ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/225637434

OntoPath: A Language for Retrieving Ontology Fragments

Conference Paper · November 2007

DOI: 10.1007/978-3-540-76848-7_60

CITATIONS
18

READS
450

4 authors:

Ernesto Jiménez-Ruiz
City, University of London
188 PUBLICATIONS **3,979** CITATIONS

SEE PROFILE

Rafael Berlanga
Universitat Jaume I
83 PUBLICATIONS **1,054** CITATIONS

SEE PROFILE

Victoria Nebot
Universitat Jaume I
39 PUBLICATIONS **721** CITATIONS

SEE PROFILE

Ismael Sanz
Universitat Jaume I
84 PUBLICATIONS **377** CITATIONS

SEE PROFILE

All content following this page was uploaded by Ernesto Jiménez-Ruiz on 22 May 2014.

The user has requested enhancement of the downloaded file.

OntoPath: A Language for Retrieving Ontology Fragments

E. Jiménez-Ruiz[1], R. Berlanga[1], V. Nebot[1], and I. Sanz[2]

[1] Departamento de Lenguajes y Sistemas Informáticos
[2] Departamento de Ingeniería y Ciencia de los Computadores
Universitat Jaume I (Castellón)
{ejimenez, victoria.nebot, berlanga, isanz}@uji.es

Abstract. In this work we introduce a novel retrieval language, named *OntoPath*, for specifying and retrieving relevant ontology fragments. This language is intended to extract customized self-standing ontologies from very large, general-purpose ones. Through *OntoPath,* users can specify the desired detail level in the concept taxonomies as well as the properties between concepts that are required by the target applications. The syntax and aims of OntoPath resemble XPath's in that they are simple enough to be handled by non-expert users and they are designed to be included in other XML-based applications (e.g. transformations sheets, semantic annotation of web services, etc.). *OntoPath* has been implemented on the top of the graph-based database *G*, for which a Protégé OWL plug-in has been designed to access and retrieve ontology fragments.

Keywords: Query Languages, Ontologies, Semantic Web, Ontology Fragments.

1 Introduction

The Semantic Web is intended to extend the current web by creating knowledge objects that allow both users and programs to better exploit the huge amount of resources. The cornerstone of the Semantic Web is the definition of widely agreed ontologies conceptualizing the knowledge behind web resources. Nowadays, several efforts are focused on building very large ontologies that are continuously growing as new knowledge is added to them by the respective communities. This happens mainly in the biomedical domain (e.g. GO[1], GALEN[2], FMA[3], NCI-Thesurus[4], Tambis[5], BioPax[6], etc).

[1] Gene Ontology: http://www.geneontology.org/
[2] Galen Ontology: http://www.opengalen.org/
[3] Foundational Model of Anatomy: http:// fma.biostr.washington.edu/
[4] NCI taxonomy/ontology: http://nciterms.nci.nih.gov/NCIBrowser
[5] Tambis Ontology: http://www.cs.man.ac.uk/~stevensr/tambis-oil.html
[6] BioPax Ontology: http://www.biopax.org/

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 897–914, 2007.
© Springer-Verlag Berlin Heidelberg 2007

However, the very large size of these ontologies as well their variety makes it difficult to deploy them in particular applications. The first problem is scalability. Most of these ontologies are expressed in OWL-DL, with different degrees of expressivity. The classification of new concepts, queries and assertions require the use of reasoners (e.g. Pellet[7], Racer[8], etc.), but they are not able to handle even medium-size ontologies [1, 2]. A second problem is the application of these ontologies to concrete applications. Such an application usually does not require comprehensive descriptions of the domains but rather a handful subset of concepts and properties from them (i.e. a local view of the domain [3]). Another important issue is their visualization in ontology editors, in which large ontologies have difficulties to be loaded and properly displayed.

This paper presents a new query language, *OntoPath*, for retrieving consistent fragments from domain ontologies. It is based on the XPath [24] syntax but it is applied over ontologies described in OWL. In contrast to other Semantic Web query languages such as SparQL [4] and RQL [5], this language is intended to extract the necessary knowledge to build new ontologies according to user and application requests. As a consequence, query results are also OWL files.

OntoPath is being applied within the Health-e-Child project [25] as a mechanism to explore large biomedical ontologies and also to perform the vertical integration of the biomedical knowledge via the definition of ontology fragments and connections between them (i.e. semantic bridges) [6]. Another direct application of *OntoPath* is the extraction of analysis dimensions for building OLAP-based multidimensional models for biomedical research [7]. Finally, it is also being applied to the compositions of Semantic Web Services. In this case, semantic services are represented as concepts and their input/output parameters as their properties. Service chains can be then obtained through Ontopath queries [8].

The rest of the paper is organized as follows. Section 2 gives a brief review of related works. An overview of the proposal is presented in Section 3. Section 4 describes how ontologies are parsed and stored into the semi-structured database G. OntoPath syntax and semantics are presented in Section 5. Section 6 is dedicated to OntoPath query processing. Experiments and examples are presented in Section 7. Finally, Section 8 gives some conclusions and future work.

2 Related Work: Ontology Modularization and Querying

The necessity of working with ontology modules is well known in the literature and several approaches can be found to modularize and query ontologies. In [1] a good review of ontology modularization formalisms is presented, among them: Distributed Description Logic [9], Modular Reuse of Ontologies [10], Package Based DL [11] and Network Partitioning [12]. Although these works propose good formalisms to automatically define ontology modules and their connections, they do not provide mechanisms to guide the modularization, that is, users do not participate in the partitioning of the ontology.

[7] Pellet reasoner: http://pellet.owldl.com/
[8] RacerPro reasoner: http://www.racer-systems.com/

Alternatively to these approaches, traversal-based ones are aimed at extracting ontology fragments according to user preferences, usually specified as a set of concepts and properties of interest. PROMPT [13], MOVE [14] and OntoPathView [18] are examples of traversal-based methods. All of them deal with pure frame-based ontologies. This is a serious limitation since most ontologies are being expressed in the standard language OWL, which assumes as underlying model the Description Logic (DL).

