Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies

Jean-Baptiste Lamy

Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies

Jean-Baptiste Lamy[a,*]

[a]LIMICS, Université Paris 13, Sorbonne Paris Cité, 93017 Bobigny, France, INSERM UMRS 1142, UPMC Université Paris 6, Sorbonne Universités, Paris, France

Abstract

Objective: Ontologies are widely used in the biomedical domain. While many tools exist for the edition, alignment or evaluation of ontologies, few solutions have been proposed for ontology programming interface, *i.e.* for accessing and modifying an ontology within a programming language. Existing query languages (such as SPARQL) and APIs (such as OWLAPI) are not as easy-to-use as object programming languages are. Moreover, they provide few solutions to difficulties encountered with biomedical ontologies. Our objective was to design a tool for accessing easily the entities of an OWL ontology, with high-level constructs helping with biomedical ontologies.

Methods: From our experience on medical ontologies, we identified two difficulties: (1) many entities are represented by classes (rather than individuals), but the existing tools do not permit manipulating classes as easily as individuals, (2) ontologies rely on the open-world assumption, whereas the medical reasoning must consider only evidence-based medical knowledge as true. We designed a Python module for *ontology-oriented programming*. It allows access to the entities of an OWL ontology as if they were objects in the programming language. We propose a simple high-level syntax for managing classes and the associated "role-filler" constraints. We also propose an algorithm for performing *local closed world reasoning* in simple situations.

Results: We developed Owlready, a Python module for a high-level access to OWL ontologies. The paper describes the architecture and the syntax of the module version 2. It details how we integrated the OWL ontology model with the Python object model. The paper provides examples based on Gene Ontology (GO). We also demonstrate the interest of Owlready in a use case focused on the automatic comparison of the contraindications of several drugs. This use case illustrates the use of the specific syntax proposed for manipulating classes and for performing local closed world reasoning.

Conclusion: Owlready has been successfully used in a medical research project. It has been published as Open-Source software and then used by many other researchers. Future developments will focus on the support of vagueness and additional non-monotonic reasoning feature, and automatic dialog box generation.

Keywords: Semantic web, Ontology-oriented programming, Automatic classification, Local closed world reasoning, Biomedical ontology, OWL

1. Introduction

A formal ontology is the specification of the concepts, their attributes and relationships, in a given domain of discourse [1]. Ontologies have two major interests: (1) they can be used to perform logical inferences for deducing new facts, with a reasoner, and (2) they can link together various pieces of knowledge from different ontologies in the Semantic Web. The W3C (World Wide Web Consortium) proposed OWL (Web Ontology Language) [2] for formalizing ontologies, and OWL ontologies are often represented as RDF graphs (Resource Description Framework) [3].

Ontologies have been used in many domains, and in particular the biomedical domain [4, 5]. Today it is one of the most complex domain of Human knowledge, and there is an increasing need for structuring and formalizing this knowledge. More than twenty years ago, the GALEN project [6] was already based on an ontology. The development of recent biomedical terminologies has also been heavily influenced by ontologies. Examples are SNOMED CT (Standardized NOmenclature of MEDicine - Clinical Terms) [7], Gene Ontology [8] and the FMA (Foundational Model of Anatomy) [9], which was recently converted to OWL 2 [10].

Many methods and tools have been proposed for the design, the edition, the maintenance, the alignment or the evaluation of ontologies, including the Protégé editor and the HermiT reasoner [11]. However, fewer options are available for *ontology programming interface*, a problem identified by Rector *et al.* [12]. Ontology programming interfaces are tools for accessing and modifying an ontology within a programming language. Typical examples of use include populating an ontology with data extracted from a database, or generating a website displaying the inferences produced by running a reasoner on the ontology. Programming languages, and especially object-oriented programming languages, are widely taught and used nowadays. Thus it is a common need to interface ontologies with these programming languages.

In the literature, three strategies have been proposed for ontology programming interfaces: query languages such as SPARQL, APIs (Application Programming Interfaces) such as OWLAPI, and ontology-oriented programming. The first two strategies are commonly used. However, with these strategies, it is still much longer and more difficult to work with ontologies than to develop an object oriented program. The last strategy is still a matter of research, but it also seems the most promising. In a previous paper [13], we highlighted the potential interest of ontology-oriented programming for the biomedical domain.

An example is the comparison of contraindications between several drugs. Contraindications can be extracted from drug databases or entered manually by an expert from official reference documents (such as Summary of Product Characteristics, SPC).

*Corresponding author

This is an author file of the article published in Artificial Intelligence In Medicine 2017;80C:11-28, DOI: 10.1016/j.artmed.2017.07.002 ;

Email address: jibalamy@free.fr (Jean-Baptiste Lamy)

However, due to synonymy, partitions and variations in granularity level, the contraindications cannot be compared "as-is" between drugs. For instance, drug A might be contraindicated with "acquired hemorrhagic disorder" and "fulminant hepatitis", and drug B with "hemorrhagic disorder" and "severe hepatic impairment", but it does not mean that drug B is not contraindicated in case of "acquired hemorrhagic disorder" and "fulminant hepatitis". A solution is to design an ontology and use a reasoner; it has been shown that ontologies could improve the quality of drug databases [14]. However, two difficulties are expected. First, drugs and disorders cannot be represented by individuals in the ontology. They can only be represented by classes, due to the subsumption relations that exist between them. But classes are more difficult to manipulate with common APIs. Second, the reasoning should only consider contraindications that are known (*i.e.* present in databases or entered by an expert). This requires *locally closed world reasoning*, which is not supported by common OWL reasoners.

The objectives of this work were to design a module for ontology-oriented programming in Python supporting OWL 2 ontologies[1], focusing on two points: (1) providing a clear, concise and easy-to-use syntax, based on both the notations of the well-known Protégé ontology editor and the dot notation common to many object-oriented programming languages, including Python, and (2) providing high-level syntactic elements for facilitating the manipulation of classes and a simple algorithm for local closed world reasoning. These two elements have been inspired by the difficulties we encountered with medical ontologies.

The rest of the paper is organized as follows. Section 2 describes two difficulties that we identified in medical ontologies. Section 3 presents the main strategies that exist for accessing and modifying an ontology in a programming language, and the approaches proposed for local closed world reasoning in the literature. Section 4 compares object models and ontologies, and details the differences. Section 5 presents Owlready, including the general architecture, the technical details and the syntax. It gives simple examples and more complex ones with Gene Ontology (GO). It also describes testing procedures on 5 ontologies and presents performance measurement and comparison with OWLAPI. Section 6 provides a use case in a medical application that illustrates how the high level syntax of Owlready can help with the two difficulties identified at the beginning of the paper. Finally, section 7 discusses the results and compares Owlready to similar approaches in the literature, before concluding.

2. Two particularities of ontologies in the medical domain

From our experience in the field, we identified two particularities of medical knowledge that complicate the design of ontologies and ontology-based reasoning. They are described in the next two subsections.

2.1. The open-world assumption is appropriate for medical data but not for medical knowledge

In medicine, knowledge is based on *evidence* as much as possible, *i.e.* knowledge comes from a proof, typically a significant difference observed during a *clinical trial* and involving a statistical test. This is known as *Evidence-Based Medicine* (EBM) [15]. A classical clinical trial design consists in recruiting two groups of patients, each group receiving a different treatment (*e.g.* drug A or B), blindly. At the end of the study, the efficacy of the treatment is evaluated for each patient, by considering biological markers or symptom evolution. If the difference between the efficacy of the two groups is statistically significant, one of the two drugs is more efficient and should be preferred; this produces a new piece of medical knowledge. When no evidence is available (and only in this case), medical knowledge should be established by considering the consensual opinion of groups of experts.

Consequently, in a medical reasoning, only evidence-based or expert-validated knowledge should be considered as true. Therefore, the *open-world assumption*, which considers that anything that is not said is possible and might be true, is not appropriate for medical knowledge. The reasoning should not make new hypotheses about medical knowledge, because such hypothesis would not be *evidence-based*. On the contrary, open-world assumption is interesting when considering *patient data*. It allows the reasoner making hypotheses about yet-unknown patient disorders or clinical conditions. This is especially useful in diagnostic systems.

For example, when a physician checks the contraindications of a drug, he considers that all contraindications are known (and listed in official texts and drug databases), even if this might not be entirely true. Thus, a medical decision-support system should only reason using known contraindications, rather than making hypotheses about potential new contraindications. This would lead to stupid alert messages such as "the patient is diabetic, the drug you are prescribing is not contraindicated with diabetes but a yet-unknown contraindication might exist. Please verify your prescription!".

In conclusion, a mix of open- and closed-world assumption is needed for the medical reasoning. This is usually referred to as *local closed world reasoning*. Other authors [16] made a similar observation, when reasoning on patient records and patient inclusion in clinical trials, and they showed that some patient data must be considered under the open world assumption while other must not. They considered patient diagnostics to be "open" (there might be yet-unknown diagnostics) and patient pharmacy data to be "closed" (all drugs prescribed to the patient are known *via* the pharmacy service).

2.2. The main medical concepts cannot be represented by individuals, but only by classes

The second particularity is that many *granularity levels* are usually considered in medicine. For example, disorders can be expressed at various granularities, *e.g.* intestine inflammatory disorders, Crohn disorder, severe Crohn disorder, Crohn disorder with cutaneous manifestations, *etc* [17]. Patient data is typically described at a fine granularity, *e.g.* "M. X suffers from Crohn disorder", while medical knowledge is described at a more general level, *e.g.* "this drug is contraindicated with intestine inflammatory disorders" (thus including Crohn disorder). Drug treatments behave similarly, for example analgesics, NonSteroidal Anti-Inflammatory Drugs (NSAID), aspirin, aspirin in the antiplatelet indication, aspirin tablet 1g, *etc*. These numerous levels of granularity complicate the representation of medical concepts because, when considering a medical concept, it is almost always possible to specify it more for defining a finer concept.

In ontologies, granularity levels are represented using *is-a* relations (*e.g.* Crohn disorder *is an* intestine inflammatory disorder). Since is-a relations do not exist between individuals, the main medical concepts, such as disorders and drug treatments, can only be represented by classes, and not individuals. For instance, disorders cannot be represented as the individuals of a Disorder

[1] In the rest of the paper, unless otherwise specified, the OWL version is 2.

