university of groningen / faculty of arts

Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen - Bolzano

Master Thesis

A PORTABLE MENU-GUIDED NATURAL LANGUAGE INTERFACE TO KNOWLEDGE BASES

Student:	Marco Trevisan
Student number:	S1809725
Address:	Via Giorgione 12 B
	Santa Maria di Sala (VE)
	30036 Italy
Phone number:	+39 041 5781062
E-mail address:	evenjn@gmail.com
Supervisors:	Prof. Gertjan van Noord, University of Groningen
	Prof. Enrico Franconi, Free University of Bozen-Bolzano

Programme:	Research Master Linguistics
Specialization:	Erasmus Mundus Masters Programme in Language and Communication Technologies
Date of submission:	December 7th, 2009

Abstract

Menu-based natural language interfaces to databases are software systems that allow their users to edit a query by composing fragments of generated natural language. The fragments available for this purpose are provided by the system through contextual menus. This thesis discusses the development of such a system which relies on a description logic reasoner to determine the available fragments, and on natural language generation to produce them. The logic of the interface, along with the query interaction model, are defined by the Query Tool framework. The focus of this work is on the language model and on the resources needed to represent a query in any knowledge domain, meeting the constraints imposed by the query language and the interaction model of Query Tool. This thesis also documents two experimental techniques to produce automatically the resources needed to interface the system with a new knowledge base. The first technique mines a corpus for natural language expressions describing a semantic relation. The second technique produces a generation template from a natural language expression assembling the appropriate lexical and syntactic elements.

Table of Contents

1 Introduction

Natural language interfaces to databases (NLI) are software systems which allow their users to access information stored in a database by typing requests in some natural language [Androutsopoulos et al. 1995]. One of the approaches in NLI development is to use natural language generation (NLG) to display both the query and the elements which can be used to further refine it. In such a system, the users are not free to type their queries. Instead, they have to compose the query by selecting and combining fragments of natural language proposed by the system. The system produces these fragments through NLG techniques on the basis of the elements of the underlying formal query language.

This thesis discusses the development of such a NLI which takes advantage of the capabilities of a specific variant of database, namely a knowledge base. A knowledge base (KB) is, roughly speaking, a database system coupled with a deduction system. The deduction system operates on a logic-based description of the constraints that hold on the data, also known as *ontology*. The deduction system uses the ontology to infer knowledge, that is, to augment the information available to the system beyond the bare data. This information can be used to answer queries which otherwise a simple database could not answer. Research in description logics achieved efficient computational inference techniques for certain classes of inference problems. In particular, some of these techniques allow to calculate efficiently whether a query is not satisfiable, i.e. it will yield no results. This feature can be used to prevent the user from carrying out an operation on the query when the result of the operation would be an unsatisfiable query [Dongilli et al. 2004].

This thesis also discusses our efforts at improving the portability of our NLI. Our system is not tailored to suit any specific KB or any particular knowledge domain: in order to use it with a KB, it must be provided with appropriate linguistic resources. We experimented with two different computational techniques with the goal of reducing the effort of creating these resources, and ideally to realize an intelligent NLI working out-of-the-box with any KB.

The first technique draws from an idea first presented in a work in variant recognition [Lin & Pantel 2001], and later employed to identify meronymy relations [Ittoo & Bouma 2009]. The system works under the hypothesis that paths in dependency trees linking the same set of words tend to have the same meaning. We use this hypothesis to address a specific sub-task of the generation of the resources needed by our NLI, that is, the collection of natural language expressions describing semantic relations defined in the ontology. To this purpose, we re-used and adapted an algorithm developed for the project described in [Ittoo & Bouma 2009]. The system we designed retrieves and ranks list of dependency paths for each relation in the system, at the condition that the required linguistic resources are provided. These dependency paths can be fed to the rule-based system to produce the resources needed by our NLI.

The second technique follows an approach to domain independent generation developed by Xiantang Sun [Sun 2009] under the supervision of Chris Mellish. In a previous work they argue that "RDF representations carry rich linguistic information,

which can be used to achieve readable domain independent generation" [Sun & Mellish 2007]. We argue that even though the information is not sufficient in general, it is still useful to partially automate the production of the resources needed to configure a NLG system for its use in a new domain. To this purpose, we built and evaluated a system which implements Sun's approach as it was laid out in a draft of his thesis [Sun 2009]. The system we built produces all the resources necessary to configure our NLI for use with a new KB, using the KB itself as the only input data. The generated resources are meant to be reviewed and corrected by systems engineers and domain experts.

This introduction continues with an overview of Query Tool, the query interface framework our NLI is based on. In the first part of this thesis, we focus on the development of our natural language interface for Query Tool. We review the requirements, we present some attempts to relate the system to previous art in this field and we discuss the design of our natural language generator: the generated language, the generation pipeline, and the bridge between knowledge base and natural language, that is, our template map. In the second part, we switch to the topic of the automated generation of the template map. We first discuss the approach we adapted from [Ittoo & Bouma 2009], and we report the results of our evaluation. Next we discuss the technique we implemented following Sun's approach, and for this technique as well we report our experimental results.

1.1 Query Tool

A query interface to a database is a system that helps the user access information stored in a database. The user interacts with the interface to encode her informative needs into a piece of data called query. The most common kind of query interfaces, often found on Web applications, are form-based interfaces. In this kind of interfaces, the query is a set of labelled fields which the user fills with values. For example, eBay[1] allows the user to retrieve auctions for objects that are available to a specific country. To do so, the user has to input the country name in a specific field of the search form. Natural language interfaces (NLIs) are another kind of query interfaces. In NLIs, the query is a natural language expression input by the user. While far less common than form interfaces, NLIs can also be found on the Web. In WolframAlpha[2], to retrieve the countries with a GDP greater than a hundred million dollars, the user may type [1] in the search box.

[1] Asian countries with a gross domestic product of more than 100 millions

Regardless of the input methods, all query interfaces eventually transform the query into a format which can be processed by the database management system, e.g. into a SQL query. In form-based interfaces this translation is often simple and immediate. Natural language interfaces instead require a complex processing of the input, which may rely on intermediate representations, e.g. first order logic formulas or DRT (discourse representation theory) frames.

Query Tool is a framework for query interfaces to knowledge bases. A knowledge base (KB) is a database system enriched with logical constraints and connected to a logic reasoning system. A key component of a KB is its ontology. The ontology of a KB is a representation of the constraints that hold on the data stored in the KB. The con-

1 An on-line auctions Web site accessible at www.ebay.com

2 A scientific knowledge base Web site accessible at www.wolframalpha.com

straints are encoded as logical formulas, and therefore they can be fed to logical reasoners for inference purposes. Another feature of ontologies is that the logical language used to specify them usually frames the data into structures more similar to the ones used in natural language than to those used in database schemas. For this reason, the ontology plays a key role in bridging the gap between machine processable data and human-readable text.

Query Tool defines an abstract model for queries on KB, and a set of operations on it. To interact with a Query Tool-based interface, the user creates a new query, and applies these operations until the query represents her informative needs. Through its query model, Query Tool controls the actions of the user and drives the interaction of the user with the KB. Query Tool's query model is abstract in the sense that it does not specify how the query should be displayed to the user. In principle, the Query Tool framework can be instantiated with any of the previously mentioned interfaces, i.e. form-based interfaces, NLIs, and many others. A Query Tool query is an intermediate representation of the user's informative needs: it will eventually be transformed into an SQL query and fed to the database management system.

The main benefit of Query Tool is that, to a limited extent, it prevents the user from composing queries which, if executed on the target KB, would retrieve no elements. At any point during the query composition process, the set of available arguments for the operations which can be carried on the query depends on the KB's ontology and on the query composed so far. To calculate the available arguments, the tool relies on the constraints described in the ontology, and on the services of a logic reasoner. For example, if the query is [2][3] and if the target KB specifies that `Male` objects cannot be in `spouse` relation with `Male` objects, then Query Tool will not allow the user to add to the query the additional constraint `Male(y)`, on the ground that the resulting query would be unsatisfiable.

[2] $Male(x) \wedge spouse(x, y) \wedge Person(y)$

[3] $Country(x) \wedge PrimeMinister(x, y) \wedge Female(y)$

A positive side effect of this functionality is that the restriction of the available arguments also qualifies the available arguments as relevant components of the data model of the KB. These elements are displayed only in contexts where they are immediately relevant to the user, and they inform the user about the capabilities of the informative system. For example, suppose that the target KB specifies that `Country` objects may relate to other `Country` objects through the `Border` relation and through the `RecognizesAsIndependent` relation. If the query is [3], the list of available operations provided by Query Tool will contain the following two operations:

- add $RecognizesAsIndependent(x, z) \wedge Country(z)$
- add $Border(x, z) \wedge Country(z)$

Users do not need to have previous knowledge about the database schema, nor about the ontology. The elements that they may use to communicate their informative needs are listed by the system and tailored to the context they specified so far.

1.1.1 The query language of Query Tool

In general, different KB management systems rely on different languages to represents queries and knowledge. Presently, the most prominent knowledge representation

3 We assume the reader is familiar with first order logic notation.

(KR) language is OWL, a W3C recommendation. Query Tool can be used to query an OWL KB, but it also supports (Extended) ER KBs and others. In general, different KR languages have different features, but certain basic features are found in most KR languages, usually disguised under a different syntax. Research in the field of Description Logics (DL) produced a theory of knowledge representation languages which allows to describe and analyse the semantics features of these languages. This theory collects results on the computational complexity of the operations defined on these languages, which in turn gives information about the complexity of reasoning with queries written in those languages.

Query Tool supports KBs with different query languages because the queries it supports rely on features which are shared by most of the existing KR languages. In other words, Query Tool allows the user to formulate queries in a language which is simple enough to be translated to all the supported KR languages. In terms of description logic, these features are known concept instantiation, limited existential restriction and conjunction.

- Atomic Concept instantiation: the query language allows to specify that an element should be an instance of a certain atomic concept. For example, it is possible to specify that each object to be retrieved is an actor.
- Limited existential restriction: the query language allows to introduce a new object Y and specify that a certain relation[4] holds between a previously-introduced object and Y. Y cannot refer to previously-introduced objects. For example, it is possible to specify that each object to be retrieved is married to another element.
- Conjunction: the query language allows to specify a list of atomic concept instantiations and limited existential restrictions that must hold at the same time for an element. For example, it is possible to specify that each object to be retrieved is a Colombian actor, singer, film director and film producer whose mother is Colombian and whose spouse is Peruvian.

In DL, queries characterized by these features are called tree-shaped conjunctive queries. Each DL query is a kind of DL formula. In the following, we refer to DL queries as DL query, and we will use the simple term *query* to refer only to the data structure defined by Query Tool to represent tree-shaped conjunctive formulas.

Tree-shaped conjunctive DL queries are defined as follows. Given a set of atomic concepts $Ac = \{c_0, c_1, c_2, .. c_i, .. c_n\}$ and a set of relations $Ro = \{r_0, r_1, r_2, .. r_i, .. r_n\}$, the syntax of the language of tree-shaped conjunctive DL queries is described by the following BNF grammar, where S is the start symbol and all the elements of $Ro \cup Ac$ are terminal symbols:

$S ::= C \mid \exists R.(S) \mid S \sqcap S$

$R ::= r_i$ for each $r_i \in Ro$

$C ::= c_i$ for each $c_i \in Ac$

The “square cap” symbol $\sqcap$ is used for conjunctions, while the construct starting with the “there exists” symbol $\exists$ is used for limited existential restrictions.

The language can be given a direct model-theoretic semantics, but for simplicity we provide instead a semantic-preserving map m from DL queries to formulas of model-

4 More precisely, object relations, also known as roles. Quert Tool cannot deal with relations between objects and data type literals yet.

theoretic first order logic. We define the map recursively on the structure of the DL formulas, as follows[5]:

$m(S) = m(S, x)$ — x is a new variable

$m(C, x) = C(x)$

$m(\exists R.(S), x) = \exists y.(R(x, y) \wedge m(S, y))$ — y is a new variable

$m(S \sqcap S, x) = m(S, x) \wedge m(S, x)$

According to this mapping, [3] is mapped to [4], where x is a free variable.

[4] $A \sqcap \exists R.(B \sqcap C) \sqcap D$

[5] $A(x) \wedge \exists y.(R(x, y) \wedge B(y) \wedge C(y)) \wedge D(x)$

The language defined via this map is the fragment of first order logic consisting of all formulas with a single free variable, composed using only unary predicates, binary predicates, conjunctions, and existential quantifications, such that the undirected graph induced by the applications of binary predicates (i.e. each application P(x, y) corresponds to an edge between the vertex x and the vertex y) does not contain loops.

1.1.2 The query model of Query Tool

The query model defined by Query Tool is a data structure for representing tree-shaped conjunctive DL queries. In Query Tool, a query is a labeled directed n-ary tree, where each node is labeled with a sequence of node labels, and each edge is labeled with an edge label. Each node label refers to an atomic concept of the ontology, each edge label refers to a relation of the ontology. When the context allows to do so, we will identify node labels with the atomic concepts they refer to. For example, we will write about the concepts labelling a node instead of about the concepts referred by the node labels of a node. We will use this approach also with edge labels.

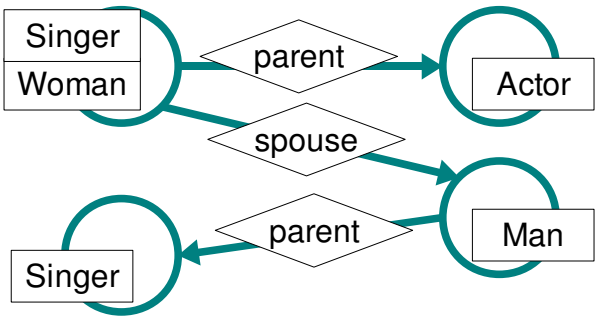

Illustration 1: A Query Tool query

The root of the tree is the node which has no entering edges. The query in illustration 1 is rooted at the node labeled with Singer and Woman.

The function roll-up associates each node of the Query Tool query to a tree-shaped conjunctive DL query. Each DL query defines a concept, in the sense that it identifies a set of elements, the elements which satisfy it. For this reason, each node represents

5 Note that in this definition we abuse of the notation in that, for each DL concept, we use the same symbol to denote the concept in the DL language and the corresponding unary predicate in the first order logic. We abuse the notation in the same way with DL relation symbols.

an entity instantiating the concept associated with the node. For example, the last node in illustration 1 represents a singer whose father is married to a female singer mother of an actor, whereas the first node represents a female singer mother of an actor married to the father of a singer.

The roll-up of the root is the formula [6]. The roll-up of the node labeled with Man is [7], where $spouse^{-1}$ is the inverse of the relation spouse.

[6] $Woman \sqcap Singer \sqcap \exists parent.(Actor) \sqcap \exists spouse.(Man \sqcap \exists parent.(Singer))$

[7] $Man \sqcap \exists parent.(Singer) \sqcap \exists spouse^{-1}.(Singer \sqcap Woman \sqcap \exists spouse.(Actor))$

Each node of the tree is associated with the formula [8], where c_0, c_1, c_2, ... c_N are the labels of the node, r_0, r_1, r_2, ... r_M are the labels of the edges leaving the node and the inverse of the labels of the edges entering the node. S_0, S_1, S_2, ... S_M are the formulas associated with the nodes on the other side of those edges. The query encoded by the tree is the formula associated with the root node.