Recently, Seidenberg and Rector [2] have proposed a segmentation algorithm to extract fragments (i.e. segments) from large ontologies in OWL, more specifically the GALEN ontology. This algorithm resembles traversal-based approaches in that it starts from a concept of interest and then navigates through the concept hierarchy to identify not only related properties and concepts but also any element necessary to describe and classify them. Thus, their main aim is to keep as much as possible of the inference capabilities of the original ontology in the resulting segments. However, resulting segments use to be very large, some of them still intractable by current editors and reasoners. This is because affected axioms can be scattered across the whole concept hierarchy.

In this paper, we present a different approach for extracting fragments from OWL ontologies. Our method differs from the previous one in that we do not preserve all the axioms that are necessary to classify the extracted concepts. Instead, our approach makes explicit much of this knowledge in the resulting fragments. In other words, instead of generating new OWL with implicit knowledge to be inferred, we generate specific and explicit fragments that will not require reasoning capabilities. As a consequence, much smaller and specific fragments can be obtained. This is especially useful when generating OWL fragments that will be used in specific final applications (e.g. a data warehouses).

It is worth mentioning that there exist other query languages for the Semantic Web, such as RQL [5], SparQL [4], KAON views [15] and RVL [23]. However, these languages deal with RDF/S ontologies, which are less expressive than OWL ones. Moreover, their aim is not to build new closed and consistent ontologies as ours, but providing relational-like data satisfying user requests (e.g. all elements with a specific date and author).

3 Ontology Management System Architecture

The development of OntoPath has been carried out in the context of a new database management system aimed at storing, organising and retrieving customized pieces of knowledge to be used in specific applications [16]. This system relies on the semi-structured database G[9] and the language OWL. The former was chosen because of its great capability for storing and retrieving large graph-like data. Figure 1 summarizes the system architecture, whose main modules are described in turn.

[9] A. Rios "G Platform (Helide S.A)": http://www.helide.com, http://www.maat-g.com

- *OWL parser*[10] *and builder*. The former consists of an ad-hoc SAX-based parser, which builds the necessary objects from an input ontology to make it persistent in the database. The latter allows OWL files to be generated from database objects.

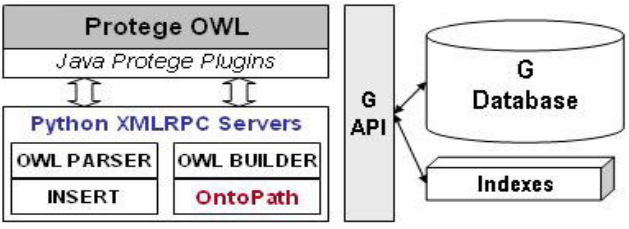

Fig. 1. Architecture of the Ontology Management System

- *G database*. It has been used as a backend to store, index and retrieve the OWL ontologies as graphs. To store ontologies four object types are defined, namely: *ontology*, *property*, *concept*, and *enumeration*; the latter for nominal lists. Figure 2 summarizes the four object types and the existing references between them. All OWL constructors have been regarded so that ontologies can be loaded and retrieved without semantic lost.

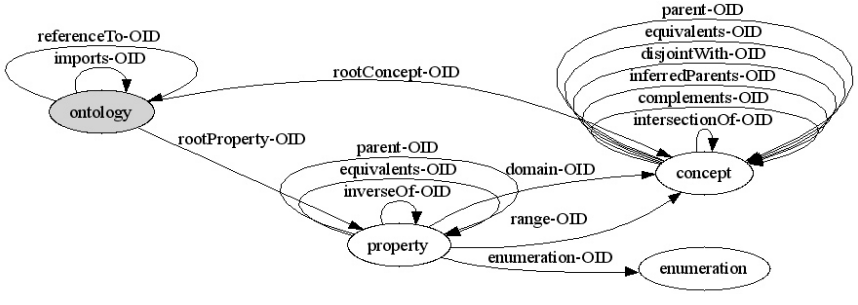

Fig. 2. Meta-schema for storing OWL Ontologies into G

- *OntoPath.* It is the proposed query language for retrieving ontology fragments (personalized modules or views) from the domain ontologies. By using OntoPath, users can specify the desired detail level in the concept taxonomies as well as the properties that are required by the target applications. OntoPath is the main focus of this work.
- *Protégé-OWL.* It is the front-end of the system, which allow users to visualize and edit the stored ontologies (fragment or whole ones). A specific plug-in has been recently designed to create and browse ontology modules and fragments [19].

[10] Python OWL Parser: http://krono.act.uji.es/people/Ernesto/Owl_Parser/

4 Ontology Parsing and Storage

From now on, we assume that ontologies are expressed in OWL-DL 1.0 (which uses a subset of the DL $\mathcal{SHOIN}$(D) [20]), and therefore we use standard Description Logic (DL) syntax when discussing semantic issues in the paper.

As previously mentioned, ontologies are parsed and stored in the G database as graphs (see Figure 2), where nodes represent ontology entities (e.g. concepts, properties and nominals) and edges between nodes represent their different relationships (e.g. *subClassOf*, *domain*, *range*, *intersectionOf*, etc.) From now on, we represent G database records as follows:

$$R= Type(att_1=v_1,\ldots,att_n=v_n)$$

Where *Type* is the type of the record (e.g. *ontology*, *concept*, etc.) , and $att_i=v_i$ states that the record attribute att_i takes the value v_i. Multi-valued attributes are also allowed, taking the form $att_i=v_1,..,v_k$. In order to support the heterogeneity of the database records, attributes lists associated to a record type can be of arbitrary length (i.e. optional attributes are allowed) and formats (i.e. an attribute can take values from different data types at each record). Additionally, values can be references to other database records. For this purpose, record unique identifiers are used. As an example, the following records represent a simple ontology with two concepts and one property:

O=*ontology*(*name*='Simple.owl', rootConcept=C1, rootProperty=P1)
*C*1=*concept*(*name*='Thing')
*P*1= *property* (*name*='PropertyThing')
*C*2=*concept*(*name*='Person', *subClassOf*=*C*1)
P2= *property*(*name*='hasFriend', *range*=*C*2, *domain*=*C*2, *subPropertyOf*=*P*1)

P1, P2, *C*1, *C*2 and *O* are the unique identifiers of the corresponding data records. To denote an attribute *att* of a record *R*, we use *R.att*, for example C2.*subClassOf*.

