An intelligent query interface based on ontology navigation

Enrico Franconi
franconi@inf.unibz.it

Paolo Guagliardo
paolo.guagliardo@stud-inf.unibz.it

Marco Trevisan
marco.trevisan@stud-inf.unibz.it

KRDB Research Centre
Free University of Bozen-Bolzano

ABSTRACT

In this paper we present a formal framework and an experimental software supporting the user in the task of formulating a precise query – which best captures their information needs – even in the case of complete ignorance of the vocabulary of the underlying information system holding the data. Our intelligent interface is driven by means of appropriate automated reasoning techniques over an ontology describing the domain of the data in the information system.

We will define what a query is and how it is internally represented, which operations are available to the user in order to modify the query and how contextual feedback is provided about it presenting only relevant pieces of information. We will then describe the elements that constitute the query interface available to the user, providing visual access to the underlying reasoning services and operations for query manipulation. Lastly, we will define a suitable representation in "linear form", starting from which the query can be more easily expressed in natural language.

Author Keywords

query formulation support automated reasoning description logics ontology visual graphical intelligent user interface

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces—*Graphical user interfaces (GUI)—Natural language —Interaction styles*

INTRODUCTION

Recent research showed that adopting formal ontologies as a means for accessing heterogeneous data sources has many benefits, in that not only does it provide a uniform and flexible approach to integrating and describing such sources, but it can also support the final user in querying them, thus improving the usability of the integrated system.

We introduce a framework that enables access to heterogeneous data sources by means of a conceptual schema and supports the users in the task of formulating a precise query over it. In describing a specific domain, the ontology defines a vocabulary which is often richer than the logical schema of the underlying data and usually closer to the user's own vocabulary. The ontology can thus be effectively exploited by the user in order to formulate a query that best captures their information need. The user is constantly guided and assisted in this task by an intuitive visual interface, whose intelligence is dynamically driven by reasoning over the ontology. The inferences drawn on the conceptual schema help the user in choosing what is more appropriate with respect to their information need, restricting the possible choices to only those parts of the ontology which are relevant and meaningful in a given context.

The most powerful and innovative feature of our framework lies in the fact that not only do not users need to be aware of the underlying organisation of the data, but they are also not required to have any specific knowledge of the vocabulary used in the ontology. In fact, such knowledge can be gradually acquired by using the tool itself, gaining confidence with both the vocabulary and the ontology. Users may also decide to just explore the ontology without actually querying the information system, with the aim of discovering general information about the modelled domain.

Another important aspect is that only queries that are logically consistent with the context and the constraints imposed by the ontology can be formulated, since contradictory or redundant pieces of information are not presented to the user at all. This makes user's choices clearer and simpler, by ruling out irrelevant information that might be distracting and even generate confusion. Furthermore, it also eliminates the often frustrating and time-consuming process of finding the right combination of parts that together constitute a meaningful query. For this reason, the user is free to explore the ontology without the worry of making a "wrong" choice at some point and can thus concentrate on expressing their information need at best.

Queries can be specified through a refinement process consisting in the iteration of few basic operations: the user first specifies an initial request starting with generic terms, then refines or deletes some of the previously added terms or introduces new ones, and iterates the process until the resulting query satisfies their information need. The available operations on the current query include addition, substitution and

Workshop on Visual Interfaces to the Social and Semantic Web (VISSW2010), IUI2010, Feb 7, 2010, Hong Kong, China.
Copyright is held by the author/owner(s).

deletion of pieces of information, and all of them are supported by the reasoning services running over the ontology.

The framework is implemented in the form of an experimental software, called Query Tool, which relies on a web-based client-server architecture consisting of three components:

- the query logic (QueLo), responsible of "reasoning" over the ontology in order to provide only relevant information w.r.t. the current query;
- the natural language generation (NLG) engine, that given a query and a lexicalization map for the ontology produces an English sentence;
- the user interface (UI), that provides visual access to the query and editing facilities for it, allowing to interact with the QueLo sub-system while benefiting from the services of the NLG engine.

OVERVIEW

In the paper we will introduce the Query Tool's formal framework and functional API, that constitute the basis on top of which different types of user interfaces (UIs) can be devised. Previous implementations of the Query Tool ([1, 15]) experimented mostly with tree-based and menu-based UIs. In this section we first describe the behaviour of the Query Tool using a generic representation over an abstract user interface. Later on in the paper we will then introduce a concrete interface based on the generation of natural language.

Consider a scenario in which we have a conceptual schema, say an OWL ontology, we know nothing about. In such situation, the Query Tool reveals to be particularly useful in that it allows to discover information about the ontology and the modelled domain, even when its vocabulary is completely ignored. What we call *intensional navigation* of the ontology is the process of building a query, starting from a very general request which is then refined by adding or deleting constraints according to the user's information need. In our abstract representation, the default initial query looks like the one shown in Figure 1 that generically asks for some "thing". Four operations are available for manipulating the query: *add* for the addition of new terms and relations; *substitute* for replacing a portion of the query with a more general, equivalent or more specific term; *delete* for discarding parts of the query; and *weaken* for making a portion of the query as general as possible.

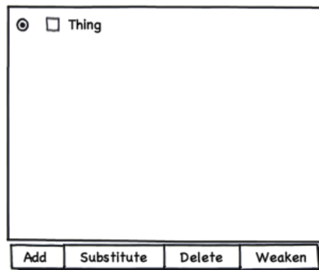


Figure 1: Initial query

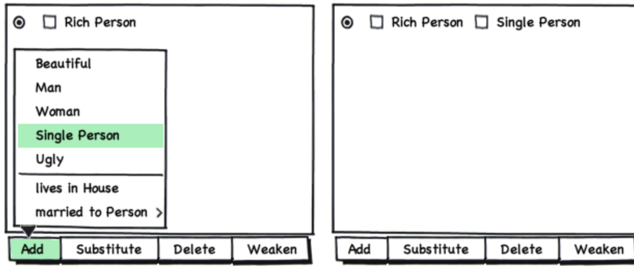
The first step in the refinement of our query consists in being more specific about what we are looking for. This can be achieved by selecting something within the query and asking for a substitution. In our example, we tick the check-box associated with the term **Thing** and then press the **Substitute** button. As shown in Figure 2, we are presented with a three-part menu listing all the possible substitutions available for the selected portion of the query: terms that appear at the top are more general than the selection, the ones in the middle are equivalent, while those at the bottom are more specific. Moreover, these terms are organised in sub-menus according to the taxonomic information defined in the ontology. Thus, we can easily navigate the different options choosing the desired level of detail for the substitution. In our case, instead of **Person** we choose the further specific term **Rich Person** as a replacement for the selection, resulting in the query visible on the right side of Figure 2.

