Improving Responsiveness of Ontology-Based Query Formulation

Ivan Zorzi, Sergio Tessaris, and Paolo Dongilli

Free University of Bozen-Bolzano, Italy
<lastname@inf.unibz.it>

Abstract. Recent research showed the benefits of adopting formal ontologies as a means for accessing heterogeneous data sources. The use of an ontology not only provides a uniform and flexible approach for integrating and describing these sources, but it can be used to improve the usability of an integrated system by guiding the final user to formulate his/her information needs. More precisely, the task of formulating queries can be supported by an intelligent use of the ontology describing the information sources. In fact, it has been proved that an appropriate use of automated reasoning techniques can support a user in formulating a precise query —which best captures her/his information needs— even in the case of complete ignorance of the vocabulary of the underlying information system holding the data. Previous work has been carried out on intelligent interfaces for query formulation and this paper describes how to improve usability of such systems by reducing the calls to reasoning services.

1 Introduction

In this paper we introduce the reader to the features of an intelligent query interface. We simply call it Query Tool and it was devised to enable users to access heterogeneous data sources by means of an integrated ontology. The Query Tool supports the users in formulating a precise conjunctive query, where the intelligence of the interface is driven by reasoning services running over a given logic-based ontology.

The ontology, which describes a given domain, defines a vocabulary which is richer than the logical schema of the underlying data, and it is meant to be closer to the user's wide vocabulary. The user can exploit the ontology's entities to formulate the query, and she is guided by such a richer vocabulary in order to understand how to express her information needs more precisely, given the knowledge of the system. This latter task —called intensional navigation— is the most innovative functional aspect of our interface. Intensional navigation can help a less skilled user during the initial step of query formulation, thus overcoming problems related to the lack of schema comprehension and enabling her to easily formulate meaningful queries. Queries can be specified through an iterative refinement process supported by the ontology via intensional navigation. The user may specify her request using generic terms, refine some terms

of the query or introduce new terms, and iterate the process. All details are thoroughly described in the coming sections.

Furthermore we draw the attention of the reader towards the optimisation techniques we are applying to the Query Tool in order to improve the usability of the system. Improvements are made working on three fronts: reducing as much as possible the calls to the reasoner, storing the taxonomy, and caching query information.

The paper is organised as follows. First of all we describe the technologies and techniques underlying the system, then we present the actual system (Query Tool) from the user perspective, with a complete exposition of the functionalities of the interface. Afterwards we illustrate the interaction with the reasoning services followed by a section on the optimisations of such a system. Finally, in the discussion section, we show how we are also leveraging natural language generation technologies to enhance user interaction with the interface.

2 Background

2.1 Ontology mediated access to data sources

The purpose of the presented Query Tool is to support query formulation in the context of information access mediated by ontologies. More specifically, the scenario in which we consider the deployment of the tool consists of one or more data sources providing their own query language (e.g. they can be relational sources). The information provided by the sources is described by means of a global ontology together with mappings relating the ontology vocabulary to the vocabulary of the data sources. We do not impose any constraint on the kind of mappings and/or architecture underlying the integration system.

The Query Tool relies on the availability of an ontology providing the vocabulary for the queries and a query engine capable to retrieve the data. These minimal requirements enable the Query Tool to be used in simple cases in which data are retrieved from a knowledge base (see [10]) as well as more complex architectures in which query answering requires complex processing (e.g. using rewriting [2]).

The ontology language adopted by the tool is OWL-DL (see [6]), therefore the conceptual model exposed to the user centres around the concept of classes and properties. While the user is guided to the construction of queries structured in terms and properties which can be refined (see the next sections for details), the system generates conjunctive queries composed by unary (classes) and binary (attribute and relation) predicates.

