

# Semantic Discovery of the User Intended Query in a Selectable Target Query Language\*

Carlos Bobed, Raquel Trillo, Eduardo Mena, Jordi Bernad  
IIS Department, University of Zaragoza  
María de Luna, 50018 Zaragoza, Spain  
{cbobed, raqueltl, emena, jbernad}@unizar.es

## Abstract

*The syntactic approach of most of web search engines still has the drawback of not considering the semantics of the keywords entered by the user. So, users usually have to browse many hits looking for the information they want.*

*In this paper, we present a system that, given a set of keywords with well defined semantics, automatically generates a set of formal queries, in the query language of the user's choice, which attempt to capture what the user had in mind when she or he wrote those keywords. The system uses ontologies and a Description Logics reasoner to perform a semantic enrichment of user keywords to improve the discovering of possible user queries and to reject semantically inconsistent queries.*

**Keywords:** Semantic Web, ontology-based query generation, Description Logics reasoners.

## 1. Introduction

The huge amount of information available on the Web has made users dependent of web search engines. These are usually fed with a set of keywords that are a simplification of the user's information need and, most of them, do not consider the semantics of the user keywords. So, users tend to browse many hits looking for the information they want.

Thus, to answer what the user is looking for, it would be useful to know the meaning intended with each input keyword. However, knowing the intended meaning of each keyword is not like knowing the intended user query. For example, given the meaning of the keywords *vehicle* and *house*, the user could be asking for either recreation vehicles or houses with a parking lot. So, it would be interesting to find queries that represent the possible semantics intended by the user and express them in a formal query language.

---

\*This work has been supported by the CICYT project TIN2007-68091-C02-02. We would like to thank Sergio Ilarri for his valuable comments.

In this paper, we present a system that translates a set of keywords with well defined semantics (they are mapped to ontology concepts, roles or instances), into a set of formal queries (in the query language of the user's choice) which attempt to capture what the user had in mind when she or he wrote those keywords. The expressivity of the chosen output query language determines which queries are possible. Besides, the system uses a Description Logics (DL) [2] reasoner a) to semantically enrich queries with new ontology terms that relate user keywords, b) to filter generated queries that are inconsistent according to the provided ontology, and c) to avoid generating equivalent queries.

The rest of the paper is as follows. In Section 2, we give an overview of the architecture of the system. The technique used to deal with different output query languages is presented in Section 3. The main steps of the query generation process are detailed in Section 4. In Section 5, the inconsistent query filtering method is presented. In Section 6, we explain the semantic enrichment of user keywords. Related works are presented in Section 7. Finally, conclusions and future work can be found in Section 8.

## 2. Architecture of the System

Our system takes as input a list of keywords with well defined semantics (semantic keywords). Plain keywords could be chosen by the user from the set of terms of one or several ontologies [3], or could be mapped automatically to ontology terms using the techniques described in [8]. So, each keyword has the semantics of the corresponding ontology term, which can be a concept, a role, or an instance. Most of the query generation process is done using not the specific keywords but their type, which improves the performance as we will see in the following.

Along this section we overview the main steps to transform a list of semantic keywords into a list of possible user queries (see Figure 1):