As explained in next sections, *OntoPath* query processing relies on traversing both the concept hierarchy and the association graph between concepts. The former consists of the links present at *subClassOf* record attributes, whereas the latter consists of the property records linking concepts through its domains and ranges.

As OWL properties can appear in many different contexts (i.e. class restrictions), we introduce another record type named inferred property (*i-property*). An *i-property* record has a similar structure to original *property* records, but its interpretation must be restricted to the context it happens. For example, considering the class definition $C \sqsubseteq A \sqcap \exists R.B$, the record *i-property*(*name*=*R*, *domain*=*C*, *range*=*B*) means that in the context of *C*, some instances of *A* are related through *R* to some instances of *B*. For this reason, *i-property* records will have attached the DL expression from which they where extracted.

Ontology instances (i.e. individuals in DL jargon) are also stored as database records, preserving their references through database object references. For example, the following instances belong to the previous ontology:

I1=instance(name='John', hasFriend=I2, type=C2)
I2= instance(name='Peter', hasFriend=I1, type=C2)

Following subsections describe how complex DL expressions resulted from OWL constructors can be approximated to enrich as much as possible database records, and therefore to make explicit the necessary knowledge for solving *OntoPath* queries.

4.1 Approximating DL Expressions

As mentioned in the introduction, large ontologies cannot be classified by current DL reasoners due to their inherent exponential complexity. This means that concepts that are not explicitly related to others cannot be properly classified, and will be unconnected to the rest of the ontology. However, there are some simple and frequent patterns appearing in OWL definitions that can be treated without requiring tableau-like reasoning. The application of these patterns establishes further relations between concepts and properties in the database, and they will provide a more complete vision of the ontology as a graph.

The current list of patterns is as follows:

1. **Inferred parents:** if a concept is defined as an intersection (both equivalent and a subset) of a set of concepts, we can infer that it is a subclass of them.

   $C \equiv C_1 \sqcap \ldots \sqcap C_n \Rightarrow C.subClassOf=C_1,\ldots,C_n$

2. **Inferred children:** if a class is defined as the *equivalent* of the union of a set of classes, these classes are its subclasses.

   $C \equiv C_1 \sqcup \ldots \sqcup C_n \Rightarrow C_1.subClassOf.append(C)\ ,\ \ldots,\ C_n.subClassOf.append(C)$

3. **Inferred domains:** If a restriction of the form ∃R.D or ∀R.D is found as a class definition, the corresponding property record is build:

   $C \sqsubseteq \exists R.D \Rightarrow i\text{-}property(name=R, domain=C, range=D)$.

   Both *C* and *D* must be named classes. Notice that this pattern is also applied to cardinalities, but the i-property is created without a defined range (qualified cardinalities are not considered).

4. **Creation of new classes:** If an axiom of the form $C \sqsubseteq \exists R.(D \sqcap \exists S.E)$ is found, being *C* and *D* named classes, then a new named class *D'* ($D' \equiv D \sqcap \exists S.E$) is created, with *D'.subClassOf=D*. Notice that by applying pattern 3, the following property record is also created: *i-property*(*name=R*, *domain=C*, *range=D'*). Finally, the pattern 3 is recursively applied to the sub-expression ($\exists S.E$). New class names are generated with the elements of the expression: *D'.name=D*_with_*S_E*.

5. **Nominals:** the occurrence of nominals is treated as a special case of pattern 3 if they appear inside a restriction: *allValuesFrom*, *someValuesFrom* and *hasValue* cases. The correspondent *i-property* record is created (inferred domain), an also an *enumeration* record for the set of nominals. Notice that for *hasValue* cases (∋) the enumeration record will have only one instance value associated.

   $C \sqsubseteq \exists R.\{i_1, i_2\ldots, i_n\} \Rightarrow i\text{-}property(name=R, domain=C, enumeration=E)$.

   $C \sqsubseteq \ni R.\{ i_1\} \Rightarrow i\text{-}property(name=R, domain=C, enumeration=E)$.

   $E=enumeration(name=R\text{-}nominals, list\text{-}of\text{-}values=i_{1,\ldots}, i_n)$

These patterns are applied when parsing the OWL file. For each class definition, a DL expression is built from the corresponding OWL constructors, and it is normalized (conjunctive form) before applying the patterns. These DL expressions are also stored in the database records of their classes.

In order to better understand the proposed patterns, let's consider the following class definitions examples in DL. The first one has been extracted from the Pizza[11] ontology and the second one has been taken from a portion of the GALEN ontology expressed in OWL[12]:

Pizza Example

$SpicyPizza \equiv Pizza \sqcap \exists hasTopping.(PizzaTopping \sqcap \exists hasSpiciness.Hot)$

By applying the previous patterns, the following records are generated; notice that a new class has been created (*PizzaTopping_with_ hasSpiciness_Hot*) and new explicit relationships are stored (i-properties *i-hS* and *i-hT*):

```
P = concept(name=Pizza, ....)
SP = concept(name=SpicyPizza, subClassOf=P)
PT = concept(name=PizzaTopping, ...)
H = concept(name=Hot,...)
PTH = concept(name= PizzaTopping_with_ hasSpiciness_Hot, subClassOf=PT)
hT = property(name=hasTopping, ...)
hS = property(name=hasSpiciness, ...)
i-hT = i-property(name=hasTopping, subPropertyOf=hT, domain=SP, range=PTH)
i-hS = i-property(name=hasSpiciness, subPropertyOf=hS, domain=PTH, range=H)
```

Figure 3 shows the representation of the previous registers with all the DL constructors represented explicitly in the graph by means of relationships between concepts

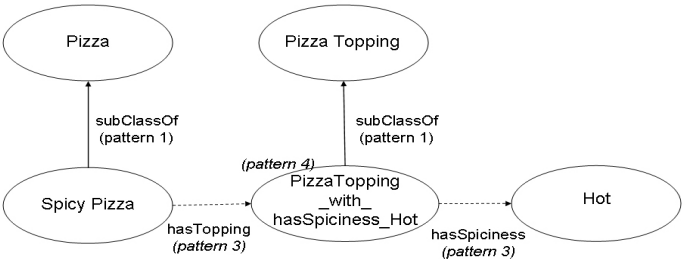

Fig. 3. Graph representation for Pizza example

Let us emphasize that the creation of class *PizzaTopping_with_ hasSpiciness_Hot* (by means of pattern 4) is also proposed in the Pizza-OWL Tutorial[13] in which they create manually the class *HotPizzaTopping*.