Figure 2: Example of specialisation

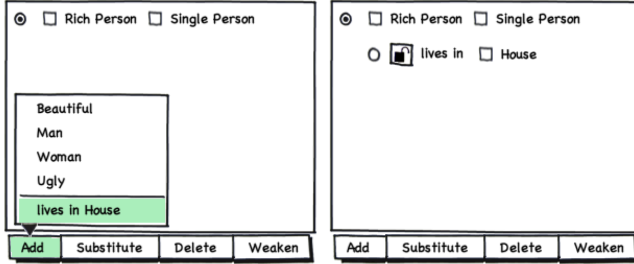
Another way of modifying a query is to add constraints in the form of new terms or relations. As shown in Figure 3, upon clicking on the add button a two-part menu is displayed, containing suitable terms and relations that can be safely added to the query. Terms are shown in the upper part of the menu, while relations in the lower one. The chosen item is inserted in a specific place of the query, that can be selected by means of the radio button present in each line. Figure 3a shows how the new term **Single Person** is added to the first (and only) line of the query, while Figure 3b shows how adding a relation results in the creation of a new line, indented w.r.t. the first one and consisting of the name of the relation **lives in** followed by the label **House** associated with its range. Observe that the menu of Figure 3b, compared to that of Figure 3a, does not include the term **Single Person** and the relation **married to Person** as possible options. In fact, the former is already present in the query, thus it would be redundant to propose it again; the latter became incompatible with the query due to the addition of the previous term, and this means that in our ontology a person who is rich and single (or perhaps just single) cannot be married to anyone.

A query can be made more general or "weaker" in a variety of ways, one of which is the substitution with a more general term. Other possibilities are given by deletion and weakening, both of which remove selected elements from the query but with distinct approaches and outcomes. The difference between them is shown in Figure 4: while in Figure 4a deleting the selected portion causes the second row of the query to disappear, in Figure 4b weakening the same portion preserves the row, although the term **House** is replaced with the generic term **Thing**.

Suppose that our ontology states that a rich man who is married to a beautiful woman and lives in a beautiful house is in-

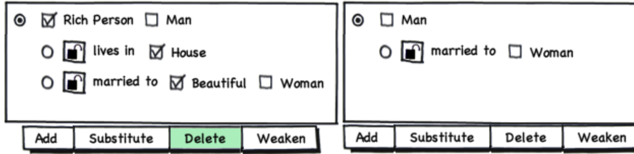


(a)

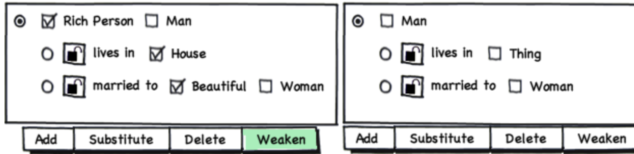


(b)

Figure 3: Addition of (a) a new term and (b) a new relation



(a)



(b)

Figure 4: Example of (a) deletion and (b) weakening

deed a lucky person. As a result of the substitution shown in Figure 5, where the whole query is replaced with the (more general) term **Lucky Person**, the second line disappears. In some situations this "side-effect" is undesired, because we would perhaps like to operate on that part of the query later on. The closed padlock icon visible in the third line of the query indicates that that line is protected against such an "accidental" deletion and would not inadvertently disappear as the result of the substitution. However, note that locked portions of the query are still fully affected by explicit deletion.

THE QUERY LOGIC

The Query Tool's framework and its functional API (QueLo) have been recently defined from a formal point of view in the technical report [7], with the purpose of precisely describing its components and the abstract operations provided for query manipulation. Here, we concisely summarise the main definitions introduced in [7] and formally prove the most important of Query Tool's properties, namely that it generates only "meaningful" queries.

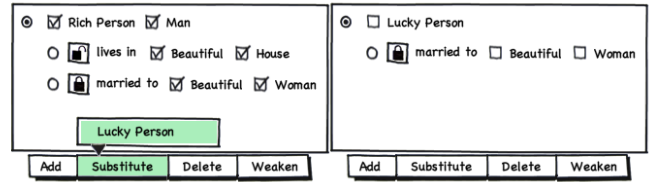


Figure 5: Preventing side-effects of substitution

Formal framework

From the point of view of the QueLo sub-system, a query is a labelled tree where each node corresponds to a variable and it is associated with a set of concept names from the ontology, while each edge is labelled with a role name. Therefore, for what concerns the expressive power, the Query Tool can represent tree-shaped conjunctive queries only.

Let **N** be a countably infinite set of node names, **C** be a finite set of concept names and **R** a finite set of role names, and let **N**, **C** and **R** be pairwise disjoint. A *query* Q is a quintuple $\langle V, E, o, \mathcal{V}, \mathcal{E}\rangle$ in which (V, E) is a directed tree rooted in $o \in V$, with set of nodes $V \subseteq \mathbf{N}$ and with set of edges $E \subseteq V \times V$; $\mathcal{V}$ is a total function, called *node-labelling function*, associating each node with either a non-empty set of concept names or with the singleton $\{\top\}$; and $\mathcal{E}$ is called the *edge-labelling function* that associates each edge with a role name. A query consisting of exactly one node, whose set of labels is a singleton, is called *atomic*. For an edge $e = \langle x, y\rangle$, we indicate its initial node x with $\mathsf{init}(e)$ and its terminal node with $\mathsf{ter}(e)$.

Given queries S and Q, we say that S is a *subquery* of Q, and write $S \subseteq Q$, iff $V(S) \subseteq V(Q)$, $E(S) \subseteq E(Q)$, each node $n \in V(S)$ is s.t. $\mathcal{V}_S(n) \subseteq \mathcal{V}_Q(n)$ and every edge $e \in E(S)$ is such that $\mathcal{E}_S(e) = \mathcal{E}_Q(e)$. We say that S is a *complete subquery* of Q (in symbols $S \subseteqq Q$) if it also holds that, for every $n \in V(S)$, $\mathcal{V}_S(n) \supseteq \mathcal{V}_Q(n)$ and every descendant of o_S in Q is a node in S. A *selection* within a query Q is a subquery S of Q, which is called *simple* if $S \subseteqq Q$ or S consists of exactly one node, namely its root o_S, such that $\mathcal{V}_S(o_S)$ is a singleton or is equal to $\mathcal{V}_Q(o_S)$. Every selection S within a query Q partitions the nodes of Q into *selected*, which belong to $V(S)$, and *unselected*, belonging to $V(Q) \setminus V(S)$. The selected nodes can be further partitioned into *totally selected*, having all of their labels selected, and *partially selected*, which have some, but not all, of their labels selected. An example of query as represented is shown in Figure 6, which also shows the compact graphical notation we use for representing a selection within a query: selected nodes are drawn using a double circle and selected labels within each of them are underlined.