- *Permutations of keyword types:* To discover all the pos-

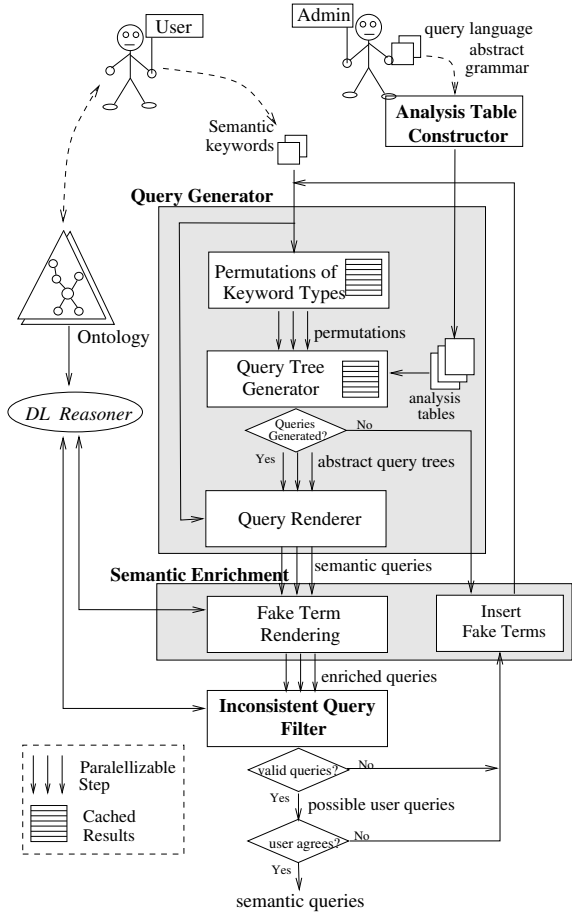


Figure 1. From keywords to formal queries

sible queries, the system obtains all the permutations of the type of terms corresponding to input keywords. The results of this step are cached to optimize future iterations. More details can be found in Section 4.1.

- *Generation of abstract query trees:* For each permutation, the system generates all the possible abstract queries according to the syntax of the selected output query language. These abstract queries are represented as abstract query trees (AQTs from now on), where nodes are operators and leafs are typed gaps (concept, role or instance gaps), see Section 4.2 for more details.

To generate AQTs, the system consults the analysis tables corresponding to the abstract syntax of the selected output language. These tables are generated using a special annotated grammar *only when a new output query language is made available to the system*. See Section 3 for more details.

- *Query rendering:* For each AQT generated, gaps in leafs are filled with the user keywords matching the

gap type. After being filled, query trees are rendered into a full query in the selected language.

- *Semantic enrichment:* The system is able to consider new terms when no query is generated or satisfies the user. This is done by adding new fake terms to the original list of user keywords (“Insert Fake Terms” step) and later by rendering new fake terms with extra ontology terms (“Fake Term Rendering” step). This process is performed using a DL reasoner (see Section 6).
- *Inconsistent query filtering:* After being rendered, the system filters the redundant and inconsistent queries using a DL reasoner compatible with the chosen output query language as explained in Section 5.

### 3. Analysis Table Construction

This step builds the analysis tables [1] that are used by the *Query Generator* to build all the possible syntactically correct AQTs corresponding to the input user keywords, according to the selected output query language.

The construction of these tables is performed only once for each new output query language. The context-free grammar  $G$  of the new query language must be transformed into an abstract context-free grammar  $\mathcal{G} = \{\mathcal{S}, \mathcal{N}, \mathcal{T}, \mathcal{P}\}$ , where the “syntactic sugar” has been removed:  $S$  is the starting symbol<sup>1</sup>;  $N$  contains a nonterminal for each operator of the language, including the three term types;  $T$  includes three terminals  $C$ ,  $R$ , and  $I$  (corresponding to concept, role, and instance gaps); and  $P$  contains one production (rule) for each nonterminal: a) for operators, the rule produces the ordered list of the term types of its operands, b) for a term type, it produces the operators of that type. In Figure 2, we show the transformation of a subset of the BACK<sup>2</sup> language.

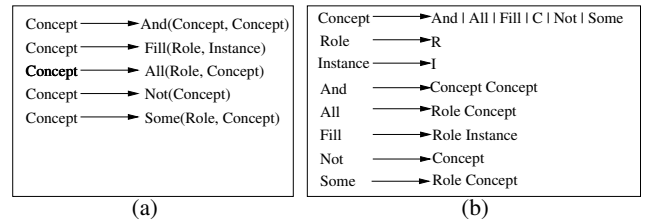


Figure 2. (a) Sample BACK grammar  $G$  and (b) the resulting abstract grammar  $\mathcal{G}$

The previous process is performed by the system administrator (see Figure 1) who specifies the abstract grammar  $\mathcal{G}$  adding more information about the operators of the new

<sup>1</sup>In our prototype, it is “Concept”.

<sup>2</sup>Due to space limitations, we use BACK [6] in examples as it almost lacks “syntax sugar”.

query language. Every operator is annotated with three attributes to help the system to avoid generating semantically equivalent queries: *sym*, which says whether the operator is commutative; *assoc*, which says whether the operator is associative; and *invol*, which says whether a unary operator is involutive. Moreover, there is an attribute (*writeTemplate*) which is used to translate the AQTs generated into specific syntactically correct formal queries.

Finally, the module *Analysis Table Constructor* (see Figure 1) takes this abstract grammar as input and, using bottom-up parsing techniques [1], generates the analysis tables needed during the query generation process.