[11] Pizza OWL: http://www.co-ode.org/ontologies/pizza/
[12] Galen ontology in OWL: http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/galen.owl
[13] Pizza-OWL Tutorial: http://www.co-ode.org/resources/tutorials/protege-owl-tutorial.php

Galen Example

$AcuteIschaemicCardiacPathology \equiv CardiacPathology \sqcap$

$\exists isConsequenceOf.(Ischaemia \sqcap \exists hasChronicity.(Chronicity \sqcap \exists hasState.Acute))$

For the previous DL expression two classes are created by applying pattern 4 recursively (*Ischaemia_with_hasChronicity_Chronicity_with_hasState_Acute* and *Chronicity_with_hasState_Aute*). Next we present the created records for the DL expression:

CP = concept(name= CardiacPathology, ...)
AICP = concept(name= AcuteIschaemicCardiacPathology, subClassOf=CP…)
I = concept(name= Ischaemia, …)
C = concept(name= Chronicity,…)
A = concept(name= Acute,…)
ICA=.concept(subClassOf=I, name= *Ischaemia_with_hasChronicity_Chronicity_with_hasState_Acute*, …)
CA = concept(name=*Chronicity_with_hasState_Aute*, subClassOf=C)
iCO = property(name=*isConsequenceOf*, …)
hC = property(name=*hasChronicity*, …)
hS = property(name=*hasState*, …)
i-iCO = i-property(name=*isConsequenceOf*, subPropertyOf=iCO, domain=AICP, range=ICA, …)
i-hC = i-property(name=*hasChronicity*, subPropertyOf=hC, domain= ICA, range=CA …)
i-hS = i-property(name=*hasState*, subPropertyOf=hS, domain=CA, range=A,…)

Finally, it is worth mentioning that for the Galen portion (3002 classes and 413 properties) are generated more than two hundred classes following pattern 4 and near two thousand inferred properties following patterns 3 and 5.

4.2 Expressivity of Stored and Retrieved Ontologies

During the storage of an ontology, a DL reasoner can be applied to classify all its concepts and instances according to the logic subsumption relationship. However, if such a reasoner is not available or it cannot be applied to the ontology due to its size, relations between concepts will be only approximate as some subsumption relationships cannot be detected. This is especially critical when union, negations or complex axioms (i.e. $C \sqsubseteq \exists R.D \sqcup \exists S.E \sqcup (F \sqcap \neg G)$) are included in the concept definitions, as these constructors can only be treated under very strict conditions. In these cases, a set of anonymous classes are created to maintain the graph structure, and the original DL expressions are kept in the concept records in order to be reconstructed when required.

Next table summarizes the treatment given in the storage for each class or role constructor. Notice that, OWL DL ontologies using constructors that are not treated will produce approximate answers (i.e. possibly incomplete) to OntoPath queries. Fortunately, many domain ontologies seldom use negation and union in the concept definitions.

Table 1. Treated DL constructors

Constructors	Treatment
$\mathcal{AL}$ - ∀R.C - Universal Restrictions	Pattern 3
$\mathcal{AL}$ - Concept Intersection	Pattern 1
$\mathcal{E}$ - ∃R.C - Existential Qualifications	Pattern 3
$\mathcal{C}$ - Complex Concept Negation	Not treated.
$\mathcal{U}$ - Concept union	Pattern 2, only the case of concept equivalence.
$\mathcal{H}$ - Role Hierarchy	Stored role attribute *subPropertyOf*
$\mathcal{R}$ - Complex Role Inclusion Axioms	Not considered for OWL 1.0
$\mathcal{I}$ - Inverse Properties	Stored role attribute *inverseOf*
$\mathcal{O}$ - Nominals	Pattern 5
$\mathcal{N}$ - Cardinality Restrictions	Pattern 4
$\mathcal{Q}$ - Qualified Cardinality Restrictions	Not considered for OWL 1.0
(D) - Data Type Properties	Stored role attribute *PropertyType*

5 OntoPath Queries

An OntoPath query is a tree relating concepts and properties of the ontology. As in XPath, trees are expressed through paths with predicates that impose conditions through the traversed nodes. However, unlike XPath (and SparQL), the answer to an OntoPath query is not a set of pointers to the nodes in the graph satisfying the specified conditions, but a consistent and closed fragment of the ontology (i.e. a sub-graph) containing the required concepts and properties. Consistent means that all concepts in the fragment are satisfiable, whereas closed means that the fragment includes all the necessary concepts and properties to make it consistent. Notice that closeness and consistency may involve changes over the original knowledge, mainly over frontier concepts of the ontology fragment, in which a partial definition of them has been extracted. Nevertheless, the main objective is to define a party's view of a domain [3].

5.1 Query Expressions

An OntoPath query contains two types of tree nodes: concept nodes and property nodes. A property node have only one child concept node, whereas a concept node can have more than one child property node. Tree nodes also contain predicates that must be satisfied in the retrieved fragment. Figure 4 shows the interpretation of a simple OntoPath query like Disease/related-to/Gene, where an ontology fragment is extracted with diseases (if exist) directly related by means of the related-to property to the Gene concept.

The simplest query in OntoPath consists of specifying a concept *C*. The result consists of an ontology fragment containing all their sub-concepts (i.e. the concept taxonomy under *C*) an all its instances. No object properties are retrieved nor relations between instances involved in the fragment. Data type properties associated to these concepts (direct and inferred ones) are always included in the fragment.

