A Structured Semantic Query Interface for Reasoning-Based Search and Retrieval

Dimitrios A. Koutsomitropoulos[1], Ricardo Borillo Domenech[2], and Georgia D. Solomou[1]

[1] High Performance Information Systems Laboratory (HPCLab), Computer Engineering and Informatics Dpt., School of Engineering, University of Patras, Buidling B, 26500 Patras-Rio, Greece
[2] Servicio de Informática, Universitat Jaume I, Rectorado, 12071, Castellón, Spain
kotsomit@hpclab.ceid.upatras.gr,
borillo@uji.es,
solomou@hpclab.ceid.upatras.gr

Abstract. Information and knowledge retrieval are today some of the main assets of the Semantic Web. However, a notable immaturity still exists, as to what tools, methods and standards may be used to effectively achieve these goals. No matter what approach is actually followed, querying Semantic Web information often requires deep knowledge of the ontological syntax, the querying protocol and the knowledge base structure as well as a careful elaboration of the query itself, in order to extract the desired results. In this paper, we propose a structured semantic query interface that helps to construct and submit entailment-based queries in an intuitive way. It is designed so as to capture the meaning of the intended user query, regardless of the formalism actually being used, and to transparently formulate one in reasoner-compatible format. This interface has been deployed on top of the semantic search prototype of the DSpace digital repository system.

Keywords: Semantic Web, queries, ontologies, entailment, guided input.

1 Introduction

The growing availability of semantic information in today's Web makes ontology-based querying mechanisms necessary. Europeana for example counts over 10M of semantic objects corresponding to heritage and collective memory resources [14]. And this currently forms only the tip of the iceberg: Vast amounts of Linked Data exist and continuously emerge out of DBpedia, social applications, open government data and other sources.

However, querying the Semantic Web is not a straightforward task, especially in case of expressive ontology languages, like OWL and OWL 2 where inference holds a key part. In addition to the current lack of protocols and standards for efficiently searching through ontological information, one has to cope with the added complexity Semantic Web queries inherently bear with:

- *A priori ontological knowledge*: In order to formulate an expressive query and to bound results, the user needs to know in advance class names, properties and individuals that consist the ontology's contents. Alternatively, suitable mechanisms are necessary to expose this information to the user.
- *Expert syntaxes,* like Description Logics (DLs), SPARQL, Manchester Syntax, that are usually difficult to read and comprehend, let alone to compose from scratch for non-expert users.
- *Inherent complexity*: The added-value of ontological, entailment-based querying does not surface unless a suitable query expression is as elaborate as possible. In order to surpass the level of relational queries, one needs to delve into complex combinations of classes, properties and restrictions, thus formulating expressive conjunctive queries, impossible to express or answer using relational techniques [9].

Finally, *NLP approaches* are not always a sound solution: While they can produce meaningful results in a number of cases [8], there is virtually no guarantee that the intended user query will actually be captured. Parsing a free text sentence may or may not correspond to a successful query expression or to the one that the user would have meant to.

Therefore in this paper we propose a *structured* querying mechanism and interface that helps to construct and submit entailment-based queries to web ontology documents. The main idea is to aid the user in breaking down his intended query expression into several *atoms*. These atoms are then combined to form allowed expressions in Manchester Syntax, as the closest to our purposes regarding user-friendliness. At the same time, the interface tries to be as intuitive as possible by automatically disallowing (graying out) nonsensical combinations (for example, select a restriction without selecting a property first), offering dynamic auto-complete choices and classify them as per class (type) or relation, disclosing namespace prefixes when possible, marking the various fields with NL-like labels and presenting results based on their class or type.

Based on this idea we have developed a prototype application as an add-on to the DSpace digital repository system, latest version (1.6.2 and 1.7.0)[1]. This work builds upon and evolves earlier efforts for creating a semantic search service for DSpace [10]. The novel semantic search interface is backed up by a new *DSpace Semantic API* that supports a pluggable design for reasoners as well as OWL 2.0 and the newest OWL API v.3. Most importantly, our Semantic API is designed along the same principles as its predecessor, i.e. to remain independent of the DSpace business logic and to be agnostic to the rest of the user interface or even the underlying ontology.

In the following we first review current approaches for querying the Semantic Web and point out our decisions for the interface design (Section 2). Then we describe the design and architecture of the DSpace Semantic API, which the querying services are

[1] `http://www.dspace.org/`

based on (Section 3). Section 4 describes the user-perceived functionality of our interface and presents some indicative examples. Finally, section 5 and 6 summarize our conclusions and future work.

Our prototype is openly available at: http://apollo.hpclab.ceid.upatras.gr:8000/jspui16/semantic-search. Source code is maintained as a Google Code project[2] where instructions and latest developments can be found.

