsequent chapters.

Chapter 2

State of the Art

2.1 The Semantic Web and Linked Data

The Semantic Web is an evolution of the World Wide Web and is based on the standards established by the World Wide Web Consortium (W3C) [Berners-Lee et al., 2001, Heath and Bizer, 2011]. While the traditional Web is focused on multiple documents connected by links that have no semantic value, the goal of the Semantic Web is to create a Web of Data by attributing semantic value to both the data and the links that connect them, and to make these data and links machine-readable.
In 2006, during a conference on the Semantic Web, Tim Berners-Lee coined the term Linked Data to indicate a standard for publishing structured data using vocabularies that can be linked and interpreted by machines[1]. Berners-Lee described Linked Data as "the Semantic Web done right"[2] and further added "while the Semantic Web is the goal or the end of this process, Linked Data provides the means to achieve that goal" [Bizer et al., 2009].
Linked Data is based on a set of rules[3], known as the Linked Data Principles, for publishing data on the Web in a way that all published data becomes part of a single global data space:

1. Use URIs as names for things;

2. Use HTTP URIs so that people can look up those names;

[1]https://data.europa.eu/en/news/origin-linked-data
[2]https://www.w3.org/2008/Talks/0617-lod-tbl/
[3]https://www.w3.org/DesignIssues/LinkedData.html

3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL);

4. Include links to other URIs, so that they can discover more things [Bizer et al., 2009, Heath and Bizer, 2011].

The Uniform Resource Identifiers (URIs) and HyperText Transfer Protocol (HTTP) are two fundamental technologies for Linked Data. While URIs provide a means to identify any entity that exists in the world, the HTTP protocol provides a universal mechanism to retrieve resources and information about them. URI and HTTP are integrated by the Resource Description Framework (RDF) which is a fundamental technology for the Web of Data. RDF provides a graph-based data model with which to represent, describe, and link together data representing entities that exist in the world (Section 2.2).
Using URIs, the HTTP protocol, and the RDF data model, Linked Data builds directly on the general architecture of the Web. The Web of Data can therefore be seen as an extension of the traditional Web and has many of the same properties:

- The Web of Data can contain any type of data;

- Anyone can publish data on the Web of Data;

- Entities are connected by RDF links, creating a global data graph that enables the discovery of new data sources [Bizer et al., 2009].

An example of the application of Linked Data Principles was the Linked Open Data (LOD) project, founded in January 2007 and supported by the W3C. The LOD is simply a cloud that hosts a collection of datasets, published following the Linked Data Principles.
The initial goal of the project was to launch the Web of Data by identifying the first datasets available under open licenses, converting them to RDF, and publishing them on the Web. Since its beginning, thanks to its open nature, the project has grown significantly to involve large organizations in publishing their datasets and linking them to existing datasets. The nature of the content in the cloud is different, in fact the datasets hosted within it include for example data about people, organizations, places, books, movies, music, drugs, reviews and so on.
The size of the LOD project is presented in Figure 2.1, where each node in the figure corresponds to a distinct dataset published as Linked Data. An example of

an open RDF dataset contained in the cloud is DBpedia, which is a set of RDF triples extracted from Wikipedia info-boxes [Auer et al., 2007]. As can be seen, DBpedia sits at the center of the cloud and, due to the fact that it provides URIs and RDF descriptions for many common entities, it acts as a hub to which other datasets are linked.

The LOD, in about 14 years, has recorded a remarkable growth, in fact in 2007 it hosted only 12 datasets while in the version of May 2021 has reached 1301 databases with 16283 links within it[4]. The growth of LOD is the real evidence that the development of the Web of Data is actually taking place.

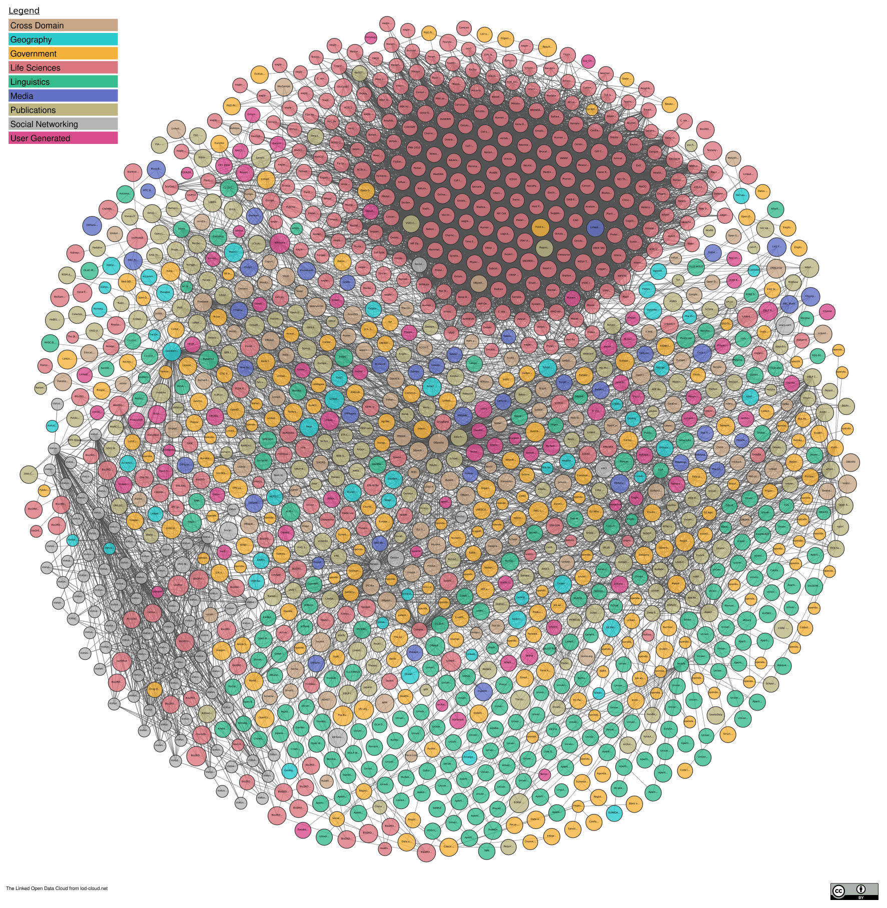


Figure 2.1: A May 2021 snapshot of the Linked Open Data Cloud from `https://lod-cloud.net`. The cloud began in 2007 with only 12 datasets and, in this version, has reached 1301 datasets within it.

[4]`https://lod-cloud.net/`

2.2 RDF

The Resource Description Framework (RDF) is a W3C standard model for representing information about resources on the Web. In recent years, RDF has become the de-facto standard for publishing and interlinking data on the Web. RDF allows us to make statements about resources[5]. The format of a statement has the following structure:

```
<subject> <predicate> <object>
```

The *subject* and *object* represent the two resources that are in a relationship, while the *predicate* represents the nature of their relationship. The relationship goes from the subject to the object and is called, according to RDF, a property. Due to the fact that RDF statements are composed of three elements they are called *triples.* The combination of triples generates a directed graph, called RDF graph or RDF datasets, where the subjects and objects of the triples correspond to the nodes of the graph, while the predicates form the arcs. Figure 2.2 shows an example of RDF graph that is derived from the following example triples[5]:

```
<Bob> <is a> <person>.
<Bob> <is a friend of> <Alice>.
<Bob> <is born on> <the 4th of July 1990>.
<Bob> <is interested in> <the Mona Lisa>.
<the Mona Lisa> <was created by> <Leonardo da Vinci>.
<the video 'La Joconde a Washington'> <is about> <the Mona Lisa>
```

In RDF, a resource is represented through an *International Resource Identifier (IRI)*, a *literal* value, or an *blank node.* An IRI identifies a resource, it can appear in all three positions of a triple, and, from a technical point of view, is a generalization of the URI that can also contain non-ASCII characters in the IRI character string. A literal is a string representation of a certain value which can be associated with a datatype or a language tag. Literals may only appear in the object position of a triple and the default value is string. Blank nodes, on the other hand, can be used to denote resources without explicitly naming them with an IRI and can appear in the subject and object position of a triple.
The RDF data model is typically used in combination with vocabularies, which are collections of classes and properties that attribute semantic information to resources. RDF provides an RDF data modeling vocabulary, called RDF Schema

[5]`https://www.w3.org/TR/rdf11-primer/`

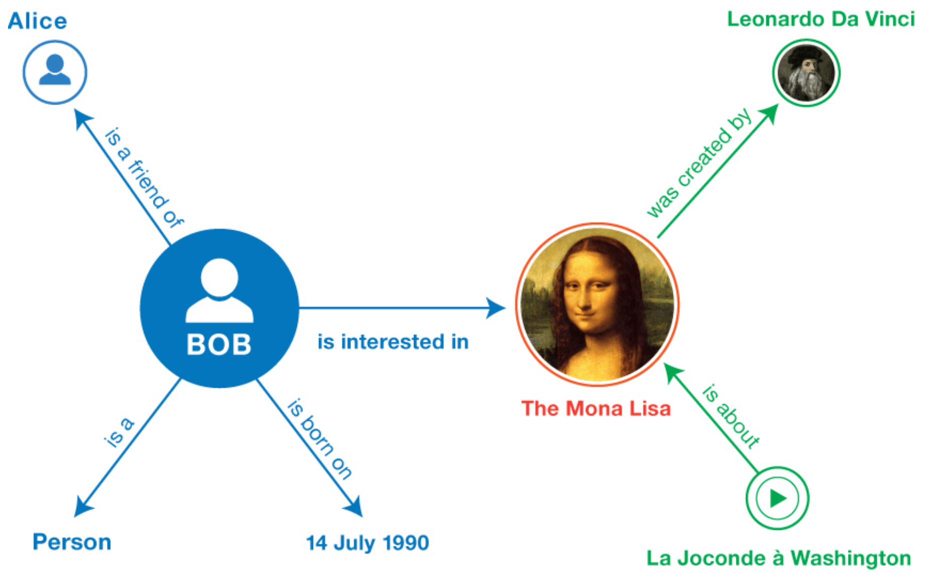


Figure 2.2: An example of RDF graph from `https://www.w3.org/TR/rdf11-primer/`.

(RDFS), which is a set of mechanisms for defining the semantic characteristics of RDF data. RDF Schema uses the notion of *class* to specify the categories that can be used to classify resources and the *type* property to declare the relationship between an instance and its class. With RDF Schema, one can also create hierarchies of classes/subclasses and properties/subproperties through the *subClassOf* and *subPropertyOf* properties. While type restrictions on subjects and objects of particular triples can be defined through the *domain* and *range* restrictions properties.

For even more comprehensive semantic modeling of RDF data, RDF Schema is many times supported by the use of the OWL vocabulary (Section 2.3), while the standard query language SPARQL (Section 2.4) is used to retrieve information within RDF graphs.

Furthermore, RDF recommends the use of the XSD (XML Schema Definition), which specifies how to formally describe elements in an XML document, and provides a set of 19 primitive data types (boolean, string, double, float, date, gYear, etc.) useful for declaring datatypes associated with literals.

To be published on the Web, an RDF graph must first be serialized using one or more RDF serialization formats. Below we present the most common serialization formats and for each of them we give a small example, based on a small part of the Figure 2.2. Specifically, in the examples we describe that Bob is a person and knows Alice. To do this we use the RDF vocabulary FOAF (Friend of a Friend) which is used to describe persons and the relationships between them. So, the most common RDF serialization formats are:

- **Turtle**: a plain text and compact serialization format:

  ```
  @prefix foaf: <http://xmlns.com/foaf/0.1/> .

  <http://example.org/bob#me> a foaf:Person ;
      foaf:knows <http://example.org/alice#me> .
  ```

- **JSON-LD**: a JSON-based serialization format:

  ```
  [
    {
      "@id": "http://example.org/bob#me",
      "@type": [
        "http://xmlns.com/foaf/0.1/Person"
      ],
      "http://xmlns.com/foaf/0.1/knows": [
        {
          "@id": "http://example.org/alice#me"
        }
      ]
    }
  ]
  ```

- **RDFa**: a serialization format that embeds RDF triples in HTML documents:

  ```
  <div xmlns="http://www.w3.org/1999/xhtml"
    prefix="
      foaf: http://xmlns.com/foaf/0.1/
      rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
      rdfs: http://www.w3.org/2000/01/rdf-schema#"
    >
    <div typeof="foaf:Person" about="http://example.org/bob#me">
      <div rel="foaf:knows" resource="http://example.org/alice#me">
      </div>
  ```

```
  </div>
</div>
```