## 4. Query Generation

In this section, we present the steps of the automatic generation of possible queries for the input semantic keywords.

### 4.1. Permutation of Keyword Types

The first step is to calculate all the permutations of the types of the semantic keywords for two main reasons: 1) to focus on the kind of knowledge represented by the user keywords, and 2) to be independent of the order in which the user entered them.

For the set of semantic keywords  $K = \{k_i\}$ , we define  $KT = \{t_i\}$ , where  $t_i$  is the type of  $k_i$  ( $C$  for a concept,  $R$  for a role, and  $I$  for an instance). This step calculates all the permutations  $KP = P_{|KT|}$  of the set  $KT$  associated to the list of input semantic keywords. For example, given “person drives bus” mapped to the homonym terms in ontology *People+Pets*<sup>3</sup> (their types are  $C$ ,  $R$ , and  $C$ , respectively), then  $KP = \{\langle R C C \rangle, \langle C R C \rangle, \langle C C R \rangle\}$ .

### 4.2. Query Tree Generator

For each permutation  $p \in KP$ , and consulting the analysis tables of the selected query language (see Section 3), the system builds the corresponding well formed AQTs. These are built using bottom-up parsing techniques [1]. Let us denote by  $T_h^p$ , the set of AQTs with maximum height  $h$  where the terminals  $C$ ,  $R$ ,  $I$  appear in preorder in the AQT in the same order as in permutation  $p$ ;  $h$  limits the complexity of the generated queries. In Figure 3, we show two different AQTs for permutation  $\langle R C C \rangle$ , generated from analysis tables of abstract grammar in Figure 2.b. The system considers the annotated properties of operators to avoid generating semantically equivalent trees, and executes concurrently for each permutation. Moreover, dealing with AQTs performs better than using the specific keywords entered by the user directly because the subtrees generated can be cached and shared across concurrent threads.

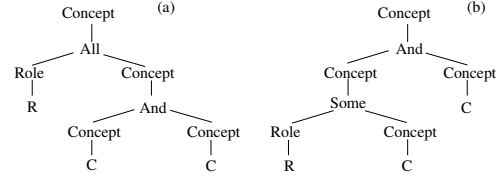


Figure 3. Sample abstract query trees for permutation  $\langle R C C \rangle$  in BACK

## 4.3. Query Renderer

During this step, concurrently for each AQT  $t \in T_h^p$ , the system creates a set of query trees  $Q^t$ ; each  $q \in Q^t$  is the result of using each user keyword  $k$  to replace a gap  $g$  in  $t$ , where  $g \in \{C, R, I\}$  and the type of  $k$  is  $g$ . During this substitution process, symmetry and associative properties of operators are taken into account to avoid generating query trees that are semantically equivalent. Finally, the system visits all the query trees and renders them into formal queries expressed in the selected output language, by consulting the *writeTemplate* attribute (see Section 3). For “person drives bus”, the AQT in Figure 3.b, is rendered into “And(Some(drives,bus),person)” and “And(Some(drives,person),bus)”.

## 5. Semantic Filtering of Inconsistent Queries

Despite being syntactically correct, some of the queries generated during previous steps may be inconsistent according to the ontology from which the user keywords were chosen (in previous examples, ontology *People+Pets*). The generation of these inconsistent queries cannot be avoided in earlier stages because the system cannot know whether an AQT will lead to an inconsistent query until it is rendered. As example, the generated query “And(Some(drives,person), bus)”, which literally means “retrieve buses that drive persons”, makes no sense.

To filter inconsistent queries we take advantage of the capability of DL [2] reasoners to classify expressions according to a given ontology. Thus, the system assigns a unique id to each query (e.g. “ $q_2 = \text{And}(\text{Some}(\text{drives,person}), \text{bus})$ ”) and asserts them into the knowledge base. Once all the queries have been asserted, the reasoner classifies them *at the same time* and queries classified as equivalent to the bottom concept (*nothing* in some systems, which cannot have instances) are removed (e.g.,  $q_2$  is inconsistent as  $\text{bus} \not\sqsubseteq \text{domain}(\text{drives})$  and also  $\text{person} \not\sqsubseteq \text{range}(\text{drives})$ ).