2 Background

As long as there is not yet a standard query language specifically for OWL ontologies, a search mechanism that intends to utilize a formal query language has to choose among either a DL-based or a RDF-based approach. The former category is more closely related to rule languages and logics. The latter includes SQL-like languages, aiming at retrieving information from RDF documents. No matter what approach is actually followed, Semantic Web query effectiveness highly depends on the mechanisms employed to actually construct the query, as discussed in the previous section.

2.1 Syntaxes for OWL Query Languages

Some known languages for querying RDF data are SPARQL [15], SeRQL[1], RDQL[16] and more. But SPARQL is the one that has been recognized as the de facto standard for the Semantic Web. Because SPARQL has been mainly designed for querying RDF documents, its semantics are based on the notion of RDF graphs and thus it has no native understanding of OWL vocabulary.

Even when a bridging axiomatization is offered (like for example with SPARQL-DL [18]), OWL-oriented queries in SPARQL can become extremely verbose, especially in case complex OWL expressions are involved. To this end, some SPARQL variants have been recently proposed, such as SPARQLAS[3] and Terp [17], bearing a more OWL-friendly profile: they both intend to facilitate those who are not familiar with SPARQL to write queries in this language, by allowing the mix with OWL syntactic idioms, like OWL functional syntax and OWL Manchester Syntax, respectively.

Nevertheless, DL-based query languages, compared to the RDF-based ones, have more well-defined semantics w.r.t. OWL since they were designed exactly for querying OWL (and OWL 2) documents. For example nRQL [4], OWL-QL [3] and OWLlink's ASK protocol [11] fall in this category.

Provided that querying an ontology is often about finding individuals or instances, another simple yet powerful approach is to directly employ DL syntax to write these queries. For example, the query tab of Protégé 4[4] follows this practice. It uses the Manchester syntax [6], which is a "less logician like" and more user-friendly syntax for writing OWL class expressions.

[2] `http://code.google.com/p/dspace-semantic-search/`
[3] `http://code.google.com/p/twouse/wiki/SPARQLAS`
[4] `http://protegewiki.stanford.edu/wiki/DLQueryTab`

When a user poses a query, a parser maps the given expression into a concept expression (class). Consequently, all instances classified by the reasoner under this particular concept are retrieved. In this sense, Manchester syntax can be used as an entailment-based querying language for OWL documents and this is the approach we follow in our implementation.

2.2 Query Formulation

The existing approaches for performing search on the Semantic Web can be roughly divided in two categories [2]: those using structured query languages, like the ones described previously, and those expressing natural language or keyword-based queries. The first category includes systems that use a formal language for evaluating queries. The latter category is comprised of applications that accept either a whole phrase (expressed in natural language) or simple keywords as queries. The NL-based approaches usually require an additional reformulation step where posed queries are translated into class expressions or triples, depending on the target language. Such a system is PowerAqua [12], where a user query is translated from natural language into a structured format. Similarly, in keyword-based systems, keywords are matched to parts of an RDF graph or to ontology elements and then evaluated against the knowledge base. QUICK [20] belongs to this category and is based on the idea of assigning keywords to ontology concepts.

The parsing and reformulation process that is necessary in NL-based and keyword based approaches restricts systems functionality, as it involves further query analyzing, and sometimes makes handling complex requests difficult or even impossible. On the other hand, the use of formal structured languages in query interfaces assumes that users have at least a basic understanding of the language's syntax as well as of the underlying ontology's structure. This requirement leads to more expert-user oriented applications, not suitable for common users who are not familiar with the logic-based Semantic Web.

A way to bridge the gap between the complexity of the target query language and end users is to develop a guided input query interface. This is a practice followed along several years, by applications querying database systems, in order to facilitate the formulation of more complex SQL requests. For example, an advanced search facility in a digital library system, like DSpace, utilizes drop down menus with Boolean operators so as to help users in setting restrictions when searching the system's database. When searching knowledge bases, though, where ontologies are involved, things become much more complicated.

Several semantic search systems that guide users in structuring their requests have been proposed in the literature, but this is mostly about systems that use SPARQL and SPARQL-like languages. In addition, they are usually focused on graphical or visual techniques, like NITELIGHT [19] and Konduit [13]. To the best of our knowledge no other system using DL-based query languages exists, that follows the idea of controlled input forms for the structured formulation of semantic queries.

3 The DSpace Semantic API

In this section we focus on the design and implementation of the semantic search service, which has been developed as an add-on to the new DSpace 1.7. We describe the main components of the Semantic API and then we point out its interaction with inference engines in order to support entailment-based queries. First however we briefly introduce the DSpace ontology model which acts as the underlying knowledge base for queries.

3.1 The DSpace Ontology

The first step in developing any semantic search service is to identify or construct the target knowledge base or ontology, which queries will actually be performed against. In our case, we construct the DSpace ontology on-the-fly, following a sophisticated procedure fully described in [9] and based on the interoperable system's mechanisms for exporting resources' metadata through OAI-PMH.