- **RDF/XML**: an XML-based syntax for serializing RDF graphs:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:about="http://example.org/bob#me">
    <foaf:knows rdf:resource="http://example.org/alice#me"/>
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  </rdf:Description>
</rdf:RDF>
```

2.3 Ontologies and OWL

An ontology is a set of precise descriptive statements about a part of the world, usually referred to as the domain of interest or the object of the ontology[6]. It includes machine-interpretable definitions of the basic concepts in the domain and the relations among them. Some of the reasons why ontologies are developed are:

- To share common understanding of the structure of information among people or software agents;
- To enable reuse of domain knowledge;
- To make domain assumptions explicit;
- To separate domain knowledge from the operational knowledge;
- To analyze domain knowledge [Noy and Mcguinness, 2001].

The Web Ontology Language (OWL) is a language for expressing ontologies. The current version of OWL, called OWL 2, was developed by the W3C and published in a first version in 2009 and refined with a second version in 2012. OWL 2 is

[6]https://www.w3.org/TR/owl2-primer/

therefore an extension and revision of the original version of OWL that was published in 2004 by the W3C.
OWL 2 is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. It is a computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit[7]. OWL documents, known as ontologies, can be published in the World Wide Web, and may refer to or to be referred from other OWL ontologies[6]. In OWL 2, the statements that are made in an ontology are called *axioms*. The objects are denoted as *individuals*, categories as *classes*, and relationships as *properties*. Properties are further subdivided: object properties relate objects to objects (like a person to their spouse), datatype properties assign data values to objects (like an age to a person), while annotation properties are used to encode information about the ontology (like the author and creation date of an axiom). OWL extends the expressivity of RDFS with additional modeling primitives. For example, OWL defines the *equivalentClass* and *equivalentProperty* primitives, which provide powerful mechanisms for defining mappings between terms from different vocabularies, which, in turn, increase the interoperability of datasets modeled using different vocabularies. Another example of an OWL primitive that is very useful in the context of Web of Data is *inverseOf*, which allows one to declare that one property is the inverse of another, and thus strengthen the semantic value of statments [Heath and Bizer, 2011]. OWL, along with RDF and SPARQL, is part of the W3C's Semantic Web technology stack.

2.4 SPARQL

SPARQL is an RDF query language that is able to retrieve and manipulate data stored in the RDF format. In 2004, the RDF Data Access Working Group, a W3C group, released a first public working draft of the SPARQL query language. In January 2008, SPARQL became an official W3C recommendation and one of the key technologies of the Semantic Web. The most recent version, SPARQL 1.1, was released in 2013[8]. Since RDF Databases are directed labeled graphs, SPARQL is essentially a graph-matching query language. SPARQL queries are

[7]`https://www.w3.org/OWL/`
[8]`https://www.w3.org/TR/rdf-sparql-query/`

composed of three parts:

1. *Pattern matching*: includes features of pattern matching for graphs, such as patterns, optional parts, union of patterns, nesting, filtering values of possible matchings, and the possibility of choosing the data source to be matched by a pattern;

2. *Solution modifiers*: once the output of the pattern has been computed, allow to modify these values by applying operators such as projection, distinct, order, and limit;

3. *Output types*: a SPARQL query can return boolean values, selections of values of the variables which match the patterns, construction of new RDF graphs built from the matching values, and description of resources [Pérez et al., 2009].

The result set of a SPARQL query can be serialized using one of four common formats recognized by SPARQL, namely Extensible Markup Language (XML), JavaScript Object Notation (JSON), Comma Separated Values (CSV), and Tab Separated Values (TSV)[9].

2.5 Protégé

Protégé is a free, open source ontology editor with full support for the OWL 2 Web Ontology Language. It was designed by Mark Musen in 1987 and has since been developed by a team at Stanford University [Gennari et al., 2003]. The software is the most popular and widely used ontology editor in the world[10] [Musen, 2015].

Protégé supports creation and editing of one or more ontologies in a single workspace via a completely customizable user interface. Visualization tools allow for interactive navigation of ontology relationships. Advanced explanation support aids in tracking down inconsistencies. Refactor operations available including ontology merging, moving axioms between ontologies, rename of multiple entities, and more[11].

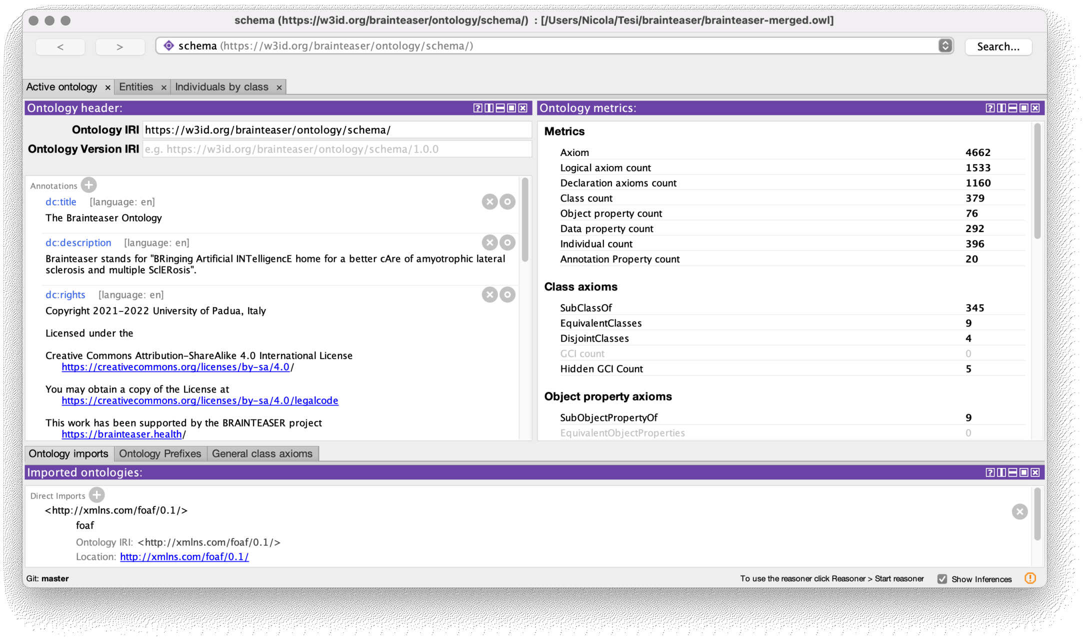
Figure 2.3 represents the main Protege workspace, which displays the "Active Ontology" tab by default. This tab shows an overview of the "active" ontology,

[9] `https://www.w3.org/TR/sparql11-overview/`

[10] `https://protege.stanford.edu/shortcourse/`

[11] `https://protege.stanford.edu/products.php`

including metrics about its content, annotations about the ontology, and the imported ontologies. The drop-down box on the toolbar shows the current active ontology, i.e. the one where all changes occur. At the top right of the toolbar is the Search button, which can be pressed to open the search window[12].



schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]

schema (https://w3id.org/brainteaser/ontology/schema/)

Ontology IRI https://w3id.org/brainteaser/ontology/schema/

Ontology Version IRI e.g. https://w3id.org/brainteaser/ontology/schema/1.0.0

dc:title [language: en]
The Brainteaser Ontology

dc:description [language: en]
Brainteaser stands for "BRinging Artificial INTelligencE home for a better cAre of amyotrophic lateral sclerosis and multiple SclERosis".

dc:rights [language: en]
Copyright 2021-2022 University of Padua, Italy

Licensed under the

Creative Commons Attribution-ShareAlike 4.0 International License
https://creativecommons.org/licenses/by-sa/4.0/

You may obtain a copy of the License at
https://creativecommons.org/licenses/by-sa/4.0/legalcode

This work has been supported by the BRAINTEASER project
https://brainteaser.health/

Metrics

Axiom	4662
Logical axiom count	1533
Declaration axioms count	1160
Class count	379
Object property count	76
Data property count	292
Individual count	396
Annotation Property count	20

Class axioms

SubClassOf	345
EquivalentClasses	9
DisjointClasses	4
GCI count	0
Hidden GCI Count	5

Object property axioms

SubObjectPropertyOf	9
EquivalentObjectProperties	0

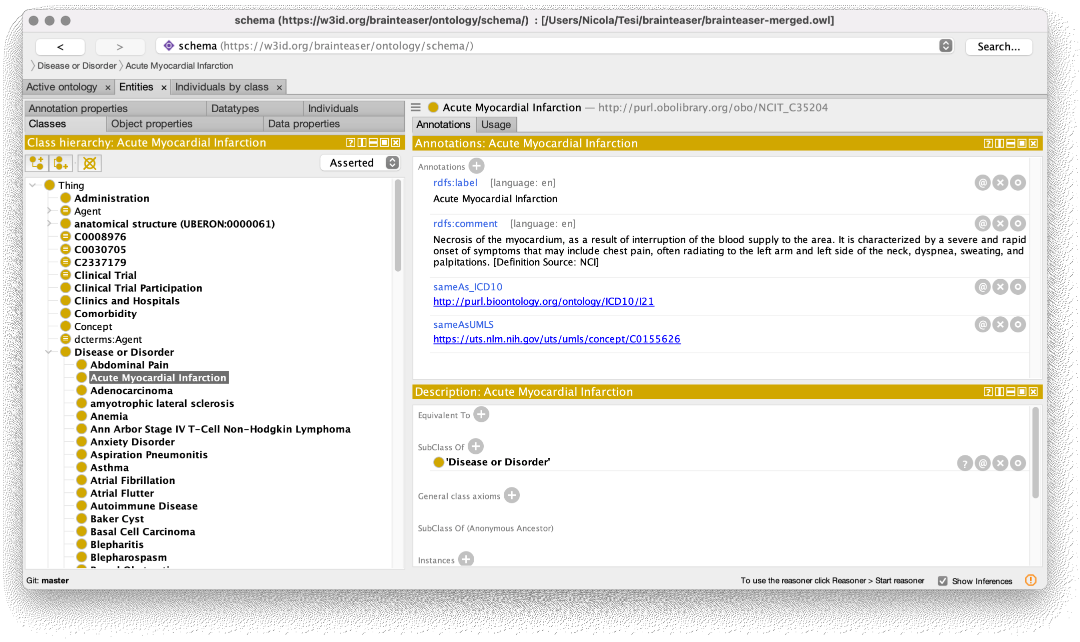
<http://xmlns.com/foaf/0.1/>
foaf
Ontology IRI: <http://xmlns.com/foaf/0.1/>
Location: http://xmlns.com/foaf/0.1/

Git: **master** To use the reasoner click Reasoner > Start reasoner ☑ Show Inferences

Figure 2.3: A snapshot of Protégé "Active Ontology" tab. This tab shows the information and metrics of the ontology and also lists the ontologies imported from outside, in this case the FOAF ontology.

While Figure 2.4 shows the "Entities" tab, which is the most important part of the ontology editor. From this tab, it is possible to explore all the classes, properties, and individuals of an ontology. Each tab consists of several views that can be resized, removed, floated, split, and layered in various ways. Most views implement hypertext navigation so that links can be followed easily regardless of which view is used. The left view displays the hierarchies of classes or properties and the buttons to add and remove a class/subclass or a property/subproperty. While the right view displays the description and characteristics of the class or property selected in the hierarchy tab. Navigating back and forth is possible with the left and right arrow buttons in the toolbar, which act just like a web browser. The "Search" button in the toolbar performs a global search in the loaded ontologies[11].

[12]http://protegeproject.github.io/protege/getting-started/



schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]
<
>
schema (https://w3id.org/brainteaser/ontology/schema/)
Search...
Disease or Disorder 〉 Acute Myocardial Infarction
Active ontology
Entities
Individuals by class
Annotation properties
Datatypes
Individuals
Classes
Object properties
Data properties
Class hierarchy: Acute Myocardial Infarction
Asserted
Annotations
Usage
Annotations: Acute Myocardial Infarction
Description: Acute Myocardial Infarction
Equivalent To
SubClass Of
General class axioms
SubClass Of (Anonymous Ancestor)
Instances
Git: master
To use the reasoner click Reasoner > Start reasoner
Show Inferences

Thing
- Administration
- Agent
- anatomical structure (UBERON:0000061)
- C0008976
- C0030705
- C2337179
- Clinical Trial
- Clinical Trial Participation
- Clinics and Hospitals
- Comorbidity
- Concept
- dcterms:Agent
- Disease or Disorder
  - Abdominal Pain
  - Acute Myocardial Infarction
  - Adenocarcinoma
  - amyotrophic lateral sclerosis
  - Anemia
  - Ann Arbor Stage IV T-Cell Non-Hodgkin Lymphoma
  - Anxiety Disorder
  - Aspiration Pneumonitis
  - Asthma
  - Atrial Fibrillation
  - Atrial Flutter
  - Autoimmune Disease
  - Baker Cyst
  - Basal Cell Carcinoma
  - Blepharitis
  - Blepharospasm

Acute Myocardial Infarction — http://purl.obolibrary.org/obo/NCIT_C35204

Annotations

rdfs:label [language: en]
Acute Myocardial Infarction

rdfs:comment [language: en]
Necrosis of the myocardium, as a result of interruption of the blood supply to the area. It is characterized by a severe and rapid onset of symptoms that may include chest pain, often radiating to the left arm and left side of the neck, dyspnea, sweating, and palpitations. [Definition Source: NCI]

sameAs_ICD10
http://purl.bioontology.org/ontology/ICD10/I21

sameAsUMLS
https://uts.nlm.nih.gov/uts/umls/concept/C0155626

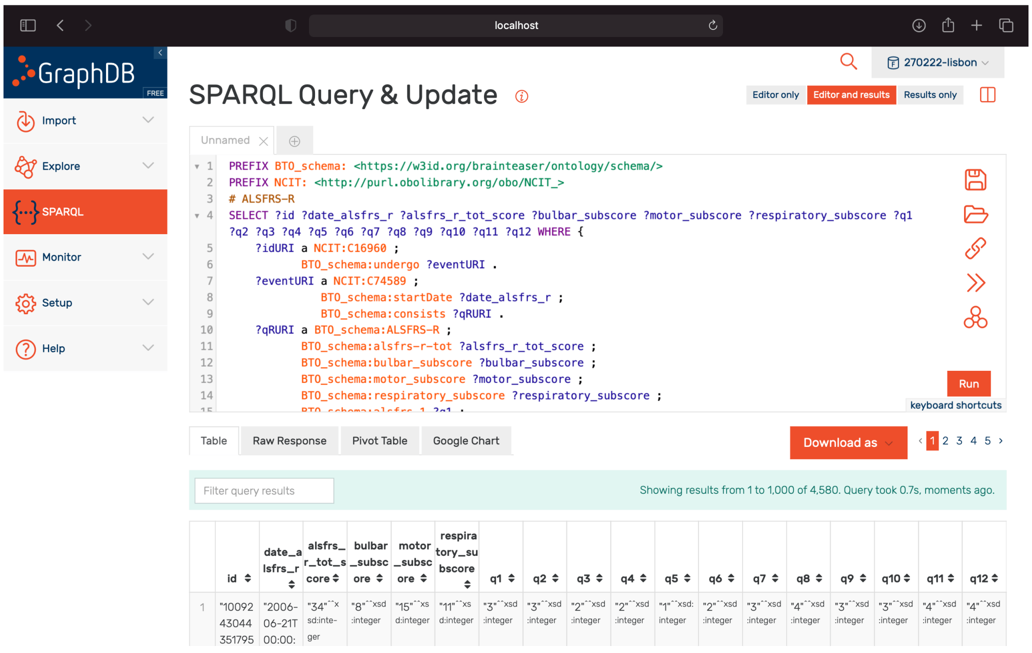
SubClass Of
'Disease or Disorder'

Figure 2.4: A snapshot of Protégé "Entities" tab. In the class hierarchy on the left side, the class "Acute Myocardial Infarction" is selected. The right side shows information about the selected class.

These images illustrate only the main tools of Protégé, but many other tools and functionalities are present within it. For more information see the official documentation at `http://protegeproject.github.io/protege/`.

2.6 GraphDB

Ontotext GraphDB is a highly efficient and robust graph database and knowledge discovery tool compliant with W3C Standards and with RDF and SPARQL support. GraphDB can perform instances and relationship exploration, handle real-time queries and inferences, and derive metrics and statistics about datasets. In this section we will focus only on two main functionalities: "SPARQL" and "Visual graph" tools. For more information on the other features of GraphDB see the official documentation at `https://graphdb.ontotext.com/documentation/free/`. Figure 2.5 shows the "SPARQL" tab, which is responsible for querying the imported RDF datasets with SPARQL queries. Within this tab there is an editor for writing the query. This is an editor that provides syntax highlighting and namespace autocompletion for easy reading and writing. In the right part of the editor there are also some icons that allow, for example, to save a query, to open one that has been previously saved or to exclude from results the data that has been inferred. The query is executed using the “Run” button and the results are automatically displayed below the editor. The results are displayed as a table by default (other options are Raw response, Pivot table and Google Charts) and can be exported, through the “Download as” button, in several formats such as JSON, XML, CSV, TSV.



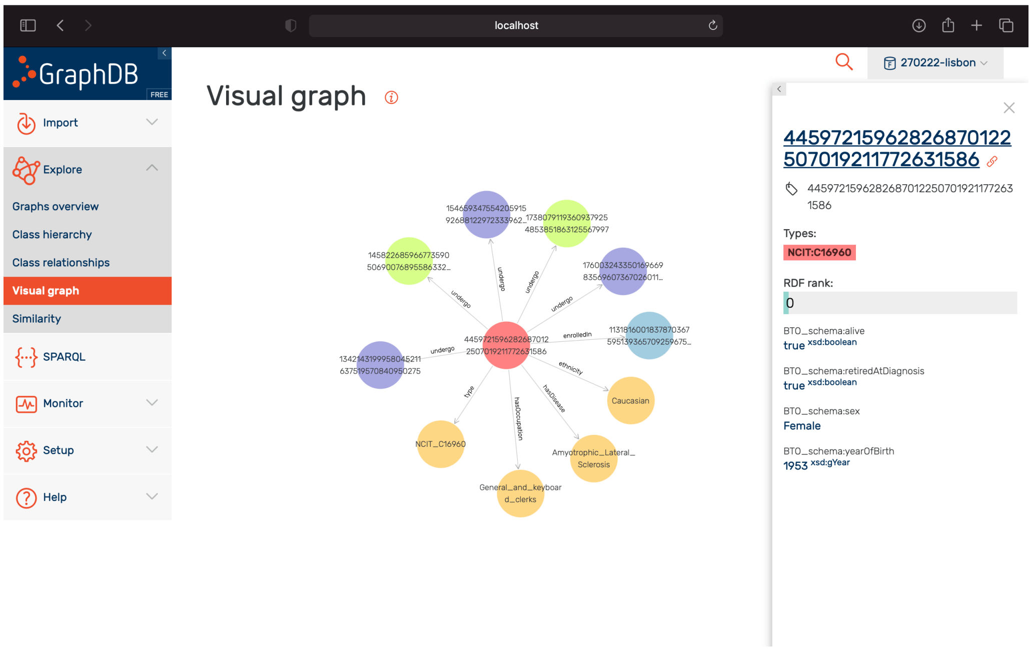
```
PREFIX BTO_schema: <https://w3id.org/brainteaser/ontology/schema/>
PREFIX NCIT: <http://purl.obolibrary.org/obo/NCIT_>
# ALSFRS-R
SELECT ?id ?date_alsfrs_r ?alsfrs_r_tot_score ?bulbar_subscore ?motor_subscore ?respiratory_subscore ?q1
?q2 ?q3 ?q4 ?q5 ?q6 ?q7 ?q8 ?q9 ?q10 ?q11 ?q12 WHERE {
    ?idURI a NCIT:C16960 ;
            BTO_schema:undergo ?eventURI .
    ?eventURI a NCIT:C74589 ;
              BTO_schema:startDate ?date_alsfrs_r ;
              BTO_schema:consists ?qRURI .
    ?qRURI a BTO_schema:ALSFRS-R ;
            BTO_schema:alsfrs-r-tot ?alsfrs_r_tot_score ;
            BTO_schema:bulbar_subscore ?bulbar_subscore ;
            BTO_schema:motor_subscore ?motor_subscore ;
            BTO_schema:respiratory_subscore ?respiratory_subscore ;
```

Showing results from 1 to 1,000 of 4,580. Query took 0.7s, moments ago.

	id	date_alsfrs_r	alsfrs_r_tot_score	bulbar_subscore	motor_subscore	respiratory_subscore	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12
1	"1009243044351795	"2006-06-21T00:00:	"34"^^xsd:integer	"8"^^xsd:integer	"15"^^xsd:integer	"11"^^xsd:integer	"3"^^xsd:integer	"3"^^xsd:integer	"2"^^xsd:integer	"2"^^xsd:integer	"1"^^xsd:integer	"2"^^xsd:integer	"3"^^xsd:integer	"4"^^xsd:integer	"3"^^xsd:integer	"3"^^xsd:integer	"4"^^xsd:integer	"4"^^xsd:integer

Figure 2.5: A snapshot of the GraphDB “SPARQL” tab.

Figure 2.6 instead shows the tool “Visual graph” inside the “Explore” tab. This tool allows to explore the graph of RDF data without using SPARQL. Using a search input field, a starting resource is chosen for exploring the graph. The graph that is displayed shows the chosen resource in the center and all links to other resources around it. It is possible to limit the number of links displayed (20 by default) and choose the types of classes and properties to show using the button with the settings icon. Nodes that belong to the same class have the same color and the size of the nodes reflects the importance of the node by RDF rank. When a node is clicked, the drop-down menu on the right is opened with more information about that node. In addition, each node can in turn be expanded or collapsed, and if it is not of interest it can be hidden.
This is just one feature provided within the “Explore” tab, in fact it also allows to explore hierarchies between classes and consult class relationships.



44597215962826870122 507019211772631586

44597215962826870122507019211772631586

Types:
NCIT:C16960

RDF rank:
0

BTO_schema:alive
true xsd:boolean

BTO_schema:retiredAtDiagnosis
true xsd:boolean

BTO_schema:sex
Female

BTO_schema:yearOfBirth
1953 xsd:gYear

Figure 2.6: A snapshot of the "Visual graph" tool of the GraphDB "Explore" tab.

Chapter 3

Requirements analysis

As mentioned in Chapter 1, the purpose of this thesis is to design and develop the part of the Brainteaser Ontology (BO) that deals with retrospective data on Amyotrophic Lateral Sclerosis (ALS). The ALS part of the BO, and more generally the entire BO, was co-designed in close collaboration with medical partners and domain experts. We used this approach to incorporate expert knowledge into the BO and, at the same time, to validate all design choices. To this end, we operated iteratively, producing several intermediate versions of the ontology and discussing them with our domain experts.

As a design approach, classes and properties already defined in external ontologies were used for the classes in the BO ontology whenever possible. Reusing existing ontologies as much as possible is generally a good practice when developing ontologies [Noy and Mcguinness, 2001]. There are different libraries of reusable ontology on the web, such as the one we used, namely Ontobee [Ong et al., 2017]. In detail, Ontobee allowed us to search for some existing ontology classes that we imported into BO ontology. By reusing entities and properties already defined in other ontologies, we are not only able to reinforce collaboration and data consistency across databases, but to ensure the authority of the semantic meaning of these resources.

New custom classes and properties were created only when it was impossible to find their exact correspondence in some existing ontology available online or included in OntoBee. Each of these new classes or properties were always discussed with the medical partners to verify that these new concepts correctly represented the corresponding real-world concepts and to ensure the semantic quality of the ontology.

The first design phase of the BO involved all project partners and focused on defining what types of data should be handled in the ontology. As mentioned in Section 1.2, it emerged that the ontology needs to handle anonymized raw data (both retrospective and prospective), data generated by the AI models, and data from the evaluation challenges. From this point, the design of the ontology focused on the specific part of the retrospective data. In a next step, we received simplified samples of restrospective data from medical partners to start understanding their characteristics and potential issues in detail. In addition, medical partners were elicited through forms in order to collect requirements that they considered important for both ALS and MS diseases.
The requirements gathering and analysis phase ended up with a preliminary draft of the BO schema. From this draft and feedback from subsequent discussion meetings on the BO, new versions of the ontology were then iteratively developed until final approval by the partners. After the approval of the BO schema, full and anonymized retrospective data were received from the medical partners: Lisbon, Turin and Madrid for ALS. The inspection of these complete data allowed to further improve and refine the ontology and also to discover some problems on the instances, related to incomplete or noisy data. In conclusion, these updates to the BO and the solutions to address specific issues have been then discussed and approved in one-on-one meetings with the specific medical partner.
Overall, all these iterations led to the development of the first version of the Brainteaser ontology as well as mappers to ingest retrospective data from medical partners.

Chapter 4

Design of an Ontology for Amyotrophic Lateral Sclerosis

In this chapter we describe more in detail the part of the ontology designed to model the Amyotrophic Lateral Sclerosis (ALS) retrospective data according to the co-design approach defined in Chapter 3. In particular, in the Section 4.1 we present the principles that have guided the development of the ontology, while in the Section 4.2 we analyze the main semantic areas and the main classes contained in each of them. The complete documentation of the first version of the Brainteaser Ontology is available at `https://w3id.org/brainteaser/ontology/`.

4.1 Overall design principles