2.2 Queries

The Query Tool represents queries to the user as trees, in which nodes are labelled by classes and edges by properties. Each node of the tree correspond to a different variable and properties (edges) constitute the joins between a node

and the rest of the query. In this way the conjunctive queries generated by the system are acyclic.[1]

Users interact with the system to refine the query by a set of operations which can be performed on nodes of the query tree. Once selected, a node becomes the focus for the operations which can be divided into substitution (when a class is substituted by more general or specific one) and incremental refinement by addition of compatible classes or properties. Additionally, the system allows the deletion of part of the query.

For each focus the tool suggests the terms and/or properties which can be used to refine the query. This step requires the interaction with an OWL-DL reasoner in order to establish which properties or classes are "compatible" with the current query. This must be done in real time when the user interacts with the tool, since both the query and the focus affect the responses from the reasoner.

For more details on the query language and the user perspective over the tool, the reader is referred to [4]; in this paper we concentrate on showing how we increased the responsiveness of the system by optimising the use of the OWL reasoner.

2.3 Reasoning services

An OWL reasoner is employed to derive the information required to drive the interface. These information range from the taxonomical position of an OWL expression w.r.t. the terms of the ontology, to the satisfiability of an expression.

To allow for the maximum flexibility, the tool communicates with the reasoner by means of the DIG API (see [1]). To one side this enables the possibility of using any compliant reasoner; but on the other side the use of HTTP as underlying transport introduces additional overhead in terms of network connections.

For this reason, one of the first goal we wanted to achieve is to minimise the number of calls to the reasoner (see Section 5).

2.4 An Example

Now we want to present an example that will be referred throughout the paper to better understand the operations involving the reasoner. To do so, we employ an excerpt of the Wine Ontology which is shown in Fig. 1. We adopted the UML notation to represent the *is-a* relationships among terms and we introduced constraints of disjointness where needed.

We have *Wine_Descriptor* as root concept and *Wine_Taste* and *Wine_Colour* as specialisations of Wine_Descriptor.The concept *Wine* has a property *has_Colour* towards concept Wine_Colour; the inverse of this property is *colour_Of* when seen from concept Wine_Colour. Wine_Colour specialises in *Red* and *White* while Wine in *Red_Wine*, *White_Wine*, and *Table_Wine* respectively. Because of lack of space, we are not going to present the axioms of this sample ontology that can anyway be represented in OWL-DL.

[1] From the technical point of view cyclic queries wouldn't pose any problem; however, usability tests conducted in the context of a previous project suggested that the users don't find co-references intuitive enough.

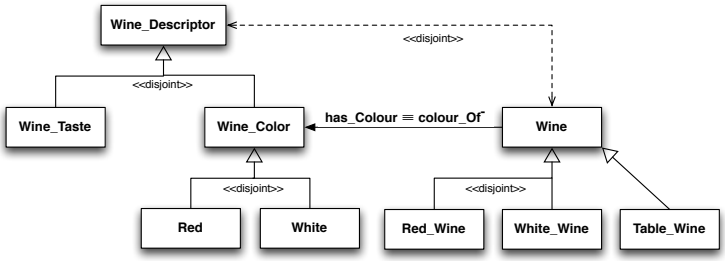

Fig. 1. An excerpt of the Wine Ontology.

3 Query Tool

In this section we present a brief description of the end user system. It is a Java-based application adopting the Standard Widget Toolkit (SWT) [11] for creating the graphical user interface. The system requires at least JRE 1.4 and a DL reasoner providing a DIG 1.x interface.

The query interface is provided with four Tabs:

- Admin: administrative interface used to load the ontology and connect to the reasoner.
- Compose: main query composition interface.
- Query: displays the actual query, mainly for debugging purposes.
- Results: displays the results of the query evaluation.