DSpace metadata follow the Dublin Core (DC) specification by default, while it is possible to import and use other metadata schemata as well. In our particular implementation, we have also enhanced the system's metadata schema with learning object (LOM) metadata, specifically tailored for its usage as an institutional repository (http://repository.upatras.gr/dspace/). The resulting ontology comprises of axioms and facts about repository items and is expressed in OWL 2 (Fig. 1).

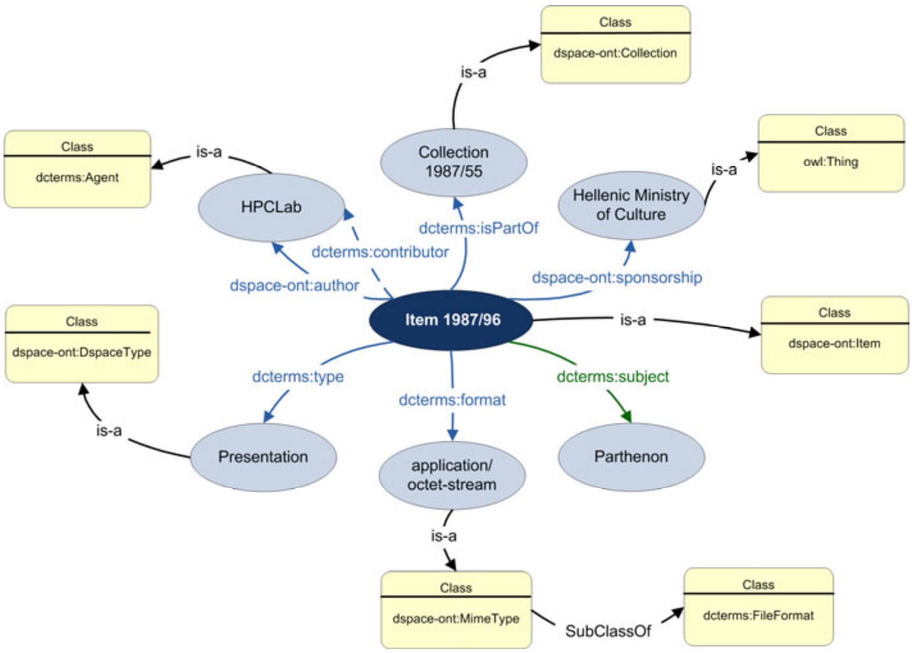


Fig. 1. An example instance of the DSpace ontology

3.2 Design and Architecture

The semantic search service uses several APIs to perform search and inference against the ontology. The OWL API [5] is used as the basis for ontology manipulation and interaction with reasoners. Compared to our previous efforts [10] we have now migrated to the latest OWL API v. 3.1.0 to support proper handling of OWL 2 idioms as well as to better interface various reasoners. For the latter, we have also upgraded to FaCT++ v. 1.5.0[5] and added the ability to "hot-swap" between reasoners dynamically, adding support also for Pellet[6].

Figure 2 depicts the different components of the semantic search service in relation to the DSpace infrastructure.

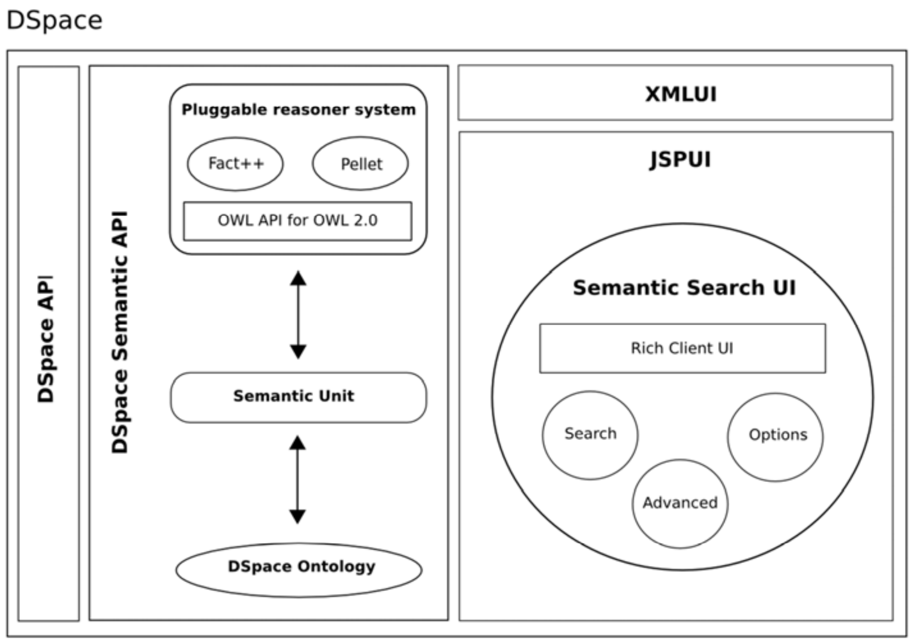


Fig. 2. The architecture of the semantic search service for DSpace

All these components are part of the DSpace Semantic API. The DSpace Semantic API is defined at the same level as the DSpace API. This new API can be used in the rest of DSpace modules without problems, like the JSP user interface (JSPUI), XMLUI, REST API or LNI. In our implementation, the DSpace Semantic API interacts with the JSPUI module by means of a new user interface for querying DSpace digital objects using an ontology (see Section 4).