[8] $c_0 \sqcap c_1 \sqcap c_2 \sqcap \ldots c_N \sqcap \exists r_0.(S_0) \sqcap \exists r_1.(S_1) \sqcap \exists r_2.(S_2) \sqcap \ldots \exists r_M.(S_M)$

The semantics of description logic formulas does not depend on the order of the elements appearing in conjunctions. However, the query model of Query Tool maintains an ordering among the labels of each node and among the edges leaving that node. The reason is that any user interface instantiating the Query Tool framework must display the elements of the query in a certain order. If no order was enforced, the labels or the children of a node could be scrambled after every operation on the model, confusing for final users of the interface.

Query Tool allows the user to mark nodes as *distinguished* to select the entities that should be visualized as part of the answer to the query. In addition, Query Tool allows the user to mark edges as *sticky*, to make sure that deletion and substitution operations do not remove the nodes they enter.

1.1.3 Operations of the query model

Query Tool defines four operations for the query model, each operation taking different arguments. For each query, for each operation, Query Tool determines the combinations of arguments the user can apply. The available combinations of arguments depend on the target KB and on the query, and therefore they are re-computed online after every modification to the query. The four operations Query Tool provides are:

- addCompatible(node, concept)
- addProperty(node, relation, concept)
- substitute(selection, concept)
- delete(selection)

These operations are defined formally in [Guagliardo 2009]. In the following, we describe the operations one by one with the help of some examples. We will apply the operations on the toy query depicted in illustration 2, targeting the toy ontology depicted in illustration and 3.

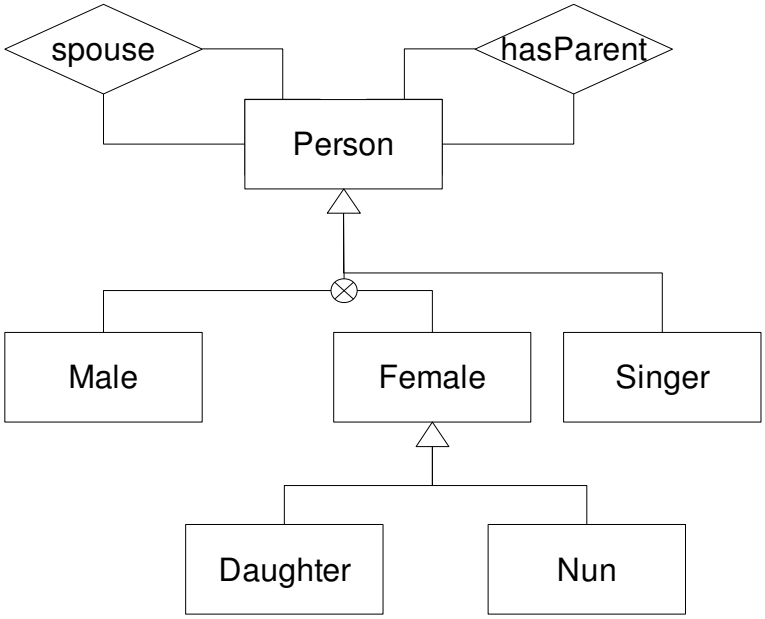

Illustration 2: A toy ontology

The toy ontology contains a concept (Person) generalizing four other concepts (Man, Woman, Singer, Nun, Daughter). Woman is specialized by Nun and Daughter. Woman and Man are disjoint. This implies that Nun and Man are incompatible concepts, as well as Daughter and Man are. The ontology contains two relations: hasParent and spouse. For both relations, the domain is Person and the range is Person. The ontology also contains two constraints which are not depicted in the illustration. First, Nun objects do not participate in spouse relations. Second, every Daughter object participates in a hasParent relation in first position, i.e. on the side of children.

Illustration 3: Toy query

The toy query displayed in illustration 3 is a tree with two nodes connected by one edge. The first node is labeled with the concept Woman, the other node is labeled with the concept Person. The edge is directed from the first node to the second, and it is labeled with the relation parent. The DL query encoded by the toy query is [9].

[9] $\text{Woman} \sqcap \exists \text{hasParent}.(\text{Person})$

The operation addCompatible takes as input a node of the query and a concept, and it adds the concept as a label to the input node, at the end of the sequence of labels of the node. The concepts available as arguments for this operation are those concepts which do not generalize, nor specialize, nor are equivalent to, nor are incompatible with the concept represented by the roll-up of the node. For example, on the toy query, Query Tool allows to apply the operation addCompatible on the node labeled with Woman, with argument the concept Singer. Any other concept argument would not be allowed, since all the other concepts are either generalizations (Person) or specializations (Nun) of the concept associated with the node argument, that is [9], or they are equivalent to it (Daughter) or they are not compatible with it (Man). After

the operation is applied, the resulting query is the one depicted in illustration 4, associated with the DL query [10].

[10] Woman ⊓ Singer ⊓ ∃hasParent.(Person)

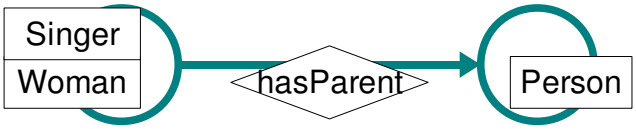

Illustration 4: Toy query after AddCompatible

The operation addProperty takes as input a node of the query, a relation and a concept, and it adds a new edge, pointing to a new node, to the sequence of edges leaving that node. The edge is labeled with the input relation, and the new node is labeled with the input concept. The arguments available for this operation are those concepts and those relations that can be used to instantiate a limited existential restriction compatible with the roll-up of the node argument. For example, on the toy query, Query Tool allows to apply the operation addProperty on the node labeled with Woman, with arguments the relation spouse and the concept Person. After the operation is applied, the formula associated with the resulting query is [11]. The resulting query is shown in illustration 5. If the node labeled with Woman was labeled with Nun instead, Query Tool would not have allowed the addProperty operation with argument spouse, since the ontology specifies that no Nun can participate in spouse relations.

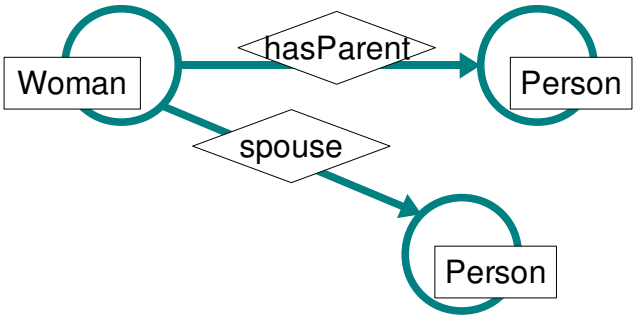

Illustration 5: Toy query after AddProperty

[11] Woman ⊓ ∃hasParent.(Person) ⊓ ∃spouse.(Person)

The operation substitute takes as input a concept and a selection, that is either a node label, a node, or an entire sub-tree of the query. If the selection is a node label, the operation replaces it with the input concept. Otherwise, the operation replaces the node or the entire sub-tree with a node labeled with the concept. The concepts available as arguments are those concept which generalize, specialize, or are equivalent to the concept associated with the selected portion. For example, Query Tool allows to Substitute the whole toy query with the concept Daughter. After the operation is applied, the resulting query would be a single node labeled with Daughter. Query Tool also allows to substitute the single label Person in the toy query with any concept subsumed by Person, e.g. Man. Illustration 6 shows the result of this latter substitution.

Illustration 6: Toy query after Substitute

The operation `delete` simply takes as input a selection, and deletes it. If the selection is a single node, and the node has children, delete will not have any effect. The exact behaviour of substitute and delete is actually more complex, as Query Tool allows to protect portions of the query from being removed or replaced, by marking edges as sticky. If some sticky edge would be removed as an effect of a `delete` or `substitute` operation, the edge is not removed instead. Also, every node and every edge on the path from the root of the query to the node pointed at by the sticky edge is not removed. Node labels of the nodes in of the selected node or of the selected sub-tree are removed regardless of sticky edges.

2 Natural Language Interface

Query Tool comes with two graphical user interfaces: the tree interface and the natural language interface. Each of these interfaces provides a visual representation of the query, widgets that allow to perform operations, and visual access to the arguments for these operations. The natural language interface represents the query as English text. This interface realizes the operations `addCompatible`, `addProperty`, `substitute`, and `delete` as addition, substitution and deletion of portions of the text.

This section of the thesis describes the problems we faced during the realization of this interface, and the solutions we adopted. The following part deals with the requirements of the system, then we give an overview of similar work in this field. Next, we describe the design of our NLI starting from the language used to generate the natural language representation of the query, its syntax and its lexicon. The generator relies on the features available in this language to build the text. Once the language is defined, we proceed with describing the template map, our bridge between the elements of the ontology and the linguistic resources necessary to represent them as English text. Finally, we describe the generation procedure.

2.1 Requirements

The natural language interface for Query Tool must fulfil four basic requirements:

- the query must be presented as English text;
- the operations must be realized as insertion, deletion or substitution of portions of the text;
- there must be an orderly association between the query elements and some portions of the text;
- the execution of the query operations should affect the text minimally.

In the following, we will discuss each of these requirements in more detail. The first requirement is that the interface must present the query to the user as English text. For example, the query depicted in illustration 7, associated with the formula [1], could be presented as [2].

[1] `Man ⊓ ∃hasParent.(Singer)`

[2] I am looking for a man. One of his parents is a singer.

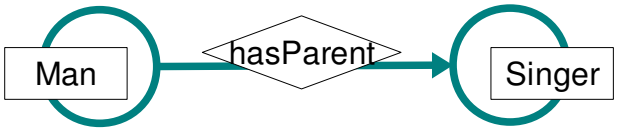

Illustration 7: Query Tool query for `Man ⊓ ∃parent.(Singer)`

The second requirement is that the interface must realize all query operations as addition, deletion or substitution of portions of the text. For example, substituting `Sing-`

er with Woman in [1] could be realized as substituting a singer with a woman in [2]. Adding ∃spouse.(Actor) to the node labeled with Man could be realized as adding His spouse is an actress right before One of his parents is a singer. Query Tool allows to extend the query with arguments chosen among a set of available values. The interface should present the available choices as natural language expressions, as fragments of text.

The third requirement is, the interface must associate portions of the text with edge labels and node labels of the query, and it must order these portions according to the order of the associated labels in the query. This means that the addition of a label to a node must result in the insertion of text after the text associated with the other labels of the node, and before the text representing the labels of the edges leaving the node. The addition of an edge leaving a node must result in the insertion of text after the text representing the labels of the node, and before the text representing the labels of the other edges leaving the node. For example, adding Actor to the node labeled with Man could change [2] into [3]. Adding ∃spouse.(Actor) to the node labeled with Man could change [2] into [4].

[3] I am looking for a man. He is an actor. One of his parents is a singer.

[4] I am looking for a man. His spouse is an actress. One of his parents is a singer.

In addition to the order of the node labels of a node, and in addition to the order of the edges leaving a node, the interface must also follow the order of the nodes obtained visiting the query breadth-first. Let us elaborate a bit on this point. Given a query, the interface must produce an English text as the concatenation of a sequence of n strings t_0, t_1, ... t_n where n is the number of labels of the query. Each of these strings must be associated with exactly one label of the query, and vice versa, i.e. there must be an invertible function f associating each label of the query to one of the strings in the sequence. In addition, f must be such that the sequence $f(t_0)$, $f(t_1)$, ... $f(t_n)$ is the sequence of labels obtained visiting the query (which is a directed ordered n-ary tree) breadth-first. Breadth-first means starting from the root, and listing, for each node, the label of the edge entering the node, if any, followed by all the labels of the node, in their order.

For example, the Query Tool query for [5], depicted in illustration 8, contains eight labels. Query Tool's query model specifies that the labels and the outgoing edges of a node are ordered. For the sake of this example, we can assume that the ordering is the one given by the illustration lay out – upper labels before lower labels. With this information, we can calculate the sequence of labels for the query, which is [6].

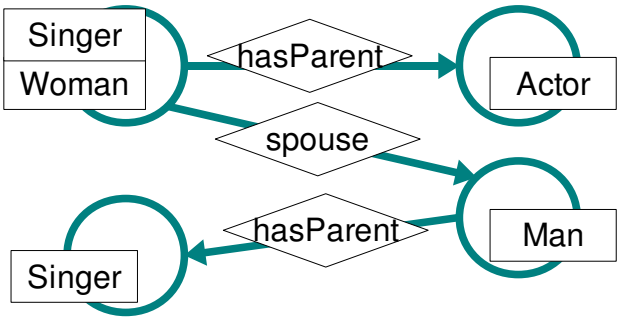

Illustration 8: Query Tool query for Singer ⊓ Woman ⊓ ∃hasParent.(Actor) ⊓ ∃spouse.(Man ⊓ ∃hasParent.(Singer))

[5] Singer ⊓ Woman ⊓ ∃hasParent.(Actor) ⊓ ∃spouse.(Man ⊓ ∃hasParent.(Singer))

[6] [Singer, Woman, hasParent, Actor, spouse, Man, hasParent, Singer]

[7] I am looking for a singer. She is a woman, one of her parents is an actor, her spouse is a man. One of the man's parents is a singer.

The interface could represent the query using the text [7], the concatenation of the eight strings listed in the box below, which f maps orderly into the labels listed in [6].

- I am looking for a singer.
- She is a woman,
- one of her parents is
- an actor,
- her spouse is
- a man.
- One of the man's parents
- is a singer.

The fourth requirement is that the query operations should affect the text the least possible. In particular, they should not affect portions of text which are not directly involved in the operation. They should modify the text in a discreet way: for example, the addition of ∃spouse.(Actor) as in the previous example should result in the insertion of His spouse is an actress with no changes to the surrounding text.

Ideally, the interface would realize the replacement or deletion of part of the query as the replacement or deletion of all and only the portions of text associated with the labels contained in that part of the query. We believe that if that was the case, the user would get quickly uncomfortable with the behaviour of the interface. The text representing the query would sound extremely unnatural, for no anaphoric reference could appear in the text. Anaphoric references such as She, her and the man in [7] are sensitive to the replacement of the elements of the text they refer to. Replacing a man with a woman in [7] would cause One of the man's parents to become One of the woman's parents. Anaphoric references would not be the only problem: finite verbs, like “is”, must agree with the number of their subjects, and therefore are affected by the same issue. For these reasons, there is a conflict between between the linearity of the behaviour of the interface and the range of linguistic features available to produce the text, which in turns impacts the text readability.

Last but not least, one more requirement which we did not mention at the beginning. The interface should be easily configurable to work with any KB, and software developers with basic or no linguistic knowledge should be able to configure the interface to query KB in any domain.

2.2 Previous Art

The requirements we just presented describe a problem that we may frame into the literature in at least three different ways, not mutually-exclusive:

- a natural language interface to a database
- a natural language generator
- a controlled natural language editor

Each of these characterizations captures some of the features our system. Let us briefly see how.

> “A natural language interface to a database (NLI) is a system that allows the user to access information stored in a database by typing requests expressed in some natural language.” [Androutsopoulos et al., 1995]

The definition fits almost perfectly. Our system is not the typical NLI in that the user cannot type freely: instead, the user may only choose among the portions of text proposed by the system. [Androutsopoulos et al., 1995] also report of a system, NLMenu [Tennant et al. 1983], later evolved into LingoLogic [Thompson et al. 2005] which is similar to ours in this respect. NLMenu is a menu-based natural language understanding system. Rather than requiring the user to type his input to the system, input to NLMenu is made by selecting items from a set of dynamically changing menus. Active menus and items are determined by a predictive left-corner parser that accesses a semantic grammar and lexicon. (Adapted from [Tennant et al. 1983].)