Initially the user is presented with the *Admin* Tab (see Figure 2). Here, some preliminary operations necessary for the query formulation have to be executed:

1. **Connection setup**: one of the operations the user has to carry out consists in testing the reasoner connection. A reasoner with reference to the ontology is used by the system to drive the query interface: in particular, it is used to discover the terms and properties which are proposed to the user to manipulate the query.
2. **Loading and managing ontology files**: all the operations the system provides cannot be accomplished without loading an ontology. The interface allows the user to specify an ontology in DIG 1.x format to be loaded into the system. Once the ontology is loaded into the system, the user has also the possibility to adjust the content of that ontology, depending on her needs; if the user wants that the modifications take a permanent effect, she can save them back to the file. As a matter of fact, users might frequently have the necessity to extend an ontology in order to obtain different results or to correct it as a consequence of unexpected behaviour.
3. **Loading a metadata file**: the interface gives the possibility to the user to specify a metadata file to be loaded into the system. Metadata files contain valuable information about the terms in the ontology; that information concerns essentially the lexicalisations of those terms. Actually, as the terms

contained in the ontology could be expressed by a sort of shorthand, their lexicalisations are provided so that the user can deal with clearly understandable terms.

4. **Customising lexicalisations**: given the metadata file, the interface offers to the user the opportunity to apply desired variations to the lexical information of the terms. Those variations can be saved back to a metadata file or just saved temporarily in the system. The query to be generated should be as unambiguous as possible: if the user can assign to the terms the lexicalisations which best give significant importance to her, the query formulation will be transparent and therefore the really intended result will be retrieved.

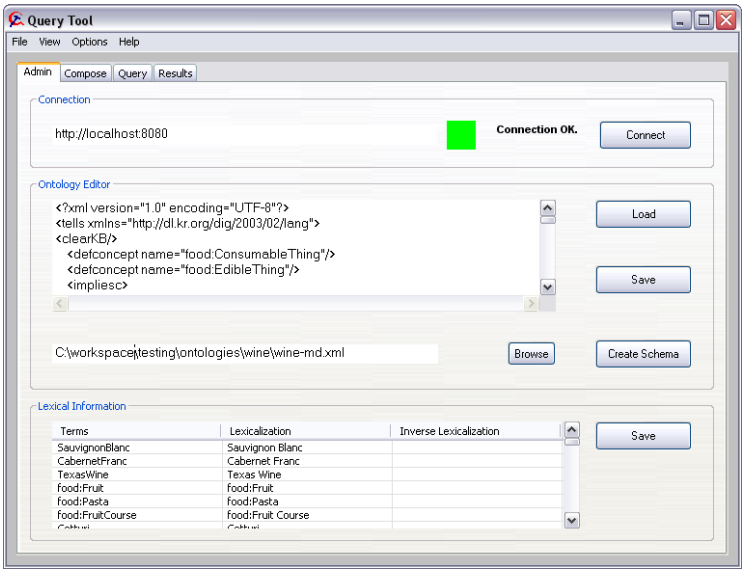

Fig. 2. Administrative interface of the Query Tool.

As you can see in Figure 2, the reasoner connection has been tested by means of the "Connect" button. An ontology has been loaded ("Load" button) and also a metadata file ("Browse" button). Subsequently, the "Create Schema" button has been clicked and all the lexicalisations of the ontology terms are presented in the "Lexical Information" table. Here the user can change the lexicalisations by clicking on the cell corresponding to the lexicalisation she wants to modify.

In the *Compose* Tab (see Figure 3) the user can formulate the query by means of pop-up menus presenting the possible operations. Initially the user is presented with a choice of different starting terms (all the concepts in the ontology or a subset defined by means of the metadata file): she selects the first term to be added in the query. Subsequently, the interface gives the possibility to perform the following operations:

– **Add compatible terms**: other terms specified in the ontology can be added to the query. The compatible terms are automatically suggested to the user by means of appropriate reasoning tasks on the ontology describing the data sources. Indeed, the system suggests only the operations which are compatible with the current query expression.
– **Substitute terms**: the system gives the opportunity of substituting the selected term of the query with a more specific or more general term. It can also be the case that in the ontology there are terms which are equivalent to the selected one: in this case the user is offered to replace the selection with an equivalent term.
– **Delete terms**: as the query is specified through an iterative refinement process, it could be the case that the user needs to delete some terms from the query.
– **Add or delete properties**: analogously, the user can add properties to the query. A property can be a relation or an attribute. The interface suggests a list with the possible alternatives. The user can specify some restriction values to attributes.