Query langage (SPARQL) :

```
PREFIX <file://path/to/drug.owl>
SELECT (SUM(?drug_price) AS ?total_price)
WHERE {
    ?order :name "My order" .
    ?order :has_drug ?drug .
    ?drug :has_price ?drug_price .
}
```

Application Programming Interface (Java + OWLAPI) :

```
OWLDataFactory df = OWLManager.getOWLDataFactory();
public static float getOrderCost(OWLIndividual order) {
  float cost = 0.0;
  OWLObjectProperty drugProperty = df.getOWLObjectProperty(IRI.create("file.owl#has_drug"));
  OWLDataProperty priceProperty = df.getOWLDataProperty(IRI.create("file.owl#has_price"));
  Iterator drugs = order.listPropertyValues(drugProperty);
  while(drugs.hasNext()) {
    OWLIndividual drug = (OWLIndividual) drugs.next();
    Float price = (Float) drug.getPropertyValue(priceProperty);
    cost = cost + price.floatValue();
  }
  return cost;
}
```

Ontology-oriented programming (Python + Owlready) :

```
class Order(Thing):
    def get_cost(self):
        cost = 0.0
        for drug in self.drugs:  cost = cost + drug.price
        return cost
```

Figure 1: Examples of source codes for accessing an ontology in a computer program, using a SPARQL query (top), OWLAPI in Java (middle) and ontology-oriented programming in Python (bottom). The three examples compute the cost of a drug order, considering a single box for each drug in the order (thus it just computes the sum of the price of each drug in the order).

class, but only as subclasses of Disorder. Classes are usually more complex to manipulate than individuals, and thus this point complicates the use of ontologies in the biomedical domain. For example, in order to assert that Crohn disorder is a chronic disease, we need a single (subject, predicate, object) RDF triple if Crohn disorder is represented by an individual in the ontology:

(crohn_disorder, is_chronic, true)

On the contrary, if Crohn disorder is represented by a class, we need a "role-filler" constructs (noted $\exists R.\{i\}$ in Description Logics (DLs) notations, or "R value i" in Protégé, R being a property and i an individual or a literal). In OWL, it is represented by a restriction which requires 4 RDF triples and a blank node (_bn):

(Crohn_disorder, rdfs:subclassOf, _bn)
(_bn, rdf:type, owl:Restriction)
(_bn, owl:onProperty, is_chronic)
(_bn, owl:hasValue, true)

Moreover, it is not possible to assert directly that all individuals of a class A are related to all individuals of a class B [18]. For example, there is a well-known contraindication between NSAID and hemorrhagic disorders. However, this contraindication cannot be represented in OWL by a single restriction, because both NSAID and hemorrhagic disorders are classes and not individuals. OWL (version 2) can only express the fact that "NSAID are contraindicated with *some* hemorrhagic disorders" or "NSAID are contraindicated with *only* hemorrhagic disorders", but not "NSAID are contraindicated with *all* hemorrhagic disorders". Consequently, this contraindication can only be modeled in OWL by (1) creating a third class, Contraindication, (2) creating an individual of this class, the NSAID-HemorrhagicDisorders contraindication, and (3) relating the NSAID drug class and the HemorrhagicDisorder class to the contraindication individual using two "role-filler" constructs.

In the literature, Schulz *et al.* [19] also noticed that RDF triples (corresponding to individuals and their data) were not enough for representing medical knowledge and even patient data.

3. Related works

3.1. Ontology programming interfaces

Many tools have been proposed for interfacing RDF graphs or OWL ontologies, and many of them are listed on the W3C website[2]. However, the existing tools provide little help for dealing with the two difficulties we identified in the previous section. Several tools propose closed world reasoning, but options are more limited for *local* closed world. Most of the tools focus on RDF, some of them support OWL but they rarely provide specific syntax for facilitating the use of classes and role-filler constructs.

We can distinguish three main strategies for accessing an ontology in a programming language (Figure 1). The first strategy is the use of a *query language*, such as SPARQL (SPARQL Protocol and RDF Query Language) [20]. SPARQL is a query language proposed by the W3C. It allows searching and modifying RDF graphs. It has been inspired by SQL (Structured Query Language), developed for relational databases. However, SPARQL has two important drawbacks. First, it is not an object-oriented approach: a query must be written for each access to the ontology, even the simplest (*e.g.* get the value of the functional property p for individual x). This may be tedious, especially if the number of different queries is high.

[2] https://www.w3.org/2001/sw/wiki/Tools

Second, SPARQL is based on RDF and not OWL. Thus, SPARQL is data-oriented, it performs queries but no inference. Moreover, it is possible to query the individuals in the ontology, but manipulating the classes, for example adding new restriction (such as $\forall R.C$), is more complex, although still possible [21]. This point is particularly problematic in the biomedical domain, since we have seen that most medical concepts, such as disorders and drug treatments, are classes in ontologies.

The second strategy is the use of an *Application Programming Interface* (API). These APIs provide objects and functions for manipulating the elements that compose an ontology (*i.e.* classes, individuals, properties, annotations, restrictions, *etc*). In this strategy, a class in the ontology is an *instance* of the OWLClass class in the object-oriented programming language. Several similar APIs exist, such as OWLAPI [22] and Jena[3] in Java, or owlcpp [23] for C++ with binding for other languages (including Python).

These APIs are widely used, but they focus on performance rather than ease of use. They lead to complex and voluminous source codes (see example in Figure 1). Moreover, since an OWL class is considered as an instance in the programming language rather than a class, it is not possible to take full advantage of object-oriented programming. In the example Figure 1, the sum computation is implemented in a static function while, to strictly stick to the object paradigm, it should have been implemented in a method of the Order class (*e.g.* Order.getCost()). However, APIs cannot associate methods with OWL classes.

The third strategy is *ontology-oriented programming*. It is based on object-oriented programming, a well-known and successful paradigm. It takes advantage of the similarities that exist between ontologies and object models [24]: classes, properties and individuals in an ontology correspond to classes, attributes and instances in an object model [25].

Ontology-oriented programming tries to connect, or even to unify, ontologies and the object model of a given programming language, *e.g.* a class in the ontology becomes a class in the programming language. The W3C showed that this approach could reduce the length of the source code by a half [25]. In Figure 1, ontology-oriented programming not only reduces the length of the source code, but also adheres more strictly to the object paradigm, by defining a get_cost() method in the Order class of the ontology.

We can distinguish two approaches for ontology-oriented programming. The *static* approach automatically generates source codes for the entities present in the ontology (*i.e.* classes, properties, *etc*). It relies on a "translator" program that reads the ontology description (typically an OWL file) and produces one or more source files containing the classes in a given programming language. The static approach facilitates the access to the content of the ontology and performs type-check at compile time. In contrast, because of its static nature, this approach cannot perform inference or automatic classification. Several static approaches have been proposed for Java [26, 27] and C# [28].

On the other hand, the *dynamic* approach consists of a dynamic "translation" between the ontology and the object model, at runtime. This approach allows access to the ontology but can also execute a reasoning engine and update the object model according to the produced inferences. For example, the automatic classification can dynamically change the classes to which an individual belongs. Moreover, this approach also allows the integration of methods in the ontology's classes. However, it cannot perform type-checking, since types can change at runtime.

A dynamic tool has been proposed in Common Lisp [24, 29], using a specific subsumption algorithm rather than an external reasoner. A prototype in Python has also been achieved using metaclasses [30]; however, the reasoning and the expression of restrictions remain quite limited. Other authors aimed to design new programming languages specific to ontology-oriented programming, such as Go! [31] (not to confound with the Go programming language from Google).

More recently, a semi-dynamic approach has been proposed in Java [32]. It can perform automatic classification of individuals with a reasoner, with some limits (it is not possible to change the class of an instance in a static programming language such as Java, therefore new instances are created after classification, and the programmer should no longer use the older ones).

3.2. Local closed world reasoning

Many situations require local closed world reasoning in ontologies or databases. We described evidence-based medical knowledge in section 2; in the literature other authors mentioned natural language interfaces [33], Semantic Web Service policies [34] or aerial vehicle planning [35]. However, most of the works were not focused on local closed world reasoning, but more generally on non-monotonic features, which also includes *defeasible inheritance* or default rules (*e.g.* one can consider that the heart is located on the left of the body *unless otherwise specified* for a given patient).

The integrity constraint approach [36] considers some OWL axioms as usual axioms when reasoning on individuals (ABox), but as relational database integrity constraints when reasoning on classes (TBox).

Another approach is the extension of OWL by epistemic operators for non-monotonic features [37]. These operators distinguish known facts (K) and assumed facts (A). Further works lead to hybrid MKNF (Minimal Knowledge and Negation as Failure) knowledge bases [38] which integrate Description Logics and production rules in the same framework. The authors proposed specific algorithms for reasoning on these knowledge bases. Then, other authors defined well-founded semantics based on hybrid MKNF knowledge bases, compatible with OWL [39].

The grounded circumscription approach [40] is an adaptation of circumscriptive Description Logics which remains fully decidable. It restricts the extensions of some classes and properties to a given set of individuals (for classes) or individual pairs (for properties).

Another approach extended the $\mathcal{SROIQ}$ Description Logic with an *NBox* (Negation As Failure Box) specifying the classes or properties to close [41]. Individuals that are not asserted as belonging to a class in the *NBox* are considered as belonging to the complement of the class. This method has been implemented in the TrOWL reasoner[4].

Local closed world assumption was also studied for databases. Databases usually rely on closed world assumption, but local closed world is often desirable, when the database coverage is incomplete. In this context, information is separated in two parts: the first one (M) includes known data, and the second one ($\mathcal{L}$) includes metadata that indicates in which situations the coverage of the database is complete. $\mathcal{L}$ is composed of local closed world assumptions (LCWA), *e.g.* everything is known about the patients in service XYZ, but not for those of other services. Circumscription and quantifier elimination techniques were used for defining

[3] https://jena.apache.org/

[4] http://trowl.org

Static object models (*e.g.* **Java**)	**Dynamic object models** (*e.g.* **Python**)	**Formal ontologies** (*e.g.* **OWL**)
Closed-world assumption	Closed-world assumption	*Open-world assumption*
Implicit disjoints between classes (two classes are disjoint if none of them is a child of the other, two classes cannot be equivalent)	Implicit disjoints between classes (two classes are disjoint if they have no common descendant, two classes cannot be equivalent)	*No implicit disjoint between classes (disjoint classes must be stated explicitly, two classes can be equivalent)*
Implicit distinction between instances (all instances are distinct, two instances cannot be the same)	Implicit distinction between instances (all instances are distinct, two instances cannot be the same)	*No implicit distinction between individuals (distinct individuals must be stated explicitly, two individuals can be the same)*
Single inheritance	*Multiple inheritance*	*Multiple inheritance*
Single instantiation	Single instantiation	*Multiple instantiation*
The class of an instance cannot be changed at runtime	*The class of an instance can be modified at runtime*	*The classes of an individual can be modified to more specific classes by the reasoner*
The superclass of a class cannot be changed at runtime	*The superclasses of a class can be modified at runtime*	*The superclasses of a class can be modified to more specific superclasses by the reasoner*
Attributes are defined for a given class	Attributes are defined for a given class, *but can be considered as independent from the class (duck-typing)*	*Properties are independent from classes (they are first-order constructs with inheritance support)*
No annotation support	No annotation support	*Annotations are supported*

Table 1: Comparison of static object models, dynamic object models and ontologies. Features of ontologies are shown in blue and italic, and different features in red.

a tractable inference method for local closed world reasoning in database [35]. The authors represented LCWA by conjunctions of literals. In a more complete work, Denecker *et al.* [42] studied the semantics of the LCWA. They showed that the complexity of database requests in a local closed world is unacceptably high. The authors proposed approximate methods with acceptable complexity. They also proposed specific categories of queries and locally closed databases for which the problem is tractable.

3.3. Final words

In conclusion to this related works section, several strategies have been proposed for ontology programming interfaces. Various approaches have also been proposed for performing local closed world reasoning. However, there is no existing ontology programming interface supporting local closed world reasoning and including high-level syntax for manipulating classes, the two needs we identified in section 2. For local closed world reasoning, all approaches found in the literature involved new constructs (*e.g. NBox*) and specific reasoning methods. We will propose a different, simpler but more limited, method in section 5.5, based on the automatic generation of universal constraints.

4. Comparison of object models and formal ontologies

We have seen in section 3 that object models and formal ontologies are similar in many aspects. Several object models exist (actually one per object-oriented programming language), and two main categories can be distinguished: *static* object models (*e.g.* in Java), in which classes are compiled and statically typed at compile time, and *dynamic* object models (*e.g.* in Python), which are dynamically typed at runtime. Dynamic object models are generally more permissive, for instance the class of an instance can be changed at runtime. New attributes can also be added to an instance, even if these attributes were not declared at the class level. Table 1 shows the differences between static object models, dynamic object models and formal ontologies. It appears clearly that dynamic object models are closer to ontologies than static ones.