The main difference between our NLI and NLMenu is that our NLI relies on the capabilities of KBs instead of using a semantic grammar to prevent the user from composing unsatisfiable queries. Our approach has two benefits over semantic grammars.

The first advantage is that our system is more effective than semantic grammars in preventing the user to formulate queries that lead to no results. For each element of the query, the set of extensions which can be attached to it depends on the whole query, and not just on the element. By contrast, the semantic grammar of NLMenu is context-free: this means that, for example, the query “a car” could be extended specifying “with an electric engine” and later with "running on diesel". Our system would forbid the second extension on the ground of logical incompatibility.

The second advantage is that our system requires less effort to be configured to interface a new database. NLMenu and other NLIs based on semantic grammars need a dedicated semantic grammar for each database, and each such grammar is to be hand-crafted. Our system instead relies on the ontologies which have been developed for these KB systems.

> “Natural language generation systems combine knowledge about language and the application domain to automatically produce documents, reports, explanations, help messages, and other kinds of texts.” [Reiter & Dale, 1997]

The first and the third of the four requirements listed in the previous chapter describe a NLG system. The KB provides the interface with knowledge about the application domain, in form of an ontology. The interface may combine this knowledge with user-input linguistic resources and with a hard-coded language model in order to transform Query Tool queries into text. However, our system also allows to edit the generated text, to insert, replace or remove portions of it. This kind of activity is slightly out of the scope of classic NLG literature, which focuses only on producing the text. Also, an important subtask of NLG, called text planning, is not a concern for our system, as the content to be represented and the order of its representation are fixed. As a system generating text from ontologies, the system we want to build is very similar to NaturalOWL [Androutsopoulos et al. 2008]. NaturalOWL generates English and Greek descriptions of individuals (e.g. items on sale or museum exhibits) and classes (e.g. types of exhibits) from OWL DL ontologies annotated with suitable linguistic and user modeling information. (Adapted from [Androutsopoulos et al. 2008].)

> "A computer-processable controlled natural language is a well-defined subset of a natural language that can be translated unambiguously into a formal target language (and optionally vice versa)." (Rolf Schwitter, quoted from a draft of the CNL Manifesto, 2009)

A controlled natural language (CNL) is a language engineered to be read and written almost like a natural language. The lexicon and syntax of a CNL is restricted with respect to the natural language it is modeled on, either prohibiting certain words or syntactic constructs, or by allowing only selected ones. The focus of the current research in this field are languages with formal syntax and semantics, free of ambiguity, that can map each sentence of the language into a single logical formula coherent with the meaning of the sentence. An editor for such a controlled natural language can be used as a KB authoring system [Bernstein & Kaufmann 2006] but also as a query interface [Bernstein et al. 2005].

An editor of this kind has many of the features of our system. Like in NLI and NLG systems, the medium of representation of the query is text in a natural language. Unlike NLG systems, the text can be edited, but unlike a NLI, the text cannot be edited freely, but only according to the rules of the CNL. Our system differs from CNL editors such as the one described in [Kuhn 2009][6] or the one developed for the Grammatical Framework [Ranta 2004][7] in one point. In these systems the syntax, together with the lexicon, determines the possible expansion of a text. For example, at the beginning of a sentence, the user has the option start with a definite article, `the`, or with an indefinite one, `a`. In our system, instead, the semantics determines the possible expansions of the text, as the operations available through menus operate directly at the semantic level. The Query Tool query model is so simple that the user can control it relatively easily with buttons and menus. Should it become more complex, for example allowing disjunction, negation, or cycles in the query graph, then it may be more convenient for the user to edit the text at the syntactic level instead of browsing a forest of menus to pick the desired function.

One challenge in the design of controlled natural language lies in keeping its syntax unambiguous but still user-friendly. This challenge does not appear in Query Tool because the system does not need to interpret the natural language representation of the query, so no ambiguity needs to be resolved.

In addition to the systems described above, there is another system in the literature sharing many features with ours: WYSIWYM [Power et al. 1998], also known as Conceptual Authoring. This system does not fall in any of the previous characterizations, but it does use NLG technologies and an interaction model similar to that of NLMenu. WYSIWYM editing (acronym for "what you see is what you mean") is a knowledge editing method in which text in natural language is the medium to view and edit knowledge. The text employed to this purpose is generated by the system. The user can edit the text only by clicking on "anchors" in the text and choosing from a list of semantic alternatives, presented as expressions in natural language. Each choice directly updates the KB, from which a new text is then generated. (Adapted from [Power et al. 1998].)

One difference between WYSIWYM and our system is that the first knowledge is represented using domain-specific object-oriented models, while ours relies on description logic, with the benefit that the list of semantic alternatives is calculated using production-level DL reasoners. Adapting WYSIWYM to a new domain requires adapt-

6 E.g. http://attempto.ifi.uzh.ch/webapps/acewikigeo/
7 E.g. http://www.cs.chalmers.se/~aarne/GF/demos/index.html

ing the domain-specific model and, following this model, the associated linguistic resources (templates). Our system relies on existing DL-based formalisms for domain knowledge representations. This allows to reuse existing knowledge schemas (ontologies) and possibly NL technologies designed for them.

WYSIWYM has been used also as a natural language interface to databases [Hallet et al. 2007], but WYSIWYG originates as a tool for editing machine-processable knowledge and high-quality text representing that knowledge. Our system is a tool for editing a query in a very simple query language powered by DL reasoners. The text produced by our system will on average be not longer than a paragraph describing a small set of related elements. A text produced with WYSIWYM will usually be much longer and much more structured, where the structure of the generated text may vary greatly across different domains. Instructions for operating a washing machine cannot be generated in the same way as descriptions of museum exhibits. For this reason, the complexity of the generation facilities used in WYSIWYM is on another level with respect to the NLG system Query Tool needs.

Query Tool blends selected features of all the systems presented above, namely menu-based query editing, natural language generation to represent the query being edited and the natural language expressions which can be used to edit it. The novelty of the approach is the DL-based query representation and the DL-reasoning used to restrict the elements which can be used to edit the query.

The work described in this thesis is not the first attempt at designing a natural language interface for Query Tool. The first version of the tool adopted a simple solution based on concatenation of canned text. The second solution [Dongilli 2008] relied on KMPL, a grammar development environment working with systemic functional grammars. In a subsequent stage, we developed the prototype of a NLG system [Trevisan 2008] following the documentation of NaturalOWL [Androutsopoulos et al. 2008]. A different project [Perez 2009] investigated on the syntactic features the NLG system could use in the generated text, with the aim of defining a controlled natural language simple to support and expressive enough to be used with real-world ontologies. The NLG system we are going to describe relies on many ideas and insights introduced in the project mentioned above.

2.3 Language

In this section we describe the syntactic and lexical features we selected to support in our NLG system. This selection follows from an analysis of the requirements and an analysis of a corpus of ontologies.

2.3.1 Design goals

Two goals drive the choice of the features to include in the generation language:

- Expressiveness – the language should be flexible enough to describe any query in any domain, within the limits of the underlying query language
- Readability – the language should permit the creation of text that humans would enjoy reading

In the following, we will review the elements which contribute to define the meaning of the text to be produced.

Introducing the query

The text our system generates describes the information need of the user. Query Tool always presents an initial query at the beginning of each interaction, and this initial query, if represented without any contextual information, like in [8], may be obscure for users accustomed to type their query in an empty text box.

[8] A thing.

We could use expressions like [9-12] to contextualize the query itself. Among those, the formulation which most faithfully corresponds to the system's interpretation of the user's input is [11], but we chose instead to rely on the simpler [12].

[9] Please tell me about ...

[10] I am searching for ...

[11] Please show me a list of all the elements that, according to the information you possess, are ...

[12] I am looking for ...

As an aside, we adopted [8] as the representation of the concept `Top`, which is the concept subsuming all other concepts. However, depending on the target ontology, a more appropriate expression could be [13].

[13] A thing or a person.

Representing a query

A query in Query Tool is the description of a set of features which identify the set of objects exhibiting them. For example, the query `student(x)` identifies the set of all objects which exhibit the feature of being a student. Queries in Query Tool are recursive structures: every node is the root of a query, which is a sub-query of the query rooted at the parent of the node. Each of these sub-queries is, in turn, a description of a set of features which identify the set of objects exhibiting them. In a similar way, humans use natural language to identify a set of objects by listing the features that characterize them. There are many ways to describe such a set of elements in natural language, but all of them rely on a noun phrase. We could represent [14] using any of [15-19].

[14] `student(x) ⋀ actor(x)`

[15] I am looking for students who are also actors.

[16] I am looking for a student who is also an actor.

[17] I am looking for all the students who are also actors.

[18] I am looking for a list of students who are also actors.

[19] I am looking for some students who are also actors.

Plural expressions like [15, 17, 18, 19] are more accurate considering that the system will return the list of qualifying elements. However, [17] cannot be split into two separate sentences, and the other plural forms become verbose when split (see for example [24]). Instead of using singular forms in sub-queries and plural forms in the main query, we chose to use singular forms whenever possible. This reduces the complexity of the generation and reduces the amount of linguistic information necessary

to port the application to a new domain. The downside is that sometimes plural form is more adequate than singular form. For example, [15] sounds better than [16].

Representing relations

When an edge labeled with the relation R leaves a node N and enters a node M, it means that for each element x in the set of elements described by the query rooted in N, there exists at least one element y among those described by the query rooted in M, such that R(x, y). For example, [20] describes the set of all teachers who have a child who is a student. A faithful representation of [20] would be any of [21-25]. In this case it is not handy to use a plural expression to represent the query rooted in the node associated with the variable x. The reason is, splitting the sentence requires to introduce an "each of" expression, which in turn introduces the singular form. Compare [24] with [25]: [24] is more coherent with the fact that the system will return a list of teachers, but also more wordy.

[20] teacher(y) ∧ mother-of(y, x) ∧ student(x)

[21] Please show me the list of all elements such that each element is a teacher and it is the mother of at least one element who is a student.

[22] I am looking for the teachers who are mothers of at least one student.

[23] I am looking for the teachers who are mothers of some students.

[24] I am looking for some teachers. Each of them is the mother of at least one student.

[25] I am looking for a teacher. It is the mother of at least one student.

Since every node is associated with a sub-query, and each sub-query is represented by a noun phrase, the representation of an edge will always connect two noun phrases. It is in general easy for humans to produce a sentence out of a triple (node, edge, node), but the process cannot easily be automatized, because the natural language expressions which can be used to represent a relation do not fit into a single syntactic category. A relation may be represented by a proposition [26], by an adjective [27], by a noun [28], by a verb [29], or even by a more complex expression [30].

[26] I am looking for a flat **opposite** a park.

[27] I am looking for a food **typical** of a country.

[28] I am looking for a woman **mother** of a teacher.

[29] I am looking for a shop **selling** shirts.

[30] I am looking for a person who **has been co-authoring a book with** a Nobel laureate.

While in principle one could need more than one sentence to represent a relation, this is rarely the case, and most triples (node, edge, node) can be described with a single clause.

Representing concepts

When a node X is labeled with the concept C, it means that each element x in the set defined by the query rooted in X possesses the feature represented by C. While in principle even a whole clause could be necessary to represent a concept, like in [34],

most concepts can be represented using nouns [31] or adjectives [32], or noun phrases [33].

[31] I am looking for a limousine.

[32] I am looking for a blue car.

[33] I am looking for a pick-up truck with air conditioning.

[34] I am looking for a ship destroyed during WWII.

Again, while in principle one could need more than one sentence to represent a concept, we assume that a noun phrase will be adequate in most cases.

Readability

The requirements for the NLI specify that each label of the query should be associated with a textual element, and they also specify the order of these elements in the text. Therefore, it is not possible to improve the readability of the text by re-ordering its elements. However, there are other dimension of the language which affect the readability of the text.

One dimension is the lexical choice. The same concept may be represented using different expressions in different contexts. We did not work on this feature, as it would require more effort from the knowledge engineers, and it would be hard to support it without targeting a specific domain. Another dimension is ellipsis, that is, the possibility of removing elements of text which are redundant for its comprehension. Another dimension is the choice of referents in anaphoric references.

The language for the generation is a fragment of English we crafted by selecting syntactic and lexical features according to the observations given in the previous section. In the following we describe the design choices and the resulting language, syntax and lexicon.

2.3.2 Syntax

The syntax of the generated language is defined by the following context free grammar, described in [35-48] and [51-53].

[35] λE.LIST := E | E CONJ E | COMMALIST<E> CONJ E

[36] λE.COMMALIST := E | E COMMA COMMALIST<E>

[37] COMMA := ,

[38] DOT := .

[39] CONJ := and

The fist two rules are parametrized on a syntactic category. They will be used in the following rules to define lists of nouns, adjectives, adverbs and other categories.

[40] TEXT := S | S TEXT

[41] S := LIST<C> DOT

[42] C := NP VP

This rules above simply specify how a text is composed out of sentences (S), and how sentences are composed out of a list of clauses (LIST<C>) followed by a dot (DOT). The constituents of a clause are a noun phrase (NP) and a verb phrase (VP).

[43] NP := PRONOUN (OF NP) | (DET) (LIST<ADJ>) LIST<NOUN> (OF NP)

[44] DET := a | the | some | its

[45] ADJ := OPENADJ

[46] OF := of

[47] PRONOUN := it | some | one

[48] NOUN := thing | OPENNOUN

[49] the Black Pearl cursed crew

[50] a rubber chicken with a pulley in the middle

A noun phrase can either be a pronoun or a sequence of common nouns preceded by a sequence of adjectives, introduced by an optional determiner. Any noun phrase may be followed by another noun phrase introduced by the preposition *of*. This allows for reasonably complex noun phrases describing a set of concepts, but it does not allow to handle more complex noun phrases like [49] or [50]. However, we will later see that these complex noun phrases can still be inserted verbatim in the lexicon, as common nouns: they will be handled by the system as canned text.

[51] VP := VERB OPENBODY (PREP) NP

[52] VERB := have | be | look | OPENVERB

[53] PREP := OF | for | by | in | with | ...

A verb phrase always starts with a verb and always ends with a noun phrase, optionally introduced by a preposition. In between, the grammar allows arbitrary strings, which are not further processed during generation. Note that the grammar does not admit sentences like [54], even though sentences like these could be an elegant text describing a node labeled with two concepts, `black` and `tea`. In such cases, the generation system will have to use a formulation like [55].

[54] The tea is black.

[55] The tea is a black thing.

As these rules show, the grammar already defines a minimal lexicon. We will see in the next section that the lexicon can be extended to meet the specific application needs. Each user-defined lexical element is categorized either as an OPENVERB, an OPENNOUN, or an OPENADJ, depending whether it is a verb, a noun, or an adjective. A fourth category, OPENBODY, can be used to insert arbitrary strings within a verb phrase. Typically, they will be additional complements and/or adverbs.

Altogether these rules define a pretty simple fragment of English, sophisticated enough to handle adjectives properly, and to allow anaphora. Anaphora can be realized using definite articles in place of indefinite ones, or using pronouns instead of nouns. The context-free grammar does not make it explicit, but verbs and noun phrases can be elided. We will discuss ellipsis in the chapter dedicated to natural language generation.

The language uses agreement features to ensure, for example, that pronouns match the noun phrase they refer to, and that the verb form matches the subject of the clause. Agreement features for noun phrase elements are gender, number and person. Because of the structure of the Query Tool queries, anaphoric references can only appear in subject position. For this reason, pronouns do not need case marking features, as their case will always be nominative.