The Brainteaser Ontology (BO) is the result of iterative design work done in relationship and agreement with medical partners and domain experts. As anticipated in Chapter 3, after an initial phase of meetings to discuss the characteristics of the ontology and after numerous intermediate versions of the same, the ontology was refined once medical partners sent the definitive, anonymized retrospective datasets. Regarding the part of the ontology that deals with modeling the ALS data, the datasets were provided by three clinical institutes: “Instituto de Medicina Molecular João Lobo Antunes” (iMM, Portugal), “Universitá degli Studi di Torino” (UNITO, Italy), and “Servicio Madrileño de Salud” (SERMAS, Spain). After analyzing the datasets and working together with the clinicians, according to shared co-design principles, we refined the ontology, carefully identifying the classes and properties to be modified or added, in order to obtain

the first final version of the Brainteaser Ontology. In choosing the classes to include in the BO, we maximized the reuse of concepts defined in ontologies and vocabularies already available and known on the web, thus limiting the creation of new custom classes to a minimum. In Figure 4.1 we represent the first version of the BO as a graph where nodes are classes and edges are typed relationships amongst the classes. Classes (nodes) represent real-world resources such as a person, a visit or an anatomical part, while relationships (edges) describe how the classes interact one with each other. It is important to note that in Figure 4.1, all nodes with dark red color correspond to classes purely related to modeling Multiple Sclerosis, and will not be treated in this chapter.
Overall, BO is composed of 379 classes, 76 object properties, 292 data properties, 396 named individuals and 20 annotation properties.
The complete documentation of the first version of the Brainteaser Ontology is available at `https://w3id.org/brainteaser/ontology/`. Using the service provided by the W3 Permanent Identifier Community Group, the ontology URIs will be secure and permanent over time. In this way, anyone can reuse the BO resources.

4.1.1 Semantic areas

The ontology is divided into eight main "semantic areas", i.e., groups of entities and relationships that relate to different types of concepts and aspects of our domain. Each entity is then classified into one of the eight semantic areas. These areas are explained in detail in Section 4.2 and are:

1. **Patient**: the semantic area that describes aspects of the patient, such as his or her status, residence, relatives, diseases from which he or she is suffering, and clinical trials in which he or she is participating;

2. **Genetic Data**: the information about the genetic mutations of a patient;

3. **Behavior**: containing information about the patient's behavior over time, such as smoking, physical activity, and more general aspects of lifestyle;

4. **Events**: classes that describe the possible events to be recorded for a patient, such as diagnosis, onset, and visits;

5. **Contingencies**: an area that contains classes that describe things that may happen to the patient during his or her lifetime, such as comorbidities and different types of trauma;

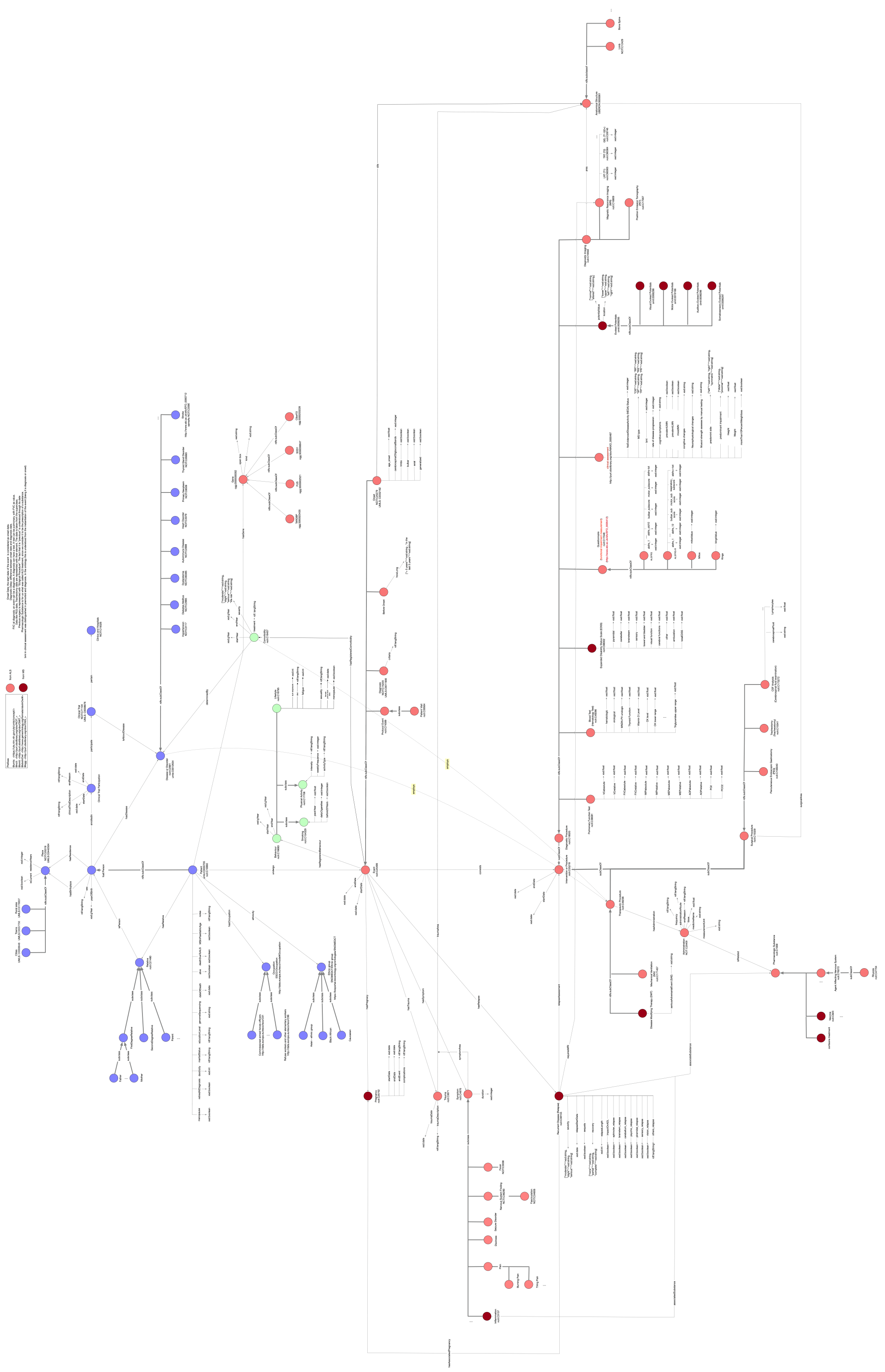


Figure 4.1: The Brainteaser Ontology.

6. **Intervention and Procedure**: containing classes on different types of procedures a patient may undergo during their medical history;

7. **Anatomical Structure**: composed of subclasses describing different parts of the human body;

8. **Symptoms**: subclasses describing symptoms of different nature that may happen to the patients.

4.1.2 External ontologies used in the Brainteaser Ontology

The ability to reuse parts of existing ontologies to generate new ones specifically tailored for some new application or context, while at the same time maintaining interoperability with other datasets, is an essential concept [Hoehndorf et al., 2015]. In the process of searching existing ontologies, the ontology repositories can help find already defined entities and relationships suitable for modeling data in a specific domain. To search for possible classes, belonging to the life sciences domain, to be imported into the Brainteaser Ontology, we mainly used Onto-Bee [Ong et al., 2017], an ontology repository where ontologies are presented as Linked Data. Table 4.1 describes the external ontologies used in the Brainteaser project. For each ontology, we report the prefix label used in the description of the Brainteaser Ontology, the URL associated with the prefix, and the name of the ontology. Among these ontologies, some of the most important ones are:

- FOAF (Friend of a Friend) [Graves et al., 2007]: an ontology describing people, their activities and their relations to other people and objects;

- The NCI (National Cancer Institute) Thesaurus OBO Edition [Kumar and Smith, 2005]: a reference terminology that includes broad coverage of the cancer domain, including cancer related diseases, findings and abnormalities;

- OGG (Ontology of Genes and Genomes) [He et al., 2014]: a well-known ontology used to model Genes, their mutations and the genome of living organisms;

- Uberon: ananatomical ontology that represents body parts, organs and tissues in a variety of animal species, with a focus on vertebrates;

- MAXO (Medical Action Ontology) [Carmody et al., 2019]: this ontology provides classes to describe the majority of the procedures that can be carried out by medical doctors while visiting or treating the patients;
- SNOMEDCT: SNOMED Clinical Terms is a collection of medical terms providing codes, terms, synonyms and definitions used in clinical documentation. It is considered one of the most comprehensive multilingual clinical healthcare terminologies in the world;
- ATCC (Anatomical Therapeutic Chemical Classification): the ATCC Ontology associates each ATC code with the corresponding active substance. It is used to model the pharmacological prescription assigned to the patients. The ontology has substantially a class for each active substance;
- Unified Medical Language System (UMLS): in particular we exploited the UMLS Metathesaurus, a large biomedical thesaurus, the biggest component of UMLS, that is organized by concept, or meaning, and it links similar names for the same concept from nearly 200 different vocabularies;
- ESCO: a multilingual classification that identifies and categorises skills, competences, qualifications and occupations relevant for the EU labour market and education. In particular, it provides a mapping to the International Standard Classification of Occupations (ISCO) to structure occupations;
- ICD-10: the International Classification of Diseases, in its tenth edition (ICD-10), contains a diagnostic and procedure coding system endorsed by the World Health Organization (WHO).

At design level, there are some cases where we imported a subset of entities from an external ontology in order to represent taxonomies of concepts and their `rdfs:subClassOf` relationships. We imported from these taxonomies only the classes that are useful for modeling the data needed for the project, i.e. we never imported the complete taxonomies, but only subsets of them. Some of the principal classes that work as roots of their respective taxonomies in the Brainteaser ontology are:

- Relative (`ncit:C21480`): the class represents a generic relative of a patient, and is the root of the taxonomy containing different degrees of kinship (such as Father, Mather, Sister, etc.);

- Occupation (`esco:model#Occupation`): the root class of possible occupation types of patients, established by the ESCO classification;
- Ethnic group (`snomed:372148003`): represents the general concept of ethnicity and is subclassified with the classes related to the different types of ethnicities;
- Symptom (`ncit:C4876`): the root class of the taxonomy describing the possible symptoms related to a patient;
- Gene (`ogg:0000000002`): root of the gene taxonomy;
- Anatomical Structure (`uberon:0000061`): root of the taxonomy that contains the classes concerning the human body locations;
- Pharmacologic Substance (`ncit:C1909`): the root class of the clinical drugs that may be prescribed to patients;
- Disease or Disorder (`ncit:C2991`): the root of the taxonomy with the possible diseases that a patient, or one of their relatives, may manyfest.

In addition, for every possible class imported from external (non-UMLS) ontologies, we relate each single concept in the BO to the respective concepts in the Unified Medical Language System (UMLS) metathesaurus and the ICD-10 classification through the relations `:sameAsUMLS` and `:sameAs_ICD10`, annotation subproperties of `owl:sameAs`. For example, the concept "amyotrophic lateral sclerosis" is inherited from the MONDO ontology (`mondo:0004976`) and the BO definition for this concept also reports that it is the same as `umls:C0002736` in the UMLS metathesaurus and the same as `icd10:G12.2` in the ICD-10 classification.

4.1.3 Brainteaser classes and properties

Not for all the concepts of the Brainteaser project it was possible to reuse classes from external ontologies. So for all those concepts that did not find a correspondence within already defined ontologies, we created new classes that allowed us to represent precisely the concept we needed. Some of these most relevant new classes are:

- **BeforeOnset**: subclass of the class "Event" (`ncit:C25499`) representing the period of time that occurred before the onset of the patient's disease;

prefix	**url**	**ontology**
foaf	`http://xmlns.com/foaf/spec/`	Friend of a Friend (FOAF) vocabulary
ncit	`http://purl.obolibrary.org/obo/NCIT_`	National Cancer Institute thesaurus (NCIT)
ogg	`http://purl.obolibrary.org/obo/OGG_`	Ontology of Genes and Genomes (OGG)
oboInOwl	`http://www.geneontology.org/formats/oboInOwl#`	OBO in OWL
uberon	`http://purl.obolibrary.org/obo/UBERON_`	Uberon
maxo	`http://purl.obolibrary.org/obo/MAXO_`	Medical Action Ontology (MAxO)
omit	`http://purl.obolibrary.org/obo/OMIT_`	Ontology for MIRNA Target (OMIT)
snomed	`http://purl.bioontology.org/ontology/SNOMEDCT/`	SNOMED Clinical Terms (SNOMEDCT)
atcc	`http://purl.bioontology.org/ontology/ATC`	Anatomical Therapeutic Chemical Classification
umls	`https://uts.nlm/nih.gov/uts/umls/concepts/`	Unified Medical Language System (UMLS)
esco	`http://data.europa.eu/esco/isco/`	European Skills, Competences, Qualifications and Occupations (ESCO)
icd10	`http://purl.bioontology.org/ontology/ICD10/`	International Classification of Diseases, tenth edition (ICD-10)
efo	`http://www.ebi.ac.uk/efo/EFO_`	Experimental Factor Ontology (EFO)
mondo	`http://purl.obolibrary.org/obo/MONDO_`	Mondo Disease Ontology (MONDO)
dcterms	`http://purl.org/dc/terms/`	DCMI Metadata Terms

Table 4.1: External ontologies exploited by Brainteaser.

- **ALSFRS** and **ALSFRS-R**: two different classes that are subclasses of the "Questionnaire" class (`ncit:C17048`) and that represent the Amyotrophic Lateral Sclerosis Functional Rating Scale in the "old" and "revised" versions respectively;
- **Clinical Trial Participation**: class representing the fact that a patient is participating in a clinical trial.

Overall, the number of new classes defined in the BO is much smaller than the number of classes imported from outside, in fact we tried to define new classes only when it was unavoidable.
Instead, the properties useful to the project, both data and object properties, are all defined as new properties of the Brainteaser Ontology. These properties allowed us to properly manage the various classes of the BO, both those imported externally and those defined by us. In the BO, URIs for both new classes and properties are prefixed with "https://w3id.org/brainteaser/ontology/schema/".

4.1.4 Named Individuals

In some cases, for the classes of some taxonomies presented in Subsection 4.1.2 (Occupation, Ethnic Group, Anatomical Structure, etc.), when they are the range of object properties, it is not necessary to register additional information beyond the concept expressed by the classes themselves. For example, when recording the occupation of a patient, it is necessary to know only the type of occupation, without any further information about it. The consequence is that every time a new triple is created with an object in one of these taxonomies, we would be forced to create a new instance. However, to avoid an explosion in the number of instances of these classes, and due to the fact that there is no need to model additional information about it, we decided to create *named individuals*, one for each class in these taxonomies. These *named individuals* are used as objects of the corresponding object properties whenever a new triple needs to be instantiated. In BO such named individuals are prefixed with "https://w3id.org/brainteaser/ontology/named-individual/".

4.2 Area-by-Area

In this section, we describe the elements that characterize each semantic area presented in Subsection 4.1.1, in particular we focus on classes and properties useful for modeling ALS data, while the MS part is not covered. We use bold font to denote ontological classes, monospace font to represent their URIs, and italics to represent properties. Prefixes for URIs derived from external ontologies are defined in Table 4.1, while the base prefix of the Brainteaser Ontology is "https://w3id.org/brainteaser/ontology/schema/".

4.2.1 Patient

The patient semantic area, shown in Figure 4.2, contains classes and properties to represent a patient's personal information, relatives, places they have lived, diseases, and participation in clinical trials.
The central element of this area is the **Patient** class (`ncit:C16960`), a subclass of `foaf:Person`, which has several data properties to describe the patient's personal information. Some of these properties are:

- *dateOfDeath* (with range `xsd:date`): if the patient is dead, it is used to represent the date of death;
- *educationLevel* (`rdf:langString`): a property that accepts a string (with an associated language tag) representing the education title obtained by the patient;
- *maritalStatus* (`rdf:langString`): a data property to indicate the patient's marital status;
- *retiredAtDiagnosis* (`xsd:boolean`): it accepts a Boolean value. If the value is "True", it indicates that the patient is retired;
- *menopause* (`xsd:boolean`): it is used to indicate the permanent cessation of menstruation;
- *alive* (`xsd:boolean`): it indicates whether the patient is still alive ("True" value);
- *deathDueToALS* (`xsd:boolean`): if the patient died due to the course of the ALS disease, this property is set to "True". For example, if the patient died in a car accident, this property is set to "False";
- *notes* (`rdf:langString`): it is used to add notes to the patient. It is simply a free text box.

The **Patient** class inherits from the **Person** class the properties *sex* and *yearOfBirth*, which have ranges in `rdf:langString` and `xsd:gYear`, respectively. These two properties allow the patient's gender and year of birth to be recorded. It is important to note that the entire date of birth is not allowed to be recorded in order to preserve patient anonymity. In addition, the **Patient** class, since it is a subclass of **Person**, is connected to the **Place** class (`ncit:C25319`) by means of the object properties hasBirthplace and hasResidence. These properties respectively allow **Place** class instances to be linked to indicate the patient's place of birth and subsequent residences. The class **Place** is subclassed into 3 classes, namely **Cities** (`umls:C0008848`), **Towns** (`umls:C0557750`), **Rural area** (`umls:C0178837`), which correspond to the three urbanization degrees of the DEGURBA classification[1]. In detail, for reasons of data anomization we have used the DEGURBA classification since we do not record precise information on place of residence or birth (e.g. zip code), but we simply indicate whether it is a densely populated area (city) or a sparsely populated area (town or rural area). In addition, the **Place** class has two properties: *isCurrent* (`xsd:boolean`) to indicate whether it is the current residence or not, and residenceYears (`xsd:integer`) to record the number of years the patient has lived in that place.
Patient is also connected through the property *ethnicity* to the class **Ethnic group** (`snomed:372148003`), which represents a generic ethnic group. This class is the root of a taxonomy of ethnicities whose classes are taken from the SNOMEDCT ontology.
To model family predisposition, we need to register the presence of the patient's relatives. To do so, we considered the class **Relative** (`ncit:C21480`), which models a generic relative of a patient, and the object property *hasRelative* which connects the class **Patient** to the class **Relative**. **Relative**, in turn, is the root class of a taxonomy of kinship classifications such as **FirstDegreeRelative**, **Father** and **Mother**, etc. (these classes are imported directly from the NCIT ontology). A relative may or may not be a patient in the database. In the first case, the instance of "Relative" is linked to the corresponding existing instance of "Patient" through the property *isPerson*. In the other case, an instance of class "Person" is created, and the instance of "Relative" is linked through *isPerson* property to this instance. One patient or one generic person can be a relative of more than one patient, and for this reason multiple instances of "Relative" may be instantiated, one for each degree of kinship occurring for that patient/person

[1]`https://ec.europa.eu/eurostat/web/degree-of-urbanisation/background`

in the database. Each of these instances will be connected through *isPerson* to the corresponding instance of **Patient** or **Person**.
The class **Occupation** (`esco:model#Occupation`) represents the patient's job. It is the root class of a taxonomy of classes representing different types of work, and a "Patient" instance is linked to **Occupation** through the property *hasOccupation*. **Person** is connected through the property *enrolledIn* to **Clinical Trial Participation** (`:ClinicalTrialParticipation`), representing the fact that a patient is participating in a clinical trial. An instance of "Clinical Trial Participation" can have a property *startDate*, *endDate*, *clinicalTrialDescription*, and *endReason*, to indicate the start and end date and description of the clinical trial, and the reason for leaving the clinical trial, respectively. Each instance of "Clinical Trial Participation" is part of a **Clinical Trial** (`ncit:C71104`) and linked to this class through the *participate* property. In turn, a clinical trial *pertains* to a certain hospital, and thus the property links the class **Clinical Trial** to the class **Clinics and Hospitals** (`ncit:C19326`).