The major differences between dynamic object models and formal ontologies are: 1) whether non-stated facts are considered as false or possible (closed- or open-world assumption, respectively), 2) whether two classes can be equivalent and class disjointness must be stated explicitly, 3) whether two instances/individuals can be the same and the fact that they are distinct must be stated explicitly, 4) whether multiple instantiation is supported, 5) whether attributes/properties are first-order constructs, independent from classes and supporting inheritance, or are defined inside a given class, and 6) whether annotations are supported.

Difference #5 (properties as a first-order construct) has been partly fixed by the recent development of *duck-typing*. Duck-typing is a kind of "programming philosophy" which can be applied to dynamic object-oriented programming languages such as Python or Perl, without having to modify the language. Although attributes are still defined at the class level, duck-typing takes advantage of the ability of dynamic languages to access an attribute value without the need of a type-check. When using duck-typing, the programmer associates the semantics of the attribute with the attribute name only, and not with the (attribute name, class) pair as usual in object models.

In conclusion, dynamic object models are closer to ontologies than static ones are. For developing our ontology-oriented

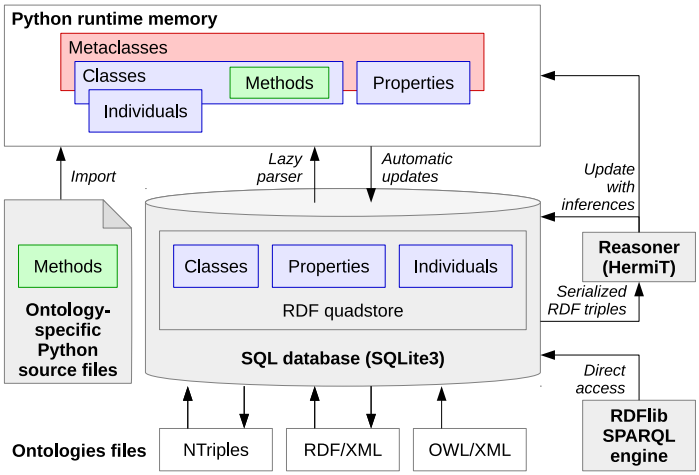


Figure 2: General architecture of Owlready.

programming module, we chose Python because it is a dynamic object-oriented programming language, because it is easy to use and widely used in the biomedical domain, and also because we already developed several tools in this language, including a generic API for medical terminologies [43].

5. Description of Owlready

5.1. General architecture

Figure 2 shows the general architecture of Owlready. It includes 5 components: (1) an optimized RDF quadstore, implemented with an SQL database in SQLite3 and stored either in memory or on disk in a file, (2) metaclasses for OWL classes and constructs, (3) optional ontology-specific Python source files, defining methods to insert into OWL classes, (4) the HermiT OWL reasoner [11], for performing automatic classification, (5) the SPARQL engine from the RDFlib Python module. Owlready is in charge of importing ontologies in the quadstore, dynamically loading their entities and wrapping them in Python objects, importing Python source files associated with the ontologies, and executing the reasoner. Owlready presents to the programmer a unified interface, mixing OWL statements (properties, class hierarchy, individuals) with Python statements (method definitions).

The quadstore relational database includes two tables: a 2-column table mapping IRI (Internationalized Resource Identifier) to shorter alphanumeric codes, and a 4-column table containing RDF quads of the forms (subject, predicate, object, ontology), where object can be an IRI or an RDF literal. Owlready uses a lazy parser for dynamically loading the entities from the quadstore on demand: when an entity (class, property or individual) is accessed in Python, it is loaded in memory from RDF, wrapped in a Python object, stored in a cache and returned. If the wrapper Python object is modified, the RDF quadstore is automatically updated, by adding, removing or changing RDF triples. When the wrapper is no longer referenced and the cache is full, it is destroyed. Then, the entity may be loaded again from RDF if the program needs to access it again. User-defined Python methods can also be associated with OWL classes; in this case, a specific annotation ("python_module") can be used to link an ontology with the name of the associated Python module. Finally, the HermiT reasoner can be used for performing automatic classification of classes and individuals (see section 5.5).

This architecture supports big ontologies, stored in the optimized quadstore. It allows a fast access for "poking around" entities in ontologies.

5.2. Mapping OWL to the Python object model

The objective of the mapping between OWL ontologies and the Python object model was to obtain the most transparent access to ontologies in Python, from a programmer's point of view. The programmer should be able to manipulate the classes and the individuals of the ontology as if they were normal Python classes and objects. Owlready can express almost all OWL 2.0 constructs, including classes, individuals, object properties, data properties, annotations, property domains and ranges, constrained datatypes, disjoints, class expressions such as intersections, unions, property value restrictions and one-of,...

Our implementation in Python relies on metaclasses. An OWL class is represented in Python by an instance of the metaclass, *i.e.* a Python class. Table 2 shows the various special methods that have been redefined in order to adapt OWL ontologies to the Python object model. These special methods are defined either at the metaclass level (*i.e.* they are applied on classes) or at the class level (*i.e.* they are applied on instances).

In the previous section, we identified 6 major differences between dynamic object models and ontologies. The following paragraphs describe how we dealt with each of them.

For difference #1 (closed- or open-world assumption), we choose to keep the closed-world assumption when programming in Python. When reasoning, the default is to rely on the open-world assumption, although we will propose a simple solution for local closed world reasoning in section 5.5.

For difference #2 (class disjoints and equivalent classes), we added an AllDisjoint() construct which states that the given classes are pairwise disjoint. OWL proposes two ways for declaring disjoint classes: the disjointWith relation (when only two classes are involved) and the AllDisjoint construct (supporting more than two classes). On the contrary, Owlready presents a single way for representing disjoints, corresponding to AllDisjoint. disjointWith relations are automatically translated into AllDisjoints, and AllDisjoints with two entities are stored in the quadstore as disjointWith relations. We also added an "equivalent_to" attribute to classes. We reimplemented the Python methods for checking subclasses and instances, to take into account equivalent classes. For example, if $A \equiv B$, when testing $C \sqsubseteq A$, we also test for $C \sqsubseteq B$.

For difference #3 (individual distinction and same individuals), we added an AllDistinct() construct which states that the given list of instances are distinct.

For difference #4 (multiple instantiation), when an instance receives more than one class, we automatically create a new class inheriting from all the classes, and we associate this new class to the instance. This emulates multiple instantiation in Python.

For difference #5 (properties as first-order constructs), we defined properties as subclasses of the ObjectProperty or the DataProperty class. This allows inheritance relations between properties in Python.

For difference #6 (annotations), annotations behave like properties, but they can be applied to any entities (unlike properties that are applied to individuals) and, when applied to classes, their values are not inherited by subclasses. Owlready emulates this behavior in Python, using the __getattr__() and __setattr__() special methods.

OWL provides various datatypes, which are represented by RDF literals. Owlready supports the following datatypes: boolean, integer, float, string, normalized string, localized string, date, time and datetime. A normalized string is a string without line break. A localized string is an RDF literal in a given language. Literals are stored in the quadstore as a string which is

	Special method	Effect	Why we redefined it?
Metaclass	__new__	Create a new class (*i.e.* a new instance of the metaclass)	Combine the new class with the already existent OWL class of the same name, if any
	__instancecheck__	Test if a given object is an instance of this class	Take into account OWL equivalent classes
	__subclasscheck__	Test if a given class is a subclass of this class	Take into account OWL equivalent classes
	mro	Compute the method resolution order (MRO), which is used in case of multiple inheritance	If Python cannot find a consistent MRO, generate an appropriate one. This occurs in case of complex hierarchies with multiple inheritance.
	__setattr__	Set the value of a given class attribute	Set annotation property values, and update the quadstore Define role-filler construct as class property
	__getattr__	Get the value of a given class attribute	Query the existent annotations in the quadstore, if any Query role-filler construct as class property
Class	__setattr__	Set the value of a given attribute of this instance	Set annotations or property values and update the quadstore Update the inverse property, if any
	__getattr__	Get the value of a given attribute of this instance (called only for attributes not defined yet)	Query the existent annotations or relations in the quadstore, and cache the result

Table 2: Special methods that have been redefined in the metaclass (thus executed on classes) and in classes (thus executed on instances) in order to map OWL to the Python object model.

the concatenation of two elements: a serialization of the value and an alphanumeric code identifying the datatype IRI. For example, the number 4 is stored as "4"H, where H is the arbitrary alphanumeric code associated with the following IRI : "http://www.w3.org/2001/XMLSchema#integer".

5.3. Syntax for accessing and modifying the content of an ontology

The syntax of Owlready has been heavily inspired by both the notations of the well-known Protégé editor and the dot notation, widely used in object-oriented programming, including Python. Table 3 shows the Owlready syntax, the corresponding DLs notation, Protégé notation and OWLAPI syntax.

Many object-oriented programming languages, including Python, rely on the dot notation. In order to mimic object-oriented programming, Owlready uses the dot notation at five levels:

1. to access the entities defined in an ontology. Each ontology is considered to define its own namespace, and the dot notation is used to access the entities in this namespace. For example, if a hypothetical ontology (URI http://www.test.org/drug.owl) is loaded in the "drug_onto" variable, the Drug class defined in this ontology (URI http://test.org/drug.owl#Drug) can be accessed as "drug_onto.Drug". Additional methods are provided to obtain the list of classes, properties, individuals or annotations in an ontology, and so on.
2. to access or modify the value of a property for an individual. For example, to obtain the values of the "hasIngredient" property for the individual in the "test_drug" variable, one can write "test_drug.hasIngredient". Functional properties return a single value, defaulting to None (the Python null value) if not defined. Non-functional properties return a list of values (lists are written with brackets [...] in Python, and the .append() method is used to add an item to a list). Finally, the values of inverse properties are automatically updated.
   In ontologies, properties are often considered as relations and thus named with a verb, *e.g.* hasX instead of X, but this is not usual in programming languages. Therefore, Owlready proposes the "python_name" annotation that can be used to specify a different name when using the dot notation. For instance, after defining the (hasIngredient, python_name, ingredients) annotation triple, one can now access to the values of the property in a more conventional way: "test_drug.ingredients".
3. to access or modify the annotations for any entity (class, property or individual). This works in a similar manner that for properties, but with annotation properties.
4. to access or modify "role-filler" constructs associated with a class (*i.e.* property value constraints involving individuals, noted $\exists R.\{i\}$). From an object-oriented point of view, these role-filler constructs can be seen as *class attributes*. Thus, in Owlready, we represented role-fillers as class attributes. The last line of Table 3 shows the general syntax we propose and its equivalence in formal notation. For example, to assert that all drugs belonging to the AspirinDrug class have for administration route oral (an individual of the class AdministrationRoute), *i.e.* AspirinDrug $\sqsubseteq \exists$has_for_administration_route.{oral}, one can write "AspirinDrug.has_for_administration_route = oral" (assuming that has_for_administration_route is functional). In addition, an existential constraint is automatically created on the individual using the inverse property, *e.g.* in the previous example, ($\exists$has_for_administration_route^{-}.AspirinDrug)(oral).
5. to access or modify the domain, the range and the inverse of a property. For example, the range of the has_for_administration_route property can be defined as following: "has_for_administration_route.domain = [Drug]".

Table 3: Correspondence between DLs, Protégé notations, Owlready syntax and OWLAPI syntax. A and B are classes, R and S are properties, i and j are individuals, n is a literal. The last line of the table shows the Owlready syntax for asserting role-fillers as class attributes, and the corresponding assertion in DLs.