2.3.3 Lexicon

The lexicon of the language is the set of words that can be used to write the text. These words are also known as lexemes. The set of lexemes of our language is not fixed: should new words become necessary, they can be added to the set of available words. The default lexicon of our language is essentially empty. When the knowledge engineers configure the Query Tool to access a particular ontology, they can extend the lexicon with new lexemes. To do so, they can create and load a list of lexical entries. Each lexical entry defines a lexeme by specifying some information regarding how the word changes in different contexts, i.e. given different values of its agreement features.

The lexemes are classified either as closed-class or open-class lexemes. Closed-class lexemes are prepositions, determiners, conjunctions and pronouns. Open-class elements are nouns, adjectives and verbs. The set of closed-class lexemes available is fixed, whereas the set of open-class elements can be extended.

One of the goals of this project is to make life easier for knowledge engineers who want to use Query Tool with their KB. In order to port the GUI to a new interface, the engineers must create new linguistic resources such as a new lexicon and a map from elements of the ontology to lexical elements. This process may be boring and time-consuming, and it could also be non-trivial when the linguistic information needed by the system is complex. For the benefit of the knowledge engineers and for the benefit of the linguistic resource generators, which we'll discuss later on in this thesis, our language is designed to work with minimal linguistic information and ideally with whatever kind of information is available. To this purpose, the Query Tool NLI accepts partially-specified lexical entries. This choice has an impact on the language itself, as some constructs are not available to the generator when some lexical information is missing.

Verbs

In order to support a wide range of verb tenses and moods, the lexical entries for the verbs should be rich in detail: they should include the base form of the verb, the past tense, the past participle and possibly other information. Our lexicon contains verb entries of two kinds: totally-specified verbs and partially-specified verbs. Each verb of the first kind must specify the base form (e.g. *fly*), the past tense (e.g. *flew*)†, the present participle (e.g. *flying*), the past participle (e.g. *flown*) and the simple present active third person singular (e.g. *flies*). Each verb of the second kind must specify a single form introduced by *should* (e.g. *should have been flying*), together with optional† inflection information, i.e. mood, voice, tense and aspect.

The first kind of lexical entry is suitable for use with any form of verb. The second kind of lexical entry is basically canned text, which can be used in any context a verb is needed. The optional inflection information can be used to prevent the generator to use the lexical entry to produce a form of the verb for which there is no support. The choice of verb forms based on *should* was suggested in [Perez 2009] as a solution to context-independent realization of verbs. This feature addresses the need for a simple and flexible tool which makes it easier to port the language to a different domain. These verb forms are not sensitive to the person and number of the subject, and therefore need no further treatment. More information regarding the mood and the voice is however useful in order to treat ellipsis properly.

† Feature not implemented yet in the lexicon import/export module.

Nouns

Each noun lexical entry specifies the singular or the plural form of the noun, or both. This allows to use nouns which do not have both forms, or nouns for which only one form is known. Each noun entry must also specify whether it is countable, i.e. whether it can be introduced by an indefinite article. In case it is, the entry must specify whether `a` instead of `an` should be used. Finally, each noun entry must specify one gender among the possible values (`neuter`, `feminine`, `masculine`).

While verb lexical entries come in two flavours, one to support detailed information and the other one to support partially specified information, the noun lexical entries as we defined them support also canned-text nouns. As long as the user provides all the necessary information to treat them properly, i.e. information about the determiners which can be used and about the gender, a noun lexical entry can support any range of noun phrases, from composite nouns like [56] to noun phrases like [57].

[56] Table tennis

[57] Great Wyrm's holy water sprinkler of the blessed hammer of destiny

Adjectives

Each adjective entry specifies the adjective itself, and whether `a` instead of `an` should be used as an indefinite article in noun phrases where the adjective comes right after the article.

2.4 Templates

The Query Tool natural language interface is portable, i.e. it is not crafted to target any specific knowledge domain. Instead, it is designed to edit queries over any target KB. Before it can be used to query a new KB, it must be provided with a lexicon including all the nouns, verbs and adjectives needed to generate the natural language representation of queries.

However, lexical information alone is not enough to generate text describing queries, because the connection between the concepts and relations in the KB and the lexicon is missing. The system needs information about how each concept and each relation of the ontology mentioned in the query contributes to the syntactic structure and the lexical elements of the generated text.

As explained in §2.3.1, every concept label of a node contributes an adjective or a noun to the noun phrase associated with the node. Every label of an edge contributes a whole clause. The subject of the clause is the noun phrase associated with the node the edge is leaving. The noun phrase associated with the destination node, instead, may become the direct object of the clause, an indirect object, or another component of the verb phrase of the clause.

2.4.1 Concept templates

We named “concept template” the data structure which encodes the contribution of a single concept label to a noun phrase. We designed two kinds of concept templates: noun-based concept templates and adjective-based concept templates.

- A noun-based concept template consists of a reference to a noun lexeme and a value for its number (`singular` or `plural`).
- An adjective-based concept template consists simply of a reference to an adjective lexeme. This template is to be used when a concept contributes an adjective modifying a noun. (E.g. a concept like `yellow`.)

These templates allow to associate concepts only with simple noun phrases, even if our grammar would allow more complex ones. While a more powerful set of concept templates could be useful for real applications, we didn't develop the system further in this direction. In order to create a more complex noun-based template, it is possible to craft a canned-text noun lexical entry that corresponds to the complex noun phrase to be associated with the concept, and use a simple noun-based concept template referring to that lexical entry.

2.4.2 Relation templates

We named “relation template” the data structure which encodes the contribution of a single relation label to a clause. Each relation template contributes the stub of a clause: the clause representing an edge of the query labeled with the relation associated with the relation template. We designed two kinds of relation templates: noun-based relation templates and verb-based relation templates.

The data structure of a verb-based relation template follows the structure of the clause first explained in §2.3.2. A verb-based template consists of:

- A reference to a verb lexeme. This is the main verb of the contributed clause.
- A choice of values for its voice, its tense and its aspect. These are the coordinates needed to realize the referred verb lexeme into a finite verb form.
- A boolean value flagging whether it is to be introduced by `should`.
- A string, which we named the body of the template. As explained in §2.3.2, this string allows for arbitrary canned text to be inserted right after the main verb of the clause. This string may be empty, and will typically contain adverbs, a direct objects, and/or complements.
- An optional preposition.

Collectively, the information stored in a verb-based relation template identifies a finite verb form, optionally followed by an arbitrary string and/or a preposition. The content of the arbitrary string is arbitrary in the sense that no grammaticality check is performed on it. It is up to the knowledge engineers who craft the template to avoid using strings that would not make sense combined with the rest of the clause.

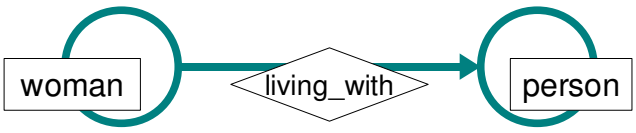

Illustration 9: A fragment of a query

Like all relation templates, a verb-based relation template contributes the stub of a clause. The intended subject of this clause is the noun phrase associated with the node the edge the relation is labeling is leaving. Illustration 9 displays a fragment of a query consisting of two nodes connected by an edge. The edge is labeled with the relation `living_with`, it is leaving the node labeled with the concept `woman`, and it is

entering the node labeled with the concept person. Let us suppose that the relation living_with is mapped to the verb-based relation template displayed in table 1. The clause associated with this fragment of query is [58], where the subject is the noun phrase associated with the node labeled with woman.

[58] The woman should be living with a person.

Ontology element	Associated template		
living_with	**verb**	**base**	live
		3rd person present	lives
		simple past	lived
		present participle	living
		past participle	lived
	voice		active
	tense		present
	aspect		progressive
	should		TRUE
	body		–
	preposition		with

Table 1: Fragment of a template map containing the template associated with the relation living_with

A verb-based template is flexible enough to handle almost all the ways a relation can contribute to a clause representing the connection between two nodes. In §2.3.1, we presented a sample of textual expressions representing relations, which we repeat here for convenience.

[59] I am looking for a flat **opposite** a park.

[60] I am looking for a food **typical** of a country.

[61] I am looking for a woman **mother** of a teacher.

[62] I am looking for a shop **selling** shirts.

[63] I am looking for a person who **has been co-authoring a book with** a Nobel laureate.

The following sentences, [64-68] show how the same expressions can be accommodated to fit into a verb-based template. In each of the following sentences, the contribution of the relation is highlighted with a bold typeface, while the underlined segment of text is the body of the verb-based relation template associated with the relation.

[64] The flat **should be opposite** a park.

[65] The food **should be typical of** a country.

[66] The woman **should be the mother of** a teacher.

[67] The shop **should be selling** shirts.

[68] The person **should have been co-authoring a book with** a Nobel laureate.

However, there is a class of expressions describing relations which cannot be modeled using the verb-based relation template as we defined it. Examples of such expressions are [69, 70, 71].

[69] The **mother** of the teacher should be a woman.

[70] The **official language** of the country should be a Semitic language.

[71] A **branch** of the company should be located in an African country.

In these expressions, the contribution of the relation is highlighted with a bold typeface: in [69] the originating relation could be `has_mother`, in [70] it could be `official_language`, in [71] it could be `has_branch`. In all of them the subject of the clause is not the noun phrase associated with the node the edge is leaving. Instead, the subject is a complex noun phrase which contains, in one of its branches, the noun phrase associated with that node.

Note that while [71] can be reformulated to fit the format of a verb-based relation template, like in [72], there is no clean way to reformulate [69] and [70] to the same end.

[72] The company should have a branch located in an African country.

That there is a difference between [69, 70] and [71]: in the first two, the noun contributed by the relation is introduced by a definite article, `the`; [71] instead is introduced by an indefinite article, `a`. This difference corresponds to the relation being functional (one is not expected to have more than one mother) or not (a company may have more than one branch).

In order to handle this class of relations, we designed a different kind of relation templates, which we named noun-based relation templates. Noun-based relation templates are similar to noun-based concept templates, in that they also contain a reference to the lexical entry of a noun, and a value for its number (`singular` or `plural`). However, the contribution of a noun-based relation template is the stub of a clause. In this clause, the verb is a simple present indicative `should be`. The subject of the clause is the main contribution of the relation: it is a complex noun phrase built around the noun designated by the template. The noun phrase associated with the node the edge is leaving still plays a relation in the clause: it is a child of the complex noun phrase, introduced by the proposition `of`.

In addition to the information already mentioned, each noun-based relation template contains a boolean flag used to specify whether the relation is functional or not. In case the relation is functional, the subject of the clause contributed by the noun-based relation template will be introduced by the definite article `the`, like in [73, 74, 75].

[73] The mother of the actor should be **...**

[74] The official language of the country should be **...**

[75] The eye of the Cyclops should be **...**

Ontology element	Associated template		
ship	**noun**	**singular**	ship
		plural	–
		countable	TRUE
		indefinite article	a
		gender	female
	number		singular
produced_in_collaboration_with	**verb**	**base**	–
		3rd person present	–
		simple past	–
		present participle	–
		past participle	produced
	voice		passive
	tense		present
	aspect		Simple
	should		FALSE
	body		in collaboration
	preposition		with

Table 2: Template map containing mappings for a concept and for a relation

In case the relation is not functional, the subject of the clause is depends on whether the noun-based template is plural or not, and whether the noun specified in it is uncountable or not. If the noun specified in the noun-based template is countable, the subject is the noun itself, introduced by an indefinite article, like in [76, 77, 78]

[76] A mother of the actor should be **...**

[77] An official language of the country should be **...**

[78] An eye of the pirate should be **...**

If the noun is uncountable, the subject will be an indefinite pronoun, some, followed by the original noun introduced by of, like in [79, 80, 81].

[79] Some of the knowledge of the system is **...** (inferred, domain-specific)

[80] Some of the music of the album is **...** (classical, jazz, rock)

[81] Some of the software of the company is ... (proprietary, shareware, open-source)

If the noun-based template specifies the noun to be plural, the subject will be an indefinite pronoun, one, followed by the original noun introduced by of, like in [82, 83, 84].

[82] One of the mothers of the actor should be ...

[83] One of the official languages of the country should be ...

[84] One of the eyes of the pirate should be ...

2.4.3 Template maps

In our system, the gap between the ontology and the lexicon is filled by a map which associates each concept to a concept template, and each relation to a relation template. We will refer to this map simply as “the template map”.

Table 2 illustrates a fragment of a sample template map. The sample contains the mapping of two ontology elements: a concept and a relation. The templates contain references to lexical elements, but the neither the templates nor the template map contain any lexical information. In the table, the referenced elements are the verb and the noun. The information contained in the referenced elements is shown for illustration purposes, highlighted with a dark background.

2.5 Generation

Our work on this part of the system is essentially a revision and an enhancement of the natural language generator documented in [Trevisan 2009]. There, we described a simple generation system based on templates similar to the ones we presented in the previous section, and generating a language similar to the one presented in §2.3.2.

For this description we adopted [Reiter & Dale 200] as the reference architectural model. The procedure to generate text from the input is divided into three modules, each module carrying out orderly a number of different tasks.

- The document planning module performs the content determination and the document structuring task.
- The microplanning module performs the lexicalization, the referring expression generation, and the aggregation.
- The surface realization module performs the linguistic realization and the structure realization.

In the following, we will explain and discuss each of these modules and tasks as they are realized in our system.

2.5.1 Document Planning

In general, the document planning module of a NLG system selects the information to be conveyed in the generated text (content determination), its order and its structure (document structuring). The output of the module is a document plan, consisting in a data structure containing messages, the atomic elements of information.

In our system, the input to this module is a Query Tool query as we defined it in §1.1.2 with the additional constraint that each node must have at least one label. When the input query contains a node with no labels, a new label referring to the concept `Top` is inserted at this stage of the process.

The requirements for the Query Tool GUI specify the pieces of information to be conveyed in the text and their presentation order in function of the input query. In our system, we identify two kinds of messages: the information provided by a node label and the information provided by an edge label. According to the requirements all labels should contribute to the meaning of the generated text, in the order they would appear by traversing the tree in pre-order (see §2.1).

The requirements do not enforce any specific structure for the document. Our observation is that the query is declarative in nature: our system only needs to develop it into a series of statements that elaborate the declared information need. Many NLG systems use Rhetorical Structure Theory to specify the structure of the text to be produced. Using the terms of the RST framework, the structure of our text would be a chain of elaborations of the initial statement which introducing the query.

This analysis of the document planning problem justifies our choice for the data structure for the document plan: a list of messages, each message being a node label or an edge label. To produce a document plan, the document planner simply traverses the query in a pre-order: starting from the root, each node is processed first collecting all the node labels in their order, and for each outgoing edge, in the order they appear, the edge label is collected and the destination node is visited recursively.

2.5.2 Microplanning

In the architecture described in [Reiter & Dale 2000], the microplanning module of the NLG system fills the gap between the document plan, which specifies the messages to be conveyed and their order, and the text specification, which specifies in full detail the syntactic structure of the final text including lexical information. Another module, the surface realizer, will transform the text specification into proper plain text.

In our generation pipeline, the microplanning process is the most complex, because at this point all the information which so far was stored on separate layers blends into a single structure. This process is subdivided into three tasks: the lexicalization, the aggregation and referring expressions generation. In our system, the tasks are performed in this order.