The *weakening* of a query Q w.r.t. a selection S within Q is the query $Q \ominus S$ obtained from Q by replacing its node-labelling function $\mathcal{V}_Q$ with a function that associates each totally selected node with $\{\top\}$, each partially selected node n with $\mathcal{V}_Q(n) \setminus \mathcal{V}_S(n)$ and each unselected node m with $\mathcal{V}_Q(m)$.

The last notion we introduce is that of *sticky edges*, which are edges that can only be deleted explicitly (that is, when performing a deletion), but never implicitly (e.g., as the con-

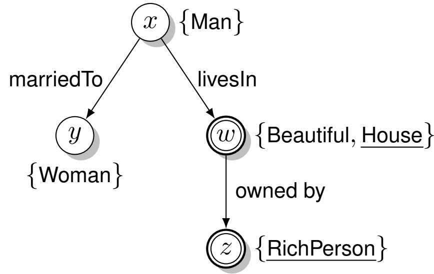


Figure 6: Example of query and a selection within it

sequence of a substitution). Sticky edges are closed w.r.t. the tree structure of the query, that is, when an edge e is sticky, then all the edges in the path from the root of the query to $\mathsf{ter}(e)$ are such. The meaning and importance of sticky edges will become more clear in the next section, where we introduce and describe the two operations delete and substitute. For the moment, sticky edges can be simply understood as immutable (to some extent) pieces of information within a query, which are not modified as a "side effect" of an operation not directly intended to do so.

Functional API

To draw the inferences that are at the basis of the query formulation tasks, we express a query as a concept of some description logic (DL) language, for which the containment test of two conjunctive queries is decidable and available as a reasoning service. In what follows, we assume the existence of an underlying knowledge base $\mathcal{K}$ in such a DL language $\mathcal{L}$ over $\mathbf{C}$ and $\mathbf{R}$. We say that C is a sub-concept of (or *subsumes*) D in $\mathcal{K}$, and write $C \sqsubseteq_{\mathcal{K}} D$, iff $\mathcal{K} \models C \sqsubseteq D$, in which case we also say that D is a super-concept of (or *is subsumed by*) C. Two concepts C and D are *equivalent* in $\mathcal{K}$, written $C \equiv_{\mathcal{K}} D$, iff one subsumes the other and vice versa. For two concept names c_1 and c_2 we say that c_1 is a *direct sub-concept* of c_2 (and that c_2 is a *direct super-concept* of c_1) iff c_1 subsumes c_2 and there is no $c \in \mathbf{C}$ equivalent to neither c_1 nor c_2 and such that $c_1 \sqsubseteq_{\mathcal{K}} c \sqsubseteq_{\mathcal{K}} c_2$.

Before introducing the functional API, let us first give some preliminary definitions. Given a query Q and $n \in V(Q)$, the operation $\mathsf{roll\text{-}up}(Q, n)$ translates Q into an $\mathcal{L}$-concept w.r.t. n and it is defined as $\mathsf{enc\text{-}rollup}(Q, n, n)$, where enc-rollup is the recursive procedure described in Algorithm 1. We use $\mathsf{roll\text{-}up}(Q)$ as an abbreviation for $\mathsf{roll\text{-}up}(Q, o)$, where o is the root of Q. The concept $\mathsf{roll\text{-}up}(Q, n)$ is called the *context* of Q w.r.t. n, expressing the informative content of Q from the point of view of a specific node, which we call the *focus*. Queries Q_1 and Q_2 are *equivalent*, in symbols $Q_1 \equiv Q_2$, iff $\mathsf{roll\text{-}up}(Q_1) \equiv_{\mathcal{K}} \mathsf{roll\text{-}up}(Q_2)$. We say that a query Q over a consistent knowledge base $\mathcal{K}$ is *satisfiable* iff its roll-up is such in $\mathcal{K}$ (that is, $\mathcal{K} \not\models \mathsf{roll\text{-}up}(Q) \sqsubseteq \bot$).

The functional API of the Query Tool is structured in two main parts:

- the underlying reasoning services, consisting of the operations getComp, getRel, getSupers, getEquiv, getSubs;
- the operations for query manipulation, including addRel, addComp, weaken, substitute and delete.

Algorithm 1 Calculate $\mathsf{enc\text{-}rollup}(Q, n, m)$

Input: a query Q and two nodes $n, m \in V(Q)$
Output: a concept C expressing Q in the description logics language $\mathcal{L}$

```
 1: C ← c, for some c ∈ 𝒱(n)
 2: for all x ∈ 𝒱(n) such that x ≠ c do
 3:     C ← C ⊓ x
 4: end for
 5: for all children x of n in Q such that x ≠ m do
 6:     R ← ℰ(⟨n, x⟩)
 7:     C ← C ⊓ ∃R . enc-rollup(Q, x, n)
 8: end for
 9: if n ≠ o then
10:     Let p be the parent node of n in Q
11:     if p ≠ m then
12:         R ← ℰ(⟨p, n⟩)
13:         C ← C ⊓ ∃R⁻ . enc-rollup(Q, p, n)
14:     end if
15: end if
16: return C
```

Given a query Q and a node n, we say that a concept name c is *compatible* with Q focused in n iff $c \sqcap \mathsf{roll\text{-}up}(Q, n) \not\sqsubseteq \bot$, while a role name r is such iff $\exists r^{-}. \mathsf{roll\text{-}up}(Q, n) \not\sqsubseteq \bot$. The operation $\mathsf{getComp}(Q, n)$ returns a directed acyclic graph (DAG) G, whose nodes are all the concept names that are compatible with Q focused in n and that are neither sub- nor super-concepts of $\mathsf{roll\text{-}up}(Q, n)$, and whose edges are all the pairs of concept names $c_1, c_2 \in V(G)$ such that c_1 is a direct sub-concept of c_2. In other words, the output of getComp is a taxonomy of concept names which are compatible with the query and not in hierarchy with the context. The operation $\mathsf{getRel}(Q, n)$ returns a DAG G, whose nodes are all the pairs $\langle r, c\rangle$ of role names and concept names such that r is compatible with Q focused in n and c is a sub- or a super-concept of $\exists r^{-}. \mathsf{roll\text{-}up}(Q, n)$, and whose edges are the pairs $\langle\langle r, c_1\rangle, \langle r, c_2\rangle\rangle \in V(G) \times V(G)$ such that c_1 is a direct super-concept of c_2.