The *Semantic Unit* is the core component and the mediator between all the facts and relations defined in the DSpace ontology and the inference engine (Fact++, Pellet or another OWL API-compliant reasoner).

[5] `http://code.google.com/p/factplusplus/`
[6] `http://clarkparsia.com/pellet/`

3.3 The Semantic Unit

When the implementation of the semantic layer began, a core piece with all the basic and necessary resources to execute semantic queries was created. This main unit was called *Semantic Unit* and was designed as a singleton class to be available to the entire system and initialized at system startup with the DSpace ontology and other default values.

This unit is responsible for the following topics:

- *The OWL ontology manager*. An `OWLOntologyManager` manages a set of ontologies. It is the main point for creating, loading and accessing ontologies. An `OWLOntologyManager` also manages the mapping between an ontology and its ontology document.
- *The OWL ontology* itself.
- *The imports closure*. This is just the union or aggregation of the imported ontology documents, referenced by the `owl:imports` directive.
- *The short form provider*. In OWL, entities such as classes, properties and individuals are named using URIs. Since URIs can be long and not particularly readable, "short forms" of these URIs are often used for presentation in tools such as editors and end user applications. Some basic implementations of short form providers are:
  - `SimpleShortFormProvider`. Generates short forms directly from URIs. In general, if the fragment of a URI is available (the part of the URI following the #) then this will be used for the short form.
  - `QnameShortFormProvider`. Generates short forms that look like *QNames*. For example, `owl:Thing`, `pizza:MarghertiaPizza`.
- *The reasoner* used.

The Semantic Unit is also a registry for caching purposes. This allows to reuse the loaded ontologies and short form providers, avoiding reload and parsing of the whole ontology definition. As a result, when a user loads a new ontology, this is loaded and stored only once by the Semantic Unit in an internal registry. When another user asks for the same ontology, no re-parsing is needed, and the ontology is served from the registry.

This functionality is used around the system to perform some basic interactions. The Semantic Unit is used in the main module to issue queries with the values provided by the end user. Additionally, the Semantic Unit is also used in the construction of the search results page, when the user asks for a detailed view of a record and the system performs some inference against the ontology. Summarizing, the Semantic Unit will be used in every situation where the system needs to perform any operation against the loaded ontology.

3.4 Pluggable Reasoner Design

One of the design principles of the semantic search service was openness and support for different reasoners and ontologies. In this way, a proper design on source code is also needed.

To allow the Semantic API to load different reasoners dynamically, a little use of reflection and a general interface definition was used:

```
public interface OWLReasonerFactory
{
    public OWLReasoner getReasoner(OWLOntology
ontology);
}
```

This is the main interface that is required to be implemented for every reasoner that we want to incorporate to our semantic search service. This is only a factory method implementation to create the proper instance of the reasoner. That was needed because different reasoners have their own API for creating instances. Once a reasoner instance is generated, no customization is needed because of the fact that all reasoners implement the `OWLReasoner` OWL API interface.

This can be accomplished by using Java Reflection and by relying on system configuration to determine what the correct reasoner that needs to be loaded is:

```
SupportedReasoner supportedReasoner =
SupportedReasoner.PELLET;
OWLReasonerFactory owlReasonerFactory =
(OWLReasonerFactory) Class.forName(
supportedReasoner.toString()).newInstance();
OWLReasoner reasoner =
owlReasonerFactory.getReasoner(ontology);
```

All the supported reasoners are defined in a Java enumeration class:

```
public enum SupportedReasoner
{
FACTPLUSPLUS("gr.upatras.ceid.hpclab.reasoner.OWLReason
erFactoryFactPlusPlusImpl"),
PELLET("gr.upatras.ceid.hpclab.reasoner.OWLReasonerPell
etImpl");
    private String classImpl;
    SupportedReasoner(String value)
    {
        classImpl = value;
    }

    public String toString()
    {
        return classImpl;
    }
}
```