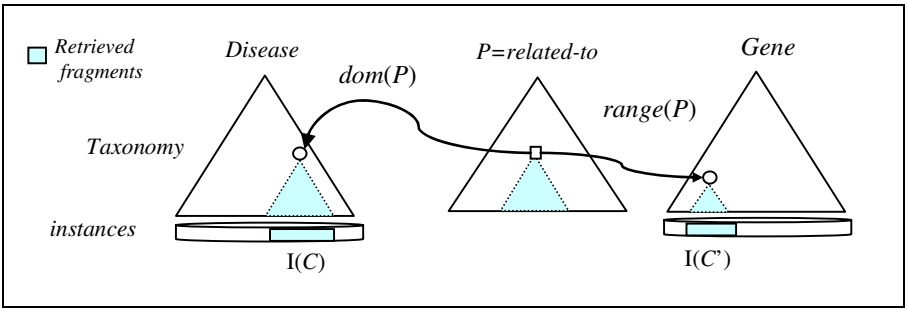

Fig. 4. Graphical interpretation of the *OntoPath* query Disease/related-to/Gene

Let *sub*(C) denote the set of all the sub-concepts of C, and *sub*(*P*) be the set of all the sub-properties of *P* and their respective *i-property* records. To state aggregation conditions over concepts, we use the syntax: *C*/*P*/*C*', where *C* and *C*' represent concepts, and *P* denotes a property. A fragment satisfying this query contains those sub-concepts of *C* and *C*', denoted R(*C*) and R(*C*') respectively, as well as those sub-properties of *P*, denoted R(*P*), such that:

- $R(C) \subseteq sub(C)$, $R(C') \subseteq sub(C')$ and $R(P) \subseteq sub(P)$
- $\forall p \in R(P)$, $dom(P) \in R(C)$ and $range(P) \in R(C')$
- $\forall c \in R(C)$, $\exists p \in R(P)$ such that $c = dom(p)$
- $\forall c \in R(C')$, $\exists p \in R(P)$ such that $c = range(p)$

We denote with I(*C*) to the direct instances (individuals) of *C*. Thus, the query *C*/*P*/*C*' also retrieves all the direct instances of all concepts in *sub*(*C*) and *sub*(*C*'). Notice that we can extend the previous query definition to paths of any length: $C_1/P_1/C_2/P_2/C_3/P_3/\ldots$

Query expressions can contain the symbol "*" to denote the concept $\top$(*Thing*), and the symbol "?" to denote any property (i.e. *sub*(?) denotes all the properties in the ontology).

Some examples of OntoPath queries are the following ones:

- RheumatoidArthritis: The taxonomy and instances below RA diseases are retrieved.
- Disease/?/RheumatoidFactor: Returns a fragment with diseases (if exist) directly related by means of any property to Rheumatoid Factor.
- AutoinmuneDisease/?/*/?/GenePTPN22: Returns a fragment with autoimmune diseases related to gene PTPN22 along with all the concepts and properties participating in their relationships.
- RheumatoidArthritis/hasTreatment/ *: Returns a fragment with RA diseases and their treatments.

Notice that the query *C*/*P*/*C*' is equivalent to the concept $C \sqcap \exists P.C'$. Similarly, the query *C*/*P*/* is equivalent to $C \sqcap \exists P.\top$ and */*P*/*C* to $\exists P.C$. One can think that fragments for them could be obtained by including their equivalent concepts in the ontology and applying some reasoner to them. This has several limitations. Firstly, for each query we

need to modify the original ontology and consequently to check the new ontology from scratch. Second, as previously mentioned reasoning is not possible for large ontologies. Finally, extracting the ontology fragment involved by a query is a difficult task from the completion graphs generated by tableau algorithms. For these reasons, we propose an ad-hoc and fast algorithm for processing OntoPath queries (see Section 6).

5.2 Predicates

As in XPath, we use square brackets [ ] to specify a predicate that must be satisfied by the involved concept or property of the query expression. If multiple conditions are required we use one []-expression for each one. They are always interpreted as a conjuction of predicates. On the contrary to XPath, we will use only []-expressions instead of the logic equivalent ones.

Twig Queries: Predicates allow us to express twig queries, that is, tree-like aggregate conditions over the concepts and properties of the query. These have the form:

$$C[P_1/C_1][P_2/C_2] \ldots$$

This is equivalent to the DL query $C \sqcap \exists P1.C1 \sqcap \exists P2.C2$.

When a twig query is required, the condition for the ontology fragment associated to each branching node (e.g. *C*) is as follows:

$$\forall c \in \mathrm{R}(C),\ \forall P_i \in children(C),\ \exists p \in \mathrm{R}(P_i),\ \text{such that } c=dom(p)$$

Future extensions of *OntoPath* will include negation and disjunction in the predicates. However, such an extension requires the inclusion of reasoning mechanisms for classifying the appropriate property domains and ranges participating in the results.

Querying Metadata: These predicates are useful for selecting parts of the ontology that were created by different authors or that contain useful annotations for retrieving relevant concepts to certain applications (e.g. lexicon, keywords, etc.) They have the following syntax:

Node[@*annotation*]
Node[@*annotation* <op> *value*]

Here *Node* is either a concept or a property of the ontology. The first expression means that the specified annotation for the OWL element exists, whereas the second one indicates that the assigned value for the specified annotation must satisfy the expressed condition. For this purpose, <op> represents any binary operator over atomic data types (e.g. <, >, ==, !=, *like*, etc.) Instances including annotated elements will be included in the resulting fragment. For example, the query Disease[@source=NCI] retrieves all the disease concepts stemming from NCI thesaurus, along with the instances created for them.

Notice that reasoning over annotations is undecidable (belongs to OWL-Full category), and therefore we cannot use a reasoner to build fragments under metadata conditions.