A person (so both a patient and a relative who is not present in a clinical trial) may have one or more diseases. **Person** is connected to **Disease or Disorder** class (`ncit:C2991`) through *hasDisease* property. In addition, the class **Clinical Trial** is linked to **Disease or Disorder** through *isAboutDisease* property to indicate which disease is being studied in the specific trial.
Finally, an instance of class **Patient** can register one or more instances of class **Event** (`ncit:C25499`) by means of the object property *undergo*.

4.2.2 Genetic Data

A patient may have data regarding their genetic makeup. These data are very useful to study cases of hereditary ALS. Specifically, the inheritance of ALS is attributed to mutations in 12 different genes, the most common of which are SOD1, FUS and TARDBP [Andersen and Al-Chalabi, 2011].
As shown in Figure 4.3, to model this genetic data, the ontology links the class **Patient**, through the property *hasGene*, to the class **Gene** (`ogg:0000000002`), which is subclassified with the following types of genes: **SOD1** (`ogg:3000006647`), **FUS** (`ogg:3000002521`), **C9orf72** (`ogg:3000203228`), and **TARDBP** (`ogg:3000023435`). In addition, the instances of **Gene** class can have two properties: *kind* and *open box*. The first one allows to express the type of mutation present on the gene, while the second one allows to add more information through an unstructured text.

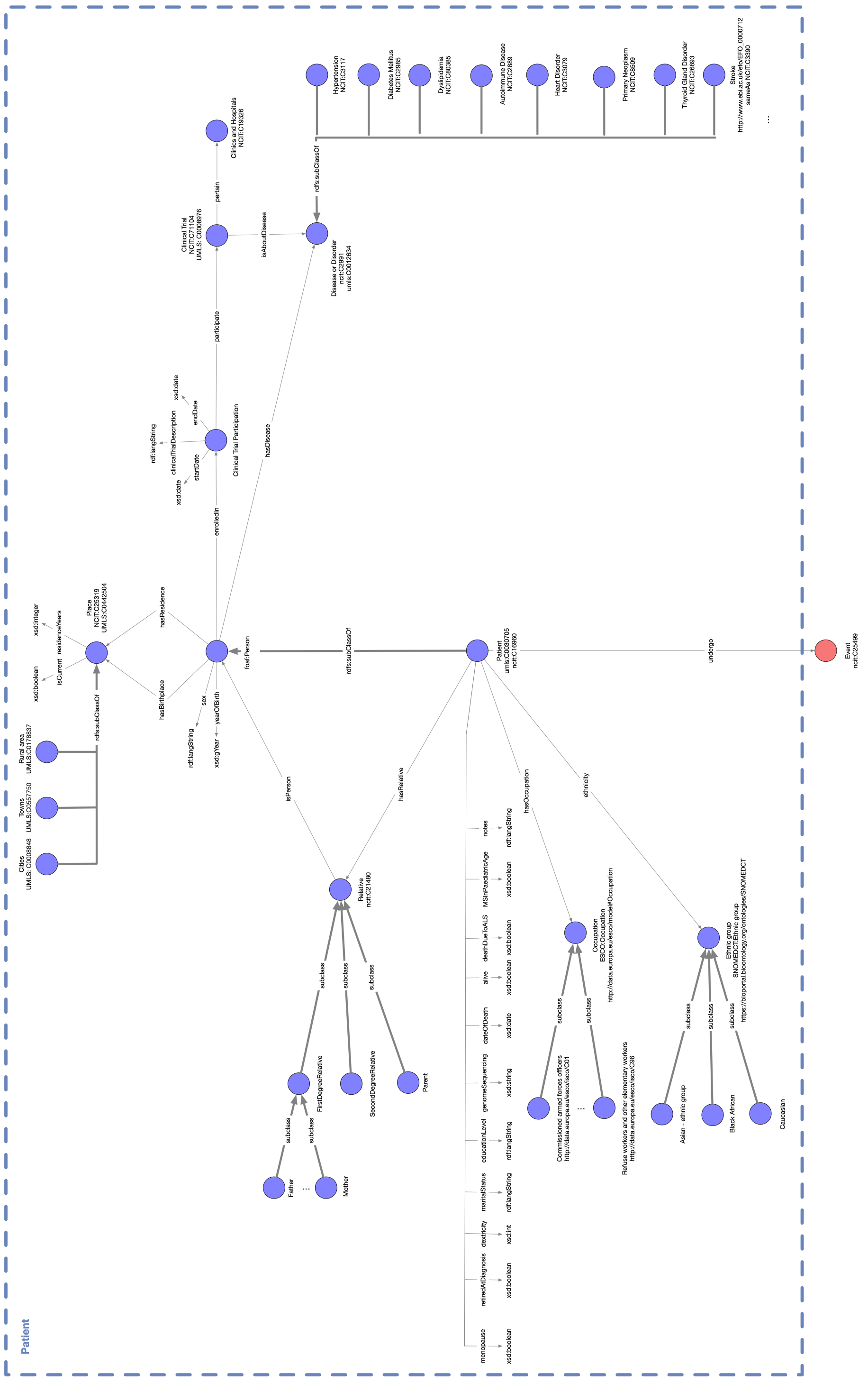


Figure 4.2: Patient semantic area, including the classes Patient, Person, Relative, Place, Clinical Trial, Clinical Trial Participation, and Disease or Disorder.

4.2.3 Behaviour

In this semantic area, patient behavioral data, and in particular their evolution over the years, are modeled through the use of classes and properties. As reported

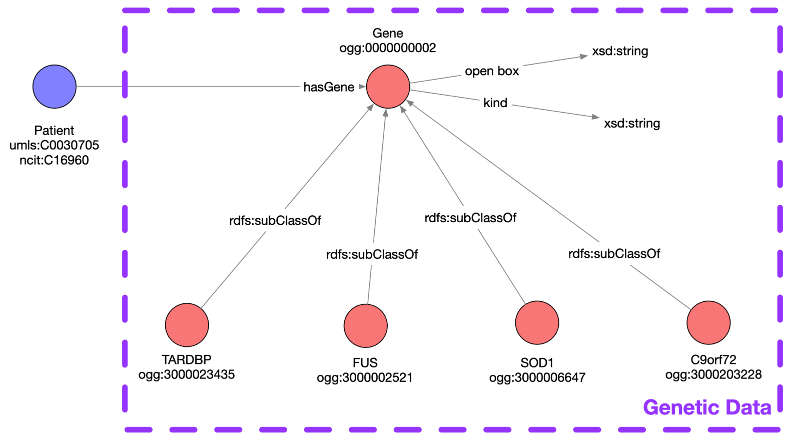


Figure 4.3: Semantic area containing the information about a patient's gene and their mutations.

in Figure 4.4, the main class of this area is **Behaviour** (`ncit:C19683`), which has the properties *startYear* and *endYear* (both with range `xsd:gYear`), which describe the start and end year of such behavior. Each instance of the **Behaviour** subclasses therefore offers the possibility to register the time interval in which such behavior was sustained. In addition, the class **Event** is linked to **Behaviour** through the property *hasRegisteredBehaviour*.
Different types of behavior are modeled through the use of subclasses of the main class **Behaviour**. Such subclasses are:

- **Smoking** (`ncit:C154329`): represents the patient's smoking habit. It has the properties *packYear* (with range `xsd:float`) and *dailyCigarettes* (`xsd:integer`). The first property relates to a quantification of lifetime tobacco exposure and is calculated by multiplying the number of packs of cigarettes smoked per day by the number of years the person has smoked (one pack-year corresponds to smoking 20 cigarettes per day for one year)[2], while the second refers to the average number of cigarettes smoked per day;

- **Physical Activity** (`ncit:C17708`): describes the physical activity usually exerted by the patient. Its properties are *intensity* (`rdf:langString`), *weeklyFrequence* (`xsd:integer`) and *activityType* (`rdf:langString`);

- **Lifestyle** (`ncit:C16795`): acts as a container for different types of behavioral information regarding the patient. Such behaviors are modeled as data properties of the **Lifestyle** instance, and are: *sunExposure* (`xsd:int`), *diet*

[2] `https://www.cancer.gov/publications/dictionaries/cancer-terms/def/pack-year`

(`xsd:int`), *fatigue* (`xsd:int`), *sexuality* (`rdf:langString`), *femalePeriodStartDate* (`xsd:date`), and *menopause* (`xsd:boolean`).

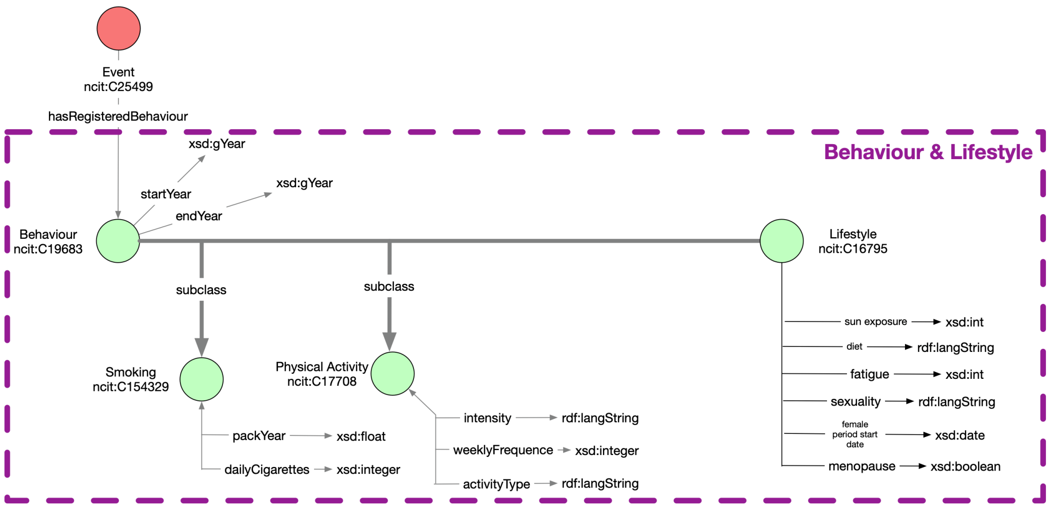


Figure 4.4: Behaviour and Lifestyle semantic area, including the subclasses Smoking, Physical Activity and Lifestyle.

4.2.4 Events

The "Events" semantic area, represented in Figure 4.5, contains some of the most important classes of the ontology, since one of the objectives of the ontology itself is to model the events related to the patient and their evolution in time. The main class in this area is **Event** (`ncit:C25499`), which describes a generic event that can happen to a patient in a medical environment. The class **Event** is characterized by the two data properties *startDate* and *endDate* (both with range `xsd:date`), which describe the time period in which this event occurred. Different types of events are represented through the use of these subclasses:

- **Protocol Event** (`ncit:C74589`): represents a planned protocol activity, such as randomization and study completion, and occurrences, conditions, or incidents independent of planned study evaluations that occur during the trial (e.g., adverse events) or before the trial (e.g., medical history). Its subclass is **Patient Visit** (`ncit:C39564`), which describes a specific type of protocol event, i.e., a patient visit to a hospital;

- **Diagnosis** (`ncit:C25279`): represents the moment when the patient is officially diagnosed with the disease;

- **Onset** (`ncit:C25279`): represents the event in which the first symptoms of the disease first occur to the patient. The **Onset**, in particular, can be of several types, which are represented as boolean properties (`xsd:boolean`): *limbs*, *bulbar*, *axial*, and *generalized*. An onset may also have the property *cerebrospinalOligloconalBands* (`xsd:integer`), which indicates, if the test was performed, the presence of such bands in the patient's bloodstream and the *age_onset*, i.e., the age in years of the patient at the time of onset. Onset is located in a part of the human body, and thus the event **Onset** is related to the class **Anatomical structure** (`uberon:0000061`) through the *site* property;
- **Before Onset** (`:BeforeOnset`): describes an event that happened before onset, whatever its nature. The property *howLong* indicates how long before onset this event happened, and can only take two string values, defining an event that happened in the last five years, or before the last five years.

An instance of Event, in turn, can be linked to various other classes. **Event** is linked to **Trauma** through *hasTrauma* property, which describes the registration of a traumatic event that happened to the patient; to **Symptom** through *hasSymptom* property, which describes the presence of symptoms recorded during an event; and finally to one or more **Interventions or Procedures** that can be applied to the patient, through the *consist* property.

4.2.5 Contingencies

The "Contingencies" semantic area, represented in Figure 4.6, contains classes that represent "things that may happen": occurrences that happen to the patient that may or may not be related to the disease. They correspond to phenomena in the patient's body that cannot be directly planned. However, each of these contingencies is recorded during an event and are therefore linked to the **Event** class via a specific property.
The classes in this area are:

- **Trauma** (`ncit:C3671`): describes the presence of a trauma on the patient's body. It also reports the properties *traumaDate* (`xsd:date`) and *traumaDescription* (`rdf:langString`). The class **Trauma** is linked to the class **Anatomical Structure** through the object property *traumaArea*. It is linked to the **Event** during which it is registered via the *hasTrauma* property;

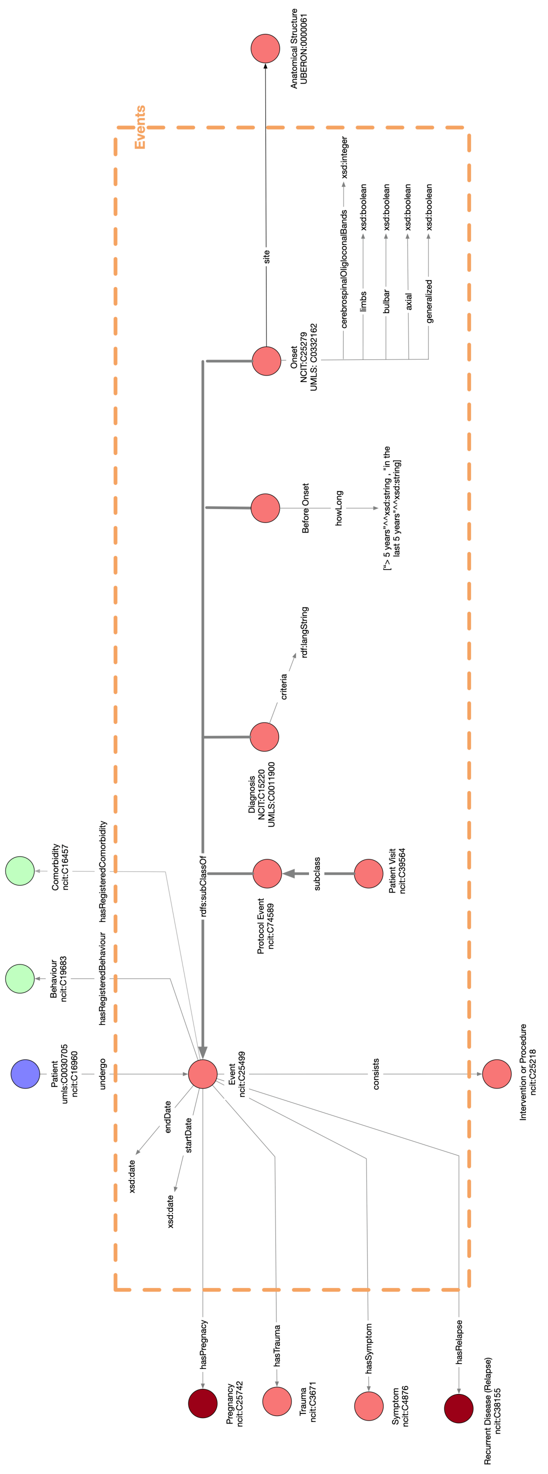


Figure 4.5: Events semantic area, including the class Event and its subclasses: Protocol Event, Diagnosis, Before Onset and Onset.

- **Comorbidity** (`ncit:C16457`): represents the presence of two or more diseases or medical conditions in a patient. The **Comorbidity** is characterized by the properties *startYear* and *endYear* (both with range `xsd:gYear`), along with *treatment* (`xsd:string`), which reports the prescribed treatment, and *severity*, which describes the severity of the comorbidity with a set of string values (slight, moderate, serious, life risk). A **Comorbidity** is recorded during an **Event**, and the two classes are linked through the *hasRegisteredComorbidity* property.

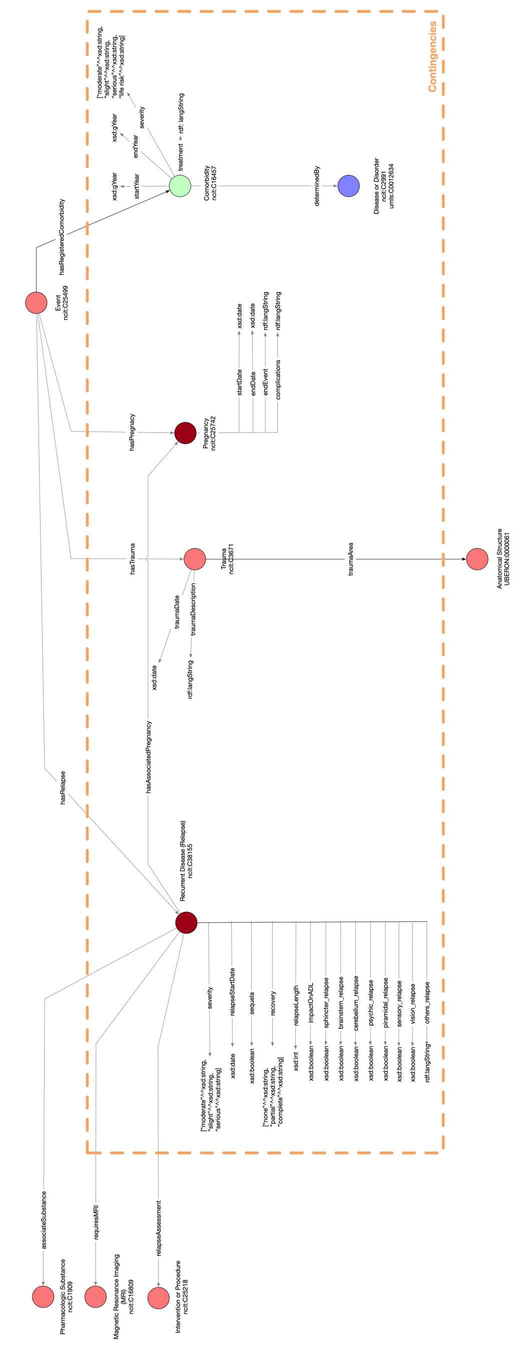


Figure 4.6: Contingencies semantic area composed of classes Trauma and Comorbidity.

4.2.6 Intervention and Procedure

This semantic area contains the classes that describe the interventions made to a patient to understand, monitor, and intervene on his or her clinical condition. The class **Event** is linked to the main class of this area, **Intervention or Procedure** (`ncit:C25218`), through the object property *consists*. This class represents "a general activity that produces an effect, or that is intended to alter the course of a disease in a patient"[3].

Instances of the class **Intervention or Procedure** may have two properties: *startDate* and *endDate*. These dates allow us to record the start and end time of the intervention. Note that, thanks to an abuse of notation, by leaving the *endDate* empty, we can also represent interventions and procedures that are atomic in time, i.e., are not distributed over multiple days. The ontology contains several subclasses of **Intervention or Procedure**, and each of them constitutes a semantic subarea depending on the nature of the procedure itself. In the following subsections we describe in detail the features of each subarea.

4.2.6.1 Therapeutic Procedure

The **Therapeutic Procedure** subclass (`ncit:C49236`), depicted in Figure 4.7, represents the action or administration of therapeutic agents to produce an effect that is intended to alter or stop a pathological process. In addition to the generic therapy represented by the same class **Therapeutic Procedure**, there is the subclass **Non-invasive ventilation (NIV)** (`ncit:C171457`).

Non-invasive ventilation is an essential part of Amyotrophic Lateral Sclerosis (ALS) treatment because it significantly improves survival, quality of life, and cognitive performance [Morelot-Panzini et al., 2019].

All possible therapeutic procedures may involve the prescription or associated use of one or more pharmacological substances. The administration of these substances is modeled through the use of the class **Administration** (`ncit:C25409`), characterized by properties that describe the administration itself: *administrationRoute* (`rdf:langString`), *endReason* (`rdf:langString`), *dose* (`xsd:float`), *medicineName* (`xsd:string`), *frequency* (`rdf:langString`), *measurementUnit* (`xsd:string`). An **Administration** is related to the administered substance which is represented through the class **Pharmacological Substance** (`ncit:C1909`), and linked through the object property *isRelated*. Among the possible sub-

[3]`http://purl.obolibrary.org/obo/NCIT_C25218`

stances that can be administered (subclasses of **Pharmacological Substance**) in the ALS domain, we have the class **Agent Affecting Nervous System** (`ncit:C78927`) and especially its subclass **Riluzole** (`ncit:C47704`).

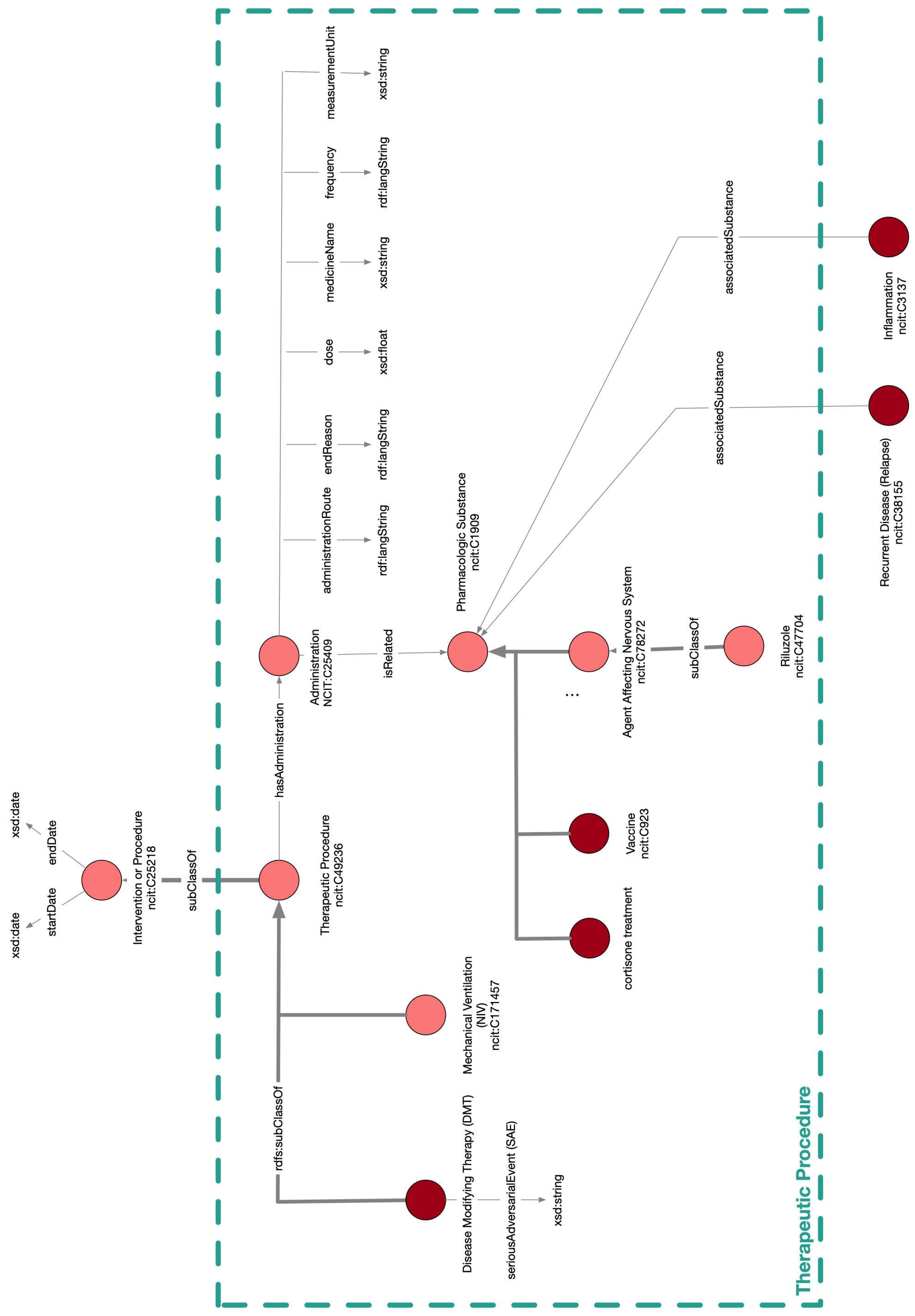


Figure 4.7: Therapeutic Procedure semantic area, sub-area of the Intervention or Procedure area.

4.2.6.2 Diagnostic Procedure

Diagnostic Procedure (`ncit:C18020`) is a subclass of **Intervention or Procedure**, and represents any procedure or test to diagnose and evaluate the progression of a disease or disorder. This, in turn, has many subclasses. Due to this, we have further subdivided the graphical representation of this sub-area into several figures to aid in readability: Figure 4.8, Figure 4.9, and Figure 4.10.

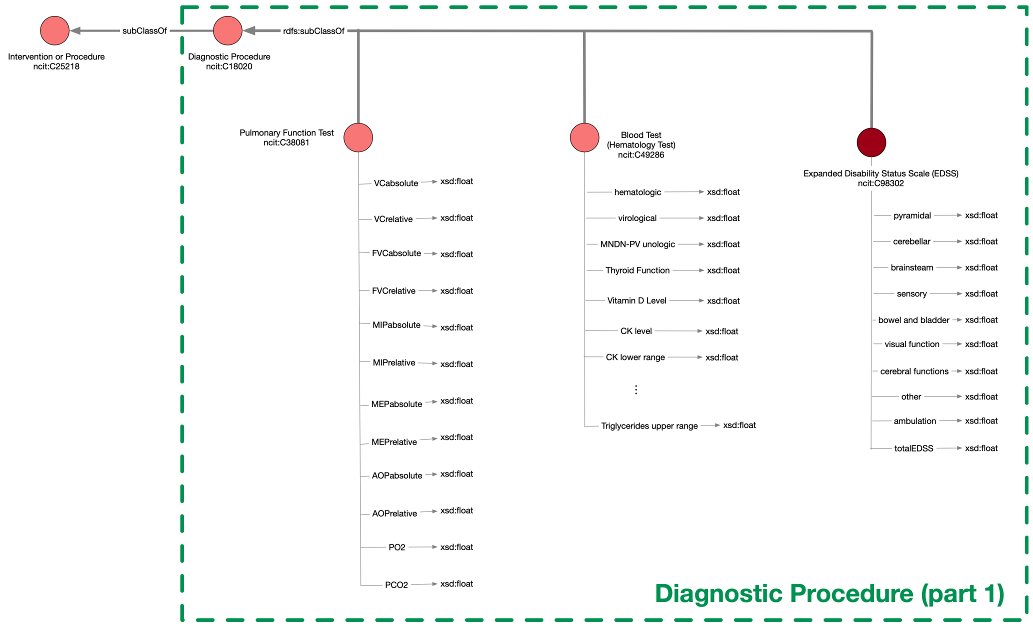


Figure 4.8: Part 1 of the Diagnostic Procedure semantic area, a sub-area of the Intervention or Procedure area. This figure reports the subclasses Pulmonary Function Test and Blood Test.

Starting from Figure 4.8, we see the **Pulmonary Function Test** (`ncit:C38081`). As the name suggests, it is used to store information about pulmonary function tests carried out on the patient. In particular, the **Pulmonary Function Test** is a foundational element of ALS management, as it is used to make decisions, including when to initiate the Non-invasive ventilation (NIV) [Lechtzin et al., 2018]. Instances of this class have several associated properties (all with range `xsd:float`), each of which corresponds to a parameter used to assess the quality of the patient's respiratory function (e.g., *FVCabsolute*, *FVCrelative*, etc...).
Another subclass is **Blood Test** (`ncit:C49286`), which represents a test to measure hematopoietic components and study hematological disorders. Relevant parameters assessed through a blood test are represented through the use of properties of this class (e.g., *CK level* (`xsd:float`)).
In the figure Figure 4.9, two more subclasses of **Diagnostic Procedure** are presented. The first is the class **Questionnaire** (`ncit:C17048`), and represents a

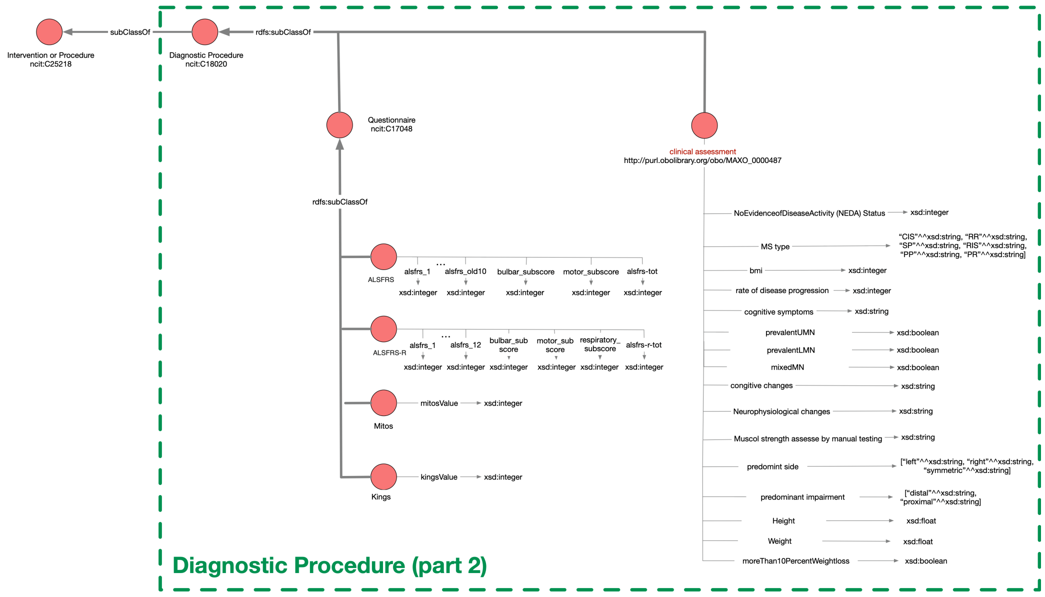


Figure 4.9: Part 2 of the Diagnostic Procedure semantic area, a sub-area of the Intervention or Procedure area. This figure reports the subclasses Questionnaire and clinical assessment.

generic questionnaire that is submitted to patients during a visit. This class is in turn subclassified with four additional classes representing different questionnaires that are very useful in the ALS domain:

- **ALSFRS** (`:ALSFRS`): this class represent the ALS Functional Rating Scale (ALSFRS) that is a validated rating instrument for monitoring the progression of disability in patients with Amyotrophic Lateral Sclerosis (ALS) [Cedarbaum et al., 1999]. The ALSFRS consists of ten questions on specific physical functions: (1) speech, (2) salivation, (3) swallowing, (4) handwriting, (5) cutting food and handling utensils, (6) dressing and hygiene, (7) turning in bed and adjusting bed clothes, (8) walking, (9) climbing stairs and (10) breathing.
  The data properties of this class are:

  – *alsfrs_1,...,alsfrs_old10* (with range `xsd:integer`): used to store the value of each of the ten questions that compose the ALSFRS. Each question is rated from 4 (normal) to 0 (no ability);[4];

  – *alsfrs-tot* (`xsd:integer`): corresponds to the sum of all ten questions of ALSFRS, with a maximum total score of 40 and a minimum total score of 0;

[4] `https://www.alspathways.com/assessing-function/`

  - *bubar_subscore* (`xsd:integer`): indicates the sum of ALSFRS questions 1, 2, and 3;
  - *motor_subscore* (`xsd:integer`): reports the sum of ALSFRS questions 4, 5, 6, 7, 8, and 9;

- **ALSFRS-R** (`:ALSFRS-R`): corresponds to the revised version of the ALSFRS (ALSFRS-R), which, compared with the original version of the ALSFRS, incorporates additional assessments of dyspnea, orthopnea, and respiratory insufficiency [Cedarbaum et al., 1999]. In other words, the 10th question of the ALSFRS, in the ALSFRS-R version, is split into 3 questions, namely question 10, 11, and 12.
  The data properties of the **ALSFRS-R** class are:
  - *alsfrs_1,...,alsfrs_12* (with range `xsd:integer`): used to store the value of each of the twelve questions that compose the ALSFRS-R. Each question, as for the ASLFRS, is rated from 4 (normal) to 0 (no ability)[4];
  - *alsfrs-r-tot* (`xsd:integer`): corresponds to the sum of all twelve questions of ALSFRS-R, with a maximum total score of 48 and a minimum total score of 0;
  - *bubar_subscore* (`xsd:integer`): indicates the sum of ALSFRS-R questions 1, 2, and 3;
  - *motor_subscore* (`xsd:integer`): reports the sum of ALSFRS-R questions 4, 5, 6, 7, 8, and 9;
  - *respiratory_subscore* (`xsd:integer`): stores the sum of ALSFRS-R questions 10, 11 and 12;

- **Mitos** (`:Mitos`): this class represents the Milano-Torino ALS staging system (ALS-MITOS), which is a tool for measuring ALS progression. The MiToS system uses six stages, from 0 to 5 and is based on functional ability as assessed by the ALS Functional Rating Scale-Revised (ALSFRS-R), with stage 0 being normal function and stage 5 being death [Fang et al., 2017]. The number of the Mitos stage is stored through the *mitosValue* data property (`xsd:integer`);

- **Kings** (`:Kings`): represent the King's staging system for the ALS progression. It uses five stages, from 1 to 5 and is based on disease burden as measured by clinical involvement and significant feeding or respiratory failure, with stage 1 being symptom onset and stage 5 being death [Fang et al., 2017]. The associated data property *kingsValue* (`xsd:integer`) stores the value of the King's stage.

While the second is the class **Clinical Assessment** (`maxo:0000487`), which represents a measurement performed in a clinical setting using clinician's observations and instrument data to inform patient care and research[5]. Some of the most important data properties associated with this class for the ALS domain are:

- *bmi* (`xsd:integer`): used to store the Body Mass Index (BMI) value, that is a value derived from weight and height of a patient;
- *Height* (`xsd:float`): indicates the height in meters of the patient;
- *Weight* (`xsd:float`): represents the weight in kilograms of the patient;
- *moreThan10PercentWeightloss* (`xsd:boolean`): stores the value "True" if the patient has lost more than 10% of his usual weight;
- *prelaventUMN, prevalentLMN, mixedMN* (`xsd:boolean`): indicates, for ALS disease, the loss of function of upper motor neurons (*prelaventUMN*), lower motor neurons (*prevalentLMN*), or both (*mixedMN*).

The last image in this subsection, Figure 4.10, contains the class **Diagnostic Imaging** (`ncit:C16502`). With this class we refer to various techniques for visualizing the inside of the body to help understand the cause of a disease or injury, and thus confirm a diagnosis. Doctors also use it to see how a patient's body is responding to treatment for a fracture or disease. To **Diagnostic Imaging** is linked the class **Anatomical Structure**, through the object property *area*, to indicate which area of the body is involved in the procedure. The subclasses of **Diagnostic Imaging** are **Magnetic Resonance Imaging**, also known as **MRI** (`ncit:C16809`), characterized by properties that describe its results, and **Positron Emission Tomography**, also known as **PET** (`ncit:C17007`).

4.2.6.3 Surgical Procedure

The class **Surgical Procedure** (`ncit:C15329`) describes a generic surgical procedure performed on a patient. As seen in the Figure 4.11, its subclasses are:

[5]`http://purl.obolibrary.org/obo/MAXO_0000487`

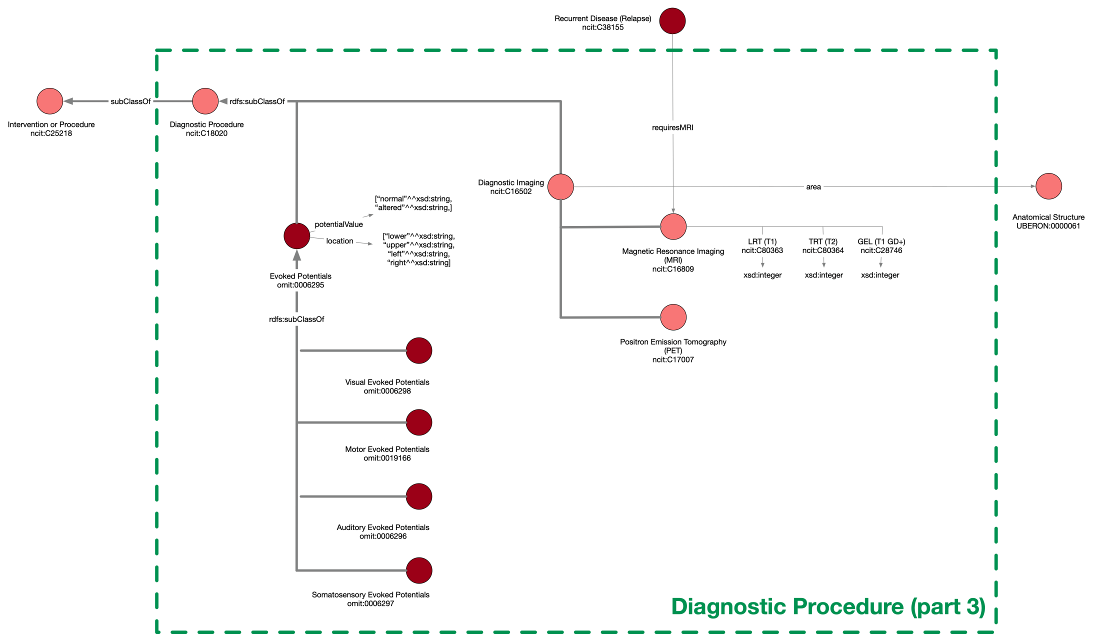


Figure 4.10: Part 3 of the Diagnostic Procedure semantic area, a sub-area of the Intervention or Procedure area. This figure reports the subclass Diagnostic Imaging.

- **Percutaneous Endoscopic Gastrostomy (PEG)** (`ncit:C10604`): a procedure in which a flexible feeding tube is placed through the abdominal wall and into the stomach, allowing nutrients, fluids and medications to be placed directly into the stomach, thus bypassing the mouth and esophagus[6];