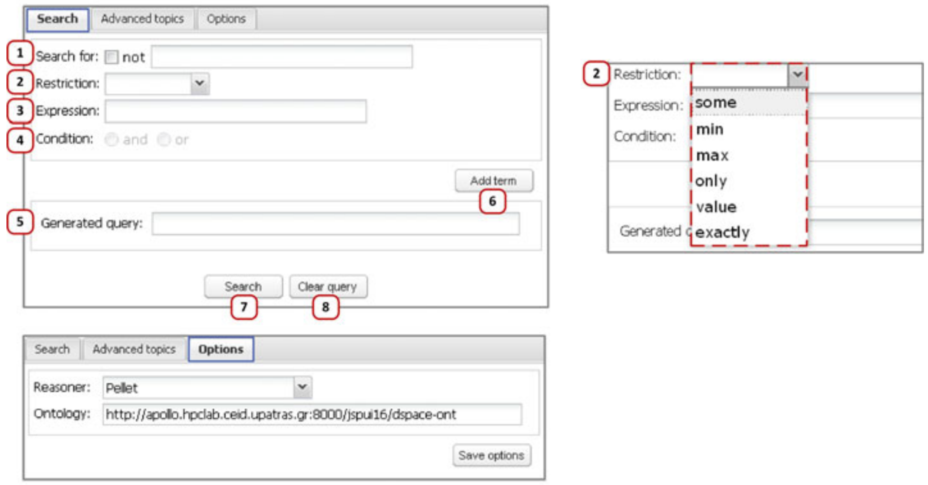
Following these guidelines two initial implementations called `OWLReasoner FactoryFactPlusPlusImpl` for Fact++ and `OWLReasonerPelletImpl` for Pellet support have been added.

4 Functionality and Examples

In this section we describe how our semantic search service interacts with users, guiding them smoothly in the construction of correct and accurate queries. First, a detailed description of the interface building components is given; then, we present how users can take advantage of this interface, by showing some indicative examples.

4.1 The Interface

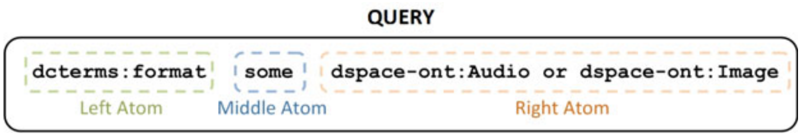
When the semantic search interface is loaded, one can distinguish among three separate tabs: *Search* (default), *Advanced topics* and *Options*.



Reasoner: Pellet
Ontology: http://apollo.hpclab.ceid.upatras.gr:8000/jspui16/dspace-ont

Fig. 3. *Search* and *Option* tab of the semantic search interface

The *Search* form contains all necessary elements for guiding users in building queries in Manchester syntax as intuitively as possible. Each component in this form corresponds to a certain building atom of the query expression. Their functionality is described in detail later in this section.



```
dcterms:format some dspace-ont:Audio or dspace-ont:Image
```

Fig. 4. The building atoms of a query expression in Manchester syntax

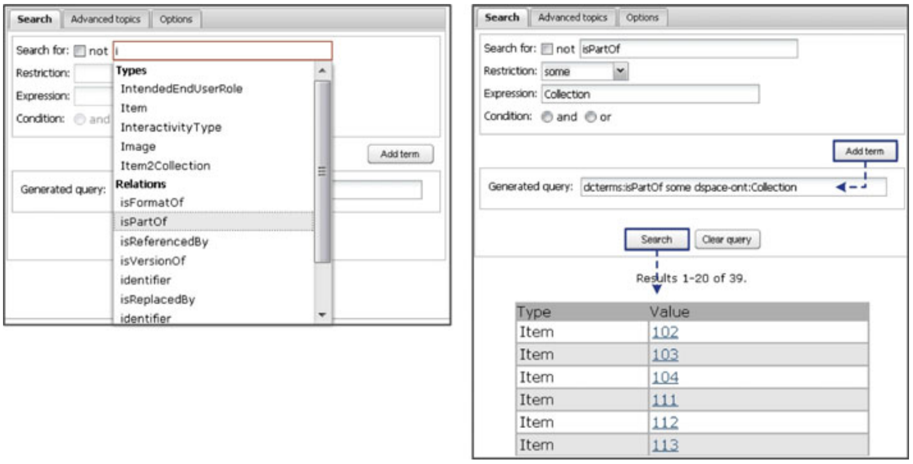
The *Advanced topics* tab is currently inactive and is reserved for future extensions of the system, like for example the support of other query syntaxes, such as SPARQL.

The *Options* tab includes options that allow users to change the ontology against which they perform their search, as well as the underlying reasoner (currently, Pellet or FaCT++). For altering the knowledge base, we only need to supply a valid URL of an OWL/OWL 2-compliant ontology. In addition, the user can switch between reasoners dynamically (Section 3.4).

Next we describe the various elements of the *Search* tab, according to their number in Fig. 3. Based on Manchester syntax's primitives for formulating an expression [7] and depending on the values entered by the user in each preceding field, subsequent fields are enabled or disabled accordingly. Figure 4 depicts the three main atoms of such a query expression. What is more, an auto-complete mechanism is enabled where necessary, for guiding users in supplementing information.