Data filters: With this predicates, only concepts having the specified features are selected. Moreover, only instances satisfying the data filter will be included in the result. The syntax is as follows:

Concept[*Datatype_Property* <op> *value*]

Here <*op*> also represents any binary operator over atomic data types. In DL these conditions are expressed via concrete domains. However, these are not fully supported by current versions of OWL. In the future, it would be interesting to have a classification of concepts involving data type properties (for example concepts involving age intervals for disease onsets) and then filter the ontology concepts according to them (for example, selecting only concepts that involve disease onsets for infants). However, now this kind of filters can be only applied to instances that explicitly set the specified data type property.

6 Query Processing

The query processor of OntoPath has been implemented as an *interpreted grammar*. The grammar output consists of a tree where each node contains the necessary actions to retrieve the required objects from the database and to build the result set with them. All these actions require some basic query over the G database (API), see Figure 5. Due to size limitations, a full description of the underlying algebra cannot be included in the paper.

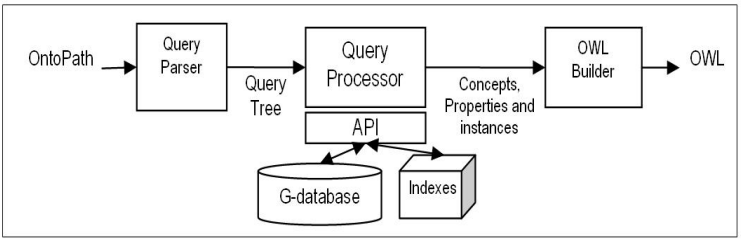

Fig. 5. Query Processing for OntoPath

Query Trees
In the rest of the section, concept nodes are denoted with *C*, *C*', etc., property nodes with *P*, *P*', etc. *parent*(X) denotes the parent node of *X* in the query tree, whereas *children*(*C*) and *child*(*P*) denote the children nodes of *C* and the unique child node of *P* respectively. The concepts or properties denoted by the query node *X* are accessed with *name*(*X*).

Our approach for query processing relies on the following principles:

- For each query concept node *C* we must keep updated the concepts in *sub*(*name*(*C*)) that covers all the sub-concepts required by the parent and children query nodes. We call it R(*C*).

- For each query property node P we must keep updated those properties in $sub(name(P))$ that satisfies the conditions imposed by the parent and child query nodes. We call it $R(P)$.
- We call empty concept, denoted as $\perp$, to any unsatisfiable concept. Thus, if $R(C)= \perp$means that the query has no solution. Similarly, if some property node has $R(P)=\varnothing$ then the query has no solution too.
- Two basic functions are necessary to calculate $R(C)$ and $R(P)$, namely:
  - *Nearest Common Descendant*: $NCD(u, v)$ returns the nearest concept that is descendant of both u and v. It follows that $NCD(u,*) = u$, $NCD(*, v)=v$, $NCD(\perp, v)= \perp$ and $NCD(u, \perp)=\perp$. If nodes u and v have not a common descendant (i.e. $sub(u)$ and $sub(v)$ are disjoint) then $NCD(u, v)= \perp$. This function can be defined over a set of concepts, denoted $NCD(S)$, so that it returns the nearest common descendant of all the nodes in S.
  - *Nearest Common Ancestor*: $NCA(u, v)$ returns the nearest common ancestor of both u and v. It follows that $NCA(u,*)=*$, $NCA(*,v)=*$, $NCA(\perp, v)= \perp$ and $NCA(u, \perp)=\perp$. This function can be defined over a set of concepts, $NCA(S)$, so that it returns the nearest common ancestor of all the nodes in S.

The initial values of R sets are calculated as follows:

- $R(P)=sub(name(P))$ if P≠?. Otherwise, it is initialized taking into account the values of its parent and child nodes as follows:
  $R(P)=\{p \mid p \in sub(?) \wedge NCD(dom(p), R(parent(P)) \neq \perp \wedge$
  $NCD(range(p), R(child(P)) \neq \perp\}$
- For concept nodes C, $R(C)=\{name(C)\}$, that is, they just contain the name associated to the concept specified in the query node.

R sets are updated as follows:

$$R(C)^{new} = NCD(\{NCD(dom(p),R(C)^{old})\}_{p \in name(P)})\}_{P \in children(C)}) \cup NCD(range(parent(C)), R(C)^{old}) \quad (1)$$

$$R(P)^{new} = \{ p \mid p \in R(P)^{old}, NCD(dom(p),R(parent(P)) \neq \perp \wedge NCD(range(p),R(child(P)) \neq \perp\} \quad (2)$$

Notice that whenever a node C changes its $R(C)$, its parent and children must be revised. Similarly, whenever a node P changes its $R(P)$, its parent and child must be revised.

Graph Indexes

The efficient evaluation of queries requires the use of index structures. The navigation of trees and DAGs (finding ancestors, descendants and siblings) has been extensively studied in the literature; following Christophides et al.'s analysis in [21], in our implementation we have adopted a variation of Agrawal, Borgida and Jagadish's interval-based encoding technique [22].

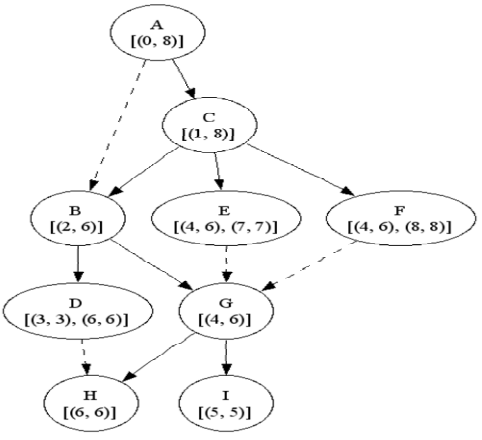

Fig. 6. Example of interval of type postorder-index assigned to a concept hierarchy

Agrawal et al's technique consists on numbering each node in the graph with integers based on a preorder and postorder traversal of the spanning tree, and then computing a set of intervals which compactly represent all the descendants of each node. This encoding is especially suited for the computation of subsumption and NCD, which are reduced to simple interval arithmetic operations. In contrast, the computation of NCA is linear in the worst case. In our tests this has not been an issue, since NCA is the least frequently used primitive; in any case, there are well-known approaches to compute NCA in constant time at little extra spatial cost, should it be deemed necessary.