DLs	Protégé	Owlready	OWLAPI
			OWLOntologyManager m = OWLManager.createOWLOntologyManager(); OWLDataFactory df = OWLManager.getOWLDataFactory(); OWLOntology o = m.createOntology(MY_IRI);
A ⊑ B	A subclass of B	class A(B): ... (or) A.is_a.append(B)	m.applyChange(new AddAxiom(o, df.getOWLSubClassOfAxiom(A, B)));
A ⊓ B	A and B	A & B	df.getOWLObjectIntersectionOf(A, B)
A ⊔ B	A or B	A | B	df.getOWLObjectUnionOf(A, B)
¬ A	not A	Not(A)	df.getOWLObjectComplementOf(A)
A ⊓ B = ∅	A disjoint with B	AllDisjoint([A, B])	m.applyChange(new AddAxiom(o, df.getOWLDisjointClassesAxiom(A, B)));
A ≡ B	A equivalent to B	A.equivalent_to.append(B)	m.applyChange(new AddAxiom(o, df.getOWLEquivalentClassesAxiom(A, B)));
{i, j,...}	{i, j,...}	OneOf([i, j,...])	df.getOWLObjectOneOf(i, j, ...)
∃R.B	R some B	R.some(B)	df.getOWLObjectSomeValuesFrom(R, B)
∀R.B	R only B	R.only(B)	df.getOWLObjectOnlyValuesFrom(R, B)
=2R.B	R exactly 2 B	R.exactly(2, B)	df.getOWLObjectExactCardinality(2, R, B)
∃R.{i}	R value i	R.value(i)	df.getOWLObjectHasValue(R, i)
∃ R.⊤ ⊑ A	R domain A	R.domain = [A]	m.applyChange(new AddAxiom(o, df.getOWLObjectPropertyDomainAxiom(R, A)));
⊤ ⊑ ∀R.B	R range B	R.range = [B]	m.applyChange(new AddAxiom(o, df.getOWLObjectPropertyRangeAxiom(R, B)));
S ≡ R⁻	S inverse of R	S.inverse = R	m.applyChange(new AddAxiom(o, df.getOWLInverseObjectPropertiesAxiom(R, S)));
A(i)	i type A	i = A() (or) i.is_instance_of.append(A)	m.applyChange(new AddAxiom(o, df.getOWLClassAssertionAxiom(A, i)));
R(i, j)	i object property assertion j	i.R = j (R is functional) (or) i.R.append(j) (otherwise)	m.applyChange(new AddAxiom(o, df.getOWLObjectPropertyAssertionAxiom(R, i, j)));
R(i, n)	i data property assertion j	i.R = n (R is functional) (or) i.R.append(n) (otherwise)	m.applyChange(new AddAxiom(o, df.getOWLDataPropertyAssertionAxiom(R, i, n)));
A⊑∃R.{i} ∧ (∃R⁻.A)(i)	-	A.R = i (R is functional) (or) A.R.append(i) (otherwise)	-

In OWL, all statements regarding a given entity do not need to be asserted in the same ontology. Owlready uses the "with <object>: code bloc" Python syntax for specifying the ontology in which any new RDF triples are asserted. When a triple is asserted outside a "with..." code bloc, it goes into the first ontology that has been associated with the entity.

5.4. Loading and exporting ontology files

Owlready natively supports three ontology file formats: NTriples, RDF/XML and OWL/XML, using simple parsers written in Python. When loading an ontology, Owlready automatically loads all imported ontologies, recursively.

To ease access to local copies of ontologies, we also defined an "onto_path" global variable. It contains a list of directories in which local copies are searched. It thus behaves similarly to the Java classpath or the Python PYTHON_PATH environment variable. If an ontology is not found in those directories, it is downloaded from its IRI, which usually corresponds to an Internet address.

Errors detected when loading an ontology are reported by raising an exception specific to Owlready (OwlReadyOntologyParsingError). Owlready uses several methods for loading ontologies, depending on the file format. For all methods, we followed the same error reporting scheme and we used the same exception when an error is encountered. In addition, if some RDF triples were already loaded when the error occurred, they are removed from the quadstore. Consequently, a failed load leaves the quadstore unchanged. When an ontology cannot be found, a HTTPError exception is raised. Since each ontology has an IRI, it is expected to be available online. Therefore, missing ontologies lead to an HTTP error.

The quadstore can be stored in memory (default behavior) or on disk (in a file). When stored on disk, it allows persistence: all ontologies previously loaded are recovered when reusing the same quadstore file. This feature is interesting for huge ontologies, because opening the database from disk takes only a fraction of a second.

Finally, Owlready can export ontologies in the NTriples and the RDF/XML formats (OWL/XML is not yet supported for exporting).

5.5. Automatic classification and reasoning

Owlready performs almost no inferences until the reasoner is explicitly executed. This behavior is similar to the one of the Protégé editor, in which the user must explicitly call the reasoner by clicking on a menu. The only exception is the values of inverse properties, which are automatically updated by Owlready.

The reasoner is executed by calling the sync_reasoner() global function. This function exports the RDF quadstore in a temporary file in the NTriple format. Then it runs the HermiT reasoner on that file. We used HermiT version 1.3.8 and we modified it in order to generate on the standard output the classification of the classes, properties and individuals in plain text (HermiT does not propose the classification of individuals in the command-line options, although the reasoner performs the classification). The sync_reasoner() function retrieves and parses the output produced by HermiT. Finally, it updates the ontologies in Owlready with the inferred statements deduced by HermiT. These statements are new is-a, is-instance-of, equivalent-to and same-as relations.

The update includes two steps: (1) updating the RDF quadstore by adding RDF triples for the inferred statements, and (2) updating the wrapper Python objects, if they have been loaded and are still available in the cache. For example, if HermiT deduced that $A \sqsubseteq B$, the sync_reasoner() function will add the (A, $subclassOf$, B) RDF triple in the quadstore. In addition, if the class A has been loaded in Python, the superclasses of the Python class A will be modified to include the class B. If the class A has not been loaded yet, but is loaded after executing the reasoner, the new (A, $subclassOf$, B) RDF triple will be taken into account when loading the class and thus the class A will include B in its superclasses, as expected.

The inferred facts can be tested in Python using the standard methods for object introspection. For example, the standard Python function issubclass(A, B) can be used for testing whether class A is a subclass of B. Owlready can also save the inferred facts in a new ontology.

In section 2.1, we have seen that local closed world reasoning is often needed for medical reasoning. Owlready relies on HermiT for reasoning, and HermiT uses the open world assumption. For permitting local closed world reasoning, Owlready provides the close_world() function. It expects a single argument, which is an individual or a class. This function acts like a "preprocessor" that generates the appropriate universal constraints needed for considering a given individual or class in a closed world. For an individual, it means that the individual relations are limited to asserted relations and existential constraints. For a class, it means that all individuals of the class are known, and that all subclasses and individuals of the given class are also considered in a closed world.

The constraints generated by close_world() depend on the asserted relations (for individuals), restrictions (for individuals and classes) and individuals (for classes). Algorithm 1 details the close_world() function. For an individual, for each property, it considers both the asserted values (set I in the algorithm) and the existential constraints (C), and it creates an appropriate universal constraint (excepted for functional properties, in which case it is not needed). For classes, the function performs 3 steps. First, if the class has individuals, it creates a *one-of* constraint (other-

Algorithm 1 Algorithm for the close_world() function. $\bigsqcup R$ denotes the union of all concepts in the set R. The algorithm is executed in Python and thus run in "closed world", *i.e.* it only considers asserted facts.

function close_world(entity e):
  if e is an individual:
    for each property p whose domain is compatible with e:
      $I = \{\ i \mid p(e, i)\ \}$
      $C = \{\ c \mid (\exists p.c)(e)\ \}$
      $R = C \cup \{\ I\ \}$
      if $R = \emptyset$: **assert** $(\forall p.\bot)(e)$
      else if p is not functional: **assert** $(\forall p.\bigsqcup R)(e)$
  else if e is a class:
    $I = \{\ i \mid e(i)\ \}$
    if $I \neq \emptyset$: **assert** e $\sqsubseteq I$
    for each property p whose domain is compatible with e:
      $I = \{i \mid \exists e' \sqsubseteq e \sqcap \exists p.\{i\}\ \}$
        $\cup\ \{i \mid \exists i',\ e(i') \wedge p(i', i)\ \}$
      $C = \{c \mid \exists e' \sqsubseteq e \sqcap \exists p.c\ \}$
        $\cup\ \{c \mid \exists i',\ e(i') \wedge (\exists p.c)(i')\ \}$
      $R = C \cup \{\ I\ \}$
      if $R = \emptyset$: **assert** $e \sqsubseteq \forall p.\bot$
      else: **assert** $e \sqsubseteq \forall p.(\bigsqcup R)$
    for each class c such as $c \sqsubseteq e$: close_world(c)
    for each individual i such as $e(i)$: close_world(i)

wise, we suppose that individuals have not been asserted, although they might exist, the only alternative being considering the class as equivalent to Nothing). Second, for each property, it creates a universal constraint, depending on the defined constraints on the class, on its ancestors, its descendants and their individuals. Third, it calls itself recursively on subclasses and individuals.

Consequently, by calling close_world() on $\top$, it is possible to reason in an entirely closed world (although usually better options exist for that). By calling close_world() on a limited set of individuals or classes, it is possible to reason in a local closed world.

For example, if we consider an ontology with the following assertions:

$A \sqsubseteq \exists p.B$

$A \sqsubseteq \exists p.\{c\}$

Then close_world(A) would generate the following universal constraints:

$A \sqsubseteq \forall p.(B \sqcup \{c\})$

If switching back to open world reasoning is needed, the best option is to assert the constraints generated by close_world() in a new ontology (using the "with..." statement as described before). Then, removing this ontology from the quadstore removes the constraints, reversing to open world reasoning.

5.6. Adding methods to OWL classes

With Owlready, an OWL class is a Python class, thus it is easy to define methods in these classes. The methods can be defined either directly in the program, or in a separate .py Python source file. Methods can take any parameters (including entities defined in the ontology) and they can be called as any other method in Python, using the dot notation. Examples will be provided in sections 5.8 and 5.9.

5.7. Implementation

Owlready has been implemented in Python 3 using agile development methods and unit tests (see section 5.10 for more details on tests). The first version of Owlready was a "proof of concept" that proposed ontology-programming in Python; it loaded the entire ontology in memory. Version 2 (which is the one described here) is based on an optimized RDF quadstore. It is more robust and supports big ontologies with millions of triples (see example with Gene Ontology in section 5.9).

Owlready is a free software, available under the GNU GPL v3 license. It can be downloaded from the Python Package Index or from BitBucket:

- https://pypi.python.org/pypi/Owlready2 (stable version)
- https://bitbucket.org/jibalamy/owlready2 (development)

The documentation of Owlready can be consulted online:

- http://pythonhosted.org/Owlready2/

5.8. Example #1: reasoning

Figure 3 shows a simple example of a drug ontology associated with a Python module with method definitions. The classes are redefined in Python for adding methods, but their parent classes and their definitions are not repeated. Thus, there is no duplicated information. The third files shown in Figure 3 is a small test program that loads the ontology (which in turn imports the drug.py Python module), creates a test drug, runs the reasoner, and calls the take() method. As the reasoner classifies the test drug in DrugCILactoseIntolerance, the implementation in DrugCILactoseIntolerance.take() will be executed. This example uses polymorphism with ontology classes.

Owlready also proposes constructs for creating an ontology entirely in Python, including class and property definitions, with full support for complex constraints. This is especially useful for generating ontologies from other source of data, *e.g.* from the content of a database. Figure 4 shows an example of an ontology fully defined in Python, corresponding to the ontology of Figure 3.