1. **Search for:** It corresponds to the outmost left (first) atom of a Manchester syntax expression. This can be either a property name or a class name. An auto-complete mechanism is triggered as soon as a word starts being typed, suggesting names for classes and properties that exist in the loaded ontology. For users' convenience, suggested values have been grouped under the title *Types* (for classes) and *Relations* (for properties) (left part of Fig. 5). The check box is used for declaring the negation of the class expression that starts being constructed. For simplicity, all prefixes are kept hidden from users and the system is responsible for adding them automatically, during the query generation process. The following two fields are not activated, unless a property name has been selected in this step.
2. **Restriction:** This represents the middle atom of the expression. Provided that a property is entered in the previous field, a number-, value- or existential restriction should now be set. Hence, the 'Restriction' drop down menu becomes active, containing respective Manchester syntax keywords.
3. **Expression:** This is a free-text field where the user can supply a single class name or expression (quantification), an individual (value restriction) or a number, optionally followed by a class (cardinality restriction). An auto-complete facility is provided for class names. This forms the outmost right (last) atom of the query expression.
4. **Condition:** From now on, the user can recursively construct more class expressions, combining them in conjuctions (*and*) or disjunctions (*or*). Consequently an appropriate *Condition* should be set for expressing the type of logical connection.
5. **Generated Query:** This field gradually collects the various user selections and inputs, ultimately containing the final query expression. It is worth noting that this is an editable text box, meaning that expert users can always bypass the construction mechanism and use it directly for typing their query.
6. **Add term:** Adds a completed expression to the *Generated Query* field. This also checks if the expression to be added is valid, and pops an error message otherwise.
7. **Search:** When pressed, evaluates the query expression as it appears in the *Generated Query* field. It also clears all other fields, thus giving the user the opportunity to further refine his initial query.
8. **Clear query:** Clears the form and makes it ready to accept new values.

Once the query has been evaluated, obtained results appear right below the search form. They are organized in the form of a two-column table, as depicted in Fig. 5: *Value* contains the retrieved entities, whereas *Type* indicates at least one of the classes


Search | Advanced topics | Options

Search for: ☐ not i
Restriction:
Expression:
Condition: ○ and

Types
IntendedEndUserRole
Item
InteractivityType
Image
Item2Collection
Relations
isFormatOf
isPartOf
isReferencedBy
isVersionOf
identifier
isReplacedBy
identifier

Add term

Generated query:

Search | Advanced topics | Options

Search for: ☐ not isPartOf
Restriction: some
Expression: Collection
Condition: ○ and ○ or

Add term

Generated query: dcterms:isPartOf some dspace-ont:Collection

Search | Clear query

Results 1-20 of 39.

Type	Value
Item	102
Item	103
Item	104
Item	111
Item	112
Item	113

Fig. 5. The auto-complete mechanism and the results table in the semantic search interface

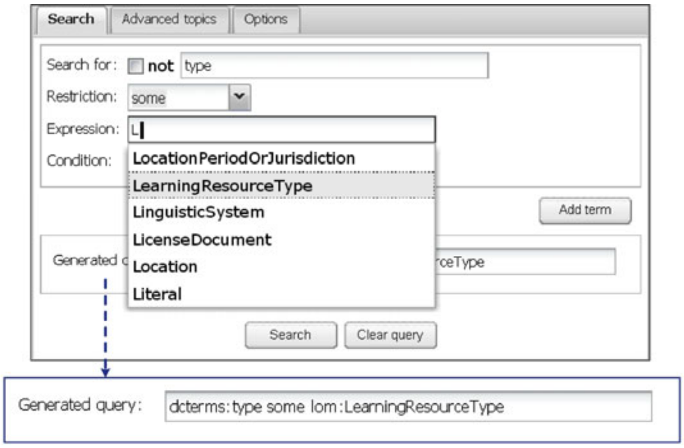
to which each entity belongs, thus providing users with a hint about their type. All retrieved entities are clickable and when selected, a separate page is loaded, containing the complete ontological information about the clicked entity. More details about this page and its elements can be found in [10].

4.2 Example Queries

First we show how a relatively simple class expression can be built through the interface. In particular, we want to retrieve all DSpace entities that have a type of `lom:LearningResourceType`. This corresponds to the following expression:

`dcterms:type some lom:LearningResourceType`

The way it is constructed through the semantic search interface is shown in Fig. 6.


Search | Advanced topics | Options

Search for: ☐ not type
Restriction: some
Expression: L
Condition:

LocationPeriodOrJurisdiction
LearningResourceType
LinguisticSystem
LicenseDocument
Location
Literal

Add term

Generated q ceType

Search | Clear query

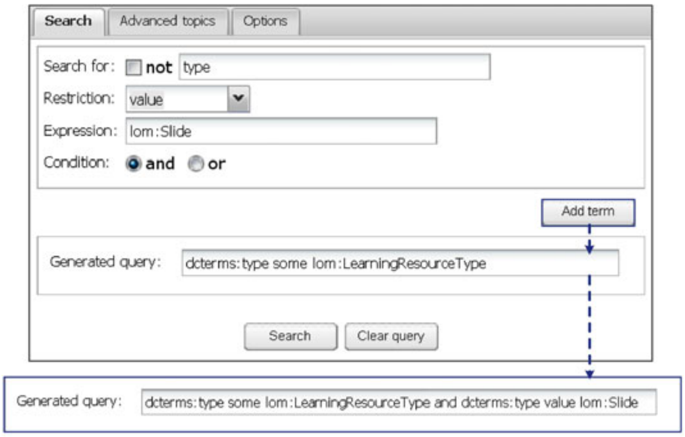
Generated query: dcterms:type some lom:LearningResourceType