2.5.3 Lexicalization

The lexicalization procedure builds the bulk of the text specification, along with some auxiliary data structures needed for the subsequent microplanning tasks.

The procedure begins with the creation of an empty text specification. The text specification is a tree of abstract syntactic specifications. Abstract syntactic specifications are the NLG equivalent of syntactic constituents in Feature Grammars. For example, the abstract specification (AS for short) of a common noun consists of a reference to a common noun lexeme plus a feature structure containing information such as the number, the person, the case marking. Given an AS of any syntactic constituent, the surface realizer can compute its surface form straightforwardly.

The procedure also creates an empty semantic map. The map will store the association between syntactic elements of the text specification and labels of the query. The procedure also creates a map between noun phrases and nodes of the query. This map will be later used in the referring expression generation and aggregation tasks.

The actual processing of the document plan begins by associating two noun phrases (NP) to each node of the query: the introductory NP and the referring NP. We called this sub-task “lexicalization of a node”. After all nodes have been lexicalized, the whole tree is lexicalized, filling the text specification with clauses built by instantiating the templates associated with the elements of the document plan. The lexicalization of nodes and the lexicalization of the tree share a common subtask, which we discuss here shortly before describing the two lexicalization sub-tasks.

Template instantiation

The template instantiation procedure produces a syntactic element using a template as a blueprint. Depending on the kind of template, the syntactic element is different. For concept templates, the output is a noun phrase. For relation templates, the output is a verb phrase missing a noun phrase.

When the input is a noun-based concept template, the output noun phrase is composed of a determiner AS and a common noun AS. The determiner is `a/an` when the grammatical number specified in the template is singular and the lexeme is countable; otherwise, the determiner is `some`. The common noun is an instance of the common noun lexeme specified in the template, with the same grammatical number specified in the template.

When the input is an adjective-based concept template, the output noun phrase is composed of a determiner AS, a common noun AS and an adjective AS. The determiner is always `a/an`, the common noun is an instance of the lexeme `thing`, with singular grammatical number. The adjective is an instance of the adjective lexeme specified in the template.

When the input is a noun-based relation template, the output is an incomplete noun phrase. It may be composed either of a determiner AS, a common noun AS, a preposition AS, like in [85, 86]. Otherwise it may be composed of a pronoun AS and an incomplete propositional phrase AS composed of a preposition AS, a determiner AS, a common noun AS, and another preposition AS, like in [87, 88]. The first configuration is returned when the common noun specified in the template is singular and countable; otherwise, the second configuration is returned.

The first determiner is either definite or indefinite when the template is functional or not, respectively. The second determiner, when present, is always definite. In this context, each preposition AS is an instance of the lexeme `of`. The common noun AS is an instance of the common noun lexeme specified in the template, with the same grammatical number specified in the template. The pronoun AS is `some` or `one` when the lexeme specified in the template is uncountable or countable, respectively.

[85] The official language of **...**

[86] An official language of **...**

[87] Some of the music of **...**

[88] One of the official languages of **...**

When the input is a verb-based relation template, the output is an incomplete verb phrase. The verb phrase is composed of a verb AS, a piece of canned text and an optional preposition AS. The verb is an instance of the verb lexeme specified in the template, the canned text is the body of the template, and the proposition, if present, is an instance of the preposition lexeme specified in the template. The verb is instantiated considering all the details provided in the template: the voice, the tense, the aspect, and whether it must be introduced by should or not.

Lexicalizing a node

The lexicalization of a node is the first sub-step of the lexicalization. The purpose of this sub-step is to produce, for each node of the query, two noun phrases, called the introductory NP and the referring NP of the node. The introductory NP is a NP which can be used to refer, for the first time in the text, to the object identified by the node. Using the terminology of [Reiter & Dale 2000], the introductory NP is used for the *initial reference* to the node. The introductory NP conveys as much information about the node as it is possible using a single common noun and as many adjectives as it is feasible. Labels associated with noun-based templates which are not represented in the introductory NP will be represented by other NPs in a different sentence.

The referring NP is a NP which can be used for all other references to the object identified by the node. Using the terminology of [Reiter & Dale 2000], the referring NP is used for *subsequent references* to the node. It is the basis for the generation of referring expressions: the referring expressions generator step will use this NP to model pronouns referring to the same object.

The sub-step processes all the node label messages associated with the input node in the order given in the query. For each such label, the system fetches from the template map the concept template associated with the label. In general, some of these labels are associated with noun-based templates, other labels are associated with adjective-based templates.

When this procedure fetches a noun-based template for the first time, the template is instantiated into a NP, and the common noun of this NP is taken to be the base noun. If all the concept templates are adjective-based, the base noun is taken to be `thing`. The introductory NP is produced as a NP composed of an indefinite article, all the adjectives produced instantiating all the adjective-based templates associated with the other labels, and the base noun. The referring NP is produced as a NP composed of a definite article and the base noun. In case all the templates are adjective-based, the referring NP is produced as a NP composed of a definite article, the adjective obtained instantiating the first template, and the base noun.

For example, consider [89], the list labels of a node. Let's also suppose that the first, second and third concepts are associated with adjective-based templates, while the other two concepts are associated with noun-based templates.

[89] Muslim filmDirector Russian singerSongwriter

According to our definition, the introductory NP for the node is [90], and the referring NP is [91].

[90] a Muslim Russian film director

[91] the film director

All the labels used to produce such NPs are marked as used, in order to avoid using them again to produce sentences in the next step, that is, tree lexicalization.

Lexicalizing the tree

Once each node has been lexicalized, the lexicalization process continues by generating one clause for each message which has not been used yet. For each node label of each node, the lexicalizer generates a clause. The subject of the clause is a copy of the referring NP of the node. The verb phrase of the clause is composed of the copula verb *should be*, and, as subject complement, the NP instantiating the template associated with the concept referred to in the node label.

For each edge outgoing a node X and entering a node Y, the lexicalizer generates a clause. Depending on the kind of template associated with the relation referred to in the edge label, the lexicalizer produces a different kind of clause.

- If the template is verb-based, the subject of the clause is a copy of the referring NP of the node X. The verb phrase of the clause is produced instantiating the template associated with the relation referred to in the edge label. The verb phrase obtained in this way is incomplete: it lacks a NP to be complete. To fill this gap, the lexicalizer uses the introductory NP associated with the node Y.
- If the template is noun-based, the subject of the clause is the NP obtained instantiating the template itself. The noun phrase obtained in this way is incomplete: it lacks a NP to be complete. To fill this gap, the lexicalizer uses the referring NP associated with the node X. The verb phrase of the clause is composed of the copula verb *should be*, and, as subject complement, the introductory NP associated with the node Y.

The first clause of the query is always half-canned, as the final text should start with [92]. The generator uses the introductory NP associated with the root node of the query to complete the clause.

[92] *I am looking for* **...**

Each of the clauses generated in the whole lexicalization process are nested in a single sentence. The next step of the microplanning will merge consecutive sentences where feasible.

Semanticization

During the microplanning stage, while the system builds the text specification, the microplanner links syntactic elements to the elements of the query they describe. There are several reasons to do so.

- The first reason is to inform the referring expressions generator whether a noun phrase is referring to an entity which was already introduced. For this reason, we want to connect each noun phrase describing a node to the node itself.
- The second reason is to inform the aggregator whether the subjects of two consecutive clauses are references to the same logical entity. This is the first condition for the aggregation of the clauses.
- The third reason is, the generator must return a one-to-one association between the query elements and some portions of the text, as specified in the requirements. Thus, the microplanner must connect each node label and each edge label

to some syntactic element of the output text, so that the surface realizer will be able to tell which portion of the final generated text is associated with each label.

For these motivations, the microplanner builds, along with the text specification, a map which connects various elements of the text specification to nodes and labels of the query. We named this map the semantic map. The semantic map connects each noun phrase to at most one node. Some noun phrases may be not connected to any node. When a noun phrase is generated during the lexicalization of a node or during the lexicalization of a node label, the system connects that noun phrase to that node.

The semantic map also connects each single edge label and node label to a single specific contiguous portion of the text. To this aim, our microplanner maps each syntactic constituent of each AS obtained by template instantiation with the edge or node label used to retrieve the template. The result is that even when, during the aggregation process, the AS are dismembered and parts of them are discarded, the single constituents that make it to the final text specification are still connected to the semantic elements that originated them. When surface realization will transform AS into text, this information will be carried on to the final text.

2.5.4 Aggregation

The aggregation step analyses the list of clauses produced in the lexicalization step in order to join adjacent clauses with the same subject into a single sentence, and to elide syntactic elements wherever it is possible to do so.

In our aggregation module, the first condition for any kind of aggregation between two consecutive clauses is that the subjects of the clauses refer to the same node. If this condition is not met, no aggregation will take place. Once this condition is met, the aggregator merges the two clauses into a single sentence, appending the second clause to the list of clauses the first clause belongs to. The clauses in the list are coordinated using a comma or a conjunction, as specified in the generative grammar (rule 41).

If the verbs of the clauses have the same voice, i.e. they are both passive or both active, the subject of the second clause is elided. (E.g. [93, 94, 95].) This elision also occurs when a clause with a passive verb follows a clause with a copula. (E.g. [96].)

[93] The actor should be a singer and the father of a boy.

[94] The film should be produced by a Bollywood company and directed by a Kannadiga artist.

[95] The firm should manufacture goods and provide services.

[96] The book should be a novel and should be written by a woman.

After the subject is elided, it may be the case that further ellipsis is possible. If both clauses have a copula and a subject predicate, the copula of the second clause is elided. (E.g. [93].)

If the clauses are both active or both passive, some elements of the verb of the second clause may be elided. The aggregator will consider the all the elements that make up the verb of the second clause, against all the elements that make up the final verb form of the first clause, in the order they appear. If the two lists of elements share a prefix, the elements in the second clause that are part of this prefix are elided. For example the aggregation of [97] and [98] produces [95], where the auxiliary should is elided.

[97] The firm should manufacture goods.

[98] The firm should provide services.

The elements considered in this operation are the atomic components of the verbs. These may be finite and infinite verb forms, including auxiliary verbs. For example, in [99] the elements of the verb are those listed in [100]. Note that these elements are sub-components of the verb phrase AS because they are sub-components of the verb of the verb phrase. In other NLG systems, the full list of auxiliaries of a verb may be not available until surface realization. This is not the case in our system.

[99] The car should be produced in an European Country.

[100] [should, be, produced]

When eliding the suffix would cause all elements of the verb to disappear, the aggregator makes sure that at least one syntactic constituent semantically linked to the edge label remains not elided. Eliding all references to a edge label would make the contribution of the edge label disappear from the generated text, and thus to a violation of the requirements. When there are no other elements in a verb phrase associated with the edge label besides the verb, the aggregator will avoid eliding all the elements of the verb. Instead, it will elide at most all of them but one.

2.5.5 Referring Expressions Generation

The aim of the referring expressions generation step is to replace noun phrases used as subsequent references with more appropriate expressions. These expressions are more appropriate in the sense that they should be just as informative as needed for the reader to understand what semantic entity the expression refers to. In general, the problem of referring expression generation is hard, since it requires to consider what entities the reader considers relevant in the context, and what characteristics of the entity are useful to disambiguate the reference. For our system, we adopted a naïve approach: subsequent references to the same node are realized as pronouns whenever they co-refer with the subject of the previous clause. Otherwise, they are realized using the referring NP of the node. For example, [101] is turned into [102], but [103] is left unchanged. We introduced an exception to this rule to accommodate the introductory clause, so that [104] becomes [105].

[101] The actor should be a singer and the actor should live in an Indian city.

[102] The actor should be a singer and he should live in an Indian city.

[103] The actor should be married to a singer. The singer should be Muslim. The actor should live in an Indian city.

[104] I am looking for a woman. The woman should an actress.

[105] I am looking for a woman. It should be an actress.

Note that when the reference appears within a prepositional phrase attached to another noun phrase, instead of turning the reference into a pronoun, the propositional phrase is replaced by a possessive determiner attached to the noun phrase. For example, [106] is turned into [107], and [108] is turned into [109].

[106] The mother of the singer should be an Indian.

[107] His mother should be an Indian.

[108] One of the parents of the actress should be an Indian.

[109] One of her parents should be an Indian.

This naïve approach fails when the reference is ambiguous. This may be the case when a pronoun matches more than one noun phrase in the previous clause (e.g. [110]), or when the referring NP of two nodes are identical (e.g. [111]).

[110] The girl$_1$ should live with a woman$_2$. She$_1$ should like strawberries.

[111] The woman$_1$ should live with an Indian woman$_2$. The woman$_2$ should like strawberries.

We acknowledge these limits, but for the purposes of our experiment we believe that this approach is sufficient.

2.5.6 Surface Realization

Surface realization is the task of transforming a text specification into a data structure called surface form. The surface form contains the final product of the generation, that is, text. In general, the surface form encapsulates the text into a structure convenient for the presentation layer of the software, which is going to display the text to the user. Depending on this layer, the surface form may be a as simple as a sequence of characters, or an HTML document, or another kind of data structure.

In our system, the surface form must convey the text plus an association between some spans of text and some elements of the query. The structure we chose is a list of cells, each cell containing a fragment of text, and each possibly containing a reference to a label of the query. The concatenation of the contents of the cells is the generated text.

Our surface realizer navigates through the list of sentence ASs which make up the text specification, and fills up the list of cells by visiting the components of each sentence AS and recursively visiting the sub-component ASs of each AS. ASs which do not have sub-component AS are transformed into a cell. For example, each common noun AS is transformed into the content of a cell using the common noun lexeme and the grammatical agreement information stored in the AS. If the semantic map connected the AS to a query label, a reference to the label is also stored in the cell. Other ASs are processed in a similar way. The surface realizer also takes care of capitalization and spacing. In detail, the first character of the content of the first cell of each sentence is capitalized, and a space between two cells is always present except when the following cell is a punctuation mark.

3 Knowledge domain portability

Most natural language interfaces to databases are bound to a specific knowledge domain, that is, they are built to handle questions regarding a specific topic, e.g. flight routes, chemical compounds, real estates [Androutsopoulos et al. 1995]. A NLI is portable when it can be adapted for use in different knowledge domains. To this purpose, the system must be adapted to cope with all the words and expressions that are typical of the target domain. To connect a portable NLI to a database in a new domain, it is usually necessary to associate linguistic resources to the elements of the database schema. For example, it may be necessary to associate to each class of elements in the database a set of synonyms which may be used to refer to elements of that class. The exact nature of the adaptation depends on the specific NLI, and it may range from modifying the source code to using a simple dedicated software to associate parts of the database schema to existing lexical resources

As we already explained in many occasions, in order to configure the Query Tool natural language interface to query a KB, the software engineer must produce a template map for the KB's ontology. This task requires a basic knowledge of some linguistic concepts such as the voice of a verb, the past participle of a verb, and others. In our vision, the software engineer would produce the template map using a software utility: this utility would allow the engineer to browse the ontology and fill a template for each concept and each relation, in the same spirit of Masque's domain editor, as presented in [Androutsopoulos et al. 1995, p. 26]. The tool may also support the engineer by displaying a sample sentence obtained using the information in the template. Ideally, we would like the system to fill the template automatically, but this task is too hard in general to accomplish. Instead, we would like to provide a tool capable of suggesting to the engineer a selection of automatically generated templates. For each relation, and for each concept, the engineer would be presented with a series of sentences generated using these templates, and given the chance of selecting one of them. The system would assign to the relation or to the concept the template used to generate the selected sentence. In the following two chapters, we will describe two experiments with some techniques that could provide a kick-start towards this challenging goal.