OWL ontology (drug.owl):

$$
\begin{aligned}
Drug &\sqsubseteq Thing \\
Ingredient &\sqsubseteq Thing \\
ActivePrinciple &\sqsubseteq Ingredient \\
Aspirin &\sqsubseteq ActivePrinciple \\
Excipient &\sqsubseteq Ingredient \\
Lactose &\sqsubseteq Excipient \\
DrugCILactoseIntolerance &\equiv Drug \\
&\quad \sqcap (\exists hasIngredient.Lactose) \\
DrugOKLactoseIntolerance &\equiv Drug \\
&\quad \sqcap \neg(\exists hasIngredient.Lactose)
\end{aligned}
$$

Annotation triple: (*drug.owl, python_module, drug.py*)

Python module with methods (drug.py):

```
from owlready2 import *
onto_path.append("/path/to/local/copy")
drug_onto = get_ontology("http://.../drug.owl")


with drug_onto:
    class DrugCILactoseIntolerance(Thing):
        def take(self):
            print("Baah!")


    class DrugOkLactoseIntolerance(Thing):
        def take(self):
            print("Ok, it is safe for me.")
```

Python script for testing (test.py):

```
from owlready2 import *
onto_path.append("/path/to/local/copy")
drug_onto = get_ontology("http://.../drug.owl")
drug_onto.load()

test_drug = drug_onto.Drug()
test_drug.hasIngredient = [
      drug_onto.Aspirin(),
      drug_onto.Lactose() ]

sync_reasoner()
test_drug.take()
# Prints "Baah!"
```

Figure 3: Example of an OWL ontology complemented by a Python module with methods. CI stands for "contraindicated with".

5.9. Example #2: using Gene Ontology

Owlready is able to load big ontologies with millions of triples, such as Gene Ontology (GO) [8]. GO includes 1,567,718 triples defined in a 178 Mb RDF/XML file. GO includes three distinct hierarchies of concepts: cellular components, molecular functions and biological processes. In the OWL version of GO, a GO concept is represented by an OWL class. Hierarchical relations between concepts are represented by *subclass of* relations, *e.g.* $GO1 \sqsubseteq GO2$. Other relations, such as *part of* relations, are represented using restriction with existential qualifier, *e.g.* $GO1 \sqsubseteq \exists part\,of.GO3$.

Python script (drug2.py):

```
from owlready2 import *
drug_onto = get_ontology("http://.../drug.owl")

with drug_onto:
    class Drug(Thing):  pass

    class Ingredient(Thing):  pass

    class ActivePrinciple(Ingredient):  pass

    class Aspirin(ActivePrinciple):  pass

    class Excipient(Ingredient):  pass

    class Lactose(Excipient):  pass

    class hasIngredient(ObjectProperty):
        domain = [Drug]
        range  = [Ingredient]

    class DrugCILactoseIntolerance(Drug):
        equivalent_to = [
            Drug
          & hasIngredient.some(Lactose)
            ]
        def take(self):
            print("Baah!")

    class DrugOkLactoseIntolerance(Drug):
        equivalent_to = [
            Drug
          & Not(hasIngredient.some(Lactose))
            ]
        def take(self):
            print("Ok, it is safe for me.")
```

Figure 4: Example of an ontology (an excerpt of a drug ontology) entirely defined in Python.

Figure 5 shows a small program that uses GO. Line #1 imports Owlready. Line #2 loads GO in the "go" variable. Line #3 defines a namespace for accessing GO concepts. This is needed because the IRIs of GO concepts do not start with the ontology IRI. Line #4 uses the namespace for accessing the concept GO_0006310.

In ontologies, entities have often arbitrary names or IDs, and their Human-comprehensive name is defined using the "label" annotation. In order to display the entities in a more friendly manner, lines #5-9 define a rendering function. This function take one parameter, the entity, and returns a string for displaying this entity. The function defined here generates strings of the form "ID:label", or "ID" if no label is available. The .first() method of the list returns the first element (here, the first label), or None if the list is empty. Line #10 prints the concept GO_0006310 again, and now its label appears, "DNA recombination".

Lines #11-14 iterate over all classes in GO, and for each class, it prints the class and its superclasses.

Figure 6 shows a more complex example with GO, and illustrates the uses of Python methods in OWL classes. In GO, the cellular component hierarchy uses *part of* relations in addition to *subclass of* relations. This example will extend the cellular component OWL class with an additional method for dealing with the *part of* hierarchy. Then, it will extract the sets of all GO concepts related to the nucleus using two different methods: linguistic and semantic. Finally, it will compare the two results and verify that all GO concepts mentioning "nucleus" are indeed related to the nucleus GO concept.

Lines #1-3 import Owlready, load GO and create a namespace as before. Line #4 begins a "with..." code block, indicating that any entity created in this code block will be located in the "obo" namespace of the GO ontology. Lines #5-19 redefine the class "GO_0005575", which is the "Cellular_component" GO concept. The new class definition declares a method called "subparts()", which returns the list of the subparts of a GO concept. This method is declared as a Python *class method* (line #6), *i.e.* it is executed on the class itself (or one of its descendants) and not on its instances. The method creates an empty list (line #8), iterates over all OWL class constructs referring to the class (line #9), tests whether the construct is a restriction on the property "BFO_0000050" (part of) (line #10) and, if so, extends the list of results with the subclasses of the construct. The ".subparts()" method is now available for concept GO_0005575 (Cellular_component), but also all its descendants (thanks to inheritance).

Line #13 creates an empty set named *linguistic*. Line #14 searches all GO concepts whose label includes "nucleus". Line #15 tests whether the concept is a descendant of GO_0005575 (Cellular_component) and, if so, line #16 adds the concept to the *linguistic* set.

Line #17 creates an empty set named *semantic*. The *semantic* set will include all concepts related to GO_0005634 (nucleus) by subclass-of and/or part-of relations (including possibly a mix of both). Thus, we need a recursive function for computing the set. Line #18-22 define the function: it has one parameters, a GO concept x, and it tests whether x is already in the *semantic* set. If not, it adds x to *semantic* and calls the function recursively on all subclasses and subparts of x (using the previously defined subparts() method). Finally, line #23 calls the function of GO_0005634 (nucleus).

Line #24 computes and prints the set difference between *linguistic* and *semantic*. Two results are found: GO_0071561 (nucleus-vacuole junction) and GO_0042025 (host cell nucleus). The first one involves the junction between nucleus and vacuole, which explains why it is not a part of the nucleus itself. The second is not the nucleus of the cell but of another cell, the host.

The supplementary computer code "go.txt" includes the entire source code for the second GO example, with additional methods (superparts(), *etc.*).

[Insert Supplementary Computer Code 1 here]

The results show that all GO concepts mentioning "nucleus" in their label are related to the nucleus, with two exceptions that can be justified. This process could be repeated for other cellular components (*e.g.* Golgi apparatus, cytoplasm, *etc.*) for performing a simple audit of GO. The entire audit is beyond the scope of this paper; however, several auditing methods have been proposed for biomedical terminologies [44], including linguistic and semantic ones.

We were unable to reproduce this second example with SPARQL, and more specifically to write a SPARQL query for obtaining the *semantic* set. Figure 7 shows an example of RDF graph with 4 concepts A, B, C, D. When computing the *semantic* set on this graph, and starting with concept A, the expected results is $\{A, B, C\}$ (because B is a part of A and C is a part of B); on the contrary D should not be included in the set because it is related

Python script (go1.py):

```
 1  from owlready2 import *
 2  go = get_ontology("http://purl.obolibrary.org/obo/go.owl").load()
 3  obo = go.get_namespace("http://purl.obolibrary.org/obo/")

 4  print(obo.GO_0006310)
 # prints obo.GO_0006310

 5  def render(entity):
 6      label = entity.label.first()
 7      if label:  return "%s:'%s'" % (entity.name, label)
 8      return entity.name
 9  set_render_func(render)

10  print(obo.GO_0006310)
# prints GO_0006310:'DNA recombination'

11  for go_concept in go.classes():
12      print(go_concept)
13      for parent in go_concept.is_a:
14          print("    is a %s" % parent)
# prints:
# GO_0000001:'mitochondrion inheritance'
#     is a GO_0048308:'organelle inheritance'
#     is a GO_0048311:'mitochondrion distribution'
# GO_0000002:'mitochondrial genome maintenance'
#     is a GO_0007005:'mitochondrion organization'
# ...
```

Figure 5: Example of a small program that imports Gene Ontology, declares a rendering function, and finally displays each GO concept with its parent classes.

Python script (go2.py):

```
 1  from owlready2 import *
 2  go  = get_ontology("http://purl.obolibrary.org/obo/go.owl").load()
 3  obo = go.get_namespace("http://purl.obolibrary.org/obo/")

 4  with obo:
 5      class GO_0005575(Thing):  # Redefines Cellular_component class
 6          @classmethod
 7          def subparts(self):
 8             results = []
 9             for construct in self.constructs():
10                if isinstance(construct, Restriction) and (construct.property is obo.BFO_0000050):
11                   results.extend(construct.subclasses())
12             return results

13  linguistic = set()
14  for x in go.search(label = "*nucleus*"):
15      if issubclass(x, obo.GO_0005575):
16          linguistic.add(x)

17  semantic = set()
18  def found(x):
19      if not x in semantic:
20          semantic.add(x)
21          for y in x.subclasses():  found(y)
22          for y in x.subparts():  found(y)
23  found(obo.GO_0005634)

24  print(linguistic - semantic)
# prints { GO_0071561:'nucleus-vacuole junction',
#          GO_0042025:'host cell nucleus' }
```

Figure 6: Example of a small program for auditing GO. The program imports GO, defines the subparts() class method, and then searches for all cellular components related to "nucleus" using a linguistic approach and a semantic approach. Finally, it compares the results of both approaches.

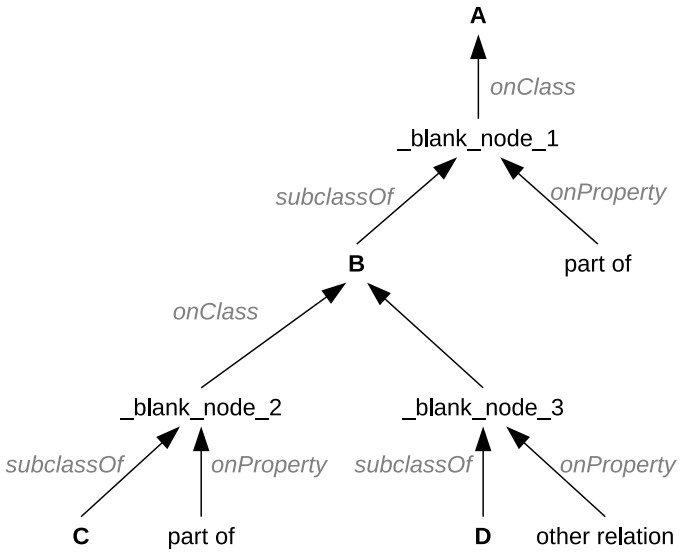


Figure 7: Example of an RDF graph with 4 concepts A, B, C, D such as $B \sqsubseteq \exists part\ of.A$, $C \sqsubseteq \exists part\ of.B$ and $D \sqsubseteq \exists other\ relation.B$.

to B with a relation that is not "part of". Using SPARQL property paths, it is possible to chain subclassOf and onClass relations (*e.g.* using the following syntax "(subclassOf | onClass)*"). However, it is not possible to restrict the query to blank nodes having the "onProperty part-of" relation. In the previous example, we can obtain the set $\{A, B, C, D\}$ but not $\{A, B, C\}$ as desired.

5.10. Tests and benchmarks

We tested Owlready by two means. First, we wrote a set of 190 unit tests. These tests represent about 40% of the volume of Owlready source code. In particular, there are 9 tests for the "role-filler as class property" syntax, and 7 for the close_world() function. Unit tests were used during the development of Owlready.