Fig. 6. Formulating a simple query with the guidance of the semantic search interface

The second query refines the previous one, by asking for those items that also satisfy the requirement to be slides. This is expressed as follows:

```
dcterms:type some lom:LearningResourceType and
        dcterms:type value lom:Slide
```

To formulate this query in the semantic search interface, we construct the appropriate class expression representing our new requirement and attach this to the previous one, by checking the 'and' condition (Fig. 7).

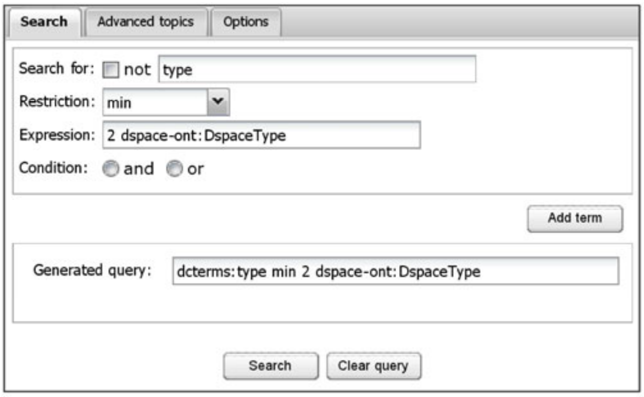


Search for: not [ ] type
Restriction: value
Expression: lom:Slide
Condition: [✓ and] or
Generated query: dcterms:type some lom:LearningResourceType
Generated query: dcterms:type some lom:LearningResourceType and dcterms:type value lom:Slide

Fig. 7. Combining query expressions using the 'and' condition

Finally, we construct a query for retrieving all DSpace items for which we have used more than one DSpace-specific types for their characterization (e.g., learning object and book, presentation and dataset, etc) (Fig. 8).

```
dcterms:type min 2 dspace-ont:DspaceType
```



Search for: not [ ] type
Restriction: min
Expression: 2 dspace-ont:DspaceType
Condition: and or
Generated query: dcterms:type min 2 dspace-ont:DspaceType

Fig. 8. A more complex query that requires manual typing for the construction of its right atom

In this case the user has to manually input the right query atom where the class name should be accompanied by the required prefix. Note also that such a cardinality-based query cannot be submitted with a traditional, keyword-based mechanism. The same holds for a whole set of other queries that are made possible only through inferencing (see [9] for more examples of such queries).

5 Future Extensions

Currently, the semantic search service is targeted towards guiding novice users in forming simple expressions. For example it is difficult – although not impossible – for someone not familiar with XSD facets to construct composite queries containing numeric or string ranges. The creation of nested queries requires particular attention as well, because the default priority in evaluating a nested expression involving Boolean operators can only be altered using parentheses. For example the expression:

```
dspace-ont:author some dcterms:Agent and dspace-ont:Item
```

evaluates differently than

```
dspace-ont:author some(dcterms:Agent and dspace-ont:Item)
```

Another possible improvement would be to display all class names in the results list. Since query results can belong to more than one class, it would be useful to see all these classes, instead of the most specific one, in the form of a tooltip.

Additionally, more checks and guided options can be added to the user interface, based on what part of the final expression is being defined by the user. For example, the ‘Expression’ field can be controlled depending on what is the user’s choice in the ‘Restriction’ field: ‘some’ and ‘only’ restrictions are always followed by class expressions; ‘value’ needs an individual or literal; and cardinality restrictions (‘min’, ‘max’ and ‘exactly’) are followed by a number and a class expression. Currently, this is circumvented by the ability to give free-text input to the ‘Expression’ box, while auto-complete would work for class names.

In any case, the actual functionality of the system is not hindered, given that the provided query box is editable; therefore someone who is familiar with ontologies and Manchester syntax has no difficulty in proceeding with complex requests.

In addition, more reasoners (e.g. Hermit[7]) and querying approaches, like SPARQL, could be accommodated. Finally, for efficiency and scalability reasons it would be preferable to integrate a persistent semantic storage mechanism with our service. Thus we would be able to support dynamic ontology updates and incremental reasoning, although these techniques are currently well beyond the state of the art.

6 Conclusions

The Semantic Web has grown by the years an extensive technological infrastructure, as it is evident by the increasing number of tools, standards and technologies that build around it. Its success however will be determined by the added-value and

[7] `http://hermit-reasoner.com/`

tangible gains it brings to the end users. To this extend, not only an adequate number of linked and open information – that would form a "Web of Data" – need to be available, but also efficient and intuitive processes for ingesting this information should be developed. Querying Semantic Web data should not put aside their underlying logic layer either: instead, entailment-based query answering must be integrated and utilized into querying systems, thus bringing the Semantic Web to its full potential.