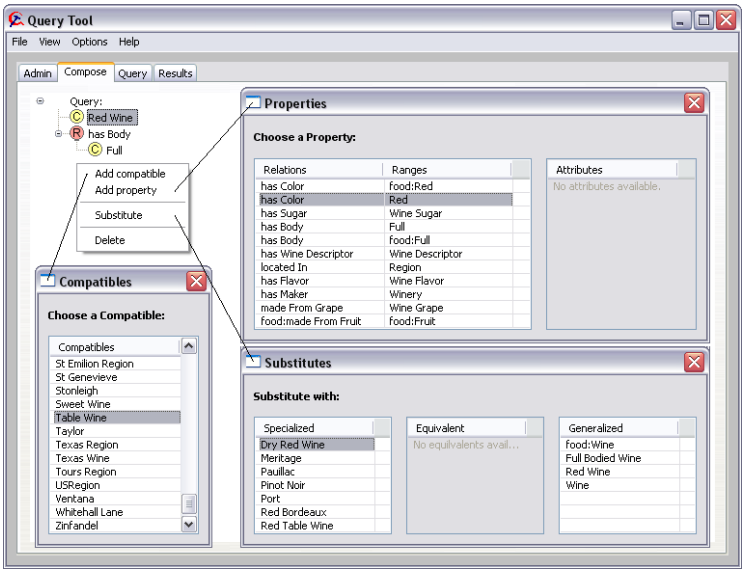

Fig. 3. Query composition interface.

The first operation to compose a query consists in selecting the starting term. By clicking on a pop-up menu ("Choose starting term") the user is presented with a windows showing all the terms that can be used as starting term.

Once the user has selected the starting term, it is possible to refine the query using again the pop-up menu. The operations allowed are listed in the pop-up

menu; the user can add a compatible term, add a property (relation or attribute), substitute the term or delete it.

If the user selects an attribute, it is possible to set it as distinguished variable or to add a restriction to the attribute. The user can also delete properties or terms from the query and select new ones.

Once the user has formulated the query, the *Query* tab shows the query in XML and DIG formats (the menu bar "Options" allows also to view the query in the corresponding SQL code). Finally, in the *Results* tab the user can retrieve the results (if any) corresponding with the formulated query.

In the menu bar, by clicking on "View" menu, the user can have a look to the log file ("View" log) of the application and also a concise description of the schema with all the taxonomy ("View" schema).

4 Reasoner interaction

In this section we describe all the operations (w.r.t. the reasoner) that users can perform during the query formulation process. Refinement of the query expression can be done by the following operations:

- addition of a compatible term;
- addition of a property;
- substitution of a term with an equivalent, more specific or more general term.

In primis we present the approach which enables the system to interact with the reasoner and then the formalisation of the above operations.

4.1 Query rolling-up

Before discussing the actual operations in terms of their use of logical deductions, we need to introduce a fundamental manipulation of the queries which enables us to exploit the reasoning services provided by a DIG reasoner. This operation is the so called *rolling up* of an acyclic conjunctive query (see [8]).

Roughly speaking, the rolling up transforms an arbitrary query without cycles into an equivalent DL expression. The key idea behind is the fact that a (sub)query of the form $P(x, y), R(y)$ is equivalent to the DL expression $(\exists P.R)(x)$. Any variable can be selected as the root of the tree (since we consider acyclic queries) and the rest of the query can be "rolled-up" starting from the leaves.[2]

To analyse the properties of a given query focused on a specific variable, say q^x, we roll-up the query using the focus as the root (with variable x associated with concept F) and then we interrogate the reasoner using the resulting complex concept $Q^{F(x)}$. In the following sections we describe in detail the procedure we employ.