3.1 Data-driven extraction of linguistic information

Unlike the next approach, which focuses on extracting templates from linguistic information available in relation identifiers, our first experiment aims at collecting additional linguistic information describing a target relation. This information could be useful to fill in a template manually, or automatically using the technique we describe in the next section. In a nutshell the approach is the following: we collect sentences which mention two entities which, in the KB, are connected through the target relation; from the parse trees of these sentences, we extract the most frequent paths. These paths contain the linguistic information we were looking for.

For example, suppose that we are looking for linguistic information about a relation R whose id is `filmdirector`. Note, incidentally, that this relation id cannot be success-

fully handled using the rule-based approach described in the previous chapter, because the tokenizer cannot cope with it. Suppose also that the KB contains some couples (a, b) ∈ R, that is to say, the KB contains information about some films and about their directors. Finally, suppose that for each element of these couples there is at least one string in the KB which can be used in natural language text to refer to the element: for example, the title of the film, or the name of the film's director. We will refer any of these strings as a natural language label (NLL) of the element.

Our approach consists in the following: for each couple instantiating the relation, we look into a corpus and collect all the sentences which contain natural language labels (NLLs) of both the elements of the couple. Using couple of NLLs like [1] we collect sentences like [2]. We parse the collected sentences using a dependency parser. We obtain a list of parse trees like [3], also depicted in illustration 10.

[1] (Jogi, Prem)

[2] Jogi (Kannada: ಜೋಗಿ) is a Kannada-language movie, directed by Prem and released on August 22, 2005.†

[3]
```
nsubj(movie-9, Jogi-1)
      dep(Jogi-1, Kannada-3)
cop(movie-9, is-6)
det(movie-9, a-7)
amod(movie-9, Kannada-language-8)
dep(movie-9, directed-11)
      prep(directed-11, by-12)
            pobj(by-12, Prem-13)
      cc(directed-11, and-14)
      conj(directed-11, released-15)
            prep(released-15, on-16)
                  pobj(on-16, August-17)
                        num(August-17, 22-18)
                        num(August-17, 2005-20)
```

For each parse tree, we discard the branches that do not contain any one of the two nlls. We obtain the path connecting the two leaves containing the two NLLs. For example, from [1] we obtain [4], also depicted in illustration 11. Leaving out the leaves we eventually get [5].

[4]
```
nsubj(movie-9, Jogi-1)
dep(movie-9, directed-11)
      prep(directed-11, by-12)
            pobj(by-12, Prem-13)
```

[5]
```
dep(movie-9, directed-11)
      prep(directed-11, by-12)
```

From [5] we can extract the string movie directed by, which is the linguistic information we were looking for. This information could be returned to the user to help him lexicalize a relation whose identifier is obscure, or it could be given as input to the the rule-based template extractor. However, [5] is one among many the paths obtained in this process, many of which are useless to our purpose. For example, the same process applied to sentence [6], leads to the path [7], from which we extract the

† Source: Wikipedia, the Free Encyclopedia, en.wikipedia.org/wiki/Jogi, retrieved on 16-08-2009.

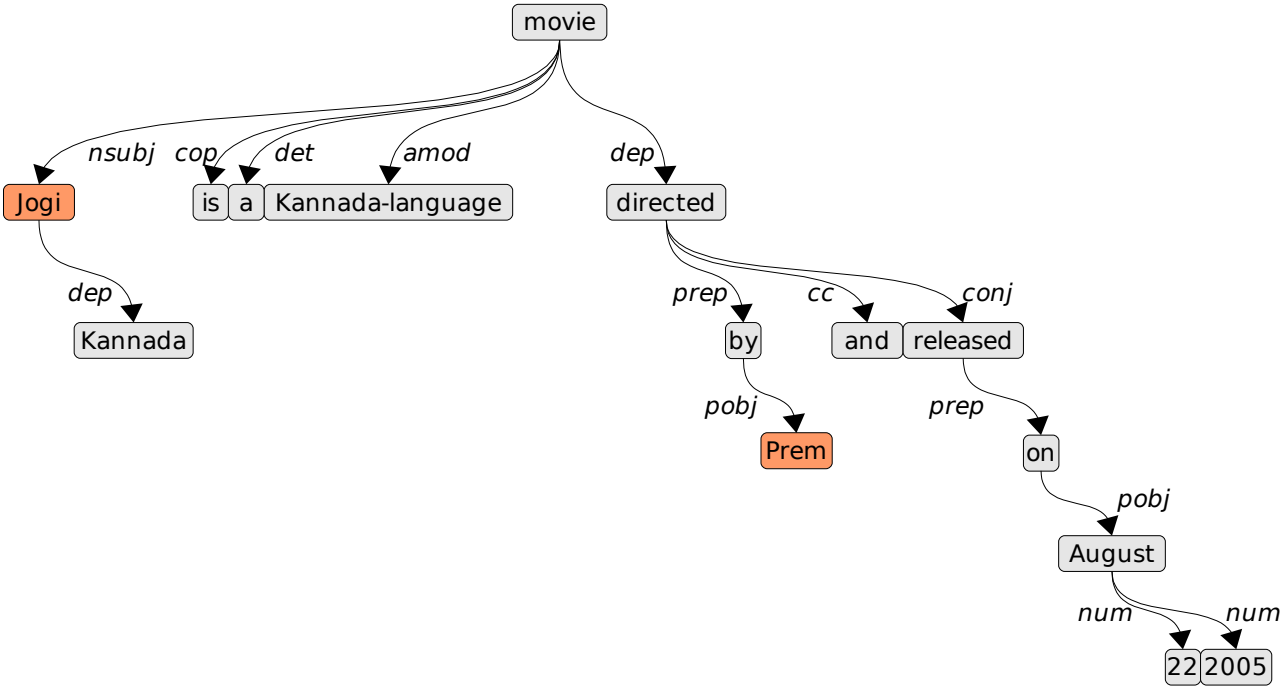

Illustration 10: Dependency parse tree

string `a story from`, which while still meaningful, does not convey the notion of film director.

[6] `Pithamagan (2003) A story from Bala after the successes of Sethu and Nanda, which starred Vikram and Surya in respectively.`

[7] `det(story-4, A-3)`
`prep(story-4, from-5)`

In order to retrieve only the relevant paths, we apply an algorithm to score each path according to the frequency of some of its features. The whole procedure returns the best scoring paths, and discards the rest. It happens that [5] is among the worst scoring paths, according to our algorithm. Instead, among the best scoring ones appears `film directed by`, possibly because the British English term is occurring more often in the retrieved sentences than its American English equivalent.

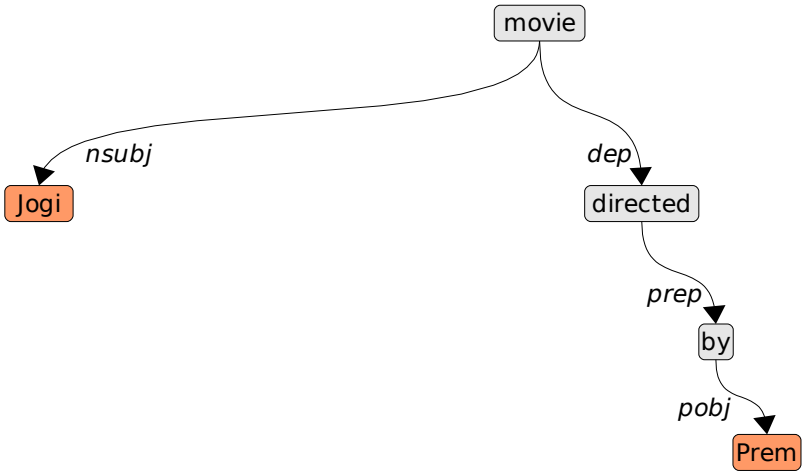

Illustration 11: Dependency path

3.1.1 Algorithm

In the following we describe the whole procedure in greater detail and in a broader context. This procedure was introduced in [Ittoo & Bouma 2009], as a way to gather lexicalized syntactic patters instantiating meronymy (part-whole) relations. We applied the same approach to gather lexicalized syntactic patterns instantiating arbitrary relations. The algorithm described here is conceptually the same used for [Ittoo & Bouma 2009], manipulated for a better presentation. The single divergence from their algorithm is the ranking of paths, which we'll discuss later.

```
1    collect (R, C, K, nll, N)
2      seeds := collectSeeds(R, K, nll)
3      paths := collectPaths(seeds, C)
4      ranked_paths := rankPaths(paths)
5      printed_best_paths := printPaths(ranked_paths[1..N])
6      return printed_best_paths
```

Text 1: Pseudocode for the data-driven linguistic information miner

Given a relation R, a corpus C, a knowledge base K, a labelling function nll(x), and the result set size N (a natural number), the algorithm returns a list of at most N strings. The pseudocode shown in text 1 breaks down the algorithm into four major steps: seed collection (line 2), path collection (line 3), path ranking (line 4) and path printing (line 5). The seed collection step is the gathering of a list of single-worded seeds for the target relation. A single-worded seed for a relation R is a couple (nll(a), nll(b)) such that (a, b) ∈ R, nll(a) ≠ nll(b), and both nll(a) and nll(b) are strings of alphanumeric characters without spaces. The pseudocode shown in text 2 shows how our algorithm collects seeds, given a relation R, a knowledge base K, and a labelling function nll.

The function nll(x) associates every object in the KB with a string named natural language label (NLL) which can be used to identify the object in a text. Such a function is to be provided by the software engineer who is configuring the natural language interface for use with a new KB. Ideally, the associated string is the most frequent among those which identify uniquely the object. In practice, there is a trade-off between precision and frequency. For example the 45th president of the U.S.A. may be referred to using any one of these expressions:

- the 45th president of the United States
- president George W. Bush
- president Bush
- George W. Bush
- the U.S. president
- the president
- he

Except the first one, all of them may be used to refer to people other than him. The first two NLLs are precise but not frequent in a general-purpose corpus. The last two

```
1    collectSeeds(R, K, nll)
2      for each (a, b) ∈ R in K
3        if nll(a) != nll(b)
4          and nll(a) is a single-worded seed
5          and nll(b) is a single-worded seed
6        then
7          seeds := seeds + [(a, b)]
8      return seeds
```

Text 2: pseudocode for seed collection

are frequent, but they seldom refer to him. In such a situation, we would recommend to choose one of the strings in the middle of the list.

However, such a situation is rather unusual. Instead, many of the elements in a KB have no obvious NLL, or they have a NLL which is seldom used in text. For example the rooms of a building, computers and other technical equipment, flights, planes, and trains. While a plane certainly has a NLL used in technical communications (usually an alphanumeric code), such a label is not found in available general-purpose corpora. It could be useful if the engineer could provide a corpus which contains these NLLs. If no useful NLL can be associated with an object of the KB, the object is useless to our approach.

While in principle our approach could work with any kind of NLL, the solution we designed requires each NLL used in a seed to consist of only one (alphanumeric) word. For example, none of the NL proposed for president Bush above is acceptable by our system (except `he`). The reason we restricted our attention to single-worded NLLs is that multi-word NLLs are infrequent: the longer the NLL is, the harder it is to encounter it in a corpus. Instead, it is more likely that the entity is referred to using expressions derived from the NLL, such as abbreviations or paraphrases, or simply using anaphors. We attempted at expanding the set of NLLs for an entity by generating expressions using words from a single initial multi-word NLL associated the entity, but it didn't help. We also tried to detect anaphors referring to the entity, without success. Eventually, we decided to focus on single-worded NLLs. This meant that the relations our system can process is a fraction of the relations contained in the KB.

Once all seeds have been collected, the algorithm proceeds by parsing sentences containing those seeds, to collect depedency paths. Text 3 shows in pseudocode how these paths are collected given a list of seeds and a corpus C. For each sentence in the corpus, for each seed `(nll(a), nll(b))`, the system checks whether the sentence contains both `nll(a)` and `nll(b)`. If that is the case, our system uses the Stanford parser to parse it (line 7)[8]. The resulting parse tree is a Stanford typed dependencies representation [de Marneffe & Manning 2008], such as the ones depicted in illustration 10 and 11. It consists of a directed tree where each node is labelled with a word of the sentence, and each arc is labelled with a dependency relation. From this tree, the algorithm extracts (line 8) the shortest path containing the node labelled with `nll(a)` and the node labelled with `nll(b)`. The algorithm discards the edge labels, the direction of the edges of this path (line 8), and the nodes labeled with `nll(a)` and `nll(b)`. Paths of length zero are also discarded. The algorithm appends this path to a list (line 9), which is eventually returned to the caller (line 10).

8 For our evaluation experiments we relied on a treebank so we did not need to parse each sentence.

```
1    collectPaths(seeds, C)
2      for each sentence in C
3        for each (a, b) in seeds
4          if sentence contains nll(a)
5            and sentence contains nll(b)
6          then
7            tree := StanfordParse(sentence)
8            path := extractPath(tree, nll(a), nll(b))
9            paths := paths + [path]
10     return paths
```

Text 3: pseudocode for path collection

The original algorithm used in [Ittoo & Bouma 2009] discarded infrequent paths. This simple approach was reasonable for their purposes, as they had to filter a single, large collection of paths. Our situation is different, because for the majority of relations few paths are collected, if any (see the following subsection, Evaluation). Therefore, we tried to improve the ranking algorithm to make the best use of the few available relations. Ranking paths using their frequency has a pitfall, illustrated in the following example. Suppose that the paths [8], [9] and [10] appear with an absolute frequency of 5, 4 and 3 respectively. According to frequency ranking, [8] is better than the other two. However, our intuition is that [10] is a variation of [9], and we would like the frequency of [10] to contribute to promote the rank of [9] above the rank of [8].

[8] (album)-(released)-(on)-(label)

[9] (album)-(released)-(by)

[10] (album)-(released)-(by)-(records)

```
1    rankPaths(paths)
2      for each path in paths
3        allsubpaths := allsubpaths + extractSubpaths(path)
4      for each path in allsubpaths
5        freq[path] := freq[path] + 1
6      for each path in paths
7        subpaths := extractSubpaths(path)
8        for each subpath in subpaths
9           composite_f[path] := composite_f[path]+ freq[subpath]
10       score[path] := score[path] / size(subpaths)
11     sort(paths, score)
12     return paths
```

Text 4: pseudocode for path ranking

Text 5 is the pseudocode of our ranking algorithm, which differs from the original in that it addresses this issue. The algorithm takes as input a list of paths, and returns it re-ordered. To do so, it first collects all the sub-paths of paths in the input list (line 3), and it computes the absolute frequency of each such sub-path (line 5). For example, if

Subpath	Absolute Frequency
(album)	3
(released)	3
(on)	1
(label)	1
(album)-(released)	3
(released)-(on)	1
(on)-(label)	1
(album)-(released)-(on)	1
(released)-(on)-(label)	1
(album)-(released)-(on)-(label)	1
(by)	2
(released)-(by)	2
(album)-(released)-(by)	2
(records)	1
(by)-(records)	1
(released)-(by)-(records)	1
(album)-(released)-(by)-(records)	1

Table 3: Subpaths and their frequency

the input list contains the three paths listed above, the algorithm produces the set of sub-paths displayed in table 3 together with their frequencies. Once the frequencies of individual sub-paths are available, the algorithm assigns to each path a value called composite frequency, computed as the sum of the frequencies of the path's sub-paths (line 9), normalized dividing by the number of those sub-paths (line 10). This composite frequency is used to sort the paths in descending order.