In this paper we have presented a straightforward approach for querying ontological information by employing the idea of structuring queries through guided input. This necessity comes naturally out of the complexity that is almost inherent in logic-based queries. Besides, current research efforts seem to coincide in trying to alleviate this very problem, no matter what approach do they actually follow – be it text- or formal-based. To our knowledge, this is the first effort to use a DL-based query language that follows the idea of controlled input forms for the structured formulation of semantic queries.

Our prototype has been built as an add-on to the DSpace digital repository system, though by design, the implementation is independent of the system's business logic. In addition, it does not depend on any specific ontology, but can load and interact with any ontology document on the Web. Thus it can serve any ontology-based searching facility or easily integrate with other repository systems or libraries.

Initial user feedback seems promising; however our next step is to make this tool widely available to the community, so as to initiate an extensive evaluation from both the developer and end user perspective.

References

1. Broekstra, J., Kampman, A.: SeRQL: An RDF Query and Transformation Language. In: 3rd International Semantic Web Conference, Japan (2004)
2. Fazzinga, B., Lukasiewicz, T.: Semantic Search on the Web. In: Semantic Web-Interoperability, Usability, Applicability (SWJ), vol. 1, pp. 1–7. IOS Press, Amsterdam (2010)
3. Fikes, R., Hayes, P., Horrocks, I.: OWL-QL - A Language for Deductive Query Answering on the Semantic Web. J. of Web Semantics 2(1), 19–29 (2004)
4. Haarslev, V., Möller, R., Wessel, M.: Querying the Semantic Web with Racer + nRQL. In: International Workshop on Applications of Description Logics (ADL 2004), Germany (2004)
5. Horridge, M., Bechhofer, S.: The OWL API: A Java API for Working with OWL 2 Ontologies. In: 6th OWL Experiences and Directions Workshop, Chantilly, Virginia (2009)
6. Horridge, M., Patel-Schneider, P.S.: Manchester Syntax for OWL 1.1. In: 4th OWL Experiences and Directions Workshop, Gaithersburg, Maryland (2008)
7. Horridge, M., Patel-Schneider, P.S.: OWL 2 Web Ontology Language: Manchester Syntax. W3C Working Group Note (2009), `http://www.w3.org/TR/owl2-manchester-syntax/`
8. Kaufmann, E., Bernstein, A.: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users? In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 281–294. Springer, Heidelberg (2007)

9. Koutsomitropoulos, D., Solomou, G., Alexopoulos, A., Papatheodorou, T.: Semantic Metadata Interoperability and Inference-Based Querying in Digital Repositories. J. of Information Technology Research 2(2), 37–53 (2009)
10. Koutsomitropoulos, D., Solomou, G., Alexopoulos, A., Papatheodorou, T.: Digital Repositories and the Semantic Web: Semantic Search and Navigation for DSpace. In: 4th International Conference on Open Repositories, Atlanta (2009)
11. Liebig, T., Luther, M., Noppens, O., Wessel, M.: OWLlink. In: Semantic Web-Interoperability, Usability, Applicability, J. IOS Press, Amsterdam (to appear)
12. Lopez, V., Motta, E., Uren, V.: PowerAqua: Fishing the Semantic Web. In: 3rd European Semantic Web Conference, Montenegro (2006)
13. Möller, K., Ambrus, O., Dragan, L., Handschuh, S.: A Visual Interface for Building SPARQL Queries in Konduit. In: 7th International Semantic Web Conference, Germany (2008)
14. Olensky, M.: Semantic interoperability in Europeana. An examination of CIDOC CRM in digital cultural heritage documentation. IEEE Technical Committee on Digital Libraries 6(2) (2010), http://www.ieee-tcdl.org/Bulletin/current/Olensky/olensky.html
15. Prud'hommeaux, E., Seaborne, A. (eds.): SPARQL Query Language for RDF. W3C Recommendation (2008), `http://www.w3.org/TR/rdf-sparql-query/`
16. Seaborne, A.: RDQL - a query language for RDF. W3C member submission (2004), `http://www.w3.org/Submission/RDQL/`
17. Sirin, E., Bulka, B., Smith, M.: Terp: Syntax for OWL-friendly SPARQL Queries. In: 7th OWL Experiences and Directions Workshop, San Francisco (2010)
18. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. In: 3rd OWL Experiences and Directions, Austria (2007)
19. Smart, P.R., Russell, A., Braines, D., Kalfoglou, Y., Bao, J., Shadbolt, N.R.: A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. In: Gangemi, A., Euzenat, J. (eds.) EKAW 2008. LNCS (LNAI), vol. 5268, pp. 275–291. Springer, Heidelberg (2008)
20. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From Keywords to Semantic Queries - Incremental Query Construction on the Semantic Web. J. Web Semantics 7(3), 166–176 (2009)