[2] Inverse roles provide the possibility of collapsing queries of the form $P(y, x), R(y)$ as well. If the ontology doesn't include transitive roles and nominals, cyclic queries can be handled in the same way (see [7]). We're considering techniques to allow more expressive languages.

4.2 Addition of a compatible term

This operation requires the list of terms "compatible" with the given query. In terms of conjunctive queries, it corresponds to add a new term to the query. The term is compatible and can be added to the query if the resulting query is satisfiable. Let us formally define a compatible term w.r.t. a query. Given an ontology Σ and a focused query $Q^{F(x)}$ we want to find all the terms $Y \in \mathbb{C}$ (where $\mathbb{C}$ are all the unary atomic terms) such that:

$\Sigma \not\models Q^{F(x)} \sqcap Y \sqsubseteq \bot$	Y is not disjoint with $Q^{F(x)}$
$\Sigma \not\models Q^{F(x)} \sqsubseteq Y$	Y is not among ancestors of $Q^{F(x)}$
$\Sigma \not\models Y \sqsubseteq Q^{F(x)}$	Y is not among descendants of $Q^{F(x)}$

The reasoning service makes use of satisfiability to check which predicates in the ontology are compatible with the current focused query. This check corresponds simply to the addition of the term Y to the focused query $Q^{F(x)}$, and verify that the resulting query is satisfiable. Actually, this operation is very expensive because the number of reasoner calls matches the number of unary predicates.

Going back to the example of Sec. 2.4, if we have a query with concept Red_Wine and we want to find all the concepts which are compatible with it, it will turn out that concept Table_Wine is compatible with the query while concept White_Wine is incompatible since it is disjoint with Red_Wine.

4.3 Addition of a property

The addition of a property requires the discovery of both a binary term and its restriction (or range). The system should check all the different binary predicates from the ontology for their compatibility. Formally, a property P is compatible with a focused query $Q^{F(x)}$ if

$$\Sigma \not\models Q^{F(x)} \sqcap \exists P.\top \sqsubseteq \bot,$$

where $\top$ represents any possible concept of the domain.

This is practically performed by verifying the satisfiability of the query $Q^{F(x)} \sqcap \exists P.\top$, for all atomic binary predicates P in the signature. Once a binary predicate is found to be compatible with $Q^{F(x)}$, repeated satisfiability is used to select the least generic unary predicate $Y \in \mathbb{C}$ such that the query $Q^{F(x)} \sqcap \exists P.Y$ is satisfiable. In other terms, the operation would consist in determining which are the compatible properties first, and then establishing which are the restrictions applicable to P. To discover all compatible properties, we need a number of reasoner calls equal to the number of properties in Σ. In addition, for each property found, to determine its restriction, we need as many reasoner calls as the number of unary predicates.

Again, returning to the example (see Sec. 2.4), this time we want to discover the properties which are compatible with the query Wine_Descriptor. As compatible properties propagate upwards in the hierarchy, property colour_of would be among the compatible properties of Wine_Descriptor. If the user instead composes a query with the concept Wine_Taste, the property colour_of would

be incompatible because the concept Wine_Taste is disjoint with the concept Wine_Colour.

4.4 Substitution of a term

Here we want to substitute a focused term of the query with an equivalent, more specific or more general term. Let us examine the substitution with a more specific term. In this case we need to perform a containment test of two conjunctive queries. Given a query focussed on concept F ($Q^{F(x)}$), we are interested in the unary terms Y subsumed by $Q^{F(x)}$, where Y must be the most general concept among the terms found (i.e. there is no other concept Y subsumed by $Q^{F(x)}$ and containing Y). Formally, given an ontology Σ and a query $Q^{F(x)}$, we want to find all the terms $Y \in \mathbb{C}$ such that:

$$\Sigma \models Y \sqsubseteq F, \neg\exists Z \in \mathbb{C} \,|\, (Z \sqsubseteq F, Y \sqsubseteq Z, Z \neq Y).$$
$$\Sigma \not\models F \sqcap Q^{F(x)} \sqcap Y \sqsubseteq \bot.$$

From Figure 1 it is possible to see that if the query is composed by concept Wine and we want to substitute it with a more specific term, we would get Red_Wine, White_Wine, and Table_Wine as candidates for the substitution since they are direct children of concept Wine.