For example, this algorithm assigns the following composite frequencies to the three paths above:

[11] (3+3+1+1+3+1+1+1+1+1) / 10 = 1.6 for [8]

[12] (3+3+2+3+2+1) / 6 = 2.33 for [9]

[13] (3+3+2+1+3+2+1 +2+1+1) / 10 = 1.9 for [10]

The ranking obtained is therefore [9], [10], [8]. In the evaluation part, we will refer to this method of ranking the paths as “the method based on composite frequency”. We will compare this method against using one similar to the one used in [Ittoo &

Bouma 2009]. We will refer to this second method method as “the method based on simple frequency”.

The last step of the procedure (line 5 in text 1) transforms the first N of the ranked paths into strings, by concatenating the content of the nodes, interleaving them with spaces. This list of strings is the output of the whole procedure.

3.1.2 Evaluation

The goal of our approach is to retrieve a small set of strings that the software engineer can use to fill in a relation template, or, in alternative, that our rule-based template extractor can process. In the following, we will refer to each such string as a textual representation of the relation. At the very core, for each input relation, our procedure is successful when among the retrieved expressions there is one that describes the relation correctly. The first metric of evaluation is therefore the frequency of relations for which our approach produces a list of strings which contains a feasible one.

This measure gives an overall idea of our achievement, but it does not give hints about what are the most problematic tasks, which parts of the solution can be improved, and which parts can't. Therefore we also evaluate our system on partial solutions: the relation between the number of couples and the number of retrieved paths, the relation between the number of retrieved paths and success, the relation between number of couples and success, the precision and recall of the filtering procedure.

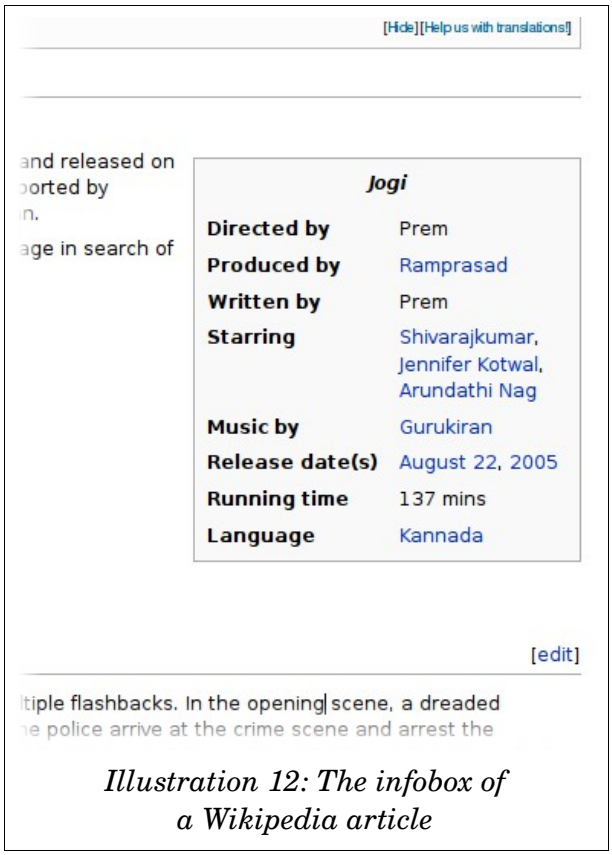

Illustration 12: The infobox of a Wikipedia article

We evaluated the system using a portion of Wikipedia as our corpus, and the Wikipedia infoboxes as our KB. The Wikipedia infoboxes is a collection of semi-structured data originating from the infoboxes of Wikipedia articles. This experiment owes much to [Wu & Weld 2008], who first employed data contained in wikipedia infoboxes to find patterns in the text using this data. In Wikipedia, the infobox of an article is a frame containing schematic information about the subject of the article, presented as labeled strings or hyperlinks. Illustration 12 showcases an infobox. These frames are inserted as simple text by Wikipedia editors, possibly borrowing them from similar articles, and possibly customizing them for the subject of the article. In this sense, they are loosely structured. Each infobox has a template name, not rendered in the page, but accessible in the source, that categorizes the subject as a person, film, city, actor, or similar. For our purposes, each value in an infobox represents the instance of a relation in the KB. For example, the infobox of the Indian film Jogi contains the value `Prem` under the label `directed_by`. This translates into the couple $(a, b) \in R$, where a is the movie titled `Jogi`, b is the person named `Prem`, and R is the relation represented by the label `direc-`

`ted_by`. Since the label with the same name may appear for in the templates of different kinds of articles, to identify a single relation we take the label together with the name of the template, which in this case is `film`.

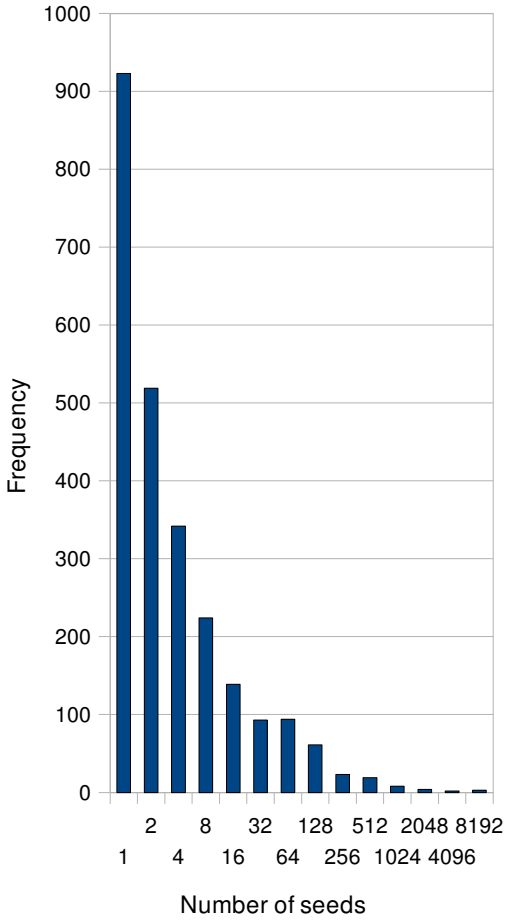

Illustration 13: Frequency distribution of the number of the seeds of a relation

We restricted our attention to couples `(a, b)` such that for each element of the couple there exists an article in Wikipedia with the same name. In other words, the elements of our KB are Wikipedia articles, and our `nll` function returns the title of the article. We did so in order to avoid considering relations which link object to data. Our expectation is that for most data values, like `137 mins` (the running time of Jogi), there is no corresponding Wikipedia article, whereas most Wikipedia articles correspond to entities that would be represented as objects in a KB. While this is the case for most articles, there are exceptions: for example, dates in Wikipedia have a corresponding article (e.g. the release date of Jogi).

The full infoboxes dataset contains more than seven millions couples, instantiating altogether more than 61 000 relations. Filtering out couples that contain an element without an associated article in Wikipedia left us with 1 868 784 instances for 19 617 relations. Moreover, since the solution we adopted is limited to handle only single-word seeds, the number of useful instances decreases to 127 644, for 2454 relations. These figures mean that our approach is limited to handle 12.6% of the 9 617 relations in our KB. For the remaining relations, the system will not retrieve any textual representation. Illustration 13 shows the frequency distribution of the number of seeds of a relation in the 127 644-seeds dataset, rounded down to a power of two. More than half of the relations have only one, two or three seeds, and more than two thirds of the relations have less than 8 seeds.

To save computational time, we retrieved textual representations for 1324 of the 2454 relations. Our target corpus was a portion of Wikipedia containing around 16 millions lines of text. For each of these relations, our system mined the corpus to find sentences containing both the words of a seed of the relation, finding 337 416 sentences in total.

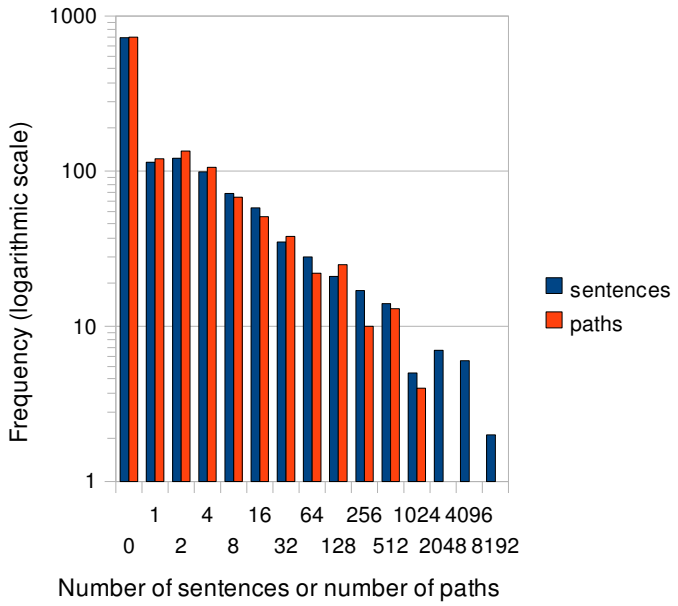

Illustration 14: Frequency distribution of the number of sentences and paths for a relation

Illustration 14 shows (in blue) the frequency distribution of the number of sentences retrieved for a relation, rounded down to a power of two. For 725 (55%) relations, no sentences containing a seed were found. For each of the 1324 relations, our system extracted a list of dependency paths from the sentences collected for that relation, collecting in total 38 746 paths. The figure is smaller than the number of sentences, as different sentences may contain the same path. Illustration 14 shows (in red) the frequency distribution of the number of paths collected for a relation. The graph shows (on the left) that most frequently, no sentences and no paths were found for the relation. On the right, a for a small number of relations (less than twenty) more than one thousands sentences were found, but these sentences often contained the same path.

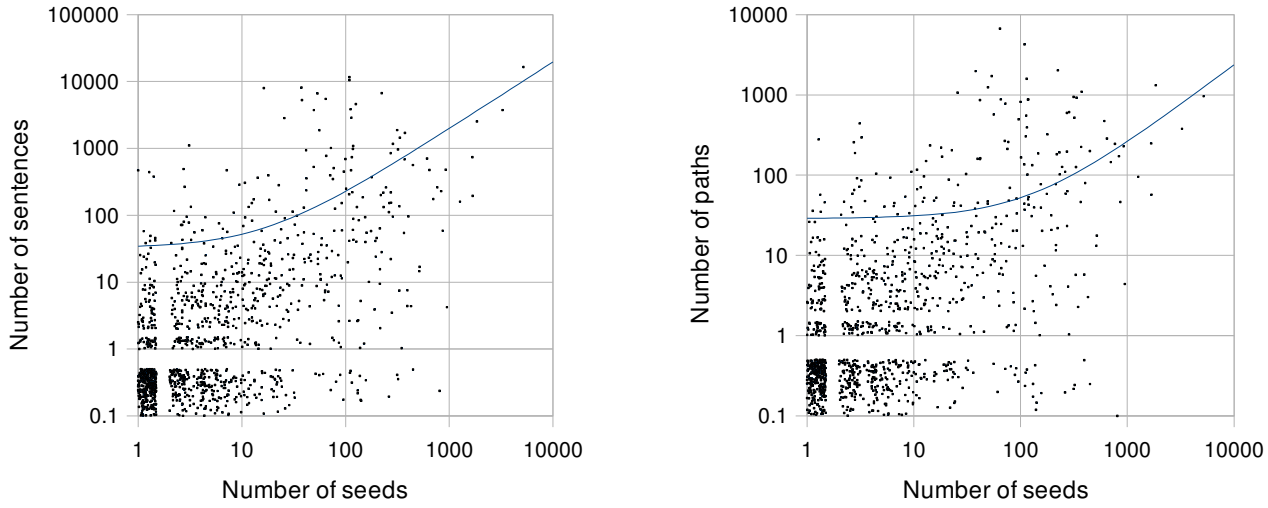

Illustration 15: Relations scattered on the real plane according to the number of their seeds and the number of their sentences and paths

Illustration 15 gives an impression of the relation between the number of seeds for a relation and the number of sentences and paths for a relation: every relation is represented by a dot. In the picture, the dots have been scattered with noise in order to avoid overlapping: this allows to visualize the frequency of the data in certain zones of the plane according to the density of the dots. The graphs display also a linear regression of the data. In our dataset, the number of sentences (and paths) is on average greater than the number of seeds used to collect them. For many relations however, no sentences (or paths) were collected, regardless of the number of seeds available.

We evaluated the path filtering algorithm on a sample of 77 relations for which a sentence was collected. We inspected the paths collected for each of these relations, to identify all those paths that our system should retrieve (the positive paths). Of these 77 relations, 21 have been left out from the sample because it was impossible for us to determine whether a positive path existed. This happened because for these relations, both the name and the available seeds were not expressive enough for understand the meaning of the relation. Of the remaining 56, 34 relations (61%) had a positive path. Illustration 16 shows how the number of positive paths grows with respect to the total number of paths (positive and negative). The graph does not include the paths for which no positive was found or identified.

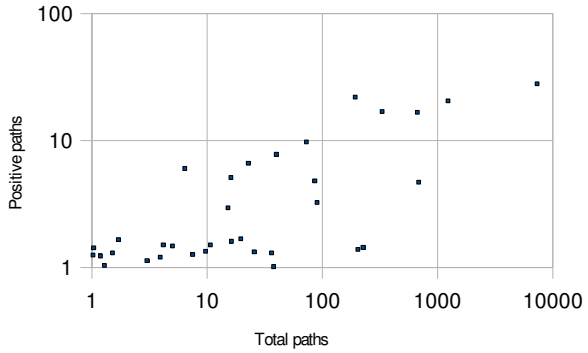

Illustration 16: Relations scattered on the real plane according to the number of their positive paths and the total number of their paths.

We evaluated our approach based on composite frequency against two baselines, using three metrics: precision, recall, and a custom metric called score. We measured these values using only the relations for which a positive path existed. By their definition, precision and recall cannot be calculated taking into consideration relations for which no positive path existed. The score of a result set is zero when the result set contains no positive path. The score is one otherwise. This metric aims at measuring how often the user will find at least one suitable textual representation among those proposed by our system. The first baseline system orders the paths randomly, the second one using simple frequency. Illustration 17 shows the precision and recall of both systems for result sets of size ranging from one to eleven. The greater the result set, the greater the recall. Illustration 18 shows the mean score.