Let S be a selection within a query Q. Then, the operations getSupers, getEquiv and getSubs return the concept names that are more general than, equivalent to and more specific than $\mathsf{roll\text{-}up}(S)$, respectively. Moreover, the concept names in the output of $\mathsf{getSubs}(Q, S)$ are additionally required to be compatible with Q focused in the root of S.

Let Q be a query and n a focus node. For a concept name c in the output of $\mathsf{getComp}(Q, n)$, the operation addComp adds c to $\mathcal{V}(n)$. More precisely, the result of $\mathsf{addComp}(Q, n, c)$ is the query Q' obtained from Q by replacing its node-labelling function $\mathcal{V}$ with $\mathcal{V}' := \mathcal{V}[n \mapsto \mathcal{V}(n) \cup \{c\}]$. For a pair $\langle r, c\rangle$ in the output of $\mathsf{getRel}(Q, n)$, the operation addRel creates a new node n' such that $\mathcal{V}(m) = \{c\}$ and an edge $e = \langle n, m\rangle$ with $\mathcal{E}(e) = r$.

Let Q and R be queries and $\widetilde{E}$ be a set of sticky edges. Then, the operation prune deletes from Q the maximal number of non-root nodes, having no incoming sticky edge (if any) and associated with the same concept names both in R and Q, such that the result is still a query.

Let S be a selection within a query Q and let $\widetilde{E}$ be a set of sticky edges. We define $\mathsf{weaken}(Q, S)$ as $Q \ominus S$ and

$$\mathsf{delete}(Q, S, \widetilde{E}) := \mathsf{prune}\big(\mathsf{weaken}(Q, S), R, \widetilde{E}\big) \ ,$$

where R is the query obtained from S by replacing $\mathcal{V}_S$ with the function on $V(S)$ associating each node n that is both in Q and S with $\mathcal{V}_Q(n) \cap \mathcal{V}_S(S)$ if such intersection is non-empty and with $\{\top\}$ otherwise, and each other node m of S with $\mathcal{V}_S(m)$. The last operation we introduce is "substitution" which, for a concept name c in the output of getSupers (*generalisation*) or getEquiv or getSubs (*specialisation*), is defined as follows:

$$\mathsf{substitute}(Q, S, \widetilde{E}, c) := \mathsf{delete}(Q', S, \widetilde{E}) \ ,$$

where Q' is the query obtained from Q by adding c to the set of concept names associated with the root of S.

Properties of the framework

We will now formally state and prove that, starting from an atomic query that is satisfiable, the query obtained by means of the operations in the Query Tool's functional API is satisfiable. In order to do that, we first prove that the operations for query manipulation preserve satisfiability, i.e., the application of each of them to a satisfiable query results in a query that is satisfiable.

LEMMA 1. *Each of the operations* addComp*,* substitute*,* addRel*,* weaken *and* delete *preserves query satisfiability.*

PROOF SKETCH. Let Q be a satisfiable query and Q' result from the application of one the above operations. In the case of deletion, weakening and substitution with an equivalent or more generic term, the resulting query Q' is equivalent or more general than the input query Q, which is satisfiable by assumption. Therefore, Q' is also satisfiable. In the case of addition of a new term/relation and substitution with a more specific term, the satisfiability of Q' is ensured by the satisfiability of Q and the explicit check for compatibility in the definitions of addComp, addRel and substitute. □

The fundamental property of the Query Tool is then proved by means of a simple induction.

THEOREM 1. *The query obtained from an initial satisfiable atomic query through a finite sequence of applications of the operations* addComp*,* addRel*,* substitute*,* weaken *and* delete *is satisfiable.*

PROOF. By an easy induction on the sequence of applications of the operations: the base case holds by assumption, while the inductive step follows directly from Lemma 1. □

THE USER INTERFACE

In this section we describe a concrete UI for the Query Tool, based on natural language generation. In the UI, the query is represented as a continuous string of natural language text, composed of a sequence of coherent text constituents called *spans*. Each of the tags occurring in the query is associated with a span by means of an injective mapping. As for each edge there is one and only one corresponding edge tag, if a span is associated with the tag of an edge we simply say that the span is associated with that edge.

The English sentence representing the query in the UI is generated by the NLG sub-system, which will be described in the next section. An example of the textual rendering of the query in natural language as displayed by the UI is given in Figure 7.

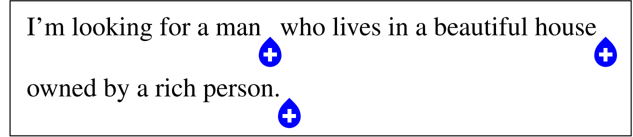


Figure 7: Textual representation of a query in natural language

Hovering

In graphical user interfaces terminology, the user *hovers* on a graphic element whenever the mouse cursor moves from some point outside the element to some point inside the element. In normal conditions, as the user hovers on the query, the system gives visual hints about its structure:

- hovering on the span associated with a tag of some node n causes the span to become *lightly highlighted*, along with all the spans associated with the tags occurring in the complete subquery rooted in n;
- hovering on the span associated with the tag of some edge e causes that span and all the spans associated with the elements of $\mathsf{tags}\big(\mathsf{ter}(e)\big)$ to become lightly highlighted.

The highlighting is such that spans associated with different tags are visualized as distinct, even when adjacent. One way of obtaining this kind of effect is, for instance, by rounding the corners of the highlighted rectangular area around each span. The only case in which highlighting on hovering does not trigger is when a menu is being displayed.

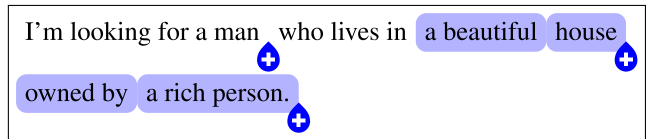


Figure 8: Hovering on the span "house"

Associated with each node of the query is a button, called the *add-button*, which is displayed as a blue onion with a white cross inside and located below the text baseline immediately after the rightmost span associated with a tag of that node. Hovering on the add-button of some node lightly highlights all the spans associated with tags of that node.