- **Tracheotomy (or tracheostomy)** (`ncit:C15341`): an opening surgically created through the neck into the trachea to allow direct access to the breathing tube[7];

- **Cerebrospinal fluid examination (CSF Analysis)** (`ncit:C173272`): the analysis of the cerebrospinal fluid, a clear, colorless liquid found the the brain and spinal cord, acting like a cushion against sudden impact or injury to the brain or spinal cord[8]. This class is characterized by the data properties *cerebrospinalFluid* (`xsd:string`) and *lymphocytes* (`xsd:float`) which describe aspects of the test results.

[6]https://digestivehealth.ws/our-services/peg/
[7]https://www.hopkinsmedicine.org/tracheostomy/about/what.html
[8]https://medlineplus.gov/lab-tests/cerebrospinal-fluid-csf-analysis/

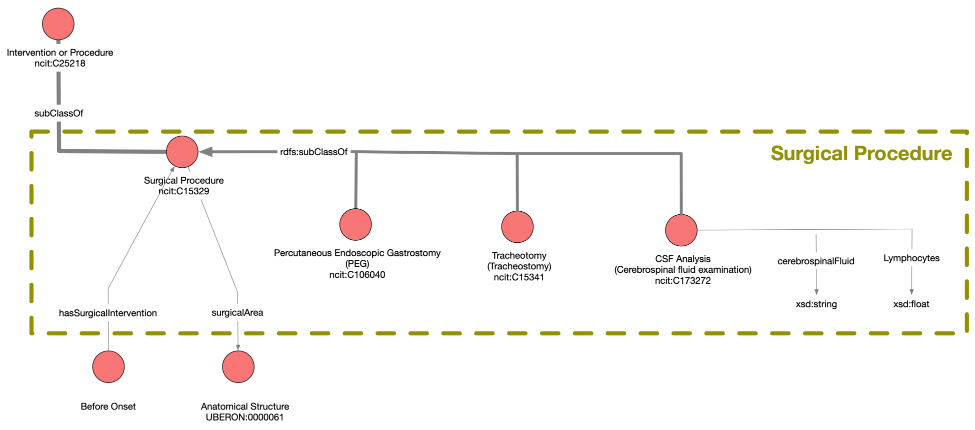


Figure 4.11: Surgical Procedure semantic area, sub-area of the Intervention or Procedure area.

4.2.7 Anatomical Structure

This area, shown in Figure 4.12, contains the class **Anatomical Structure** (`uberon:0000061`) which is the root of a taxonomy containing classes representing parts of the human anatomy used throughout the ontology (for example **Limbs**, **Bone Spine**, etc.).
At the design level, to avoid creating new instances of a certain location every time it is required, we define a named individual for each of the required anatomical structures: all resources that need to be associated with one or more anatomical structures will point to the same named individual.
In the Brainteaser Ontology, the **Anatomical Structure** class is the range of the following object object properties: *site* (connecting the **Onset** class), *traumaArea* (connecting the **Trauma** class), *area* (connecting the **Diagnostic Imaging** class), *surgicalArea* (connecting the **Surgical Procedure** class), and *symptomArea* (connecting the **Symptom** class).

4.2.8 Symptoms

The Symptoms semantic area, shown in Figure 4.13, contains classes describing the symptoms that may happen to a patient and be registered during an event.
The class **Event** is connected to the main class of this area, **Symptom** (`ncit:C4876`), through the *hasSymptom* object property. The **Symptom** class, in turn, is connected to the **Anatomical Structure** class through the *symptomArea* property, allowing to specify the area associated with a specific symptom.
The resources will rarely be of type **Symptom**, as it is too general to describe any real features of the patient's disease course. Among **Symptom** subclasses,

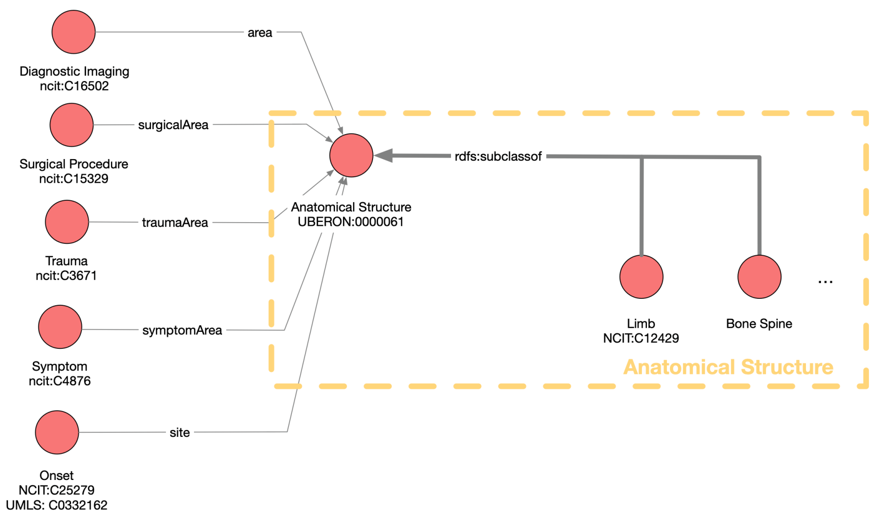


Figure 4.12: Anatomical Structure semantic area.

we list the **Fever** (`ncit:C3038`), **Nervous System Finding** (`ncit:C36280`) and its subclass, the **Fasciculation** (`ncit:C34606`). These symptoms are among the most relevant concerning the diseases modeled in this ontology.

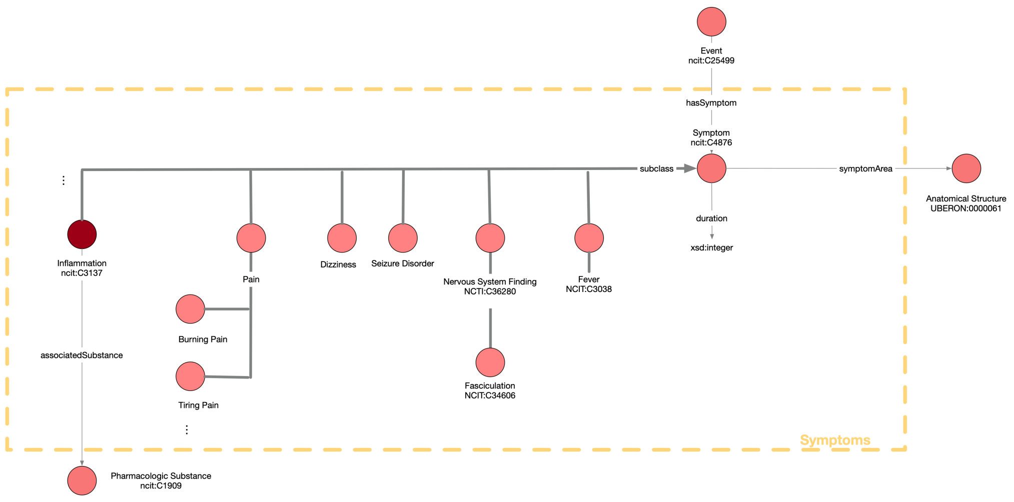


Figure 4.13: Symptoms semantic area.

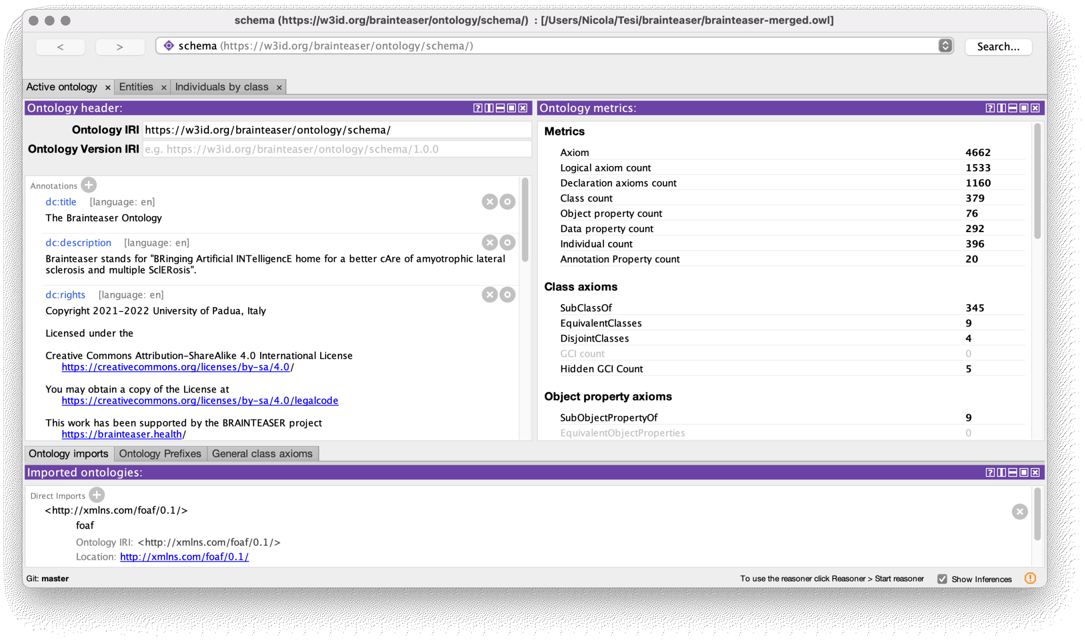
Chapter 5

Ontology and Data Mapper Development

In this chapter, in Section 5.1 we show the development of the Brainteaser Ontology using the Protégé software (introduced in Section 2.5) according to the graphical representation of the ontology given in Section 4.2. While in Section 5.2 we present the development of the ALS Data Mapper, which is a software that deals with mapping retrospective clinic data into an RDF dataset, in accordance with BO.

5.1 Ontology Development

Developing an otology using a classic text editor is not such an intuitive and quick process. For this reason, to develop the Brainteaser Ontology we used the Protégé software. As explained in Section 2.5, Protégé is an ontology editor that allows the development and management of ontologies in a very simple and clear way. Thanks to its intuitive graphical interface, it clearly speeds up the ontology creation process and helps minimize possible creation errors.
Figure 5.1 shows the main Protégé workspace, enclosed in the "Active ontology" tab, with the Brainteaser Ontology open within it. Through this tab, specifically in the "Ontology header" view, we defined the base IRI of the BO (`https://w3id.org/brainteaser/ontology/schema/`) and ontology metadata such as title (with the `dc:title` property), description (`dc:description`), authors (`dc:creator`), rights (`dc:right`), and version (`owl:versionInfo`). While the view "Ontology imports" of the tab "Active ontology", allows to import external ontologies. In our case, we imported the entire FOAF ontology.


schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]
schema (https://w3id.org/brainteaser/ontology/schema/)
Search...
Active ontology
Entities
Individuals by class
Ontology header:
Ontology metrics:
Ontology imports
Ontology Prefixes
General class axioms
Imported ontologies:
Git: master
To use the reasoner click Reasoner > Start reasoner
Show Inferences

Ontology IRI https://w3id.org/brainteaser/ontology/schema/
Ontology Version IRI e.g. https://w3id.org/brainteaser/ontology/schema/1.0.0

Annotations

dc:title [language: en]
The Brainteaser Ontology

dc:description [language: en]
Brainteaser stands for "BRinging Artificial INTelligencE home for a better cAre of amyotrophic lateral sclerosis and multiple SclERosis".

dc:rights [language: en]
Copyright 2021-2022 University of Padua, Italy

Licensed under the

Creative Commons Attribution-ShareAlike 4.0 International License
https://creativecommons.org/licenses/by-sa/4.0/

You may obtain a copy of the License at
https://creativecommons.org/licenses/by-sa/4.0/legalcode

This work has been supported by the BRAINTEASER project
https://brainteaser.health/

Metrics

Axiom	4662
Logical axiom count	1533
Declaration axioms count	1160
Class count	379
Object property count	76
Data property count	292
Individual count	396
Annotation Property count	20

Class axioms

SubClassOf	345
EquivalentClasses	9
DisjointClasses	4
GCI count	0
Hidden GCI Count	5

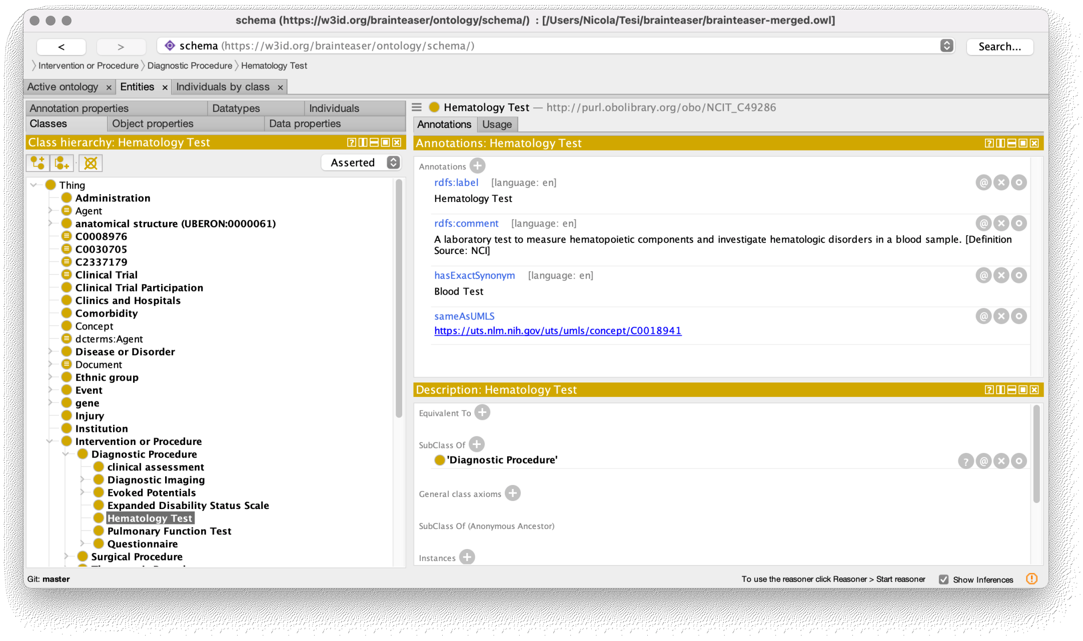
Object property axioms

SubObjectPropertyOf	9
EquivalentObjectProperties	0

Direct Imports
<http://xmlns.com/foaf/0.1/>
foaf
Ontology IRI: <http://xmlns.com/foaf/0.1/>
Location: http://xmlns.com/foaf/0.1/

Figure 5.1: The Brainteaser Ontology inside the Protégé "Active Ontology" tab.

Instead, the Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5 show different sub-tabs of the "Entities" tab, which is the most important part of Protégé where it is possible to manage classes, properties and individuals. In particular, the Figure 5.2 shows the "Classes" tab. Here we imported the single classes from the external ontologies and created all those new classes we defined. For each class, in the "Annotation" view, we reported the label (`rdfs:label`), the comment (`rdfs:comment`), and, if present, the possible synonyms (`oboInOwl:hasExactSynonym`) and reference to IDC-10 and UMLS ontologies (`:sameAs_ICD10` and `:sameAsUMLS`, annotation subproperties of `owl:sameAS`). While in the "Description" view we have defined for each class if it has equivalent classes (`Equivalent To`) or if it is subclass of another class (`SubClass Of`). Finally, in the "Class hierarchy" view, as the name suggests, the hierarchical structure of the ontology classes is shown, which is at the same time navigable. This view is useful to verify that the classes are related in the correct way.
The example shown in the Figure 5.2 shows the class `ncit:C49286` which has been assigned the label "Hematology Test", the description "A laboratory test to measure hematopoietic components and investigate hematologic disorders in a blood sample. [Definition Source: NCI]", the synonym "Blood Test" and the reference to `umls:C0018941`. In addition, we specified that it is a subclass of "Diagnostic Procedure" (`ncit:C18020`).


schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]
<
>
schema (https://w3id.org/brainteaser/ontology/schema/)
Search...
Intervention or Procedure 〉 Diagnostic Procedure 〉 Hematology Test
Active ontology ×
Entities ×
Individuals by class ×
Annotation properties
Datatypes
Individuals
Classes
Object properties
Data properties
Class hierarchy: Hematology Test
Asserted
Annotations
Usage
Annotations: Hematology Test
Description: Hematology Test
Git: master
To use the reasoner click Reasoner > Start reasoner
Show Inferences

- Thing
  - Administration
  - Agent
  - anatomical structure (UBERON:0000061)
  - C0008976
  - C0030705
  - C2337179
  - Clinical Trial
  - Clinical Trial Participation
  - Clinics and Hospitals
  - Comorbidity
  - Concept
  - dcterms:Agent
  - Disease or Disorder
  - Document
  - Ethnic group
  - Event
  - gene
  - Injury
  - Institution
  - Intervention or Procedure
    - Diagnostic Procedure
      - clinical assessment
      - Diagnostic Imaging
      - Evoked Potentials
      - Expanded Disability Status Scale
      - Hematology Test
      - Pulmonary Function Test
      - Questionnaire
    - Surgical Procedure

Hematology Test — http://purl.obolibrary.org/obo/NCIT_C49286

Annotations

rdfs:label [language: en]
Hematology Test

rdfs:comment [language: en]
A laboratory test to measure hematopoietic components and investigate hematologic disorders in a blood sample. [Definition Source: NCI]

hasExactSynonym [language: en]
Blood Test

sameAsUMLS
https://uts.nlm.nih.gov/uts/umls/concept/C0018941

Equivalent To

SubClass Of
'Diagnostic Procedure'

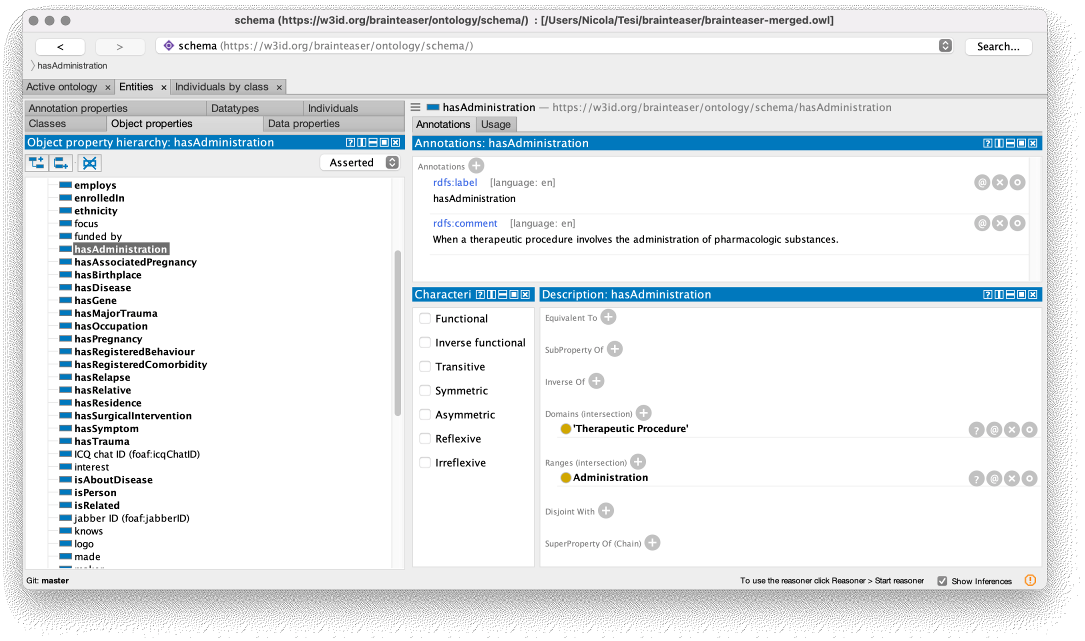
General class axioms

SubClass Of (Anonymous Ancestor)

Instances

Figure 5.2: The BO classes inside the Protégé "Entities" tab.

The Figure 5.3 presents the "Object properties" tab which includes all the object properties defined in the ontology. These are the properties that are responsible for linking the various classes in the ontology. As we can see, this image is similar to the "Classes" tab, in fact it presents the "Object property hierarchy" view to show the hierarchy of the various object properties and the "Annotations" and "Description" views to enter the information and properties of the defined object property. For each object property, in the "Annotations" view, we assigned a label (`rdfs:label`) and a comment (`rdfs:comment`), while in the "Description" view we defined its domain and range (`Domains` and `Ranges`). In addition, there is also another view, called "Characteristics" which allows to add a characteristic to the selected object property.
In the example in Figure 5.3, we defined an object property with IRI "`https://w3id.org/brainteaser/ontology/schema/hasAdministration`", label "hasAdministration", comment "When a therapeutic procedure involves the administration of pharmacologic substances.", domain the class "Therapeutic Procedure" (`ncit:C49236`) and range the class "Administration" (`ncit:C25409`).
The Figure 5.4 presents the "Data properties" tab. Here, as the name suggests, it is possible to find all the data properties of the ontology, i. e., all those properties that assign data values to resources. This tab is almost the same of the "Object properties" tab, the only thing that changes is that the range of data properties is



schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]

schema (https://w3id.org/brainteaser/ontology/schema/)

hasAdministration

Object property hierarchy: hasAdministration

employs
enrolledIn
ethnicity
focus
funded by
hasAdministration
hasAssociatedPregnancy
hasBirthplace
hasDisease
hasGene
hasMajorTrauma
hasOccupation
hasPregnancy
hasRegisteredBehaviour
hasRegisteredComorbidity
hasRelapse
hasRelative
hasResidence
hasSurgicalIntervention
hasSymptom
hasTrauma
ICQ chat ID (foaf:icqChatID)
interest
isAboutDisease
isPerson
isRelated
jabber ID (foaf:jabberID)
knows
logo
made

hasAdministration — https://w3id.org/brainteaser/ontology/schema/hasAdministration

Annotations: hasAdministration

Annotations
rdfs:label [language: en]
hasAdministration
rdfs:comment [language: en]
When a therapeutic procedure involves the administration of pharmacologic substances.

Characteri

☐ Functional
☐ Inverse functional
☐ Transitive
☐ Symmetric
☐ Asymmetric
☐ Reflexive
☐ Irreflexive

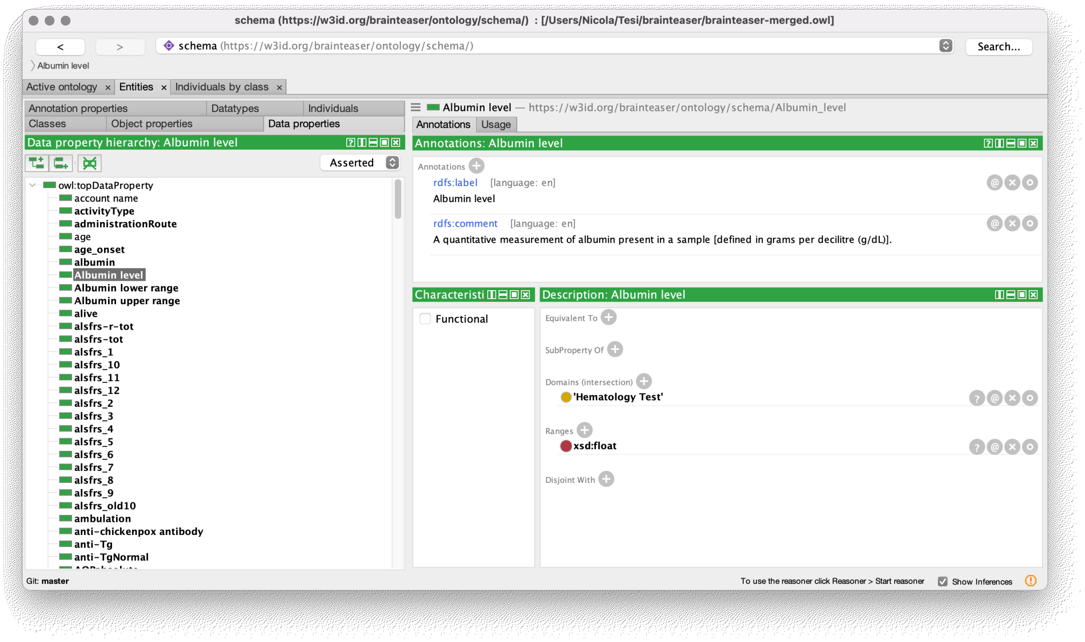
Description: hasAdministration

Equivalent To
SubProperty Of
Inverse Of
Domains (intersection)
'Therapeutic Procedure'
Ranges (intersection)
Administration
Disjoint With
SuperProperty Of (Chain)

To use the reasoner click Reasoner > Start reasoner ✓ Show Inferences

Figure 5.3: The BO object properties inside the Protégé "Entities" tab.

a datatype, and not a class as for the object properties. What we said for object properties is also valid for data properties, in fact we assign to each data property a label (`rdfs:label`) and a comment (`rdfs:comment`) in the "Annotation" view, while through the "Description" view we define the domain and the range of the selected property (`Domains` and `Ranges`).
The Figure 5.4 shows an example of a data property with IRI "`https://w3id.org/brainteaser/ontology/schema/Albumin_level`", label "Albumin level", comment "A quantitative measurement of albumin present in a sample [defined in grams per deciliter (g/dL)].", domain the class "Hematology Test" (`ncit:C49286`) and range the datatype `xsd:float`.
The Figure 5.5 instead shows the "Annotation properties" tab, which includes all those properties that allow us to encode information about the ontology. In particular in this tab we have imported the following annotation properties from external ontologies: `dcterms:title`, `dcterms:description`, `dcterms:creator`, used to add the title, description and creators of the ontology, and `oboInOwl:hasExactSynonym` used to store the possible synonyms of a class (as in the example of Figure 5.2, the class "Hematology Test" has synonym "Blood Test"). Here we have also defined two new annotation properties: `:sameAsUMLS` and `:sameAs_ICD10`, used to add reference to the UMLS and ICD-10 ontologies. For these two annotation properties, we also added a label and a comment



schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]

schema (https://w3id.org/brainteaser/ontology/schema/)

Albumin level

Data property hierarchy: Albumin level

owl:topDataProperty
account name
activityType
administrationRoute
age
age_onset
albumin
Albumin level
Albumin lower range
Albumin upper range
alive
alsfrs-r-tot
alsfrs-tot
alsfrs_1
alsfrs_10
alsfrs_11
alsfrs_12
alsfrs_2
alsfrs_3
alsfrs_4
alsfrs_5
alsfrs_6
alsfrs_7
alsfrs_8
alsfrs_9
alsfrs_old10
ambulation
anti-chickenpox antibody
anti-Tg
anti-TgNormal

Albumin level — https://w3id.org/brainteaser/ontology/schema/Albumin_level

Annotations: Albumin level

rdfs:label [language: en]
Albumin level

rdfs:comment [language: en]
A quantitative measurement of albumin present in a sample [defined in grams per decilitre (g/dL)].

Characteristics: Functional

Description: Albumin level

Domains (intersection): 'Hematology Test'

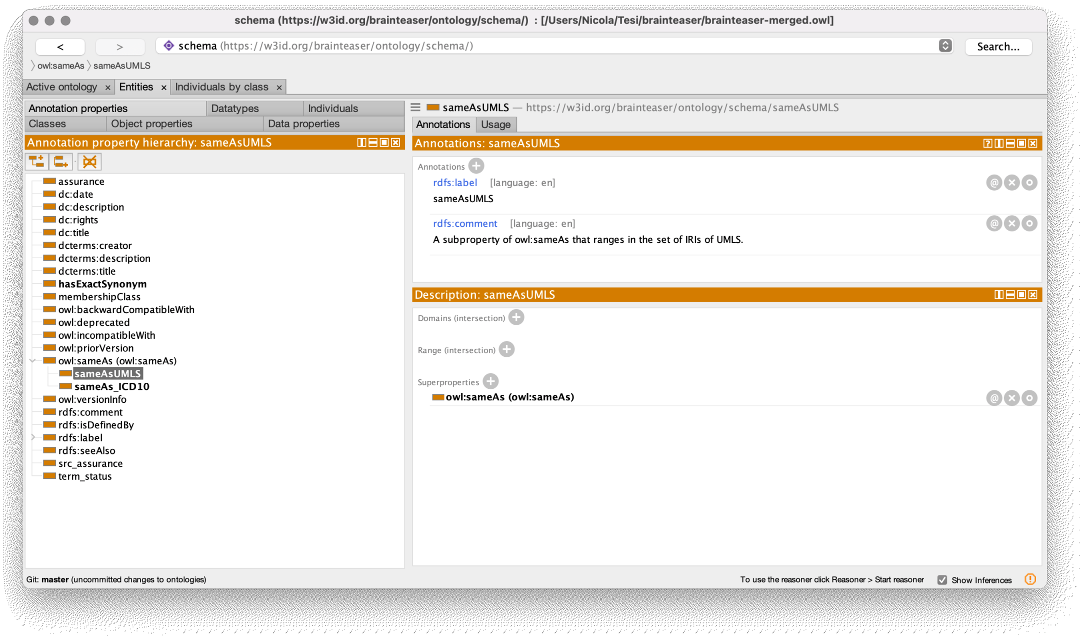
Ranges: xsd:float

Git: master

To use the reasoner click Reasoner > Start reasoner

Figure 5.4: The BO data properties inside the Protégé "Entities" tab.

(`rdfs:label` and `rdfs:comment`) in the "Annotation" view and defined that they are both subproperties of `owl:sameAs` (`Superproperties`) in the "Description" view.



schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]

schema (https://w3id.org/brainteaser/ontology/schema/)

owl:sameAs 〉 sameAsUMLS

Annotation property hierarchy: sameAsUMLS

assurance
dc:date
dc:description
dc:rights
dc:title
dcterms:creator
dcterms:description
dcterms:title
hasExactSynonym
membershipClass
owl:backwardCompatibleWith
owl:deprecated
owl:incompatibleWith
owl:priorVersion
owl:sameAs (owl:sameAs)
  sameAsUMLS
  sameAs_ICD10
owl:versionInfo
rdfs:comment
rdfs:isDefinedBy
rdfs:label
rdfs:seeAlso
src_assurance
term_status

sameAsUMLS — https://w3id.org/brainteaser/ontology/schema/sameAsUMLS

Annotations: sameAsUMLS

rdfs:label [language: en]
sameAsUMLS

rdfs:comment [language: en]
A subproperty of owl:sameAs that ranges in the set of IRIs of UMLS.

Description: sameAsUMLS

Domains (intersection)

Range (intersection)

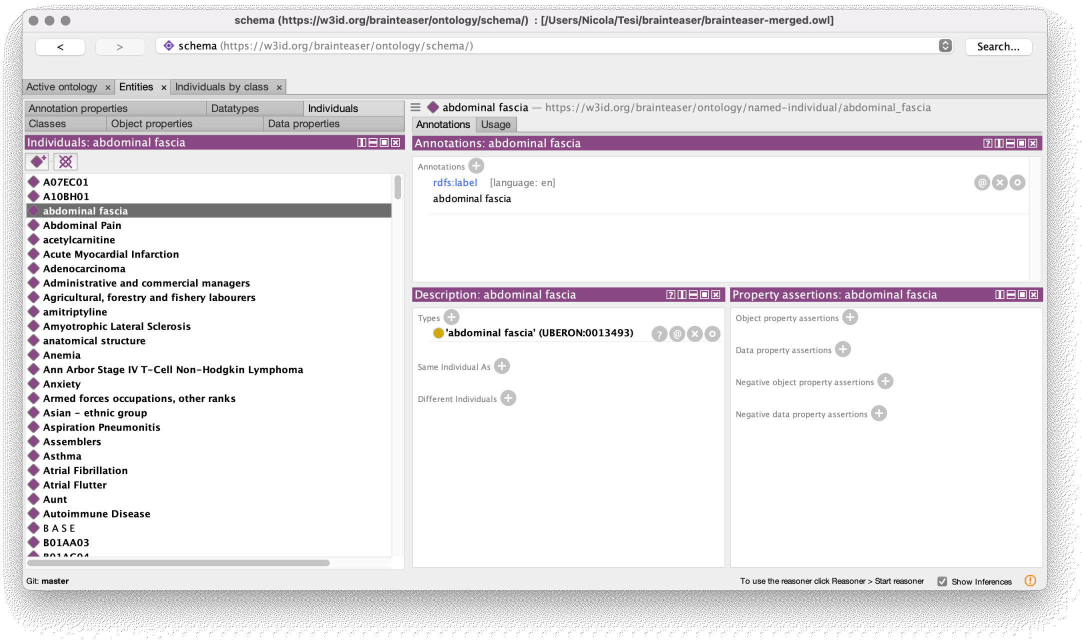
Superproperties: owl:sameAs (owl:sameAs)

Git: master (uncommitted changes to ontologies)

To use the reasoner click Reasoner > Start reasoner

Figure 5.5: The BO annotation properties inside the Protégé "Entities" tab.

Finally, in the Figure 5.6 we show the tab "Individuals", where we have defined the named individuals of some classes, according to what is explained in the Subsection 4.1.4. It is important to remember that named individuals have a different IRI prefix than classes and properties, and it is: "`https://w3id.org/brainteaser/ontology/named-individual/`". For each named individual we have associated a label (`rdfs:label`) through the "Annotation" tab and we have defined its type (`Types`) in the "Description" tab. In the example in Figure 5.6 we see defined the named individual with IRI "`https://w3id.org/brainteaser/ontology/named-individual/abdominal_fascia`", label "abdominal fascia" and type the class "abdominal fascia" (`uberon:0013493`).


schema (https://w3id.org/brainteaser/ontology/schema/) : [/Users/Nicola/Tesi/brainteaser/brainteaser-merged.owl]
<
>
schema (https://w3id.org/brainteaser/ontology/schema/)
Search...
Active ontology
Entities
Individuals by class
Annotation properties
Datatypes
Individuals
Classes
Object properties
Data properties
Individuals: abdominal fascia
abdominal fascia — https://w3id.org/brainteaser/ontology/named-individual/abdominal_fascia
Annotations
Usage
Annotations: abdominal fascia
Description: abdominal fascia
Property assertions: abdominal fascia
Git: master
To use the reasoner click Reasoner > Start reasoner
Show Inferences

A07EC01
A10BH01
abdominal fascia
Abdominal Pain
acetylcarnitine
Acute Myocardial Infarction
Adenocarcinoma
Administrative and commercial managers
Agricultural, forestry and fishery labourers
amitriptyline
Amyotrophic Lateral Sclerosis
anatomical structure
Anemia
Ann Arbor Stage IV T-Cell Non-Hodgkin Lymphoma
Anxiety
Armed forces occupations, other ranks
Asian - ethnic group
Aspiration Pneumonitis
Assemblers
Asthma
Atrial Fibrillation
Atrial Flutter
Aunt
Autoimmune Disease
B A S E
B01AA03

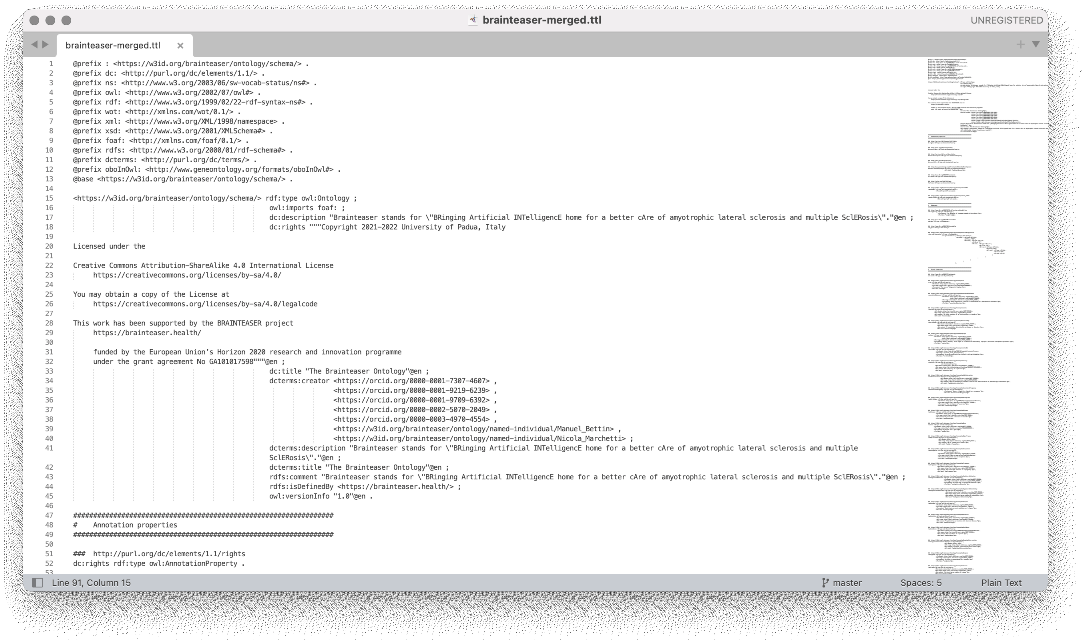
Annotations
rdfs:label [language: en]
abdominal fascia

Types
'abdominal fascia' (UBERON:0013493)
Same Individual As
Different Individuals

Object property assertions
Data property assertions
Negative object property assertions
Negative data property assertions

Figure 5.6: The BO named individuals inside the Protégé "Entities" tab.

As final note, Protégé allows to save the developed ontology in multiple formats (RDF/XML, Turtle, JSON-LD, etc.); a snapshot of the Brainteaser ontology in Turtle format opened with a text editor is shown in Figure 5.7. This image is not only intended to show the representation of the ontology in the Turtle format, but also to emphasize the fact that, in the development of an ontology, the use of an ontology editor like Protégé represents a real advantage over a development done using a classic text editor.


brainteaser-merged.ttl