6.1 Query Processing Algorithm

The query processing algorithm is as follows:

1. Initializing R sets for all query tree nodes.
2. First calculation of R sets for all query nodes. Put in a queue Q all the children and parents of the updated nodes. Finish the query processing if some concept node contains $\bot$or some property node is empty.
3. Until Q is not empty
   Pop a node from Q and revise it. That is, apply expressions (1) and (2) to update its R sets. Stop query processing if it contains $\bot$or is empty. If successfully updated then put in Q its parent and children nodes.
4. For each concept node C, for each concept $c \in \mathrm{R}(C)$, construct an OWL fragment taking into account the selected properties and groups generated in the previous steps. DL expressions will be expanded only if all its contained concepts and properties are included in the fragment.
5. For each property node P, construct an OWL fragment taking into account the inferred domain and ranges as well as the selected property taxonomy.
6. Retrieve all the instances of all the selected concepts such that they satisfy the filter conditions stated in the corresponding concept node.

If predicates over metadata are included in the query tree, we must first proceed as follows. For each concept node *C* we must select all concept in *sub*(*C*) satisfying the predicate. We must update *sub*(*C*) and re-construct the concept taxonomy under *C* for the selected concepts. Notice that NCA and NCD functions must be now applied to the re-constructed hierarchy.

The complexity of the query processing algorithm is linear with respect to the number of property records included in the query tree. This is because each iteration in step 3 can drop at least one property record from R(*P*), and the algorithm stops when either no changes are produced or some set R(*P*) becomes empty.

7 Experiments and Results

In order to check the results obtained with *OntoPath* we have designed a set of queries over well-known ontologies in the biomedical domain, namely: GALEN (a fragment in OWL format) and NCI (version 03.09d[14]). The former is more expressive and smaller than the latter.

Tables 2 and 3 summarize the results of the retrieved fragments for several queries. These queries go from very simple taxonomy retrieval to complex twig queries with different detail levels. As it can be seen, the size of the generated fragments is relatively small (see last column), although it clearly depends on the generality of the fetched query. In the NCI ontology-thesaurus, class definitions do not have complex axioms. For example, restrictions included in classes just specify the concrete range involved in certain property (*i-properties*). In this way, queries with solution are reduced to paths of three elements. In the GALEN ontology, complex axioms are included in class definitions, with numerous anonymous classes and properties. Thus, richer queries can be realized, obtaining very specific and useful fragments. For example, the first query in the GALEN ontology allow users to know which counting

Table 2. Query examples and fragment features for the NCI ontology

nciThesaurus.owl					
OntoPath Query	**Size (Kb)**	**Classes**	**Properties**	**i-properties**	**Red.**
Whole Ontology	32850	27652	109	13961	100%
Anatomy Kina	413	2204	0	0	1.2%
*/?/Cell	4581	16969	16	444	13.3%
FindingsAndDisordersKind/?/*	3403	10672	6	2556	9.9%
OrganismKind /?/FindingsAndDisordersKind	578	2217	1	0	1.7%
GeneKind/?/FindingsAndDisorders Kina	2977	9345	2	844	8.6%
/rDiseaseHasAssociatedAnatomy/	1509	4826	1	630	4.4%
Cell/?/*	447	2204	2	274	1.3%
Tissue/?/*	425	2204	2	92	1.2%
*/?/Tissue	4563	16969	16	290	13.2%
Protein Kind/?/*	5333	14851	13	11006	15.5%
*/?/Protein Kina	1458	5420	8	1712	4.2%
GeneKind/?/*	4593	13143	6	8392	13.3%
OrganismKind/?/*	3225	10331	3	0	9.4%
*/?/OrganismKind	1893	7544	5	2152	5.5%

[14] EVS-NCI (current version: 07.04e): ftp://ftp1.nci.nih.gov/pub/cacore/EVS/NCI_Thesaurus/

methods are associated to blood cells, and the last one allow users to know resistant sensitivity cases produced by proteins.

Finally, Table 4 shows some examples of *OntoPath* that also retrieves instances. As GALEN and NCI do not provide instances, we took instead the *wine*[15] ontology, which illustrates several constructors for individuals and nominal entities.

Table 3. Query examples and fragment features for the GALEN ontology

Galen.owl				
OntoPath Query	**Classes**	**Created**[16]	**Properties**	**i-properties**
Whole Ontology	3002	221	413	2168
AbsoluteMeasurement/?/Cell/?/LiquidBlood	29	10	2	18
*/hasState/resistant	8	0	1	2
*/Attribute/resistant	8	0	1	2
[hasSubprocess/][isFunctionOf/*]	3	1	2	2
/isFunctionOf/	86	20	2	24
/hasSubprocess/	26	5	1	11
Sensitivity[hasState/resistant][Attribute/presence/?/Protein]	10	2	3	3
CardiacPathology/?/Ischaemia/?/Chronicity/?/acute	7	2	3	5
/isStructuralComponentOf//?/Extremity	34	0	2	23
*/isSolidDivisionOf/UpperExtremity	7	0	1	6

Table 4. Query examples for retrieval of instances in the wine ontology

Wine.owl				
OntoPath Query	**Classes**	**Properties**	**i-properties**	**Instances**
Whole Ontology	76	13	101	161
*/locatedIn/Region	64	1	4	42
Region/?/*	1	2	0	36 (regions)
Wine	63	0	0	53 (wines)
Wine/?/*	72	9	61	160
Wine/?/Region	64	1	25	89
/hasSugar/	64	1	24	56
[hasSugar/][hasBody/*]	65	2	31	59

8 Conclusions

Ontology modularization [1] and segmentation [2] have gained an important weight in domains such as biomedicine, where available ontologies are huge. In these cases, several issues force the final users to work with a subset of the ontology: scalability problems in the reasoning over the whole ontology, visualization in ontology editors, partial knowledge of the domain, maintenance and extension, and use in concrete applications (i.e.: information extraction guided by ontologies).

The general aim of this work is to extract consistent, closed, and useful ontology fragments, suitable for concrete applications or for knowledge exploration purposes. In contrast with ontology modularization approaches, we do not advocate automatic

[15] Wine Ontology: http://protege.cim3.net/file/pub/ontologies/wine/wine.owl
[16] Classes created by means of pattern 4.