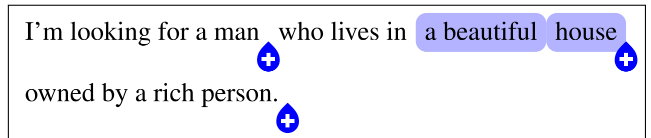


Figure 9: Hovering on the add-button at the right of "house"

Sticky edges

An edge can be marked as *sticky* by clicking on the span associated with it, at which point the text span is rendered in

boldface. Clicking on a span associated with a sticky edge unmarks the edge as sticky and reverts the text span to its default (non-bold) representation. Spans associated with node tags are not visually affected by sticky edges.

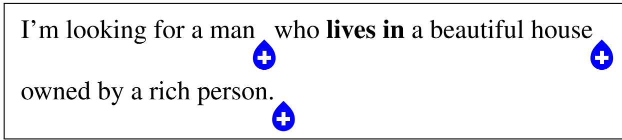


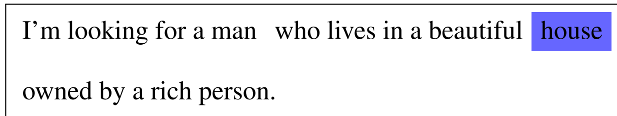
Figure 10: Clicking on "lives in" marks the associated edge as sticky

Selection

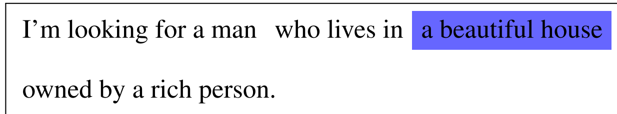
The UI provides facilities to easily select portions of the query. A simple selection can be directly specified by clicking on the span associated with a tag of some node n in one of the following ways:

- a single click results in an atomic selection, highlighting only the span on which the click occurred;
- a double click results in a node selection, highlighting all the spans associated with the elements of $\mathsf{tags}(n)$;
- a triple click results in a complete selection, highlighting all the spans in the complete subquery rooted in n.

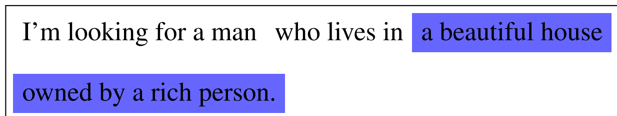
An example of each of these different types of simple selection is shown in Figure 11.



(a)



(b)



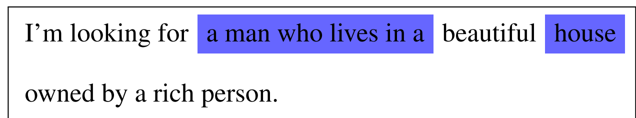
(c)

Figure 11: Simple selections obtained by (a) single, (b) double and (c) triple clicking on the span "house"

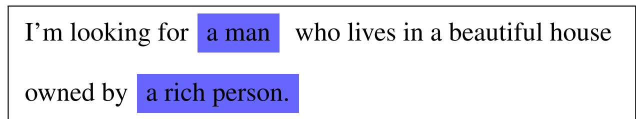
A selection can be *cleared* by clicking on an area of the UI where clicking does not have any other effect (e.g., on the white space between the lines of text representing the query, or on a span that is not associated with any tag). Clearing a selection results in an *empty selection*.

Observe that when a node has only one node label, an atomic selection on that node happens to be also a node selection, and when a node has no children, a node selection is also a complete selection. Thus, an atomic selection on a node having only one label and no children is also a node selection as well as a complete selection. We consider atomic selections to have the lowest priority and complete selections the highest, and when a simple selection belongs to more than one class, it is considered to be only of the type with higher priority. Furthermore, whenever a double click would result in the same kind of selection a single click would, it yields a complete selection instead.

A *complex selection* (non-simple) is obtained from an empty or simple selection by control-clicking (i.e., clicking while pressing the CTRL key on the keyboard) on additional spans associated with node tags, which are consequently included in the existing selection. Note that a complex selection can be *disconnected*, in the sense that it is not a well-formed subquery from the formal point of view, because there might be two selected nodes that are not connected by an edge. Examples of connected and disconnected complex selections are given in Figure 12.



(a)



(b)

Figure 12: Examples of (a) connected and (b) disconnected complex selection

From the graphic point of view, spans associated with tags in a selection are highlighted in a stronger way (e.g., a darker color) than they are when highlighted because of hovering and, unlike the highlighting effect triggered by hovering, it is not possible to distinguish between adjacent selected spans associated with the same node. When the selection includes one or more paths between nodes (that is, all of the nodes in a path within the query are selected), spans associated with edge tags are also highlighted. The visual appearance of the spans associated with tags of selected nodes or edges does not change as the result of an hovering event, as shown in Figure 13. Moreover, as the reader might already have noticed, when a non-empty selection is present, the add-buttons become invisible without changing the layout of the text.

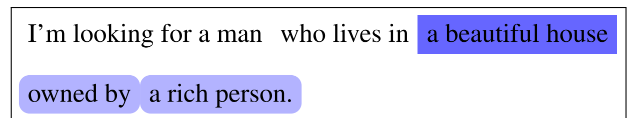


Figure 13: Hovering on either "beautiful" or "house" in presence of a non-empty selection

Addition

The query logic sub-system provides two operations, namely addComp and addRel, for refining a query through the addition of compatible terms and relations to a focus node. The UI makes these operations available to the user by means of

a pop-up menu, activated by clicking on the add-button of a node which is set as the focus.

The menu contains a list of suitable arguments for the invocation of either addComp or addRel. The menu entries are concept names and pairs consisting of a role name and a concept name, which are obtained from the output of the QueLo operations getComp and getRel w.r.t. the current query and focus. In particular, for a query Q focused in n, the menu is populated with the nodes in the graph resulting from disjoint union of the output graphs of getComp(Q, n) and getRel(Q, n), arranged in the following way:

- nodes with no incoming edge populate a menu of level 0, which is the *topmost* menu, where entries corresponding to concept names are listed before entries associated with pairs of concept/role names.
- for each node n in a menu at level k, all the nodes that are reachable from n in one step populate a sub-menu at level $k + 1$ associated with entry n.

The actual items shown to the user in the above menu structure are natural language descriptions of the node-entries (either concept names or atomic concept/role pairs) generated by the NLG sub-system.

Some of the items come with an icon on their left: an upward-pointed (resp., downward-pointed) triangle is displayed for concept names (resp., role/concept pairs) indicating that the option is associated with a nested sub-menu containing more specific (resp., more generic) options of the same type. Hovering on any of these options opens the pop-up menu associated with that item and displayed next to it. An example of the structure of the addition menu is given in Figure 14.