Second, we tested Owlready on 5 biomedical ontologies of various sizes (see Table 4 for the list of ontologies). For each ontology, we loaded it in Owlready, and we executed a script that systematically tested all individuals and classes. For each individual, the script tries to obtain its classes, and to access to the values of all properties (*i.e.* it tries all individual-property combinations). For each class, it tries to obtain its superclasses, its subclasses, its equivalent classes (including indirect ones), its ancestors, its descendants, its individuals, and the disjoint and class constructs referring to the class. The test on the 5 ontologies was performed after the development of Owlready. It led to the discovery of 3 bugs that were fixed; all of them were discovered when testing the first ontology.

We also benchmarked Owlready on the 5 ontologies: we measured the time for loading the ontology, the time for listing all classes with their superclasses (similarly to lines #11-14 in Figure 5) and the memory consumption. Table 4 shows the results with Owlready and OWLAPI. Owlready has decent performances, although not as good as OWLAPI. Owlready also uses much less memory. These differences can be explained by the programming languages (Java being several times faster than Python) but also by the different approaches and architectures (OWLAPI works at the axiom intermediate level, while Owlready works both at the RDF low level and the object high level).

	OWLAPI	**Owlready**	**Owlready**
Language	Java	Python	Python
Storage	memory	memory	disk
OBI (54,991 triples, 3,071 classes)			
http://purl.obolibrary.org/obo/obi.owl			
Memory	171 Mb	28 Mb	24 Mb
Disk space	-	-	6 Mb
Loading time	2.4 s	0.7 s	0.9 / 0.01 s
Listing time	0.3 s	0.6 s	0.6 s
Uberon (655,680 triples, 15,036 classes)			
http://purl.obolibrary.org/obo/uberon.owl			
Memory	605 Mb	122 Mb	63 Mb
Disk space	-	-	57 Mb
Loading time	8.3 s	9.2 s	9.6 / 0.02 s
Listing time	0.8 s	3.6 s	3.5 s
VTO (1,397,257 triples, 107,138 classes)			
http://purl.obolibrary.org/obo/vto.owl			
Memory	1,131 Mb	356 Mb	245 Mb
Disk space	-	-	105 Mb
Loading time	13.1 s	19.5 s	19.7 / 0.01 s
Listing time	3.4 s	16.0 s	16.2 s
GO (1,567,718 triples, 48,535 classes)			
http://purl.obolibrary.org/obo/go.owl			
Memory	1,123 Mb	330 Mb	157 Mb
Disk space	-	-	155 Mb
Loading time	14.1 s	21.3 s	22.6 / 0.01 s
Listing time	3.2 s	9.5 s	9.1 s
DRON (4,923,381 triples, 492,577 classes)			
http://purl.obolibrary.org/obo/dron.owl			
Memory	1,655 Mb	832 Mb	481 Mb
Disk space	-	-	314 Mb
Loading time	33.1 s	71.8 s	72.3 / 0.02 s
Listing time	10.5 s	183.9 s	183.4 s

Table 4: Benchmark of OWLAPI and Owlready for loading 5 ontologies of various sizes from local copies, and for listing all classes with their parent classes. For Owlready with disk storage, the first loading time corresponds to the initial import in the quadstore, and the second time corresponds to subsequent execution reusing the same quadstore.

6. Use case: reasoning on drug contraindications

The use case consists in reasoning on contraindications. It illustrates the two features of Owlready that were inspired by the particularities of medical ontologies: local closed world reasoning and the easy definition of "role-filler" on classes using the dot notation.

This use case took place during the VIIIP research project (Integrated Visualization of Information on Pharmaceutical Innovation), founded by the French drug agency. Today, information on new drugs is provided to physicians mostly by pharmaceutical company sales representatives [45, 46]. However, they are not independent of the companies and thus they may deliver biased information: a review showed that the physician's exposure to information from pharmaceutical companies was associated with higher prescribing frequency, higher costs, lower prescribing quality, or no effect, but never with net improvements in prescribing quality [47].

The objective of the VIIIP project was to design a platform for the diffusion of independent information on new drugs. As the independence of experts is sometimes questionable and difficult

	ticagrelor	heparin	aspirin
hemorrhagic disorder	**CI 1**		
acquired hemorrhagic disorder			**CI 3**
constitutive hemorrhagic disorder		**CI 2**	**CI 4**

	ticagrelor	heparin	aspirin
hemorrhagic disorder	**CI 1**	**CI**/Ok	**CI**
acquired hemorrhagic disorder	**CI**	Ok	**CI 3**
constitutive hemorrhagic disorder	**CI**	**CI 2**	**CI 4**

Table 5: Examples of contraindications for 3 drugs for thrombosis prevention, as extracted from drug databases or reference documents (top) and as interpreted by an expert (bottom). CI stands for contraindication, and Ok for the verified absence of contraindications.

to assess, the information of the platform was not based on expert opinions but rather produced from raw data extracted from drug databases and official reference texts (*Summary of Product Characteristics*, SPC). Consequently, the platform proposes tables and other visualizations comparing the properties of the new drugs with the properties of the already existing drugs for the same indication. However, drug properties (contraindications, interactions, adverse effects, *etc*) are often expressed at various granularity levels, which impairs the comparison. In this use case, we will focus on the comparison of the contraindications, *i.e.* the interactions that exist between a drug and a disorder[5].

Table 5 (top) shows 4 examples of contraindications for 3 drugs. But this table does not allow an easy comparison, because the 3 disorders (hemorrhagic disorder, acquired hemorrhagic disorder, constitutive hemorrhagic disorder) are not independent from each other. In fact, although it is not shown in the table (or specified in drug databases and reference documents), ticagrelor is *a fortiori* contraindicated with acquired and constitutive hemorrhagic disorders, because it is contraindicated with all hemorrhagic disorders (subsumption). Similarly, aspirin is contraindicated with hemorrhagic disorders, because it is contraindicated with both acquired and constitutive hemorrhagic disorders (partition). In addition, if we consider that all contraindications are known (as a physician usually does), it is possible to deduce the *absence* of contraindications, such as acquired hemorrhagic disorder for heparin. Table 5 (bottom) shows how a medical expert interprets this small dataset, including deduced contraindications and absences of contraindications ("Ok" in the table). The reasoning for determining contraindications and absences of contraindications can become complex, even for experts, when the number of drugs and disorders increases, and when multiple inheritance and many partitions are involved. The objective of the use case is to automatize this reasoning with an ontology.

We first built an ontology of contraindications. Drugs and drug therapeutic classes were extracted from the French Thériaque drug database. Disorders and contraindications were extracted from SPCs and coded manually by an expert pharmacist, to avoid the errors present in the database (we actually encountered a surprisingly high number of errors in the various drug databases we tested). Inheritance relations between disorders were obtained from the ICD10 (International Classification of Diseases, release 10) medical terminology. Partitions were determined using NLP (Natural Language Processing) methods and then reviewed by the expert. As stated in section 2.2, drugs and disorders cannot be represented by individuals, and need to be modeled as classes in order to allow *is-a* relations between them. We also defined a Contraindication class, related to both drugs and disorders. The resulting ontology belongs to the $\mathcal{ALCOI}$ family of Description Logics. Figure 8 (top) shows an excerpt of this ontology corresponding to the examples of Table 5.

For each drug, contraindications were manually extracted from reference documents (corresponding to the Table 5, top). They were then asserted in the ontology using Owlready, in five steps (see example in Figure 8, bottom, corresponding to Table 5):

1. Create an individual for each contraindication.
2. Relate each drug to its contraindications.
3. Relate each disorder to its contraindications.
4. Assert that everything is known about contraindications.
5. Assert that everything is known about drugs.

Step 1 consists in creating individuals and presents no difficulties.

Step 2 and 3 consist in creating "role-filler" constraints on classes (drugs and disorders) and existential constraints on individuals (contraindications). This can be achieved easily using the dot notation with the syntax we proposed for "role-filler": "Class.property = values" (see section 5.3).

Step 4 and 5 consist in asserting that everything is known on contraindications and drugs (*i.e.* the reasoner should not make hypothesis involving unknown contraindications). This can be achieved easily by calling the close_world() function on the Contraindication and the Drug classes (which recursively close all instances of Contraindication and all subclasses of Drug).

On the contrary, the Disorder class must remain in open-world if we want to deduce contraindications through the reasoning (using subsumption and partitions). Therefore, the reasoning occurs in a local closed world.

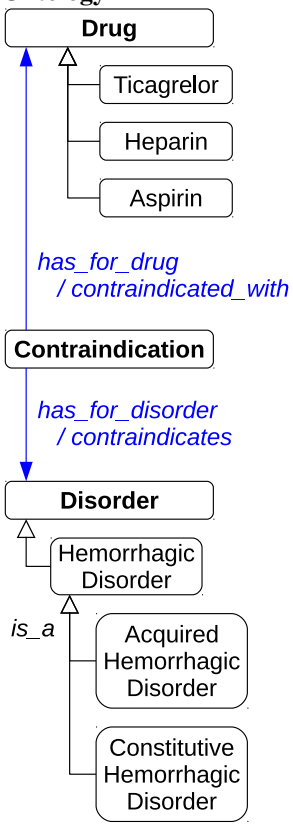
The two following classes can be used to classify the disorders with regard to aspirin:

$$DisorderContraindicatingAspirin \equiv Disorder \sqcap \exists contraindicates.(\exists has\_for\_drug.Aspirin)$$
$$DisorderOkWithAspirin \equiv Disorder \sqcap \neg\exists contraindicates.(\exists has\_for\_drug.Aspirin)$$

Similar classes can be defined for the other drugs. After running the reasoner, disorders will be automatically classified according to these defined classes. All disorders contraindicating aspirin will be asserted as subclasses of DisorderContraindicatingAspirin ("CI" in table 5, bottom), all disorders for which the absence of contraindications with aspirin can be proved will be asserted as subclasses of DisorderOkWithAspirin ("Ok" in the

[5] In practice, contraindications can involve disorders but also more general *clinical conditions*, such as pregnancy or age classes. For the sake of simplicity and brevity, we will continue to speak about disorders, although clinical conditions would be a more appropriate term from a medical point of view.