```
@prefix : <https://w3id.org/brainteaser/ontology/schema/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix ns: <http://www.w3.org/2003/06/sw-vocab-status/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix wot: <http://xmlns.com/wot/0.1/> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix oboInOwl: <http://www.geneontology.org/formats/oboInOwl#> .
@base <https://w3id.org/brainteaser/ontology/schema/> .

<https://w3id.org/brainteaser/ontology/schema/> rdf:type owl:Ontology ;
                                                 owl:imports foaf: ;
                                                 dc:description "Brainteaser stands for \"BRinging Artificial INTelligencE home for a better cAre of amyotrophic lateral sclerosis and multiple SclERosis\"."@en ;
                                                 dc:rights """Copyright 2021-2022 University of Padua, Italy

Licensed under the

Creative Commons Attribution-ShareAlike 4.0 International License
     https://creativecommons.org/licenses/by-sa/4.0/

You may obtain a copy of the License at
     https://creativecommons.org/licenses/by-sa/4.0/legalcode

This work has been supported by the BRAINTEASER project
     https://brainteaser.health/

     funded by the European Union’s Horizon 2020 research and innovation programme
     under the grant agreement No GA101017598"""@en ;
                                                 dc:title "The Brainteaser Ontology"@en ;
                                                 dcterms:creator <https://orcid.org/0000-0001-7307-4607> ,
                                                                 <https://orcid.org/0000-0001-9219-6239> ,
                                                                 <https://orcid.org/0000-0001-9709-6392> ,
                                                                 <https://orcid.org/0000-0002-5070-2049> ,
                                                                 <https://orcid.org/0000-0003-4970-4554> ,
                                                                 <https://w3id.org/brainteaser/ontology/named-individual/Manuel_Bettin> ,
                                                                 <https://w3id.org/brainteaser/ontology/named-individual/Nicola_Marchetti> ;
                                                 dcterms:description "Brainteaser stands for \"BRinging Artificial INTelligencE home for a better cAre of amyotrophic lateral sclerosis and multiple SclERosis\"."@en ;
                                                 dcterms:title "The Brainteaser Ontology"@en ;
                                                 rdfs:comment "Brainteaser stands for \"BRinging Artificial INTelligencE home for a better cAre of amyotrophic lateral sclerosis and multiple SclERosis\"."@en ;
                                                 rdfs:isDefinedBy <https://brainteaser.health/> ;
                                                 owl:versionInfo "1.0"@en .

#################################################################
#    Annotation properties
#################################################################

###  http://purl.org/dc/elements/1.1/rights
dc:rights rdf:type owl:AnnotationProperty .
```

Figure 5.7: A snapshot of the Brainteaser Ontology in the Turtle format.

5.2 ALS Data Mapper Development

In this section we present the development of the ALS Data Mapper: a software that aims to map restrospective data of ALS clinics into an RDF dataset, all in accordance with the Brainteaser Ontology. Specifically, in the Subsection 5.2.1 we present some general concepts about the development of the Data Mapper and in the Subsection 5.2.2 we explain the characteristics of the input data provided by the clinics. The Subsection 5.2.3 covers the cleaning operations on the input data, called pre-processing operations, while in the Subsection 5.2.4 we present the mapping phase. In the Subsection 5.2.5 we illustrate the files returned in output from the Data Mapper and in the Subsection 5.2.6 we present an example of data mapping. Finally, in the Subsection 5.2.7 we explain how it was possible to export in CSV the mapped data.
All the source code is available at `https://bitbucket.org/brainteaser-health/mappers/src/master/ALS/`.

5.2.1 Data Mapper overview

The ALS data mapper is the part of our software architecture dedicated to mapping retrospective clinical data of ALS patients, provided by partner clinics, into a structured RDF graph compliant with our ontology. The retrospective data come

from different clinics, each of which has adopted its own notation and standards for representing and managing knowledge. So two different clinics can assert the same knowledge in two different ways, depending on their data management system. Due to this, diffrent data mappers are needed to map data from different sources in order to standardize all the knowledge provided by clinical partners and structure it according to a graph-based model.
To reach this goal we designed a modular architecture (shown in the Figure 5.8) that in the first phase involves the creation of a dedicated module, called Data Mapper, for each partner clinic in order to correctly map its input data into an RDF graph. In detail, each Data Mapper reads the respective dataset as input, performs some pre-processing operations to clean the data, and then maps the read input data into an RDF graph. In this way, a single RDF graph is built for each dataset. Next, we merge the single graphs into a global RDF graph and as final step we serialize this graph in different formats (Turtle, RDF/XML, JSON-LD). In addition, there is also a module to export data in CSV format from the RDF graph. Such data in CSV is useful for teams involved in developing Artificial Intelligence algorithms for prediction purposes.
Each new resource that is introduced into the graph is identified by the URI prefix "https://w3id.org/brainteaser/ontology/resource/" plus a unique local ID generated during the mapping phase. Data Mappers are completely developed using the Python programming language because it allows us to use many useful libraries for our purposes and allows us to write the code in a compact and fast way. The libraries/modules used for Data Mapper development are:

- **pandas**: for input data manipulation;
- **rdflib**: for managing and serializing RDF graphs;
- **uuid**: for generating UUID codes for resource IDs;
- **mmh3**: for hashing the resource IDs;
- **csv**: for writing data into a CSV file;
- **pathlib**: for managing the file path;
- **datetime**: for managing dates;
- **tabulate**: for writing data in tabular format into txt files;
- **sys**: for redirecting standard output to a file.

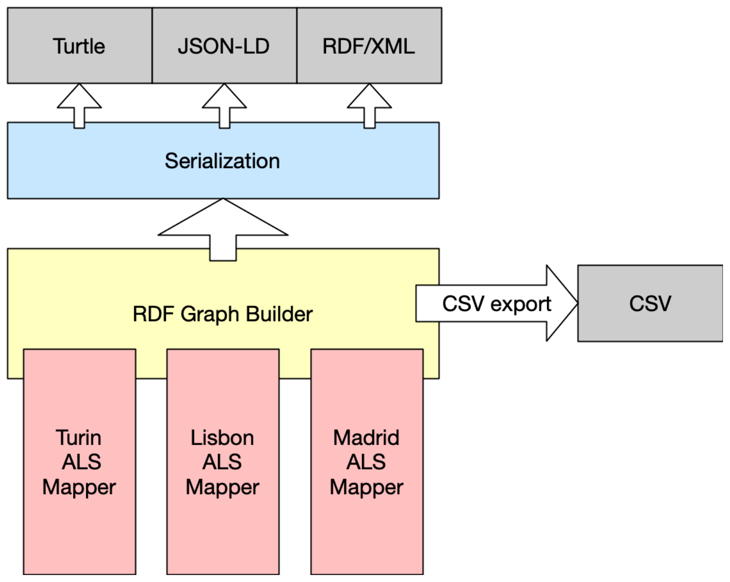


Figure 5.8: Data Mapper architecture.

5.2.2 Input data

The anonymized retrospective datasets of ALS patients are provided in files with .xlsx extension by the "Instituto de medicina molecolare João Lobo Antunes" (iMM, Portugal), "Universitá degli Studi di Torino" (UNITO, Italy), and "Servicio Madrileño de Salud" (SERMAS, Spain).
These datasets are generally structured as follows:

- A "static vars" sheet that includes:
  - Personal data;
  - Before onset data;
  - Onset data;
  - Diagnosis data;
  - Surgical and therapeutic procedures;
  - Clinical assessment data;
  - Diseases and disorders;
  - Gene mutations;
- "ALSFRS" sheet that includes data from the ALSFRS/ALSFRS-R questionnaire;
- "%FVC" sheet that includes data of the pulmonary function test;

- “OWD vars” sheet (only for iMM and SERMAS) that includes:
  - Blood test data;
  - Smoking behavior data;
  - Trauma and surgery data;

In Data Mappers, the input reading of datasets in *xlsx* format is done by exploiting the *read_excel* function of the *pandas* library. In the Code Snippet 5.1 we present an example of how we used this function.

```
staticVars = pd.read_excel(lisbonPath, sheet_name="Static Vars",
 index_col="REF", parse_dates=["Birth_year"], na_values=[" ", "NA"
 , "NaT"])
```

Code Snippet 5.1: An example of use of the pandas read_excel function

In particular, this function converts the excel sheet indicated in `sheet_name` (e.g. “Static Vars”) contained in the input file (which has a precise path, such as *lisbonPath* in our case) into a pandas DataFrame, which is a table-like structure. The parameter `index_col` is used to define a column of the excel sheet as index of the DataFrame (in the example the column with label “REF”), while the use of `parse_dates` inside the function `read_excel` allows to parse an integer/string value as datetime (e.g. “Birth_year”). Moreover through `na_values` it’s possible to declare some non-null values as null (for example “NA”), and this feature is really useful because it already allows to exclude some values that wouldn’t be useful in the mapping phase. So this is the function we used to read in input all those excel sheets useful for mapping purposes.

5.2.3 Pre-processing operations

Since we do not have the certainty that the data read in input and saved in the respective DataFrame are perfectly correct, we have identified the need to perform pre-processing operations, which are operations that try to clean all those data that are dirty. In other words, it is a matter of verifying that the data is consistent with its domain and therefore eliminating all data that does not meet this requirement. These operations are performed using pandas tools, such as the `iterrows` function to scan the DataFrame row by row, the `loc` to access a group of rows or columns, or the `isna` function to check for null values in cells, and Python operators, such as arithmetic, comparison, logical and membership operators to perform operations on some subsets of data. We also used the csv module to create new CSV files from the DataFrames generated with the input datasets. In detail, the pre-processing operations are the following:

1. Get correct ALSFRS-R tests: this is an operation that aims to return in output a new CSV file containing only all those ALSFRS-R tests that are correct. In other words, starting from the DataFrame relative to the input excel sheet "ALSFRS", we exclude duplicate tests, tests per patients whose dates are not in the correct sequence and tests with missing data (for the purposes of the project we also eliminate the "old" version of the ALSFRS tests). In this way we obtain a new dataset in CSV format with all the complete ALSFRS-R tests (i.e. those that have all the answers to the twelve questions and the date of the test). The correctness of the score/subscore sums of these tests is not checked at this stage, but will be checked during the mapper phase, with possible correction in case of wrong sums. The new CSV files will then be read as input with the pandas function `read_csv`, which works the same way as the `read_excel` function presented in Subsection 5.2.2;

2. Get correct pulmonary function tests: is an operation similar to the previous one, described in item 1. The purpose of this operation is to analyze the DataFrame corresponding to the input excel sheet "%FVC" and eliminate all those tests that are duplicated, tests per patient that are not in the correct sequence with respect to the dates, and tests that are missing the date or the result. This will create a new CSV file containing all those pulmonary function tests with all the correct data. This CSV file will then be read as input with the function `read_csv` of pandas;

3. Check "static vars" dates: this operation analyzes the onset date, the diagnosis date and, if the patient is dead, the death date. In detail, it must be verified that the dates are not null and that the onset date precedes the diagnosis date and, in turn, that the diagnosis date precedes the death date (if the patient is dead). At the end of this operation the list of patient IDs that satisfy the following requirements is returned in output. Thus the patients with null dates will be eliminated;

4. Get patient IDs with ALSFRS-R tests: with this operation we obtain the list of IDs of patients who have undergone at least one ALSFRS-R test;

5. Check "static vars" procedures: this pre-processing operation concerns NIV

(Non-invasive ventilation), PEG (Percutaneous endoscopic gastrostomy) and Tracheotomy events. In detail for each patient with at least one ALS-FRS test (the list of IDs computed in item 4) it is checked that if they have at least one of these events (NIV, PEG or Tracheotomy) the date of these events is greater than the date of the first ALSFRS-R and, if the patient died, less than the date of death. From this operation, the list of patient IDs who have the events that meet this requirement or who have not experienced these events is returned as output;

6. Check date of death with date of last visit: with this operation we want to make sure that, in dead patients, the date of death is the date after any other event. To do this we verify that the date of last visit is less than the date of death, where the date of last visit is calculated by taking the maximum date between the ALSFRS-R and pulmonary function tests. This operation returns the ID list of patients who are alive or have death as the last event;

7. Get patient IDs to be mapped: this operation, as shown in Code Snippet 5.2, must perform the intersection of the lists of IDs obtained in items 3, 4, 5 and 6. This will output the final list of patient IDs to be mapped, which contains all those subjects that passed the checks explained above and therefore have a portion of correct data.

```python
def getCorrectPatient(staticVars, alsfrs, fvc,
    patientWithCorrectDateList, patientWithALSFRS,
    patientWithCorrectEvent, patientAliveOrWithCorrectDeath):
    # initialize list
    correctList = []
    incorrectList = []
    # counter for ALSFRS and FVC
    alsfrsCounter = 0
    fvcCounter = 0
    # get unique IDs from staticVars
    idStaticVars = staticVars.index.unique()
    # compute the intersection between four groups
    for value in idStaticVars:
        if (value in patientWithCorrectDateList) and (value in
    patientWithALSFRS) and (value in patientWithCorrectEvent) and (
    value in patientAliveOrWithCorrectDeath):
            correctList.append(value)
        else:
                incorrectList.append(value)
    # get the ALSFRS number for these patients
    for value in correctList:
        alsfrsCounter += len(alsfrs.loc[alsfrs.index == value])
        fvcCounter += len(fvc.loc[fvc.index == value])
    # ...
    # return the list
    return correctList
```

Code Snippet 5.2: Developed function to get the final list of IDs that need to be mapped.

These are generally the main pre-processing operations, but they are not the only ones, in fact even in the mapping phase checks are made on individual data to verify that they are consistent with their domain.
Furthermore, all the functions developed to perform all these pre-processing operations return in output, precisely in a log file, all those visits and patients who have been excluded from the mapping phase.
In conclusion, these pre-processing operations are essential to avoid mapping those individuals who have fundamentally flawed data or who do not have sufficient visits or data for the purposes of the project. Some results obtained from this pre-processing phase are shown in the Chapter 6.

5.2.4 Mapping phase

The Data Mappers after performing the pre-processing operations described in the Subsection 5.2.3, proceed to initialize the RDF graph on which to append the RDF triples generated during the data mapping. As can be seen in the Code Snippet 5.3, we defined a utility function, called *initializeGraph*, to create the RDF graph (via the `Graph` class of rdflib) and associate with it a set of namespaces, i.e. the URI prefixes of the external ontologies used, using the `bind` function of rdflib.