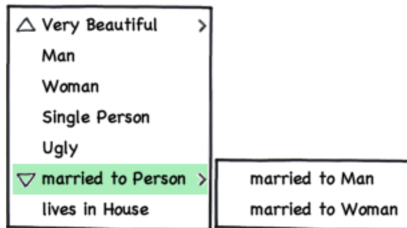


Figure 14: Structure of the addition menu

Clicking on any of the elements in the menu triggers the invocation of either addComp or addRel, according to whether the clicked item is associated with a concept name or a concept/role pair, respectively. Upon clicking, the menus disappear and the UI updates its representation of the query after the necessary changes are performed by the QueLo sub-system.

Weakening and Deletion

The user can *weaken* (respectively, *delete*) a selected portion of the query by pressing the backspace (resp., delete) key on the keyboard, which invokes the QueLo operation weaken (resp., delete) with the current query and selection as input arguments. Upon weakening (resp., deletion), the selection is cleared and the UI updates its representation to reflect the changes in the query.

Note that the operations weaken and delete, as defined in our functional API, cannot directly handle a disconnected complex selection. However, such a selection can be decomposed by the UI in a series of connected selections that are then suitable for the actual invocation of the two operations.

Observe that selected nodes can be deleted even if they have an incoming sticky edge; in other words, the result of deletion is the same independently of the presence of sticky edges in the selection. Note also that in some cases deletion produces the same result as weakening (e.g., for a node selection rooted in a non-leaf node).

Substitution

The QueLo sub-system provides the operation substitute in order to allow the substitution of a selection within the query with a more generic, equivalent or more specific term. The UI makes this operation available to the user: upon long-clicking on a selected portion (i.e., strongly highlighted) of the query a pop-up menu is displayed, listing all the possible terms with which the selection can be replaced. Such a menu is populated with concept names that are more general than, equivalent to and more specific than the selection and that are retrieved from the QueLo sub-system by means of the operations getSupers, getEquiv and getSubs, respectively. More general terms are shown at the top of the list, equivalent terms in the middle and more specific terms at the bottom. At the left of each item an icon is shown: an upward-pointing triangle for more general terms, a square for equivalent terms and a downward-pointing triangle for more specific terms.

The substitution menu has a similar hierarchical structure as the menu for addition, reflecting the taxonomic information in the output graphs of the operations getSupers, getEquiv and getSubs. In particular, some (possibly none) of the more general terms might be further generalised, in which case hovering on one such item triggers a sub-menu containing its direct super-concepts; similarly, if some of the more specific terms can be further specialised, then hovering on one such item triggers a sub-menu containing its direct sub-concepts that are compatible with the query. The same rules for further generalisation/specialisation apply to the items in the sub-menus, while equivalent terms (if any) cannot be further generalised nor specialised.

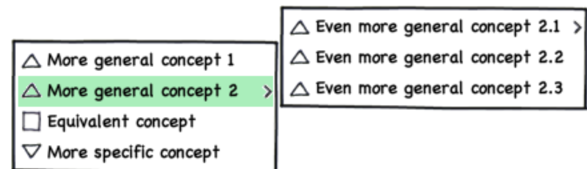


Figure 15: Structure of the substitution menu

As in the case of addition, the actual items shown to the user in the substitution menu are natural language descriptions generated by the NLG sub-system, rather than bare concept names. Clicking on any of given options triggers the invocation of substitute with the current selection and the concept name associated with the clicked item as input arguments. The selection is then cleared and the UI updates the representation of the query after the selected elements have been

replaced with the chosen term.

Observe that the operation substitute, as defined in our functional API, cannot deal with disconnected complex selection and, unlike the case of weakening and deletion, the problem cannot be overcome by converting such selection in a series of connected ones. This is due to the fact that substitution relies on the roll-up of the input selection itself, which is thus required to be tree-shaped (i.e., connected). For this reason, in the presence of a disconnected selection, the substitution operation is disabled.

NATURAL LANGUAGE RENDERING

The natural language interface of the Query Tool masks the composition of a precise query as the composition of English text describing the equivalent information needs. Interfaces following this paradigm are known as "menu-based natural language interfaces to databases". The users of such systems edit a query by composing fragments of generated natural language provided by the system through contextual menus.

In what follows, we describe how the natural language rendering of a query is achieved. We start by defining a particular linear form of the query that satisfies certain constraints, necessary to represent the elements of the query using a linear medium, that is, text. The constraints are enforced at the API level to ensure that different graphical user interfaces represent the query in a homologous way. Moreover, a consistent ordering of the query elements needs to be preserved during the operations for query manipulation to avoid confusing the end user. The linearised version of the query is then used as a guide for the language generation performed by the Query Tool's NLG engine.

Linearisation of a query

Since a concept name can be in general associated with more than one node in the same query, we introduce the notion of *tag*: given a query Q, a *node tag* is a pair $\langle n, c\rangle$ where $n \in V(Q)$ and $c \in \mathcal{V}(n)$, while an *edge tag* is a pair $\langle e, \mathcal{E}(e)\rangle$ such that $e \in E(Q)$. The set of tags relative to a node n is given by $\mathsf{tags}(n) := \{\langle n, c\rangle \mid c \in \mathcal{V}(n)\}$, while the set of all tags occurring in a query Q is defined as follows:

$$\mathsf{tags}(Q) := \{\langle e, \mathcal{E}(e)\rangle \mid e \in E(Q)\} \cup \bigcup_{n \in V(Q)} \mathsf{tags}(n)$$

A *linearisation* of a query Q is a strict total order $\lhd$ (that is, a binary relation that is asymmetric, transitive and total) on $\mathsf{tags}(Q)$ such that, for each edge $e \in E(Q)$, the following conditions hold:

$$\forall t \in \mathsf{tags}(\mathsf{init}(e)), \quad t \lhd \langle e, \mathcal{E}(e)\rangle \quad ; \tag{1a}$$

$$\exists t \in \mathsf{tags}(\mathsf{ter}(e)), \quad \langle e, \mathcal{E}(e)\rangle \blacktriangleleft t \quad ; \tag{1b}$$

and, for each node $n \in V(Q)$, it is the case that:

$$\forall t_1, t_2 \in \mathsf{tags}(n),\ t_1 \lhd t \lhd t_2 \implies t \in \mathsf{tags}(n) \ . \tag{1c}$$

where we use the expression $t_1 \blacktriangleleft t_2$ for indicating that $t_1 \lhd t_2$ and there does not exist $t \in \mathsf{tags}(Q)$ for which $t_1 \lhd t \lhd t_2$. In such case we say that t_1 *immediately precedes* t_2 or, equivalently, that t_2 *immediately follows* t_1. Note that $\lhd$ is the transitive closure of the relation $\blacktriangleleft$.