Ontology



$Drug$	$\sqsubseteq$	$\top$
$Disorder$	$\sqsubseteq$	$\top$
$Contraindication$	$\sqsubseteq$	$\top$
$Disorder, Drug, Contraindication$	*are*	*pairwise disjoint*
has_for_drug	*domain*	$Contraindication$
has_for_drug	*range*	$Drug$
$has\_for\_disorder$	*domain*	$Contraindication$
$has\_for\_disorder$	*range*	$Disorder$
$contraindicated\_with$	$\equiv$	$has\_for\_drug^-$
$contraindicates$	$\equiv$	$has\_for\_disorder^-$
$HemorrhagicDisorder$	$\sqsubseteq$	$Disorder$
$AcquiredHemorrhagicDisorder$	$\sqsubseteq$	$HemorrhagicDisorder$
$ConstitutiveHemorrhagicDisorder$	$\sqsubseteq$	$HemorrhagicDisorder$
$AcquiredHemorrhagicDisorder,$ $ConstitutiveHemorrhagicDisorder$	*are*	*pairwise disjoint*
$HemorrhagicDisorder$	$\sqsubseteq$	$AcquiredHemorrhagicDisorder$ $\sqcup ConstitutiveHemorrhagicDisorder$
$Ticagrelor$	$\sqsubseteq$	$Drug$
$Heparin$	$\sqsubseteq$	$Drug$
$Aspirin$	$\sqsubseteq$	$Drug$
$Ticagrelor, Heparin, Aspirin$	*are*	*pairwise disjoint*

	Python program	**Corresponding assertions in formal notation**

(1)	`ci1 = Contraindication()`
	`ci2 = Contraindication()`
	`ci3 = Contraindication()`
	`ci4 = Contraindication()`
		$Contraindication(ci1)$
		$Contraindication(ci2)$
		$Contraindication(ci3)$
		$Contraindication(ci4)$

(2)	`Ticagrelor.contraindicated_with = [ci1]`
	
	`Heparin.contraindicated_with = [ci2]`
	
	`Aspirin.contraindicated_with = [ci3, ci4]`
		$Ticagrelor \sqsubseteq \exists contraindicated\_with.\{ci1\}$
		$(\exists has\_for\_drug.Ticagrelor)(ci1)$
		$Heparin \sqsubseteq \exists contraindicated\_with.\{ci2\}$
		$(\exists has\_for\_drug.Heparin)(ci2)$
		$Aspirin \sqsubseteq \exists contraindicated\_with.\{ci3\}$
		$(\exists has\_for\_drug.Aspirin)(ci3)$
		$Aspirin \sqsubseteq \exists contraindicated\_with.\{ci4\}$
		$(\exists has\_for\_drug.Aspirin)(ci4)$

(3)	`HemorrhagicDisorder.contraindicates = [ci1]`
	
	`AcquiredHemorrhagicDisorder.contraindicates`
	`= [ci3]`
	`ConstitutiveHemorrhagicDisorder.contraindicates`
	`= [ci2, ci4]`
		$HemorrhagicDisorder \sqsubseteq \exists contraindicates.\{ci1\}$
		$(\exists has\_for\_disorder.HemorrhagicDisorder)(ci1)$
		$AcquiredHemorrhagicDisorder \sqsubseteq \exists contraindicates.\{ci3\}$
		$(\exists has\_for\_disorder.AcquiredHemorrhagicDisorder)(ci3)$
		$ConstitutiveHemorrhagicDisorder \sqsubseteq \exists contraindicates.\{ci2\}$
		$(\exists has\_for\_disorder.ConstitutiveHemorrhagicDisorder)(ci2)$
		$ConstitutiveHemorrhagicDisorder \sqsubseteq \exists contraindicates.\{ci4\}$
		$(\exists has\_for\_disorder.ConstitutiveHemorrhagicDisorder)(ci4)$

(4)	`close_world(Contraindication)`	$(\forall has\_for\_drug.Ticagrelor)(ci1)$ $(\forall has\_for\_disorder.HemorrhagicDisorder)(ci1)$ $(\forall has\_for\_drug.Heparin)(ci2)$ $(\forall has\_for\_disorder.ConstitutiveHemorrhagicDisorder)(ci2)$ $(\forall has\_for\_drug.Aspirin)(ci3)$ $(\forall has\_for\_disorder.AcquiredHemorrhagicDisorder)(ci3)$ $(\forall has\_for\_drug.Aspirin)(ci4)$ $(\forall has\_for\_disorder.ConstitutiveHemorrhagicDisorder)(ci4)$ $Contraindication \sqsubseteq \{ci1, ci2, ci3, ci4\}$
(5)	`close_world(Drug)`	$Ticagrelor \sqsubseteq \forall contraindicated\_with.\{ci1\}$ $Heparin \sqsubseteq \forall contraindicated\_with.\{ci2\}$ $Aspirin \sqsubseteq \forall contraindicated\_with.\{ci3, ci4\}$

Figure 8: The ontology for the use case (top) and the Python + Owlready program (bottom) for creating the contraindications of table 5, with the corresponding assertions in DLs. The numbers on the left correspond to the steps described in the text.

Dataset	**Example**	**Vitaros®**	**Pylera®**	**Ciloxan®**	**Antarene codeine®**
Number of drugs	3	8	4	9	5
Number of disorders	3	29	17	14	26
Number of asserted contraindications	4	70	30	42	50
Number of partitions	1	9	4	1	8
Number of disjoints (involving disorders)	1	640	166	82	499
Number of inferences:					
- contraindications	3	23	9	13	7
- absences of contraindications	1	23	4	0	12

Table 6: Metrics and number of inferences for the use case example and for four datasets.

table), and all disorders that are contraindicated in some situations but not all are subclasses of neither of these two classes ("CI/Ok" in the table). The automatic classification can be performed in Python, and then the issubclass() standard function can be used to test if a class is a subclass of another, *e.g.* issubclass(HemorrhagicDisorder, DisorderOkWithAspirin) would return False.

The supplementary computer code "contraindications.txt" includes the entire source code for the example of the use case.

[Insert Supplementary Computer Code 2 here]

The example we described was simple, involving only three drugs, three disorders and four contraindications. However, in practice, drugs are much more complex. Table 6 shows the metrics for the example and for four datasets that we worked on in the VIIIP project. Each dataset corresponds to a new drug recently marketed in France (Vitaros®, Pylera®, Ciloxan® and Antarene codeine®) and also includes several comparator drugs (*i.e.* already existent drugs with the same indication). Disorders were coded using ICD10 whenever possible. The table also indicates the number of inferences obtained for each dataset.

In conclusion, this use case illustrates the interest of the high-level syntax proposed by OlwReady for local closed world reasoning and for defining role-filler constructs and existential constraints, as well as its ability to deal with classes in a simple way, including the automatic classification of individuals and classes. In Figure 8 (bottom), the Python program is clearly shorter and easier to read and understand than the corresponding assertions in DLs notation.

A more advanced program can take full advantage of the possibilities offered by Python for querying drug databases or generating graphical interfaces for data entering (before creating the contraindications) and for generating an HTML website displaying tables for comparing the contraindications (after the reasoning). In a subsequent work, we also used visual analytics for facilitating the comparison of drug properties [48].

7. Discussion

7.1. The proposed syntax

The use of the dot notation allows easy access to the content of an ontology, as if the ontology was a Python module. It is possible to use the entities in the ontology as if they were Python classes and objects. This permits reusing existing Python libraries with the individuals in the ontology instead of Python objects. In addition, thanks to the "role-fillers as class attributes" syntax, the classes of the ontology can also be used as Python objects.

Arguably, the syntax of Owlready is more concise than the one of OWLAPI (see Table 3) and allows a shorter source code (in about a 1/3 ratio). OWLAPI allows the manipulation of each axiom of the ontology individually; while this can be interesting in some specific situations, it is often not a requirement. On the contrary, the proposed ontology-oriented programming syntax is shorter and easier to use, and satisfies the needs of most applications.

Using Owlready, the simple definition of an ontology (without complex logical expressions such as existential and universal constraints) is almost as easy as the creation of a class hierarchy in a programming language. It can thus be done by any developer. Then, an ontologist can add constraints in an OWL file, *e.g.* using the Protégé editor. Finally, Owlready is able to blend the object model with the ontology in a single model.

Owlready can also be used as a "glue" for accessing easily to an ontology in Python and then for connecting it to other components (*e.g.* a website). It has been used by Master students (M1 in biomedical informatics) that had almost no background knowledge in ontology and DLs, providing them an easy and object-oriented access to the entities in the ontology.

7.2. Local closed world reasoning

In section 3.2, we reviewed the various approaches for local closed world reasoning in the literature. All of them focused on non-monotonic reasoning, which is a more general and bigger problem than local closed world reasoning. The various approaches modified or extended the syntax and the semantics of DLs, which may impair their compatibility with OWL and OWL-based tools.

In this paper, we took a different approach for local closed world reasoning. This approach is very pragmatic and simple, following the "keep it simple stupid" advice from Krishnadhi [40]. It consisted in adding universal constraints to the ontology. Our close_world() function acts like a kind of "preprocessor" for OWL. This approach is 100% compatible with OWL and OWL-based tools, including the HermiT reasoner.

The proposed close_world() function actually closes individuals, classes (their list of individuals is limited to the asserted one), and their existential constraints (including role-filler). However, it does not close the properties themselves. For example, in the use case (section 6), we constrained the has_for_drug/contraindicated_with property for the four contraindications and the three Drug subclasses; however, we did not restrict this property generally, *i.e.* we do not prevent other relations involving other individuals (although this would be inconsistent, due to the range and domain of the property, and since close_world() asserted that there was no other contraindication than the four we defined). Closing properties is known to increase the complexity of the reasoning problem, and even lead to undecidability if not limited to a fixed number of properties [49].

	Goldman 2003 [28]	Kalyanpur 2004 [26]	Koide 2005 [24]	Babik 2006 [30]	Stevenson 2011 [32]	**Owlready**
Type	static	static	dynamic	dynamic	semi-dynamic	dynamic
Language	C#	Java	CommonLisp	Python	Java	Python
Classification of individuals	no	no	**yes**	?	**yes**	**yes**
Classification of classes	no	no	?	?	no	**yes**
Methods in OWL classes	no	no	?	no	no	**yes**
Syntax for OWL class expressions	no	no	**yes**	no	?	**yes**
Syntax for "role-fillers"	no	no	no	no	no	**yes**
Local closed world reasoning	no	no	no	no	no	**yes**

Table 7: Comparison of Owlready with other ontology-oriented programming approaches.

Our proposal has two limits. First, the "closed" status of a class or an individual is not asserted in the ontology, but simply translated into lower-level constraints. Consequently, if the ontology is modified later (*e.g.* if we create a fifth contraindication), it will become inconsistent. Therefore, close_world() must be called() *after* building the ontology (typically just before reasoning).

Second, the algorithm we propose for close_world() may be too permissive when inheritance is involved. For example, let us consider the following ontology: $B \sqsubseteq A$, $A \sqsubseteq \exists R.C$, $B \sqsubseteq \exists R.D$ (where A, B, C and D are classes and R is a property). Calling close_world(B) produces the following constraint: $B \sqsubseteq \forall R.(C \sqcup D)$. However, if $D \sqsubseteq C$, this would probably not be the result expected by the user, who possibly intended $B \sqsubseteq \forall R.D$. In those circumstances, an improved version of the close_world() function might test whether it is already known that $D \sqsubseteq C$, and produce more specific constraints.

Despite these limits, the simple close_world() function has proved to be useful in several situations, such as in the presented use case.

7.3. Comparison with previous ontology-oriented programming approaches

In section 3, we reviewed the existing approaches for ontology-oriented programming in the literature. Table 7 compares these approaches with Owlready. Owlready appears to be one of the most advanced approaches. In particular, it is able to perform automatic classification not only on individuals but also on classes, it is able to perform local closed world reasoning and it proposes a high-level syntax for defining "role-filler" constraints. As explained previously in section 2, these points are crucial when working on medical ontologies, but they may also be useful in other domains.

8. Conclusion and perspectives

In this paper, we first identified two difficulties encountered when working on medical ontologies. We reviewed the existing approaches for ontology programming interface, and we showed that none of them proposed a simple solution to these difficulties. We compared object models with ontologies, we found many similarities but we also highlighted some differences. Then, we presented the Owlready module for ontology-oriented programming in Python. We described the general architecture of the module, the mapping between the object model and the ontology, the syntax (including a specific syntax for manipulating classes and role-filler constraints) and the reasoning capabilities (including a simple algorithm for performing local closed world reasoning). We provided two examples with Gene Ontology. Finally, we presented a use case: a medical application for comparing contraindications. It illustrated the use of the specific syntax for manipulating classes and role-fillers, and local closed world reasoning.

There is a growing community of Python programmers in biology, bioinformatics and the medical field. Ontologies are widely used in these domains, but there was almost no solution for accessing OWL ontologies in Python before Owlready. In the biomedical domain, Owlready has already been used by other researchers for interfacing natural language processing (NLP) and verbalization tools with ontologies [50, 51]. It has also been employed for semantic reasoning on pain severity extracted from clinical records [52]. Beyond medical informatics, Owlready has been used for building an ontology-based knowledge system for helicopter transmission design [53].