The cases of substitution with more general and equivalent terms are analogous.

For the sake of clarity we report the sequence of operations needed to retrieve the substituting terms:

- query rolling-up;
- retrieval of incompatible classes: the descendants of negation of the query;
- retrieval of parents and children of the substituting term;
- filtering using incompatible terms.

We will see in Section 5.1 that a similar procedure is adopted to reduce the calls to the reasoner when looking for compatible terms.

5 Optimisation

As discussed in Section 2.3, the system relies on a DL reasoner to drive the query interface. If on one hand reasoning services with satisfiability and classification allow only to formulate consistent queries, on the other hand they introduce performance issues. Reasoner calls are expensive and should be therefore minimised as much as possible. The expensiveness of reasoner calls depends both on complexity and the fact that DL reasoners exploit the HTTP protocol to communicate.

In the following we present some optimisation techniques which can improve the usability of the system via a more responsive interface. Aim of the optimisation is to reduce the transitions between the query interface and the reasoner. Some techniques have already been exploited to reduce the number of reasoner

calls especially in the retrieval of compatible terms to be added to the query. Another important improvement comes from the storage of the taxonomy. Finally, information concerning the query can be cached during the query formulation process in order to extract some deductions to reduce reasoner calls.

5.1 Reducing reasoner calls

Concerning the refinement of the query by compatible terms, the basic policy to retrieve the compatible terms is to use the satisfiability reasoning service to check which unary predicates in the ontology are compatible with the current query. This check corresponds simply to the addition of the term to the current query, and to verify that the resulting query is satisfiable. Actually, this kind of operation is very expensive because the number of reasoner calls corresponds to the number of unary predicates in the ontology.

We adopted a different implementation in the current system. We classify the query and retrieve the its equivalents, ancestors, and descendants; then we classify the negation of the query and retrieve the descendants which are incompatible. The remaining unary predicates are the compatibles (see Section 4.2).

In reference to the addition of a property, as we discussed in Section 4.3, this operation requires the discovery of both a binary term and its restriction. One of the advantages of OWL-DL is the possibility of expressing the inverse of a role which is extremely useful for determining compatibility of binary terms. Hence, to discover the restriction of a property we use classification instead of repeated (and expensive) satisfiability. The idea is to classify the inverse of the property restricted to the query.
For example, to discover the restriction of property has_Colour applied to the query expression

$$\{x_1 \mid \text{Red\_Wine}(x_1), \text{Table\_Wine}(x_1)\},$$

we classify the expression $\exists \text{has\_Colour}^{-}(\text{Red\_Wine} \sqcap \text{Table\_Wine})$.
The reasoner returns the list of concept names more general and equivalent as range candidates of the relation has_Colour, when restricted to the domain (Red_Wine⊓Table_Wine). This method, not only lets us discover the least general predicate(s) which can be applied to the property in the given context, but also allows us to discard those properties which are incompatible with the query, i.e. bottom ($\bot$) is returned as range whenever a given property is incompatible with the query. Summarizing, we are able to both check the compatibility of a property with the query and find out the property's range by means of one single reasoner call.

5.2 Taxonomy storage

The taxonomy of the ontology provides static information concerning primitive concepts. If we store the taxonomy before starting to compose a query, initial operations like substituting a concept, would not involve the reasoner, thus improving efficiency.

The taxonomy is actually a partial order '$<$' from *Top* ($\top$), the whole domain, to *Bottom* ($\bot$), the empty set, where the partial order relation is subsumption. The partial order can be represented by a *directed acyclic graph (DAG)*, i.e. a directed graph that contains no cycles. An edge is drawn from a to b whenever $a < b$. A partial order satisfies the following properties:

- transitivity, $a < b$ and $b < c$ implies $a < c$;
- non-reflexive, not$(a < a)$.