Informally, the conditions that a strict total order must satisfy to qualify as a linearisation state that "the tag of each edge is (1a) preceded by all the tags of its initial node and (1b) followed by at least one of the tags of its terminal node" and that (1c) "between any two tags of a node there can only be (distinct) tags of the same node". Moreover, from the above conditions it can be also proved that "the tag of each edge is followed by all the tags of its terminal node".

In general, without further restrictions, a query may admit more than one linearisation, but all of them share some common properties. In particular, each linearisation $\lhd$ of a query Q is such that the minimal element is always a tag of the root and for each subquery S of Q the restriction of $\lhd$ on $\mathsf{tags}(S)$ is itself a linearisation of S. Moreover, every linearisation is compatible with the *tree-order* of the query, that is, the tags of a node n precede the tags of a node m in the linearisation whenever there is a path from a n to m in the query.

A strict[1] total order on the node set of a tree can be obtained by fixing a total order on the children of each node and performing a complete visit of the tree according to a chosen traversal strategy. If the tree is visited using a depth-first or breadth-first traversal, then the resulting order is an extension of the partial order induced by the tree itself on its node set. Thus, a linearisation of a query can always be obtained by additionally fixing a total order on the labels associated with each node. For our purposes, we decided to use a depth-first traversal where the labels (resp., children) of a node are ordered according to the sequence (resp., inverse sequence) of applications of the operation addComp (resp., addRel) to that node as focus. The motivation for this choice lies in the fact that, for cognitive reasons, we want the query to change as little as possible from the visual point of view and in such a way that changes are restricted to a limited portion of the query. As a result, every execution of addComp or addRel yields one or more new tags whose place in the linearisation is right after all the node tags relative to the node given as input argument to the operation.

Natural Language Generation module

The natural language interface (NLI) of the Query Tool relies on a natural language generation (NLG) system to produce the textual representation of the query, following an idea first presented in [12] and lately refined in [8].

NLG systems use techniques from artificial intelligence and computational linguistics to produce human-readable texts out of machine-readable data. The Query Tool uses NLG to represent the whole query, along with all the elements that the user can use to refine it, as English text. The generated text is enriched with links that connect it to the underlying logical form of the query. This allows the user to operate on the query simply by editing an English text.

Unlike most NLG systems, ours is built to let the user determine the structure of the generated text by inserting, replacing and removing snippets of it. Thus, while in the classic NLG pipeline the information to be conveyed in the text and its order is determined by the document planning module, in

[1] We only consider orders that are strict (i.e., irreflexive) unless explicitly stated otherwise.

the Query Tool it is the user who decides both the information to be displayed and its arrangement.

As the Query Tool is not tailored to any specific domain, its NLG module is simple enough to be adopted in any context and it is not bundled with all the resources that are needed to generate text out-of-the-box. Therefore, in order to use it on a specific knowledge base, the system must be provided with a *lexicon* and a *template map*. The former contains the words to be used in the generated text; the latter is the bridge between the natural language and the knowledge representation language, associating each concept/role name with a generation template. Each such template contains the syntactic and lexical information necessary to generate a fragment of text representing the associated concept or role.

We selected the syntactic features available in the templates, hence supported by the generator, in order to keep the system simple while still being expressive enough. For this purpose, we collected and analysed a corpus of more than 12.000 unique relation identifiers and we partitioned them according to the recurring syntactic patterns. For each class of the partition, we then proposed a common natural language representation template. The result of this study is a set of simple but effective templates for representing most ontology relations using natural language.

During the first stage (*microplanning*) of the generation, linguistic information stored in the template map and in the lexicon blends with the logic information encoded in the query into a single structure, known in the NLG literature as *text specification* and consisting of a list of syntactic trees with inflected lexemes on its leaves. The NLG system operates on this structure to aggregate groups of adjacent syntactic structures into single more complex structures, and to select and replace existing referring expressions with more appropriate ones. These two tasks are known in the literature as *aggregation* and *referring expressions generation*, respectively. At the same time, the system keeps track of which element of the text specification is associated with which element (either a node tag or an edge tag) of the query. An association holds when the syntactic element is the result of the instantiation of a template associated with the element of the query. These associations are used for enriching the generated text with links to the underlying query.

The linearisation of the query simplifies the effort required by the referring expressions generation, as referring expressions that need to be reworked always appear in subject position. Our algorithm replaces a subject with a pronoun whenever the previous sentence had the same subject, otherwise the subject is left unchanged. Although ambiguous expressions may occur, ambiguity is not a crucial issue as these expressions originate from user operations upon a selected element, which always becomes the target of the referring expression. Our aggregation module performs relatively simple aggregation tasks such as aggregating sentences with the same subject, eliding the subject and parts of the verb whenever it is feasible.

Once these operations are completed, the text specification is ready to be transformed into the final text. This task, known as *surface realisation*, produces a list of text tokens, some of which are connected to edge or node labels. This list is finally fed to the GUI, that displays it to the user.

Elements populating the menu for addition and substitution operations undergo a similar processing. To produce the textual representation of such an element, the system makes a temporary copy of the portion of query affected by the operation. The operation is then carried out on this portion and the resulting structure is fed to the generation pipeline used for entire queries. The outcome of the generation process is the text which will appear on the menu.

Automated generation of lexicon and template map

For the Query Tool's NLI to work with a specific knowledge base (KB) a lexicon and a template map must be provided for it. Devising these resources requires an understanding of both the domain of interest (DOI) and basic linguistic notions such as verb tenses, noun genders and countability. To ease the burden of developing these resources from scratch, we experimented with a computational technique to have the system generate them automatically. This technique follows an approach to domain independent generation proposed in [11]. The functionality we implemented allows to produce all the resources necessary to configure our NLI for use with a new KB, using as a source of data the ontology itself. However, the process is not completely reliable, therefore system engineers must review the result and make the necessary corrections. In the following, we describe the technique to handle relation identifiers, but we adapted it to handle concept identifiers as well.

The idea is based on the observation that KBs already contain some form of linguistic information. In real-world ontologies, every concept and relation has a unique identifier (ID), which most of the times is not just an arbitrary string, but a mnemonic chosen by the knowledge engineer to describe the intended meaning of the identified concept or relation. Moreover, within these IDs, certain syntactic patterns occur more frequently than others.

In our approach, each relation ID is first tokenized according to an algorithm that takes advantage of the naming conventions used by ontology engineers. Second, the tokenized ID is fed to a custom part-of-speech tagger built around QTAG [14]. The resulting tagged tokenized ID is then lightly preprocessed before being finally passed to a transformation rule, chosen among thirteen different ones, that produces a template for the template map of the NLG system.