(formal or semi-formal) techniques to partition ontologies without the participation of the final user of the ontology, which is an important limitation because the generated modules may not be useful for concrete applications. Our proposed work follows a similar approach to [13, 2], with respect to the guided module/fragment definition, where final users define their custom knowledge, instead to work with a previously module or with the whole ontology.

Currently, we are working in the definition of a well founded framework for our fragment extraction mechanism, in order to maintain and to express formally (we mean as formally a kind o representation that a reasoning system can understand; currently the maintained references does not follows any formalism, only syntactic references) the connections with the original ontology and between fragments. These connections will allow the final user to expand its previously defined fragment with more knowledge from the original ontology (or other fragments). From the literature we can emphasize, as a good staring point, the *Safe Ontology Modularization* [10] and the $\mathcal{E}$*-Connections* [17] approaches. The former one attempts to define *safe* ontology modules, whereas the latter aims at extending the OWL syntax and semantics to represent the connection between ontology modules.

Acknowledgements

This work has been partially funded by the PhD Fellowship Program of the Generalitat Valenciana, the CICYT Project TIN2005-09098-C05-04 from the Spanish Ministry of Education and Science, and the Health-e-Child European Project.

References

1. Wang, Y., Haase, P., Bao, J.: A Survey of Formalisms for Modular Ontologies. In: IJCAI 2007. Workshop SWeCKa, Hyderabad, India (January 2007)
2. Seidenberg, J., Rector, A.: Web ontology segmentation: Analysis, classification and use. In: WWW. Proceedings of the World Wide Web Conference, Edinburgh (June 2006)
3. Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., Stuckenschmidt, H.: COWL: Contextualizing Ontologies. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 164–179. Springer, Heidelberg (2003)
4. Seaborne, A., Prud'hommeaux, E.: SparQL Query Language for RDF (February 2005) http://www.w3.org/TR/rdf-sparql-query/
5. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: Proceedings WWW 2002, Hawaii, USA, USA (2002)
6. Jimenez-Ruiz, E., et al.: The Management and Integration of Biomedical Knowledge: Application in the Health-e-Child Project. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2006. LNCS, vol. 4278, Springer, Heidelberg (2006)
7. Wang, L., et al.: Biostar models of clinical and genomic data for biomedical data warehouse design. Int. Journal of Bioinformatics Research and Applications (2005)
8. Paraire, J., Berlanga, R., Llidó, D.M.: Resolution of Semantic Queries on a Set of Web Services. In: Andersen, K.V., Debenham, J., Wagner, R. (eds.) DEXA 2005. LNCS, vol. 3588, pp. 385–394. Springer, Heidelberg (2005)

9. Borgida, A., Serafini, L.: Distributed description logics: Directed domain correspondences in federated information sources. In: CoopIS/DOA/ODBASE, pp. 36–53 (2002)
10. Cuenca-Grau, B., Horrocks, I., Kazakov, Y., Sattler, U.: Extracting Modules from Ontologies: A Logic-based Approach. In: OWLED 2007. Proc. of the Third International OWL Experiences and Directions Workshop (2007)
11. Bao, J., Caragea, D., Honavar, V.: Towards collaborative environments for ontology construction and sharing. In: CTS 2006. International Symposium on Collaborative Technologies and Systems, pp. 99–108. IEEE Computer Society Press, Los Alamitos (2006)
12. Stuckenschmidt, H., Klein, M.: Structure-based partitioning of large concept hierarchies. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, Springer, Heidelberg (2004)
13. Noy, N., Musen, M.A.: Specifying ontology views by traversal. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 713–725. Springer, Heidelberg (2004)
14. Bhatt, M., et al.: Semantic completeness in sub-ontology extraction using distributed methods. In: Laganà, A., Gavrilova, M., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3045, pp. 508–517. Springer, Heidelberg (2004)
15. Volz, R., Oberle, D., Studer, R.: Implementing Views for Light-weight Web Ontologies. IEEE Database Engineering and Application Symposium (2003)
16. Jiménez-Ruiz, E., Berlanga, R.: A View-based Methodology for Collaborative Ontology Engineering: an Approach for Complex Applications (VIMethCOE). In: STICA. 1st International Workshop on Semantic Technologies in Collaborative Applications (June 2006)
17. Cuenca-Grau, B., et al.: Automatic Partitioning of OWL Ontologies Using E-Connections. In: DL 2005. International Workshop on Description Logics (2005)
18. Jiménez, E., Berlanga, R., Sanz, I., Aramburu, M.J., Danger, R.: OntoPathView: A Simple View Definition Language for the Collaborative Development of Ontologies. In: López, B., et al. (eds.) Artificial Intelligence Research and Development, IOS Press, Amsterdam (2005)
19. Jiménez-Ruiz, E., Nebot, V., Berlanga, R., Sanz, I., Rios, A.: A Protégé Plug-in-Based System to Manage and Query Large Domain Ontologies. In: 10th Intl. Protégé Conference, Budapest, Hungary (2007), http://protege.stanford.edu/conference/2007/schedule.html
20. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The making of a web ontology language. Journal of Web Semantics 1(1), 7–26 (2003)
21. Christophides, V., Plexousakis, D., Scholl, M., Tourtounis, S.: Optimizing Taxonomic Semantic Web Queries Using Labeling Schemes. Journal of Web Semantics 1(2) (2004)
22. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD 1989, ACM Press, New York (1989)
23. Magkanaraki, A., Tannen, V., Christophides, V., Plexousakis, D.: Viewing the Semantic Web through RVL Lenses. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, Springer, Heidelberg (2003)
24. Means, W.S., Harold, E.R.: XML in a Nutshell A Desktop Quick Reference, January 2001, Chapter 9: XPath: http://www.oreilly.com/catalog/xmlnut/chapter/ch09.html
25. Freund, J., et al.: Health-e-Child: An Integrated Biomedical Platform for Grid-Based Pediatrics. In: Health-Grid Conference, Valencia (2006)

View publication stats