These condition prevent cycles because $a < b < ... < z < a$ would imply that $a < a$, which is false. The only exception where the property $a < a$ holds is for the equivalent concepts.

The idea is to save not only the taxonomy but also other information pertaining each concept such as e.g. its incompatible classes and the list of incompatible properties.

5.3 Caching query information

The *focus* plays an important role during the query formulation process; in particular the system proposes the available operations on the query w.r.t. the current focus (i.e. the variable which is currently selected). The focus is crucial also for caching dynamic information concerning the query and the idea is to cache both the query and its actual classification at the focus level. In other words, we want to associate to each single variable which gets the focus the overall status of the query. Of course, cache at the single node level would be invalidated as soon as the user further refines the query.

An intuitive approach to exploit the cache would consist in modifying the system in a way that the user can only remove terms by following the exact inverse order of the one which has been used to formulate the query. This means that only the last operation can be undone. In this way we could reduce or even avoid reasoner calls because the information we need has already been cached at the node.

We know that refinement of the query is monotonic and therefore whenever the user adds new terms to the query, the domain is going to be restricted. This property can also lead us to some conclusions for reducing reasoner calls in further refinements of the query. This property does not hold when the user deletes a term from the query; in this case all the cache has to be removed.

6 Discussion

Optimising the communication and the quantity of exchanged messages behind the scenes between the Query Tool and the reasoner is only the first action we are taking to make the use of the interface more comfortable for the user.

The interaction time we are able to save with these enhancements is *partially* re-invested in the demands of a new and more complex interface we are building, based on state-of-the-art natural language generation (NLG) technologies.

The main challenge is that the query (now with partial verbalisation of single concepts and roles) is to be presented to the user in natural language with full verbalisation, and stepwise refinements of the query composed by the user are presented as natural language refinements that maintain the grammaticality of the sentences representing the query. Our solution adopts the paradigm called WYSIWYM ('What You See Is What You Meant'), a user-interface technique which uses (NLG) technology to provide feedback for user interaction [9]. The differences between our approach and that used by available systems employing WYSIWYM are explained in [5], while [3] reports our experiments in terms of discourse planning strategies of a complex concept description (query).

References

1. Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG Description Logic Interface. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003)*, volume 81 of *CEUR Workshop Proceedings*, 2003.
2. Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Ontology-based database access. In *Proc. of the 15th Italian Conf. on Database Systems (SEBD 2007)*, 2007.
3. Paolo Dongilli. Discourse planning strategies for complex concept descriptions. In *Proceedings of the 7th International Symposium on Natural Language Processing (SNLP-2007)*, Pattaya, Chonburi, Thailand, December 2007.
4. Paolo Dongilli, Enrico Franconi, and Sergio Tessaris. Semantics driven support for query formulation. In *Description Logics*, 2004.
5. Paolo Dongilli, Sergio Tessaris, and John Bateman. Leveraging Systemic-Functional Linguistics to Enhance Intelligent Database Querying. In *Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications*, Jinan, China, October 2006.
6. Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.
7. Ian Horrocks, Ulrike Sattler, Sergio Tessaris, and Stephan Tobies. How to decide query containment under constraints using a description logic. In *Logic for Programming and Automated Reasoning (LPAR 2000)*, volume 1955 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2000.
8. Ian Horrocks and Sergio Tessaris. Querying the semantic web: a formal approach. In Ian Horrocks and James Hendler, editors, *Proc. of the 2002 International Semantic Web Conference (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
9. Richard Power and Donia Scott. Multilingual Authoring Using Feedback Texts. In *Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics (COLING-ACL 98)*, pages 1053–1059, Morristown, NJ, USA, 1998. Association for Computational Linguistics.
10. Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL Query for OWL-DL. In *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007.
11. SWT. The Standard Widget Toolkit. `http://www.eclipse.org/swt`, 2007.