For the design of the transformation rules, we analysed our corpus, containing more than 12.000 relation IDs, in order to devise a partition of the domain in terms of syntactic patterns. The classes defined in this partition are s.t. to each relation of the same class can be applied a simple transformation in order to obtain a template. Each such transformation is also a uniform interpretation of the intended meaning of each relation ID in the class. Some care is needed when giving a uniform interpretation to syntactic patterns, as there are situations in which the same syntactic pattern is to be interpreted differently. For instance, the relation IDs "country_of_nationality" and "language_of_country" share the same

syntactic structure, but the first relation should be read as "the country of nationality of X is Y", while the second as "the language of X is Y". Each of the thirteen rules we defined corresponds to one class of the partition, and together they can handle 93% of the relations of the average ontology.

The system has been tested with some of the ontologies [9] developed by the national mapping agency of Great Britain and with the Pizza Ontology [6] by the University of Manchester, contributing 64 unique relations in total. From the IDs of these relations we automatically generated relation templates, which were then inspected in order to evaluate their usability in text generation. The result of the evaluation revealed that for 42 out of 64 relations (65%) the generated template is suitable for direct use with the Query Tool's NLI. The result suggests that although the generation of the template map is not totally reliable, it is nevertheless useful in that it speeds up the work of systems engineers, as they do not need to create the whole map from scratch, but only have to review the generated map and repair eventual errors. This improves the portability of the Query Tool's NLI, making it faster and easier to switch to a different knowledge base.

CONCLUSIONS AND FUTURE WORK

A Java implementation of the Query Tool has been devised, including the query logic and natural languages sub-systems and a fully functional user interface as described above. The Query Tool was implemented for the first time in the context of the European project SEWASIE [10] as a Java web-based application, later on converted into a stand-alone application [15]. Important optimisations were then introduced (see [16]) with the purpose of minimising the number of reasoner calls and thus improving the responsiveness of the tool. A difference between our implementation of the Query Tool and the previous one is that while the latter uses a DIG-based reasoning engine, the former takes advantage of the OWL-API. Most importantly, our implementation follows the formal specification given in the Query Tool's framework and functional API and complies with them.

A usability evaluation of the previous Query Tool's implementation was carried out in past work [3, 2] with the purpose of measuring its complexity of use from the user's point of view. In particular, the study aimed at determining how difficult it is for the user to formulate queries using the Query Tool and to understand the results. The outcome of the experiments showed that the tool can be easily used also by non-experienced users to query a domain in which they have no special expertise.

The framework and functional API presented in this paper consider only the addition of new relations to the query, but they could be extended to deal with attributes (i.e., properties relating a concept to a datatype) as well. The only difference with the current framework would be that a node associated with a datatype (i.e., the "range" of the attribute) cannot be the focus of a query for operations other than deletion. This basically means that such a node is always a leaf of the query tree and the only operation allowed on it is deletion. Then, since a node of this kind cannot be refined by adding a compatible term or attaching a new property, the query is never rolled-up with respect to it, thus avoiding the nonsensical eventuality that an edge associated with an attribute has to be inverted (going from the datatype to the subject). Though apparently simple, allowing for attributes poses some interesting questions, that we are currently investigating, in order to deal with concrete values from the point of view of reasoning.

The work in this paper has been partially supported by the European project ONTORULE.

REFERENCES

1. T. Catarci, T. Di Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *Proc. 16th Eur. Conf. Artificial Intelligence*, 2004.
2. T. Catarci, T. Di Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. An Ontology-Based Query Manager: Usability Evaluation. In *Proc. HCITALY 2005*, 2005.
3. T. Catarci, T. Di Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. Usability evaluation in the SEWASIE (SEmantic Webs and AgentS in Integrated Economies) project. In *Proc. 11th Int. Conf. on Human-Computer Interaction*, 2005.
4. P. Dongilli. Natural language rendering of a conjunctive query. Technical Report KRDB08-3, KRDB Research Centre, Free University of Bozen-Bolzano. `http://www.inf.unibz.it/krdb/pub/TR/KRDB08-3.pdf`, June 2008.
5. P. Dongilli, E. Franconi, and S. Tessaris. Semantics driven support for query formulation. In *Proc. of the 2004 Int. Workshop on Description Logics*, 2004.
6. N. Drummond, M. Horridge, R. Stevens, C. Wroe, and S. Sampaio. Pizza ontology. The University of Manchester. `http://www.co-ode.org/ontologies/pizza/`.
7. P. Guagliardo. Theoretical foundations of an ontology-based visual tool for query formulation support. Technical Report KRDB09-5, KRDB Research Centre, Free University of Bozen-Bolzano. `http://www.inf.unibz.it/krdb/pub/TR/KRDB09-05.pdf`, October 2009.
8. C. Hallett, D. Scott, and R. Power. Composing questions through conceptual authoring. *Computational Linguistics*, 33(1):105–133, 2007.
9. Ordnance Survey - Great Britain's national mapping agency. `http://www.ordnancesurvey.co.uk/oswebsite/ontology/`.
10. The SEmantic Webs and AgentS in Integrated Economies (SEWASIE) project. `http://www.sewasie.org/`, 2005.
11. X. Sun and C. Mellish. Domain independent sentence generation from RDF representations for the Semantic Web. In *Proc. ECAI'06 Combined Workshop on Language-Enhanced Educational Technology and Development and Evaluation of Robust Spoken Dialogue Systems*, 2006.
12. H. R. Tennant, K. M. Ross, R. M. Saenz, C. W. Thompson, and J. R. Miller. Menu-based natural language understanding. In *Proc. 21st Annual Meeting of the Association for Computational Linguistics*, pages 151–158. Association for Computational Linguistics, 1983.
13. M. Trevisan. A portable menu-guided natural language interface to knowledge bases. Master's thesis, University of Groningen, 2009.
14. D. Tufis and O. Mason. Tagging Romanian texts: a case study for QTAG, a language independent probabilistic tagger. *Proc. 1st Int. Conf. on Language Resources and Evaluation (LREC'98)*, pages 589–596, 1998.
15. I. Zorzi. An Ontology-Based Visual Tool for Query Formulation Support. Bachelor's thesis, Faculty of Computer Science, Free University of Bozen-Bolzano, 2005.
16. I. Zorzi. Improving Responsiveness of Ontology-Based Query Formulation. Master's thesis, Faculty of Computer Science, Free University of Bozen-Bolzano, March 2008.