At the same time, the queries classified as semantically equivalent are grouped. Moreover, the reasoner could even discover that a certain query is equivalent to an ontology term; for example, it discovers that “And(Some(drives,bus),

<sup>3</sup>www.cs.man.ac.uk/horrocks/ISWC2003/Tutorial/people+pets.owl.rdf

person)” is equivalent to term “bus\_driver” in ontology People+Pets. Note that using a DL reasoner to remove inconsistent and detect equivalent queries with just one classification step is an important performance improvement.

## 6. Semantic Enrichment

Due to the lack of expressivity of the chosen output query language, it could happen that no tree is generated for some specific keywords. Also, it could happen that all the queries are classified as semantically inconsistent. Besides, when some consistent queries are shown to the user, s/he could say that her or his intended query is not among them. For example, with an input “Person Fish” (mapped to the homonym concepts in ontology “Animals”<sup>4</sup>) looking for people devoured by fishes; after query generation, using BACK as output query language, only one query is found, “Person and Fish” but it is classified as inconsistent (Person and Fish are disjoint in ontology “Animals”).

In such situations, our system enriches the user’s input with some new ontology terms to find a semantic link among user keywords and so generate new possible queries. This enrichment consists of two steps (see Figure 1):

1. *Insert fake terms*: The system adds “FakeConcept” and “FakeRole” keywords to the set of user keywords, which allows inserting a new concept or role in queries, respectively. Both possibilities are explored in parallel by the system. In our example, the system generates new query trees for keywords “Person Fish FakeConcept” and “Person Fish FakeRole”; one of the queries built is “And(Person,Some(FakeRole,Fish))”.
2. *Fake term rendering*: The system must replace the fake term by a compatible ontology term. The system uses a DL reasoner to retrieve the semantically compatible terms that are candidates to replace it. In our example, one of the possibilities found is to replace “FakeRole” by role “eaten\_by” which leads to the query “And(Person,Some(eaten\_by,Fish))”, the intended one.

## 7. Related Work

In SemSearch [5], a set of predefined templates are used to define formal queries in language SeRQL. Not all the possible queries in such language are considered in those templates. Besides, SemSearch requires that at least one of the keywords entered by the user matches an ontology concept. On the contrary, our system considers all the possible queries syntactically correct defined in a query language selectable by the user and it works with any list of keywords.

<sup>4</sup>[www.cs.man.ac.uk/feetor/tutorials/Biomedical-Tutorial/Tutorial-Ontologies/Animals/Animals-tutorial-complete.owl](http://www.cs.man.ac.uk/feetor/tutorials/Biomedical-Tutorial/Tutorial-Ontologies/Animals/Animals-tutorial-complete.owl)

The system presented in [7] and Q2Semantic [4] finds all the paths that can be derived from a RDF graph, until a predefined depth, to generate the queries. However, they do not support reasoning capabilities for query optimization as opposite to our system, which discards semantically inconsistent queries and detects equivalent ones.

## 8. Conclusions and Future Work

In this paper, we have presented an approach that, taking as input a list of keywords, automatically generates a list of queries that unambiguously represent the user’s information need. These queries are syntactically correct according to the selected output query language and semantically correct according to the provided ontology. The main features of our system are that: 1) It uses a DL reasoner to infer new information, group semantically equivalent queries, and remove semantically inconsistent queries, 2) it is independent of the specific output query language, and 3) it semantically enriches the generated queries by adding ontology terms.

Our system can be used for different proposes, such as semantic information retrieval, or any other task that needs to find out the user query behind a set of semantic keywords.

As future work, we will study how to rank the generated queries and test such a ranking against real users.

## References

- [1] A. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.
- [2] F. Baader et al. *The Description Logic Handbook. Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [3] T. R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation, The Netherlands*. Kluwer Academic Publishers, 1993.
- [4] Q. L. Haofen Wang et al. Q2semantic: A lightweight keyword interface to semantic search. *5th European Semantic Web Conference, Spain*, pages 584–598. LNCS, June 2008.
- [5] Y. Lei et al. Semsearch: A search engine for the semantic web. *15th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks, Czech Republic*, pages 238–245. LNCS, October 2006.
- [6] C. Peltason. The BACK system – an overview. *ACM SIGART Bulletin*, 2(3):114–119, June 1991.
- [7] D. T. Tran et al. Ontology-based interpretation of keywords for semantic search. *6th International Semantic Web Conference, Korea*, pages 523–536. LNCS, November 2007.
- [8] R. Trillo et al. Discovering the semantics of user keywords. *Journal on Universal Computer Science. Special Issue: Ontologies and their Applications*, November 2007.