```
def initializeGraph():
    # create the graph
    graph = Graph()
    # bind the imported namespaces (from rdflib.namespace) to a
   prefix for more readable output
    graph.bind("foaf", FOAF)
    graph.bind("xsd", XSD)
    graph.bind("rdfs", RDFS)
    graph.bind("rdf", RDF)
```

```python
    # get vars from namespaces.py by excluding python vars (e.g. __init__ ) and bind the other namespaces
    namespaceList = vars(ns).items()
    for x in (namespaceList):
        if not(x[0].startswith('__') or x[0].startswith('_') or x[0] == ('Namespace')):
            # bind the other namespaces to a prefix for more readable output
            graph.bind(str(x[0]), x[1])
    # return the updated graph
    return graph
```

Code Snippet 5.3: Initialize RDF graph function

Once the RDF graph has been initialized, the Data Mappers proceed to read the DataFrames row by row through the pandas function `iterrows`. For each row of a DataFrame, we first check that its ID is included in the list of IDs to be mapped obtained in the preprocessing phase (Subsection 5.2.3), and if this check is successful we use a set of mapping functions to map all the data contained in the DataFrame.

Before presenting the mapping functions in detail, we first show two fundamental functions used within them. The first is a function we defined that allows generating the URIs of resources that are entered into the RDF graph. This function is shown in Code Snippet 5.4 and presents two different ways of generating the URIs of resources:

- For all the resources that already have an ID in the DataFrame, this ID is initially concatenated with two strings, one relating to the clinic to which it belongs and one relating to the type of resource. This allows to avoid collisions between IDs of different clinics. The string resulting from the concatenation will then be hashed using the `hash128` function of the mmh3 library, which performs a 128-bit hashing. This ID is then converted to hexadecimal (hex) and appended to the URI prefix "https://w3id.org/brainteaser/ontology/resource/" (ns.BTO_resource). Finally, the resulting string, i.e. the one composed of the URI prefix plus the hexadecimal ID, is used as an input parameter for the constructor of the `URIRef` class of rdflib, which has the task of generating the URI of the resource;

- On the other hand, for those resources that do not have an original ID, we use the function `uuid4` of the uuid library to generate a random UUID

(Universally Unique IDentifier). To standardize the way resource IDs are represented, we perform the same steps as in the previous point. In a few words, we execute the function `hash128` to hash the UUID and transform the obtained value into hexadecimal. Finally the URI is generated using the class `URIRef` class of rdflib on the string consisting of the URI prefix of the Brainteaser resources plus the obtained hexadecimal UUID.

So, at each execution of the Data Mapper the function *generateURI* will generate the same URI for all those resources that originally have an ID. While a resource that does not have an ID will have a different URI each time the Data Mapper is executed, this is due to the fact that the URI is generated randomly.

```python
def generateURI(hospitalName = None, className = None, resourceId = None):
    if (hospitalName != None and className != None and resourceId != None):
        # create the uri related to an entity for which we have an id in the original dataset
        # in this way in the future we can obtain again its uri id by calling this function
        elementURI = str(hospitalName)+"_" + str(resourceId) + "_"+ str(className)
        elementURI = URIRef(ns.BTO_resource[hex(int(mmh3.hash128(elementURI, signed=False, seed=42)))])
    else:
        # if in the original dataset we don't have an id for the resource we randomly generate it on the fly
        elementURI = URIRef(ns.BTO_resource[hex(int(mmh3.hash128(str(uuid.uuid4()), signed=False, seed=42)))])
    # return the generated URI
    return elementURI
```

Code Snippet 5.4: Generate URI function for the Brainteaser resources

The other important function used within the mapping functions is the `add` function of the rdflib library, which is responsible for adding RDF triples to the RDF graph. For example in the Code Snippet 5.5 it is possible to see that the function `add` has been used to add the triple that attributes the type of the instance (in our example the class “Diagnosis”), the triple that has as object a literal (to append the diagnosis date to the instance via the data property “startDate”), and also the triple that has as object another instance (to connect the instance of type “Patient” to the instance of type “Diagnosis” via the object property “undergo”).

The variety of input data required the development of different functions to correctly map all of this data. Furthermore since some data are represented differently from clinic to clinic we need to develop different versions of the same mapping function. So, for each type of data represented with the same standard for all clinics we have a common mapping function for all Data Mappers (Turin, Lisbon, Madrid), while for each type of data represented differently by each clinic we have developed different functions in order to standardize the way in which these data are mapped.
Based on the structure of input data presented in Subsection 5.2.2 we proceed to describe in a general way some developed mapping functions, subdividing them according to the DataFrame on which they work:

- "Static vars" DataFrame:
  - *addPatient*: which creates an instance of the "Patient" class (using its original ID) on which all its information will then be appended;
  - *addClinicalTrial*: used to add the patient's clinical trial information;
  - *addBirthYear*: used to store the year of birth of a patient;
  - *addGender*: allows to record the sex of the patient;
  - *addEthnicity*: used to map the patient's ethnicity to the corresponding named individual of the "Ethnic Group" class;
  - *addPatientStatus*: function whose purpose is to save the status (alive or dead) of the patient and, if this patient is dead, to record the date of death;
  - *addDiagnosis*: this is the function presented in Code Snippet 5.5. It is responsible for adding the "Diagnosis" event with its date to the patient;
  - *addOnset*: function that maps all data referring to the "Onset" event, such as age (in years) at onset, date, type and location of onset;
  - *addHeightWeightWeightloss*: used to map the patient's weight and height data that are usually collected at the first visit;
  - *addBeforeOnset*: for recording all information/events that occurred before the onset, such as weight before the first symptoms;
  - *addRelative*: used to add information about the patient's family history with respect to ALS disease;

  - *addOccupation*: used to map the patient's occupation to the corresponding named individual in the "Occupation" class;
  - *addRetiredAtDiagnosis*: for indicating whether the patient was retired at the time of diagnosis;
  - *addNIV, addTracheostomy, addPEG*: function that stores information about NIV, PEG, and Tracheotomy procedures performed on the patient;
  - *addSmoking*: if the patient is a smoker, this function stores that behavior;
  - *addDisease*: allows to map all diseases of the patient to the corresponding named individual of class "Disease or Disorder";
  - *addGene*: used to save information about genetic mutations that have occurred on a patient;

- "ALSFRS-R" DataFrame:
  - *addQuestionnaire*: this function, in addition to checking and correcting any incorrect sums, maps data of ALSFRS-R tests undertaken by the patient;
- "%FVC" DataFrame:
  - *addSpiro*: function that allows to record the results of pulmonary tests performed by the patient;
- "OWD vars" DataFrame:
  - *addBloodTest*: used to map all blood test values performed at diagnosis on the patient;
  - *addTraumaAndInterventionLast5Years* and *addTraumaAndInterventionMoreThan5Years*: for appending all information about the patient's trauma and surgeries that occurred before onset (in the previous five years or more);
  - *addSmokingOWD*: function that adds some information about the patient's smoking behavior, such as start and end year, number of cigarettes per day etc.

```python
def addDiagnosis(graph, patientID, patientURI, dateOfDiagnosis):
    # initialize the diagnosis event with 999 for those who do not have the date of diagnosis
    diagnosis = '999'
    # check if date is not null
    if not pd.isna(dateOfDiagnosis):
        # get diagnosis URI
        diagnosis = utils_als.generateURI()
        # add diagnosis to graph
        graph.add((diagnosis, RDF.type, classes.diagnosis))
        # add start date to diagnosis
        graph.add((diagnosis, ns.BTO_schema['startDate'], Literal(dateOfDiagnosis.date(), datatype=XSD.date)))
        # link diagnosis to the patient
        graph.add((patientURI, ns.BTO_schema['undergo'], diagnosis))
    else:
        print('[WARNING] Date of diagnosis not given for patient ID: %s, (Cannot add the diagnosis to the patient!)' % (patientID))
    # return the updated graph and the diagnosis URI
    return graph, diagnosis
```

Code Snippet 5.5: An example of a mapping function

To conclude, it is important to explain that all mapping functions, through the use of the pandas library tools and Python operators (arithmetic, comparison, logical and membership), check if the data they are going to map is consistent with its domain (for example, in Code Snippet 5.5 it is checked that the date of diagnosis is not null). If the check is positive the data is effectively mapped, but if it is negative the corresponding error is written in the log file. In this way all possible errors due to this phase are gathered in the log file.

5.2.5 Output files

The Data Mapper execution ends with the serialization of the RDF graph obtained in the previous mapping phase. The serialization is done by means of the *serializeGraph* function (shown in the Code Snippet 5.6) developed as a utility function. This function exploits within itself the `serialize` function of the library rdflib, which allows to serialize a graph RDF in multiple formats. Through its input parameter “format” it is possible to declare which format of serialization to use. In this way at the end of the serialization we obtain in output a file with the declared format. In our Data Mapper we have mainly used three serialization formats, that is Turtle, RDF/XML and JSON-LD.

```python
def serializeGraph(graph, savePath, serializationType):
    print("----- start serialization (" + str(serializationType)  +
   ") -----")
    with open(savePath, 'w', encoding="utf-8") as file:
        file.write(graph.serialize(format=serializationType))
    print("----- end serialization (" + str(serializationType)  + ")
    -----")

```

Code Snippet 5.6: Function that performs the serialization of an RDF graph.

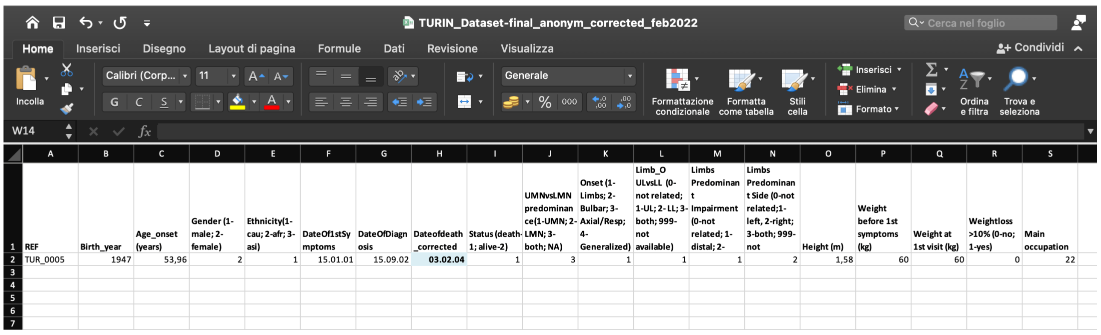
So the output files returned by the ALS Data Mapper at the end of the execution are the following:

- three different files for each clinic with extensions .ttl, .rdf, and .json that come from the serialization of the RDF dataset in Turtle, RDF/XML, and JSON-LD formats;
- three different files with extensions .ttl, .rdf and .json concerning the “merged” RDF dataset obtained by merging the RDF datasets of the different clinics and serialized in Turtle, RDF/XML and JSON-LD formats;
- a log file with a .txt extension for each clinic, containing data from pre-processing operations and errors that occurred in the mapping phase.

To conclude, an example of an RDF dataset serialized in the Turtle format is shown in the Code Snippet 5.7 of the Subsection 5.2.6.

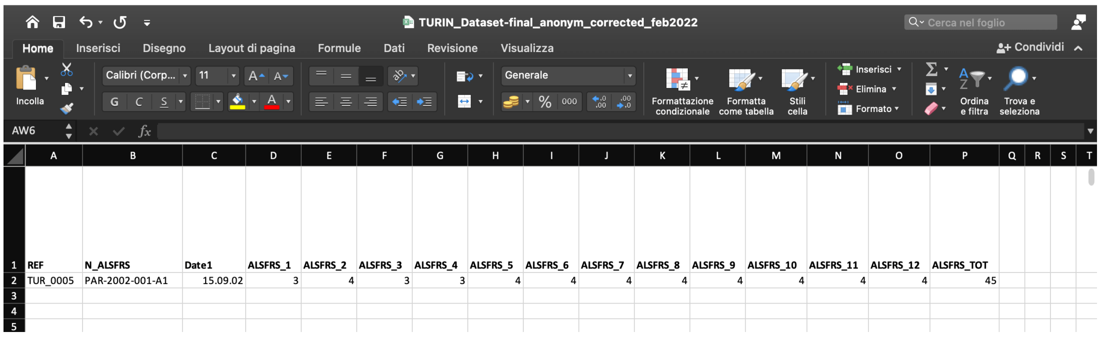
5.2.6 Data mapping example

In this subsection we present an example of data mapping performed on a patient taken from the Turin dataset. In the Figure 5.9 we present the patient’s row related to the “static vars” sheet, while the Figure 5.10 and Figure 5.11 represent an ALSFRS-R test and a pulmonary function test taken from the “ALSFRS” and “%FVC” sheets, respectively. This is a patient that has passed all the checks done in the pre-processing phase and therefore is effectively a patient that can be mapped. So, using the ALS Data Mapper of Turin, which exploits the mapping functions defined in Subsection 5.2.4, we proceed to map all its data in the RDF graph.
The resulting RDF graph obtained from the mapping phase is shown in the Figure 5.12 and Figure 5.13. We decided to split the graph representation into


TURIN_Dataset-final_anonym_corrected_feb2022

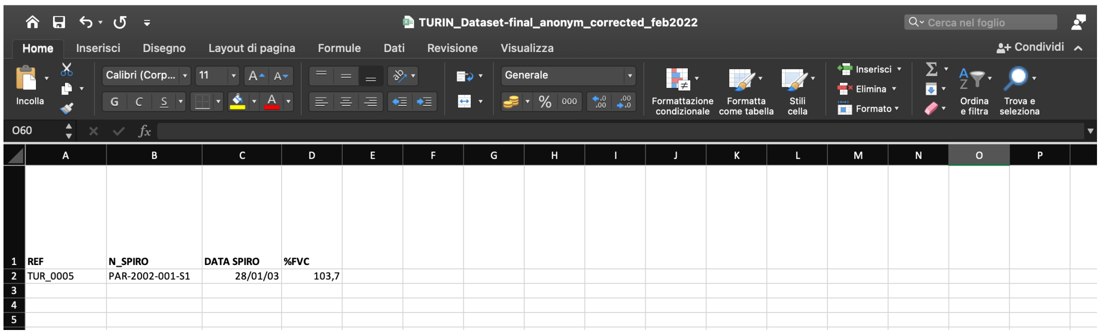
REF	Birth_year	Age_onset (years)	Gender (1-male; 2-female)	Ethnicity(1-cau; 2-afr; 3-asi)	DateOf1stSymptoms	DateOfDiagnosis	Dateofdeath_corrected	Status (death-1; alive-2)	UMNvsLMN predominance(1-UMN; 2-LMN; 3-both; NA)	Onset (1-Limbs; 2-Bulbar; 3-Axial/Resp; 4-Generalized)	Limb_O ULvsLL (0-not related; 1-UL; 2-LL; 3-both; 999-not available)	Limbs Predominant Impairment (0-not related; 1-distal; 2-	Limbs Predominant Side (0-not related;1-left, 2-right; 3-both; 999-not	Height (m)	Weight before 1st symptoms (kg)	Weight at 1st visit (kg)	Weightloss >10% (0-no; 1-yes)	Main occupation
TUR_0005	1947	53,96	2	1	15.01.01	15.09.02	**03.02.04**	1	3	1	1	1	2	1,58	60	60	0	22

Figure 5.9: An example row from the “static vars” sheet of the Turin dataset.


TURIN_Dataset-final_anonym_corrected_feb2022

REF	N_ALSFRS	Date1	ALSFRS_1	ALSFRS_2	ALSFRS_3	ALSFRS_4	ALSFRS_5	ALSFRS_6	ALSFRS_7	ALSFRS_8	ALSFRS_9	ALSFRS_10	ALSFRS_11	ALSFRS_12	ALSFRS_TOT
TUR_0005	PAR-2002-001-A1	15.09.02	3	4	3	3	4	4	4	4	4	4	4	4	45

Figure 5.10: An example row from the “ALSFRS” sheet of the Turin dataset.


TURIN_Dataset-final_anonym_corrected_feb2022

REF	N_SPIRO	DATA SPIRO	%FVC
TUR_0005	PAR-2002-001-S1	28/01/03	103,7

Figure 5.11: An example row from the “%FVC” sheet of the Turin dataset.

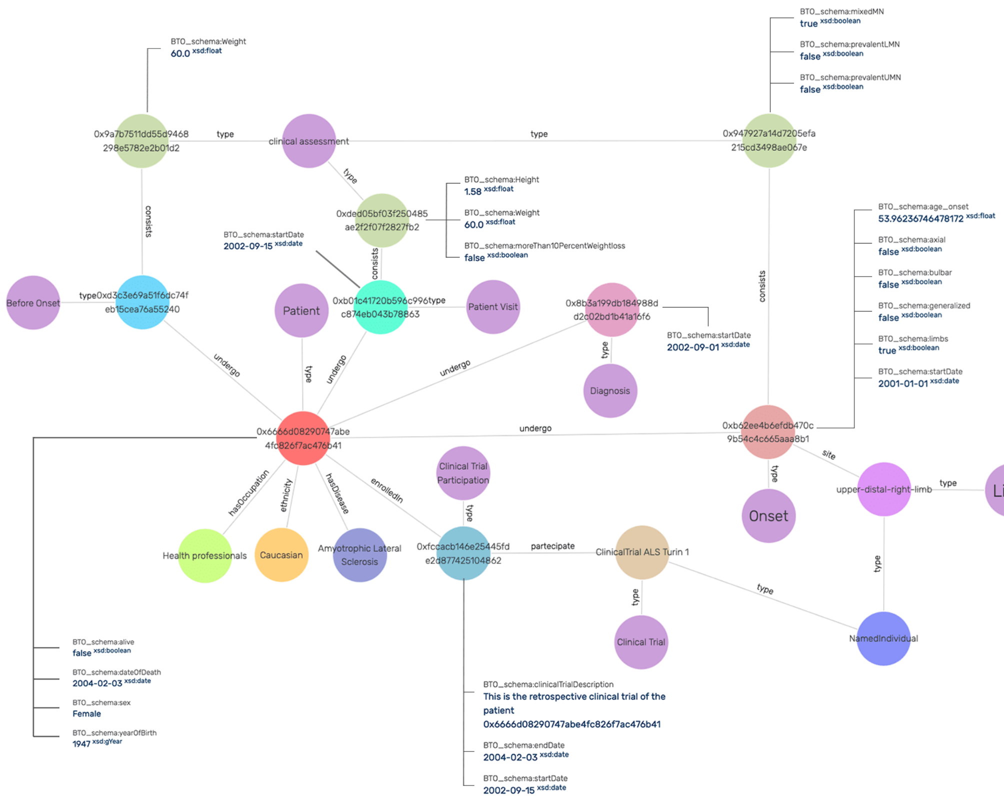


Figure 5.12: Example of RDF graph about patient static vars.

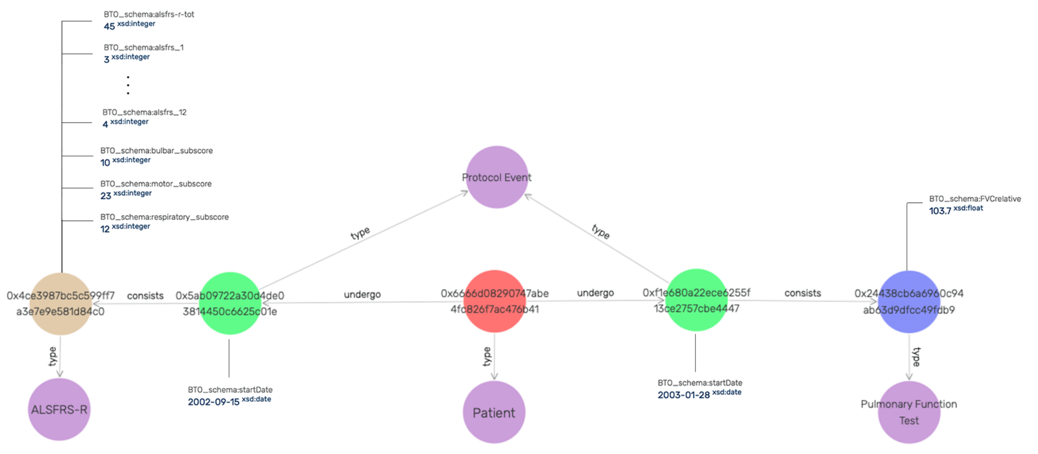


Figure 5.13: Example of RDF graph about patient visits.

two images for reasons of limited space on the page and to make these images more readable. However the node colored in red inside the two images represents the same resource "Patient" and therefore it is possible to imagine it as a single graph.
If we look at the two images in more detail, it is possible to see that the Figure 5.12 contains the graph obtained by mapping the data in Figure 5.9, i.e. the data related to the "static vars" sheet. While figure Figure 5.13 illustrates the graph obtained by mapping the data contained in Figure 5.10 and Figure 5.11, i.e., the data related to the "ALSFRS" and "%FVC" sheets.
It is important to state that the "DateOfDiagnosis" and "DateOf1stSymptoms" in the Figure 5.9 are given as the 15th day of the month, but for project reasons both dates are mapped to the first day of the month. In fact looking at the graph in the Figure 5.12 it is possible to see that these dates have been changed.
Finally, in the Code Snippet 5.7, we present the serialization of the RDF graph, seen as a union of the Figure 5.12 and Figure 5.13, in the Turtle format.

```
@prefix BTO_ni: <https://w3id.org/brainteaser/ontology/named-
   individual/> .
@prefix BTO_resource: <https://w3id.org/brainteaser/ontology/
   resource/> .
@prefix BTO_schema: <https://w3id.org/brainteaser/ontology/schema/>
    .
@prefix MAXO: <http://purl.obolibrary.org/obo/MAXO_> .
@prefix NCIT: <http://purl.obolibrary.org/obo/NCIT_> .
@prefix OGG: <http://purl.obolibrary.org/obo/OGG_> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

BTO_resource:0x6666d08290747abe4fc826f7ac476b41 a NCIT:C16960 ;
    BTO_schema:alive false ;
    BTO_schema:dateOfDeath "2004-02-03"^^xsd:date ;
    BTO_schema:enrolledIn BTO_resource:0
   xfccacb146e25445fde2d877425104862 ;
    BTO_schema:ethnicity BTO_ni:Caucasian ;
    BTO_schema:hasDisease BTO_ni:Amyotrophic_Lateral_Sclerosis ;
    BTO_schema:hasOccupation BTO_ni:Health_professionals ;
    BTO_schema:sex "Female"^^rdf:langString ;
    BTO_schema:undergo BTO_resource:0
   x5ab09722a30d4de03814450c6625c01e,
        BTO_resource:0x8b3a199db184988dd2c02bd1b41a16f6,
        BTO_resource:0xb01c41720b596c996c874eb043b78863,
        BTO_resource:0xb62ee4b6efdb470c9b54c4c665aaa8b1,
        BTO_resource:0xd3c3e69a51f6dc74feb15cea76a55240,
        BTO_resource:0xf1e680a22ece6255f13ce2757cbe4447 ;
    BTO_schema:yearOfBirth "1947"^^xsd:gYear .

BTO_resource:0xfccacb146e25445fde2d877425104862 a BTO_schema:
    Clinical_Trial_Participation ;
    BTO_schema:clinicalTrialDescription "This is the retrospective
    clinical trial of the patient 0x6666d08290747abe4fc826f7ac476b41
    "^^rdf:langString ;
    BTO_schema:endDate "2004-02-03"^^xsd:date ;
    BTO_schema:partecipate BTO_ni:ClinicalTrial_ALS_Turin_1 ;
    BTO_schema:startDate "2002-09-15"^^xsd:date .

BTO_resource:0x5ab09722a30d4de03814450c6625c01e a NCIT:C74589 ;
    BTO_schema:consists BTO_resource:0
    x4ce3987bc5c599ff7a3e7e9e581d84c0 ;
    BTO_schema:startDate "2002-09-15"^^xsd:date .

BTO_resource:0x4ce3987bc5c599ff7a3e7e9e581d84c0 a BTO_schema:ALSFRS-
    R ;
    BTO_schema:alsfrs-r-tot 45 ;
    BTO_schema:alsfrs_1 3 ;
    BTO_schema:alsfrs_10 4 ;
    BTO_schema:alsfrs_11 4 ;
    BTO_schema:alsfrs_12 4 ;
    BTO_schema:alsfrs_2 4 ;
    BTO_schema:alsfrs_3 3 ;
    BTO_schema:alsfrs_4 3 ;
    BTO_schema:alsfrs_5 4 ;
    BTO_schema:alsfrs_6 4 ;
    BTO_schema:alsfrs_7 4 ;
    BTO_schema:alsfrs_8 4 ;
    BTO_schema:alsfrs_9 4 ;
    BTO_schema:bulbar_subscore 10 ;
    BTO_schema:motor_subscore 23 ;
    BTO_schema:respiratory_subscore 12 .

BTO_resource:0x8b3a199db184988dd2c02bd1b41a16f6 a NCIT:C15220 ;
    BTO_schema:startDate "2002-09-01"^^xsd:date .

BTO_resource:0xb01c41720b596c996c874eb043b78863 a NCIT:C39564 ;
    BTO_schema:consists BTO_resource:0
    xded05bf03f250485ae2f2f07f2827fb2 ;
    BTO_schema:startDate "2002-09-15"^^xsd:date .
```

```

BTO_resource:0xded05bf03f250485ae2f2f07f2827fb2 a MAXO:0000487 ;
    BTO_schema:Height "1.58"^^xsd:float ;
    BTO_schema:Weight "60.0"^^xsd:float ;
    BTO_schema:moreThan10PercentWeightloss false .

BTO_resource:0xb62ee4b6efdb470c9b54c4c665aaa8b1 a NCIT:C25279 ;
    BTO_schema:age_onset "53.96236746478172"^^xsd:float ;
    BTO_schema:axial false ;
    BTO_schema:bulbar false ;
    BTO_schema:consists BTO_resource:0
   x947927a14d7205efa215cd3498ae067e ;
    BTO_schema:generalized false ;
    BTO_schema:limbs true ;
    BTO_schema:site BTO_ni:upper-distal-right-limb ;
    BTO_schema:startDate "2001-01-01"^^xsd:date .