A first perspective of this work is to improve Owlready with regard to properties and sub-properties. For instance, the asserted values of a property could be automatically updated when the value of a sub-property is asserted, similarly to what we did for inverse properties. A second perspective is to include in Owlready additional non-monotonic reasoning features, such as defeasible inheritance. A third perspective is to add support for vagueness in Owlready, in order to support fuzzy-ontology. Two approaches have been proposed for dealing with vagueness in ontologies: the development of fuzzy extensions to OWL based on new DLs [54], and the use of the current OWL standard using specific procedures (*e.g.* using annotations to specify fuzziness) [55]. Both approaches could be considered for integration in Owlready. A fourth perspective is to connect Owlready to graphical user interfaces, in order to generate automatically dialog boxes for editing the individuals of an ontology, or even the classes (since, as we explained, many medical concepts must actually be modeled with classes rather than individuals).

Acknowledgements

This work was supported by the French drug agency (ANSM, Agence Nationale de Sécurité du Médicament et des produits de santé) through the VIIIP project [grant number AAP-2012-013].

References

[1] N. Guarino, D. Oberle, S. Staab, Handbook on ontologies, Springer, 2009, Ch. What Is an Ontology?, pp. 1–17.

[2] C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, M. Smith, OWL 2 Web Ontology Language, W3C recommendation (2012).

[3] O. Lassila, R. Webick, Resource Description Framework (RDF) Model and Syntax Specification, W3C recommendation (1999).

[4] Yu AC, Methods in biomedical ontology, J Biomed Inform 39 (3) (2006) 252–266.

[5] B. Smith, M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters, L. J. Goldberg, K. Eilbeck, A. Ireland, C. J. Mungall, N. Leontis, P. Rocca-Serra, A. Ruttenberg, S. A. Sansone, R. H. Scheuermann, N. Shah, P. L. Whetzel, S. Lewis, The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration, Nat Biotechnol 25 (11) (2007) 1251–1255.

[6] A. L. Rector, J. E. Rogers, P. Pole, The GALEN High Level Ontology, in: Fourteenth International Congress of the European Federation for Medical Informatics (MIE-96), Copenhagen, Denmark, 1996.

[7] G. Héja, G. Surján, P. Varga, Ontological analysis of SNOMED CT, BMC Medical Informatics and Decision Making 8 (Suppl 1) (2008) –8.

[8] The Gene Ontology Consortium, The Gene Ontology project in 2008, Nucleic Acids Res 36 (Database issue) (2008) D440–4.

[9] C. Rosse, V. Mejino JL, A reference ontology for biomedical informatics: the Foundational Model of Anatomy, J Biomed Inform 36 (2003) 478–500.

[10] L. T. Detwiler, J. L. V. Mejino, J. F. Brinkley, From frames to OWL2: Converting the Foundational Model of Anatomy, Artif Intell Med 69 (2016) 12–21.

[11] B. Motik, R. Shearer, I. Horrocks, Hypertableau reasoning for description logics, Journal of Artificial Intelligence Research 36 (2009) 165–228.

[12] A. Rector, M. Horridge, L. Iannone, N. Drummond, Use Cases for Building OWL Ontologies as Modules: Localizing, Ontology and Programming Interfaces & Extensions, in: 4th Int Workshop on Semantic Web enabled software engineering (SWESE-08), 2008.

[13] Lamy JB, Ontology-Oriented Programming for Biomedical Informatics, Stud Health Technol Inform 221 (2016) 64–68.

[14] Curé O, Improving the data quality of drug databases using conditional dependencies and ontologies, Journal of data and information quality 4 (1) (2012) 20.

[15] D. J. Sheridan, D. G. Julian, Achievements and Limitations of Evidence-Based Medicine, J Am Coll Cardiol 68 (2) (2016) 204–13.

[16] C. Patel, J. Cimino, J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, L. Ma, E. Schonberg, K. Srinivas, The Semantic Web, Vol. 816-829, Springer Berlin Heidelberg, 2007, Ch. Matching patient records to clinical trials using ontologies.

[17] A. Burgun, O. Bodenreider, F. Mougin, Classifying diseases with respect to anatomy: a study in SNOMED CT., AMIA Annu Symp Proc 91-5.

[18] S. Rudolph, M. Krötzsch, P. Hitzler, All Elephants are Bigger than All Mice, Technical Report.

[19] S. Schulz, L. Jansen, Formal ontologies in biomedical knowledge representation, Yearb Med Inform 8 (2013) 132–46.

[20] SPARQL query language for RDF, W3C recommendation (2008).

[21] I. Kollia, B. Glimm, Optimizing SPARQL query answering over OWL ontologies, Journal of Artificial Intelligence Research 48 (2013) 253–303.

[22] M. Horridge, S. Bechhofer, The OWL API: A Java API for OWL ontologies, Semantic Web 2 (2011) 11–21.

[23] M. K. Levin, L. G. Cowell, owlcpp: a C++ library for working with OWL ontologies, Journal of biomedical semantics 6 (2015) 35.

[24] S. Koide, J. Aasman, S. Haflich, OWL vs. Object Oriented Programming, in: the 4th International Semantic Web Conference (ISWC 2005), Workshop on Semantic Web Enabled Software Engineering (SWESE), 2005, pp. 1–15.

[25] H. Knublauch, D. Oberle, P. Tetlow, E. Wallace, A Semantic Web Primer for Object-Oriented Software Developers, W3C Working Group Note.

[26] A. Kalyanpur, D. Pastor, S. Battle, J. Padget, Automatic mapping of OWL ontologies into Java, in: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), 2004, pp. 98–103.

[27] Zimmermann M, OWL2Java : a Java Code Generator for OWL, website. http://www.incunabulum.de/projects/it/owl2java (2009).

[28] Goldman NM, Ontology-oriented programming: static typing for the inconsistent programmer, in: Lecture notes in computer science: the SemanticWeb, ISWC, Vol. 2870, 2003, pp. 850–865.

[29] S. Koide, T. Takeda, OWL-Full Reasoning from an Object Oriented Perspective, Lecture Notes in Computer Science, ASWC 2006 4185 (2006) 263–277.

[30] M. Babik, L. Hluchy, Deep Integration of Python with Web Ontology Language, in: Proceedings of the 2nd workshop on scripting for the semantic web, Budva, Montenegro, 2006, pp. 1–5.

[31] K. L. Clark, F. G. McCabe, Ontology oriented programming in Go, Applied Intelligence 24 (2006) 3–37.

[32] G. Stevenson, S. Dobson, Sapphire: Generating Java Runtime Artefacts from OWL Ontologies, in: Lecture Notes in Business Information Processing, Advanced Information Systems Engineering Workshops, Vol. 83, 2011, pp. 425–436.

[33] Hustadt U, Do we need the closed-world assumption in knowledge representation, in: Working notes of the KI'94 workshop: Reasoning about structured objects, knowledge representation meets databases (KRDB'94), Vol. D-94-11, 1994, pp. 24–26.

[34] S. Grimm, B. Motik, C. Preist, Matching semantic service descriptions with local closed-world reasoning, in: European Semantic Web Conference, Vol. 575-589, Springer Berlin Heidelberg, 2006.

[35] P. Doherty, W. Lukaszewicz, A. Szalas, Efficient reasoning using the local closed-world assumption, International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA), Lecture Notes in Computer Science 1904 (2000) 49–58.

[36] B. Motik, I. Horrocks, U. Sattler, Adding Integrity Constraints to OWL, in: Proceedings of the OWLED 2007 workshop on OWL: Experiences and directions, Innsbruck, Austria, 2007.

[37] S. Grimm, B. Motik, Closed world reasoning in the semantic web through epistemic operators, in: Proceedings of the OWLED 2005 workshop on OWL: Experiences and directions, Galway, Ireland, 2005.

[38] B. Motik, R. Rosati, Reconciling description logics and rules, Journal of the ACM 57 (5) (2010) 93–154.

[39] M. Knorr, J. J. Alferes, P. Hitzler, Local closed world reasoning with description logics under the well-founded semantics, Artificial Intelligence 175 (2011) 1528–1554.

[40] A. Krishnadhi, K. Sengupta, P. Hitzler, Local closed world semantics: Keep it simple, stupid!, in: Proceedings of the 2011 International Workshop on Description Logics (DL2011), Vol. 745-756, 2011.

[41] Y. Ren, J. Z. Pan, Y. Zhao, Closed world reasoning for OWL2 with NBox, Tsinghua Science & Technology 15 (6) (2010) 692–701.

[42] M. Denecker, A. Cortés-calabuig, M. Bruynooghe, O. Arieli, Towards a logical reconstruction of a theory for locally closed databases, ACM Transactions on Database Systems (TODS) 35 (3) (2010) 22.

[43] J. B. Lamy, A. Venot, C. Duclos, PyMedTermino: an open-source generic API for advanced terminology services, Stud Health Technol Inform 210 (2015) 924–928.

[44] X. Zhu, J. W. Fan, D. M. Baorto, C. Weng, J. J. Cimino, A review of auditing methods applied to the content of controlled biomedical terminologies, J Biomed Inform 42 (3) (2009) 413–425.

[45] H. Prosser, S. Almond, T. Walley, Influences on GPs'decision to prescribe new drugs-the importance of who says what, Fam Pract 20 (1) (2003) 61–68.

[46] P. McGettigan, J. Golden, J. Fryer, R. Chan, J. Feely, Prescribers prefer people: the sources of information used by doctors for prescribing suggest that the medium is more important than the message, Br J Clin Pharmacol 51 (2000) 184–189.

[47] G. K. Spurling, P. R. Mansfield, B. D. Montgomery, J. Lexchin, J. Doust, N. Othman, A. I. Vitry, Information from pharmaceutical companies and the quality, quantity, and cost of physicians' prescribing: a systematic review, PLoS medicine 7 (10) (2010) e1000352.

[48] J. B. Lamy, H. Berthelot, M. Favre, A. Ugon, C. Duclos, A. Venot, Using visual analytics for presenting comparative information on new drugs, J Biomed Inform 71 (2017) 58–69.

[49] P. A. Bonatti, C. Lutz, F. Wolter, The complexity of circumscription in description logic, Journal of Artificial Intelligence Research 35 (2009) 717–773.

[50] Keet CM, Representing and aligning similar relations: Parts and wholes in isiZulu vs. English, in: Proceedings of the first International Conference on Language, Data and Knowledge, Lecture Notes in Artificial Intelligence, Vol. 10318, Springer, Galway, Ireland, 2017, pp. 58–73.

[51] C. M. Keet, M. Xakaza, L. Khumalo, Verbalising OWL ontologies in isiZulu with Python, in: Demo at the 14th Extended Semantic Web Conference (ESWC17), 2017.

[52] C. Grasso, A. Joshi, E. Siegel, Visualization of pain severity events in clinical records using semantic structures, in: IEEE Tenth International Conference on Semantic Computing (ICSC), Vol. 321-324, 2016.

[53] J. C. Zhao, L. P. Huang, L. Tian, K. Q. Wu, Knowledge system for helicopter transmission design based on ontology, in: Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences, Vol. 321-325, Wuhan, China, 2017.

[54] S. Calegari, D. Ciucci, Fuzzy ontology, fuzzy description logics and fuzzy-OWL, International Workshop on Fuzzy Logic and Applications (WILF), Lecture Notes in Computer Science 4578 (2007) 118–126.

[55] F. Bobillo, U. Straccia, Fuzzy ontology representation using OWL 2, International journal of approximate reasoning 52 (7) (2011) 1073–1094.