BTO_resource:0x947927a14d7205efa215cd3498ae067e a MAXO:0000487 ;
    BTO_schema:mixedMN true ;
    BTO_schema:prevalentLMN false ;
    BTO_schema:prevalentUMN false .

BTO_resource:0xd3c3e69a51f6dc74feb15cea76a55240 a BTO_schema:
   Before_Onset ;
    BTO_schema:consists BTO_resource:0
   x9a7b7511dd55d9468298e5782e2b01d2 .

BTO_resource:0x9a7b7511dd55d9468298e5782e2b01d2 a MAXO:0000487 ;
    BTO_schema:Weight "60.0"^^xsd:float .

BTO_resource:0xf1e680a22ece6255f13ce2757cbe4447 a NCIT:C74589 ;
    BTO_schema:consists BTO_resource:0
   x24438cb6a6960c94ab63d9dfcc49fdb9 ;
    BTO_schema:startDate "2003-01-28"^^xsd:date .

BTO_resource:0x24438cb6a6960c94ab63d9dfcc49fdb9 a NCIT:C38081 ;
    BTO_schema:FVCrelative "103.7"^^xsd:float .
```

Code Snippet 5.7: Serialization in Turtle format of the example RDF graph.

5.2.7 CSV export

In Subsection 5.2.1, specifically in Figure 5.8, we presented the software architecture of the Data Mapper, which involved developing a module capable of

exporting data in CSV format from the RDF graph. For the development of this module we used the `query` function of the rdflib library, which allows to query the RDF graph through SPARQL queries.
Unfortunately, the current version of rdflib is not optimized to run complex SPARQL queries and also does not implement all SPARQL constructs such as “IF” and “EXISTS”. In addition, running a SPARQL query that does not have the above constructs takes too much time to output the results.
We solved this problem by using GraphDB (see Section 2.6). GraphDB is one of the best RDF graph databases that allows to create a repository on which to import serialized RDF datasets, and run SPARQL queries on them in an efficient way, through its “SPARQL” tool shown in Figure 2.5. The “SPARQL” tool of GraphDB presents an editor in which it is possible to write or copy and paste ready-made queries, such as the one contained in Code Snippet 5.8. This SPARQL query, when executed on a serialized RDF dataset of Brainteaser, allows to output visits data regarding ALSFRS-R and pulmonary function tests.
Once a query is executed, GraphDB allows the results to be exported in multiple formats including CSV format, which is the useful format for AI models.
To conclude, we can say that compared to the architecture presented in Figure 5.8 we can see the “CSV export” module moved from the RDF Graph Builder to the “Serialization” module. This is due to the fact that the execution of SPARQL queries to get the data in CSV are performed on the serialized RDF datasets.

```
PREFIX BTO_schema: <https://w3id.org/brainteaser/ontology/schema/>
PREFIX NCIT: <http://purl.obolibrary.org/obo/NCIT_>
SELECT ?id ?date_spiro ?fvcValue ?date_alsfrs_r ?alsfrs_r_tot_score
    ?bulbar_subscore ?motor_subscore ?respiratory_subscore ?q1 ?q2 ?
    q3 ?q4 ?q5 ?q6 ?q7 ?q8 ?q9 ?q10 ?q11 ?q12 WHERE {
{
    # Spiro
    SELECT ?id ?date_spiro ?fvcValue WHERE {
        ?idURI a NCIT:C16960 ;
            BTO_schema:undergo ?eventURI .
        ?eventURI a NCIT:C74589 ;
            BTO_schema:startDate ?date_spiro ;
            BTO_schema:consists ?testURI .
        ?testURI a NCIT:C38081 ;
            BTO_schema:FVCrelative ?fvcValue .
        bind( substr( (str(?idURI)), 48) as ?id)
    } ORDER BY ?id ?date_spiro
}
UNION
{
    # ALSFRS-R
    SELECT ?id ?date_alsfrs_r ?alsfrs_r_tot_score ?bulbar_subscore ?
   motor_subscore ?respiratory_subscore ?q1 ?q2 ?q3 ?q4 ?q5 ?q6 ?q7
   ?q8 ?q9 ?q10 ?q11 ?q12 WHERE {
        ?idURI a NCIT:C16960 ;
            BTO_schema:undergo ?eventURI .
        ?eventURI a NCIT:C74589 ;
            BTO_schema:startDate ?date_alsfrs_r ;
            BTO_schema:consists ?qRURI .
        ?qRURI a BTO_schema:ALSFRS-R ;
            BTO_schema:alsfrs-r-tot ?alsfrs_r_tot_score ;
            BTO_schema:bulbar_subscore ?bulbar_subscore ;
            BTO_schema:motor_subscore ?motor_subscore ;
            BTO_schema:respiratory_subscore ?respiratory_subscore ;
            BTO_schema:alsfrs_1 ?q1 ;
            BTO_schema:alsfrs_2 ?q2 ;
            BTO_schema:alsfrs_3 ?q3 ;
            BTO_schema:alsfrs_4 ?q4 ;
            BTO_schema:alsfrs_5 ?q5 ;
            BTO_schema:alsfrs_6 ?q6 ;
            BTO_schema:alsfrs_7 ?q7 ;
            BTO_schema:alsfrs_8 ?q8 ;
            BTO_schema:alsfrs_9 ?q9 ;
            BTO_schema:alsfrs_10 ?q10 ;
            BTO_schema:alsfrs_11 ?q11 ;
            BTO_schema:alsfrs_12 ?q12 .
        bind( substr( (str(?idURI)), 48) as ?id)
    } ORDER BY ?id ?date_alsfrs
}
}ORDER BY ?id ?date_alsfrs_r ?date_spiro
```

Code Snippet 5.8: An example of a SPARQL query to get visits from the RDF dataset of Brainteaser resources.

Chapter 6

Dataset statistics

The purpose of this chapter is to show some statistical data on datasets provided by ALS partner clinics. For each of these datasets we will present both some general statistic data related to their content and the results obtained after performing the pre-processing operations presented in the Subsection 5.2.3. The latter results are intended to show how many patients were actually mapped to the respective RDF datasets and how many had to be discarded due to some unmet requirements. In this chapter we will focus in particular on the Turin (Section 6.1) and Lisbon (Section 6.2) datasets, for which the development of the respective Data Mapper has been completed, while the Madrid dataset is not taken into account as the development of the corresponding Data Mapper is still in progress. Finally, in Section 6.3, we present the statistical data obtained on the merged RDF dataset.

6.1 Turin dataset

The statistic data regarding the dataset provided by the "Universitá degli Studi di Torino" (UNITO, Italy), our clinical partner, are represented in the Table 6.1. This table presents the total number of patients, some general statistical data on patient characteristics, and the total number of ALSFRS-R tests and pulmonary function tests contained within the dataset. In detail, the Turin dataset contains 3257 patients, 15006 ALSFRS-R tests and 2890 pulmonary function tests. On these patients and visits, the pre-processing operations described in Subsection 5.2.3 were performed, which led to obtaining additional statistical data indicating which patients satisfied certain requirements and which did not. It is important to note that the statistical data obtained from the pre-processing op-

	Turin Dataset	**Lisbon Dataset**
Total Patients	3257	1562
Gender	1781 *Male* 1476 *Female*	884 *Male* 678 *Female*
Ethnicity	3248 *Caucasian* 7 *African* 2 *Asian*	1530 *Caucasian* 27 *African* 5 *Asian*
Birth Year	1906 - 1990	1991 - 2022
Diagnosis Year	1995 - 2018	1994 - 2021
Patient status	564 *Alive* 2693 *Death*	561 *Alive* 999 *Death*[1]
Total ALSFRS/ALSFRS-R	15006	7446
Total PFT[2]	2890	2631

[1] Two patients do not present status
[2] Pulmonary Function Tests

Table 6.1: Statistics on input datasets.

erations are taken from the log file generated as output by Data Mapper. Based on the subdivision of the pre-processing functions presented in Subsection 5.2.3, we proceed to describe the statistical data obtained:

1. Get correct ALSFRS-R tests:

   ```
   Total number of ALSFRS/ALSFRS-R: 15006
   Number of correct ALSFRS-R: 14979
   Number of incorrect ALSFRS/ALSFRS-R: 27 (duplicated ALSFRS-R)
   ```

   This operation allowed us to identify 14979 correct ALSFRS-R tests compared with 15006 total, since 27 tests were duplicates;

2. Get correct pulmonary function tests:

   ```
   Total number of Spiro: 2890
   Number of correct Spiro: 2873
   Number of incorrect Spiro: 17 (duplicated Spiro)
   ```

   Through this operation, as happened for the ALSFRS-R tests, 17 duplicate pulmonary function tests were detected, so the total number of correct tests is 2873. In the log file we have used the term “Spiro” to refer to pulmonary function tests, so the two terms can be considered equivalent;

3. Check “static vars” dates:

   ```
   Number of patients with onset/diagnosis/death (if patient is
    ↪  dead) in the correct sequence: 3255
   Number of patients with onset/diagnosis/death (if patient is
    ↪  dead) not in the correct sequence: 2
   ```

   The function detected 3255 patients meeting the following requirement: dateOfOnset $\leq$ dateOfDiagnosis $\leq$ dateOfDeath (if the patient is dead). So only 2 patients out of a total of 3257 have onset/diagnosis/death not in the correct order;

4. Get patient IDs with ALSFRS-R tests:

   ```
   Number of patients with at least one ALSFRS-R: 1990
   ```

   With this operation, we obtained the patients who had at least one ALSFRS-R test performed (1990). Unfortunately, 1267 patients of 3257 have never had an ALSFRS-R test and are therefore excluded from the next mapping step;

5. Check "static vars" procedures:

```
Number of patients with at least one ALSFRS-R: 1990
Number of patients with no events (NIV and PEG and
 ↪  Tracheostomy): 784
Number of patients with events (NIV or PEG or Tracheostomy)
 ↪  after the first ALSFRS-R and before the date of death:
 ↪  1099
Number of patients with no events (NIV and PEG and
 ↪  Tracheostomy) + Number of patients with events (NIV or
 ↪  PEG or Tracheostomy) after the first ALSFRS-R and before
 ↪  the date of death: 1883
Number of patients with incorrect events (NIV or PEG or
 ↪  Tracheostomy): 107
```

   This operation is performed on patients who have at least one ALSFRS-R (1990), since it must verify that: dateOfFirstALSFRS-R $\leq$ eventDate (NIV/PEG/Tracheostomy) $\leq$ dateOfDeath (the second comparison is performed only if the patient is dead). The results obtained show that 1099 patients meet the requirement and 784 patients have no event (NIV and PEG and Tracheostomy). The patients in these two sets (1883 in total) are exactly those subjects who should be mapped, while the 107 patients who do not meet the requirement should be discarded;

6. Check date of death with date of last visit:

```
Number of patients with at least one ALSFRS-R: 1990
Number of alive patients: 282
Number of patients with date of death >= date of last visit:
 ↪  1677
Number of alive patients + Number of patients with date of
 ↪  death >= date of last visit: 1959
Number of patients with date of death < date of last visit:
 ↪  31
Number of patients with invalid status: 0
```

With this operation we want to obtain all those patients (among those with at least one ALSFRS-R) who are alive (282), or have death as the last event (1677). This operation is necessary to avoid mapping those patients who have a visit after death, which in reality is unfeasible. So from the data obtained we see that 31 patients do not meet this requirement;

7. Get patient IDs to be mapped:

```
Number of patients: 3257
Number of patients with correct data: 1854 (total ASLFRS-R:
 ↪  14413, total Spiro: 2514)
Number of patients with incorrect data (excluded from
 ↪  mapping): 1403
```

This final function has the task of intersecting all the corrected patient sets obtained in the previous pre-processing operations in order to obtain the final list of patients to be mapped. For the Turin dataset we see that 1854 are the patients that will be mapped, while 1403 are excluded from the mapping. In addition 14413 ALSFRS-R tests and 2514 pulmonary function tests will also be mapped, which correspond to the tests performed on the 1854 patients.

6.2 Lisbon dataset

The Lisbon dataset was provided to us by the "Instituto de medicina molecolare João Lobo Antunes" (iMM, Portugal), which is one of the three clinical partners for ALS. The statistical data concerning the Lisbon dataset are shown in the Table 6.1, where there are the total number of patients, some general statistical data on their characteristics, and the total number of ALSFRS-R tests and pulmonary function tests contained in the dataset. In detail, within the Lisbon dataset there are 1562 patients, 7446 ALSFRS-R tests, and 2631 pulmonary function tests. As also explained in the Section 6.1, the pre-processing operations (described in the Subsection 5.2.3) were performed in order to obtain additional statistical data indicating which patients could be mapped and which could not. It is important to note that the statistical data of the pre-processing operations that we are going to present below are taken from the Lisbon log file. Following the subdivision of the preprocessing functions presented in Subsection 5.2.3, we proceed to describe the obtained statistical data:

1. Get correct ALSFRS-R tests:

   ```
   Total number of ALSFRS/ALSFRS-R: 7446
   Number of correct ALSFRS-R: 7050
   Number of incorrect ALSFRS/ALSFRS-R: 396 (1 test with wrong
    ↪  dates in the sequence and 395 test with missing data)
   ```

   This operation allowed us to identify 7050 correct ALSFRS-R tests and 396 incorrect tests, as they had missing data or had the dates not in the correct sequence;

2. Get correct pulmonary function tests:

   ```
   Total number of Spiro: 2631
   Number of correct Spiro: 2580
   Number of incorrect Spiro: 51 (missing data)
   ```

   Through this operation, 2580 correct pulmonary function tests and 51 tests with missing data were identified. In the log file we have used the term "Spiro" to refer to pulmonary function tests, so the two terms can be considered equivalent;

3. Check "static vars" dates:

   ```
   Number of patients with onset/diagnosis/death (if patient is
    ↪  dead) in the correct sequence: 1530
   Number of patients with onset/diagnosis/death (if patient is
    ↪  dead) not in the correct sequence: 32
   ```

   The function was able to detect 1530 patients who met the following rule: dateOfOnset $\leq$ dateOfDiagnosis $\leq$ dateOfDeath (if the patient died). So, 32 patients out of a total of 1562 have onset/diagnosis/death not in the correct order;

4. Get patient IDs with ALSFRS-R tests:

   ```
   Number of patients with at least one ALSFRS-R: 1383
   ```

   With this operation, we obtained that 1383 patients had at least one ALSFRS-R test performed. So, 179 patients of 1562 have never had an ALSFRS-R test and are therefore excluded from the next mapping step;

5. Check "static vars" procedures:

```
Number of patients with at least one ALSFRS-R: 1383
Number of patients with no events (NIV and PEG and
 ↪  Tracheostomy): 29
Number of patients with events (NIV or PEG or Tracheostomy)
 ↪  after the first ALSFRS-R and before the date of death:
 ↪  677
Number of patients with no events (NIV and PEG and
 ↪  Tracheostomy) + Number of patients with events (NIV or
 ↪  PEG or Tracheostomy) after the first ALSFRS-R and before
 ↪  the date of death: 706
Number of patients with incorrect events (NIV or PEG or
 ↪  Tracheostomy): 677
```

This operation is performed on patients who have at least one ALSFRS-R (1383), since it must verify that: dateOfFirstALSFRS-R $\leq$ eventDate (NIV/PEG/Tracheostomy) $\leq$ dateOfDeath (the second comparison is performed only if the patient is dead). The results obtained show that 677 patients meet the requirement and 29 patients have no event (NIV and PEG and Tracheostomy). These patients (706 in total) are exactly the subjects that should be mapped. Unfortunately there are 677 patients who should be discarded;

6. Check date of death with date of last visit:

```
Number of patients with at least one ALSFRS-R: 1383
Number of alive patients: 483
Number of patients with date of death >= date of last visit:
 ↪  900
Number of alive patients + Number of patients with date of
 ↪  death >= date of last visit: 1383
Number of patients with date of death < date of last visit: 0
Number of patients with invalid status: 0
```

With this operation we want to obtain all those patients (among those with at least one ALSFRS-R) who are alive (483), or have death as the last event (900). This operation is necessary to avoid mapping those patients who have a visit after death, which in reality is unfeasible. Thus, in this dataset all 1383 patients meet the requirement;

7. Get patient IDs to be mapped:

```
Number of patients: 1562
Number of patients with correct data: 705 (total ASLFRS-R:
 ↪  4698, total Spiro: 1652)
Number of patients with incorrect data (excluded from
 ↪  mapping): 857
```

   This final function has the task of intersecting all the corrected patient sets obtained in the previous pre-processing operations in order to obtain the final list of patients to be mapped. For the Lisbon dataset we have that 705 are the patients that will be mapped, while 857 are excluded from the mapping because they are patients with incorrect data. In addition, 4698 ALSFRS-R tests and 1652 pulmonary function tests will be mapped, which correspond to the tests performed on the 705 patients.

6.3 Merged dataset

To conclude we want to present some statistical data on the merged version of the serialized RDF dataset, which was obtained by merging the Turin and Lisbon RDF datasets. This merging operation resulted in a total of 2559 patients. where 1854 are from the Turin dataset and 705 are patients from the Lisbon dataset. In the same way, the 14413 ALSFRS-R and 2514 pulmonary function tests of Turin were combined with the 4698 ALSFRS-R and 1652 pulmonary function tests of Lisbon, thus obtaining a total of 19111 ALSFRS-R and 4166 pulmonary function tests.
All these statistics and additional patient-related information of the merged RDF dataset are presented in the Table 6.2.

	Merged Dataset
Total Patients	2559[1]
Gender	1387 1172 *Female*
Ethnicity	2541 *Caucasian* 15 *African* 3 *Asian*
Birth Year	1912 - 1993
Diagnosis Year	1995 - 2021
Patient status	400 *Alive* 2159 *Death*
Total ALSFRS/ALSFRS-R	19111[2]
Total PFT[4]	4166[3]

[1] 1854 from Turin, 705 from Lisbon
[2] 14413 from Turin, 4698 from Lisbon
[3] 2514 from Turin, 1652 from Lisbon
[4] Pulmonary Function Tests

Table 6.2: Statistics on merged RDF datasets.

Chapter 7

Conclusions and Future Work

The Brainteaser Ontology (BO) has the purpose to jointly model both Amyotrophic Lateral Sclerosis (ALS) and Multiple Sclerosis (MS) data. In this thesis, the goal was firstly to design and develop the part of the BO that deals with retrospective data management for Amyotrophic Lateral Sclerosis (ALS). Both the ALS and MS parts of the BO were co-designed in close collaboration with medical partners and domain experts. We used this approach to incorporate expert knowledge into the BO and, at the same time, to validate all design choices. To this end, we operated iteratively, producing several intermediate versions of the ontology and discussing them with our domain experts. After receiving and analyzing the anonymized retrospective data from the partner clinics, we refined the final details of the BO and the resulting version was approved by the domain experts. This last developed version became the first official version of the BO (Brainteaser Ontology v1.0), which documentation can be found at `https://w3id.org/brainteaser/ontology`.

Subsequently to this, the second goal was to develop the ALS Data Mapper that is responsible for mapping the retrospective data provided by the ALS partner clinics into a structured RDF graph compliant with the Brainteaser Ontology. Since the retrospective data comes from different clinics (Turin, Lisbon and Madrid), each of them using their own notation and standards for knowledge management, it was necessary to develop a modular architecture with several Data Mappers, one for each ALS clinic, with the aim of unifying all the knowledge provided by the partners and collecting it in a single RDF graph.

The complete development of the Turin and Lisbon data mappers allowed us to map their retrospective datasets, obtaining in output the first RDF dataset of Brainteaser resources. This RDF dataset is obtained by merging the Turin and

Lisbon RDF datasets and contains within it 2559 patients and 23277 visits (19111 ALSFRS-R and 4166 Pulmonary Function Tests).
The merged RDF dataset was then imported on GraphDB and queried with SPARQL queries, in order to export two datasets in CSV format: one containing the patients data and one containing the visits data. At the moment these CSV datasets are used by teams involved in AI modeling, and they are already available for the first Open Evaluation Challenge which is focused on predicting the ALS progression.
The source code of the Data Mappers, the serialized RDF datasets in different formats (Turtle, JSON-LD, RDF/XML) and the CSV datasets are available at `https://bitbucket.org/brainteaser-health/mappers/src/master/ALS/`.
As future work, it will be necessary to finish the development of the Madrid Data Mapper in order to map its data as well and thus complete all the part related to retrospective data. Then it will be necessary to start developing a new version of the ontology for the integration with the prospective data, and a Data Mapper for mapping them. In conclusion the Brainteaser Ontology and the Brainteaser resources will be integrated with the EOSC (European Open Science Cloud) services.

References

[Andersen and Al-Chalabi, 2011] Andersen, P. M. and Al-Chalabi, A. (2011). Clinical genetics of amyotrophic lateral sclerosis: what do we really know? *Nature Reviews Neurology*, 7(11):603–615.

[Auer et al., 2007] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific american*, 284(5):34–43.

[Bizer et al., 2009] Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked Data: The Story so Far. *International Journal on Semantic Web and Information Systems*, 5:1–22.

[Carmody et al., 2019] Carmody, L. C., Zhang, X. A., Vasilevsky, N. A., Mungall, C. J., Matentzoglu, N., and Robinson, P. N. (2019). Medical Action Ontology (MAxO). *Proceedings of the 10th International Conference on Biomedical Ontology (ICBO 2019).*

[Cedarbaum et al., 1999] Cedarbaum, J. M., Stambler, N., Malta, E., Fuller, C., Hilt, D., Thurmond, B., Nakanishi, A., Group, B. A. S., complete listing of the BDNF Study Group, A., et al. (1999). The ALSFRS-R: a revised ALS functional rating scale that incorporates assessments of respiratory function. *Journal of the neurological sciences*, 169(1-2):13–21.

[Fang et al., 2017] Fang, T., Al Khleifat, A., Stahl, D. R., Lazo La Torre, C., Murphy, C., LicalS, U.-M., Young, C., Shaw, P. J., Leigh, P. N., and Al-Chalabi, A. (2017). Comparison of the King's and MiToS staging systems for ALS. *Amyotrophic Lateral Sclerosis and Frontotemporal Degeneration*, 18(3-4):227–232.

[Gennari et al., 2003] Gennari, J. H., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubézy, M., Eriksson, H., Noy, N. F., and Tu, S. W. (2003). The evolution of Protégé: an environment for knowledge-based systems development. *International Journal of Human-computer studies*, 58(1):89–123.

[Graves et al., 2007] Graves, M., Constabaris, A., and Brickley, D. (2007). Foaf: Connecting people on the semantic web. *Cataloging & classification quarterly*, 43(3-4):191–202.

[He et al., 2014] He, Y., Liu, Y., and Zhao, B. (2014). OGG: a Biological Ontology for Representing Genes and Genomes in Specific Organisms. In *ICBO*, pages 13–20. Citeseer.

[Heath and Bizer, 2011] Heath, T. and Bizer, C. (2011). Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136.

[Hoehndorf et al., 2015] Hoehndorf, R., Schofield, P. N., and Gkoutos, G. V. (2015). The role of ontologies in biological and biomedical research: a functional perspective. *Briefings in bioinformatics*, 16(6):1069–1080.

[Kumar and Smith, 2005] Kumar, A. and Smith, B. (2005). Oncology ontology in the NCI thesaurus. In *Conference on Artificial Intelligence in Medicine in Europe*, pages 213–220. Springer.

[Lechtzin et al., 2018] Lechtzin, N., Cudkowicz, M. E., de Carvalho, M., Genge, A., Hardiman, O., Mitsumoto, H., Mora, J. S., Shefner, J., Van den Berg, L. H., and Andrews, J. A. (2018). Respiratory measures in amyotrophic lateral sclerosis. *Amyotrophic Lateral Sclerosis and Frontotemporal Degeneration*, 19(5-6):321–330.

[Morelot-Panzini et al., 2019] Morelot-Panzini, C., Bruneteau, G., and Gonzalez-Bermejo, J. (2019). NIV in amyotrophic lateral sclerosis: the 'when' and 'how' of the matter. *Respirology*, 24(6):521–530.

[Musen, 2015] Musen, M. A. (2015). The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12.

[Noy and Mcguinness, 2001] Noy, N. and Mcguinness, D. (2001). Ontology Development 101: A Guide to Creating Your First Ontology. *Knowledge Systems Laboratory*, 32.

[Ong et al., 2017] Ong, E., Xiang, Z., Zhao, B., Liu, Y., Lin, Y., Zheng, J., Mungall, C., Courtot, M., Ruttenberg, A., and He, Y. (2017). Ontobee: a linked ontology data server to support ontology term dereferencing, linkage, query and integration. *Nucleic acids research*, 45(D1):D347–D352.

[Pérez et al., 2009] Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45.