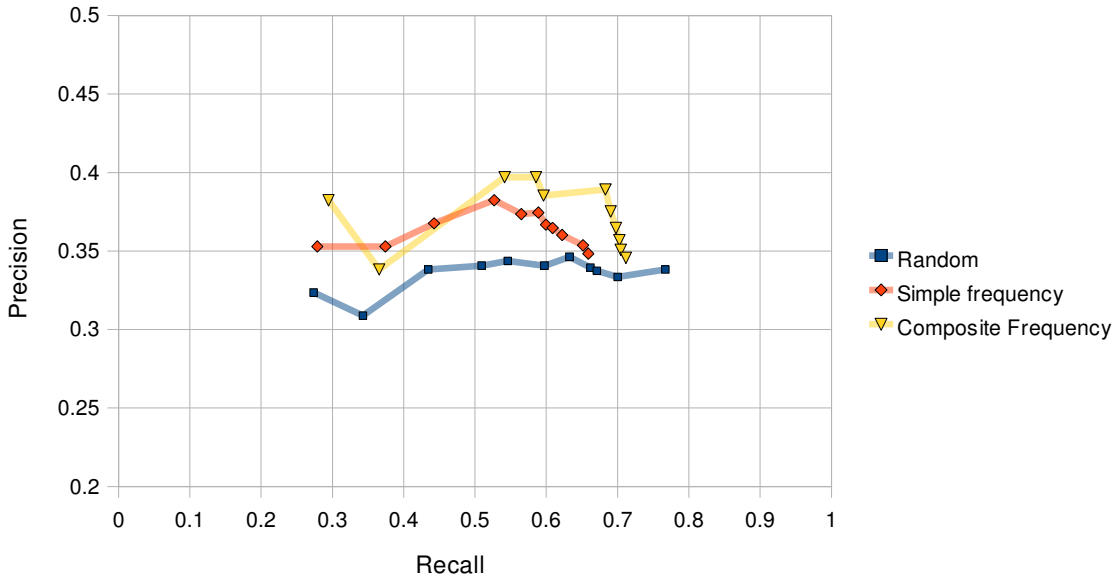

Illustration 17: Mean precision and recall on all the relation ids with a positive path

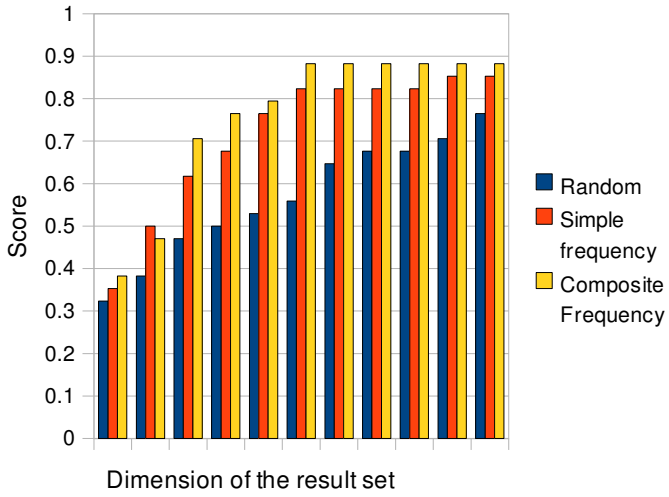

Illustration 18: Mean score on all the relation ids with a positive path

We can recap the results of the evaluation taking into accout the limitations we assumed so far. We limited our scope all the relations for which at least one single-word seed was collected. These amount to 12.6% of the relations in our corpus. We considered a sample of 2454 relations. For 599 (45%) of them, at least one sentence containing a seed was collected. We considered a sample of 77 of the 599 relations. We limited our attention on the 56 relations we could understand, and we found out that for 34 (61%) of them, one or more of the collected sentences contained a positive path. When asked to retrieve five paths representing one of these 34 relations, in 80% of the cases, our ranking algorithm returns one or more of these positive paths.

These figures suggest that:

- in 22% (0.45 * 0.61 * 0.80 = 0.22) of the cases, our system will return a list of at most five expressions, one or more of them being useful;
- in 55% of the cases, it will return no suggestions;

- in the remaining 27% (0.45 * (0.61 * 0.20 + 0.39) = 0.22) of the cases, it will return a list of at most five useless expressions.

Let us recall here that the relations targeted by our system are the relations R for which the KB contains a couple (a, b) ∈ R where nll(a) and nll(b) are single words, which in our dataset consist of 12.6% of the relations.

3.2 Rule-based template extraction

The second approach we experimented with draws from Chris Mellish and Xiantang Sun's Triple-Text NLG system [Sun & Mellish 2007]. The idea is based on the observation that KBs already contain linguistic information. In real-world ontologies, every concept, and every relation, has a unique identifier (ID), and most of the times it 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. These IDs possibly contain all the information needed to generate natural language descriptions of the identified elements [Mellish & Sun 2005]. Xiantang Sun's proposed dissertation [Sun 2009] discusses a procedure to extract syntactically rich templates from relation IDs. We adapted this procedure to better fit our data, but the underlying idea is the same: tokenize the relation ID, tag it with a part-of-speech tagger, and apply pattern-based rules to transform the tokenized, tagged relation ID into a relation template. In our approach, each relation ID is first tokenized according to an algorithm which 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 ([Tufis & Mason 1998], as cited in [Sun 2009]). The tagged tokenized ID is then lightly preprocessed and finally fed to a transformation rule, chosen among thirteen different rules, that produces a relation template.

relation ID	Sample generated sentence
altitude[9]	Its altitude should be a negative number.
communicationDeclined	It should have declined an official communication.
connectedAtContact	It should be connected at something.
constantRenamed	It should have been renamed with a Polish name
numberOfLevels	Its number of levels should be a number greater than ten.
parent_face	Its parent face should be a yellow face.
relation_Has_Domain	Its domain should be an abstract class.
tastes	It should taste bitter.
wnDerivedFromAdjective	It should derive from an adjective of time.

Table 4: Sample of templates extracted from relation IDs

9 In this chapter, unless otherwise specified, relation IDs are taken from our corpus.

In order to design an effective method following this idea, we built a corpus of almost 200 ontologies, collecting several of those freely available on the Web. We collected only ontologies which contained English words in the relation IDs. Ontologies which systematically assign arbitrary sequences of letters and digits to relation IDs were discarded. From these ontologies we extracted more than 12000 unique relation IDs.

Table 4 shows a sample of relation IDs from our corpus along with a sentence the Query Tool natural language interface should generate on the basis of the template we would like to programmatically associate to the identified relations. We anticipate that the rule-based template extraction system can actually handle most of the relation IDs displayed in the table, and extract templates contributing clauses similar to those displayed in the table. One notable exception is the target template for the `constantRenamed`: the target clause for this relation contains a preposition, `with`, that cannot be extracted from the relation ID. In this chapter, we explain how our extraction process works, following in parallel, and contrasting with, Sun's approach to the same task.

3.2.1 Tokenization

The first step in the process from a raw relation ID to a relation template, is the tokenization, that is, the transformation of the input relation ID into a sequence of alphanumeric strings. The task is relatively easy because ontology IDs usually contain only alphanumeric characters, possibly underscore, and no punctuation: ontology languages usually restrict the alphabet available for use in concept and relation IDs. Our tokenizer splits the input string at each non-alphanumeric character (e.g. `film_director`), after a letter following a digit (e.g. `40weeks`), after a digit following a letter (e.g. `Rambo2`), after an upper-case letter following a lower-case letter (e.g. `movieDirector`). Moreover, the input string is split after an upper-case letter following an upper-case letter when the former is preceding a lower-case letter (e.g. `USSenator`, `POSTagger`).

```
13   tokenize(a string S of length N)
14     for i in [0..N-1]
15       if S[i] is not a letter nor a digit
16         split between S[i-1] and S[i+1] and discard S[i]
17       else if S[i] is a digit, i > 0 and S[i-1] is not a digit
18         split between S[i-1] and S[i]
19       else if S[i] is not a digit, i > 0 and S[i-1] is a digit
20         split between S[i-1] and S[i]
21       else if S[i] is an upper-case letter, i>0 and S[i-1] is
     not an upper-case letter
22         split between S[i-1] and S[i]
23       else if S[i] is an upper-case letter, 0 < i < N-1, S[i-1]
     is an upper-case letter and S[i+1] is a lower-case letter
24         split between S[i-1] and S[i]
```

Text 5: Pseudo-code for our tokenizer

The pseudo-code of our tokenizer is shown in text 5. Our approach is rather simple, and it is easy to see that it fails to handle correctly strings with digits mixed with letters, like 3D or 22nd. Moreover, hyphenated words like e-mail or brother-in-law will be transformed in sequences of unrelated words. However, we believe that this limitation does not impact our solution substantially. Table 5 shows a sample of relation IDs, selected from our corpus, and the corresponding tokenized relations IDs according to our tokenizer.

relation ID	Tokenized relation ID
communicationDeclined	communication Declined
relation_Has_Domain	relation Has Domain
topicPattern_Irrelevant	topic Pattern Irrelevant

Table 5: Sample input and output of our tokenizer

3.2.2 Part-of-Speech Tagger

After the relation ID has been tokenized, the next step in the process is to assign a part-of-speech tag to each token. To this purpose, we rely on QTAG as Sun did, preprocessing the input to improve the accuracy on the particular kind of input we are interested in. QTAG is a PoS tagger trained on the Penn treebank, and it performs worse on our data than it would on traditional text. Our data consists of sequences of words without determiners and with traces of English syntax: they resemble noun or verb phrases, but they usually miss some elements to be grammatical. [Sun 2009] discusses in detail the pre- and post- processing of the input and output of QTAG to improve its precision for this class of input.

Sun's corpus of relation IDs differs substantially from ours: in his corpus, half (49.5%) of the relation IDs start either with has or with is. In our corpus, these relations make up less than 3% of the total. We suspected that our particular dataset demanded for a different customization of QTAG. We worked on Sun's customizations to handle some specific issues we encountered in a small sample of our dataset. To design these specific customizations, we selected a random sample of 200 relation IDs from our dataset and we hand-tagged it. After that, we compared the output of QTAG, and the output of our implementation of Sun's customization of QTAG, with our hand-crafted tags, in order to get directions on how to improve the tagger. These observations led the development of the customizations we are going to describe.

Our custom PoS tagger incrementally transforms the tokenized string depending on the feedback of QTAG. The algorithm starts by invoking QTAG to tag the initial tokenized string, and it proceeds applying the four following rules, one by one, in order. After every application, QTAG is invoked to tag the transformed tokenized string, and the result is the input for the next step. After the four rules have been applied, the final tag assignment is returned.

- If the tagged string contains tokens which are tagged as unk[10], those tokens are replaced with an unambiguous common noun (e.g. story). This is a temporary re-

10 QTAG uses the tag ??? to mark unknown words. We modified the tagset of QTAG replacing ??? with unk. In this document, all tag are written in lower case, using a sans-serif font.

placement that should help disambiguate the subsequent tokens; the unk tags will be restored in the final output tags. Unknown tags are frequently assigned to our data because the relation IDs often contain abbreviations and terminology which is highly domain dependent, as most of the ontologies represent the formalized, shared, expert's knowledge on a narrow, specialistic domain. We follow Sun's suggestion in treating these unknown words as nouns.

- If the tagged string contains a nns tag T, followed by one of [nn, nns, np, nps, cd, in, to], T is tested to find whether it could be tagged vbz, by inserting it into a test context (It T). If the test reveals that the token can be tagged as vbz, the token is replaced with an unambiguous third-person-singular present tense verb (e.g. brings). This test aims at recovering third-person-singular present tense verbs that QTAG wrongly classifies as plural nouns. The rule relies on our intuition that composite nouns involve nouns in their singular form more often than in their plural form.
- If the tagged string contains tokens tagged as vbd, those tokens are replaced with unambiguous past participle verbs (e.g. forgotten). This rule relies on the observation that ontologies rarely model events within their historical dimension: rather, they are used to model timeless states of affairs. This is especially true with databases, which usually model a snapshot of the current state of affairs rather than a history of the world (e.g. Bob is married to Alice versus Tom has married Alice). Under this assumption, past participle forms are more likely to occur than past tenses.
- If the tagged string contains a token tagged as vbn preceded by a token tagged as jj, jjs, jjr, od[11], those tokens are replaced with an unambiguous adjective (e.g. yellow). This rule relies on the fact that in such situations the past participle is used as an adjective, and the template extractor does not need to distinguish between participial and non-participial adjectives, while it useful for our purposes to distinguish between adjectival and non-adjectival participles.

Tokenized relation ID	Tagged relation ID
communication Declined	communication:nn Declined:vbn
relation Has Domain	relation:nn Has:hvz Domain:nn
topic Pattern Irrelevant	topic:nn Pattern:nn Irrelevant:jj

Table 6: Sample input and output of the PoS tagger

Table 6 shows a sample of tokenized relation IDs and the tagged tokenized relation IDs obtained from them. In the rest of this chapter we will refer to tokenized, tagged relation IDs as TT-IDs. When representing TT-IDs in text, the PoS tag assigned to each token will be displayed right after the token, preceded by a colon.

To evaluate the taggers, we hand-tagged two further samples of a hundred and two hundred relation IDs, randomly selected from our corpus. We evaluated three PoS taggers on these samples: the basic QTAG, our implementation of Sun's customized QTAG, and our variation of Sun's customization. For the purpose of this evaluation, any unk tags output by the taggers was replaced with a nn tag. Table 7 shows the

11 od is a QTAG tag for ordinal number

precision of these taggers on two samples of relation IDs randomly selected from our corpus.

	QTAG	Sun's custom QTAG (our implementation)	Our custom QTAG
Sample 1 (100 relation IDs)	0.8	0.77	0.83
Sample 2 (200 relation IDs)	0.89	0.9	0.93

Table 7: Precision of different PoS taggers

3.2.3 Partitioning the set of relation IDs

The rule-based template extractor takes as input a TT-ID and returns a relation template filled with strings extracted from the input. The idea behind this extractor is the same idea behind Sun's approach: relation IDs may be not grammatically correct with respect to an English grammar, but certain syntactic patterns occur more frequently than others. For example, [Sun & Mellish 2007] report that, in their corpus, 5% of all the relation IDs consist of a verb followed by a preposition. For some syntactic patterns occurring in his corpus, [Sun 2009] designed a rule to transform each relation matching the pattern into a template for the generation. The algorithm to extract templates from tagged relation IDs consists simply of applying the rule associated with the syntactic pattern instantiated by the relation ID. relation IDs that do not instantiate any of the syntactic patterns covered by the transformation rules are handled by a default rule.

relation class by syntactic pattern	Sun's corpus	Our corpus
Starting with has	37.2%	2.4%
Starting with is	12.3%	0.2%
Ending with a preposition	11.3%	9%
Only one word	18%	5%
Only two words	12.5%	21%
All the rest	8.3%	61%

Table 8: Frequency distribution of the classes of the first level of Sun's partition of relation IDs by syntactic pattern

This simple idea works as long as relation IDs with similar syntactic patterns can be transformed in a similar way: designing a transformation rule for a pattern means giving an interpretation to all the relation IDs instantiating that specific pattern. For some syntactic patterns, it may be the case that different relation IDs matching the pattern need to be transformed in radically different ways. For example, while all relations consisting of a single noun can probably be transformed in the same way (e.g. mother, director, birthday, sister, branch, customer), the same is not true for all relations ending with a preposition (e.g. towards, produced by, belongs to, director of, can access

†o). In order to design an effective solution following this approach, we must partition the set of relation IDs into classes of relations can be transformed in the same way. Moreover, each class of relation IDs must be defined in terms of syntactic patterns.

Table 8 shows the six pairwise-disjoint[12] top-level classes defined by Sun for use in his system, along with the relative size of those classes in his corpus and in our corpus. Sun refines these top-level classes into 16 pairwise-disjoint subclasses, shown in table 9. These classes do not correspond directly to a rule, as Sun defines twenty-two rules (plus one fall-back rule). As table 9 shows, the second-level partition of syntactic patterns defined by Sun captures a small fraction (22%) of our corpus.

relation class by syntactic pattern	Sun's corpus	Our corpus
is + ... + preposition	11.58%	0.01%
is + ... + adjective	0.33%	0.05%
is + ... + noun	0.74%	0.16%
is + to + verb	0.08%	0.0%
has + ... + noun	36.45%	2.16%
has + ... + preposition	0.57%	0.18%
has + ... + adjective	0.16%	0.05%
NP + ... + prep	5.83%	0%
adj + prep	0.82%	0.4%
adj	0.99%	0.4%
noun	14.2%	3.82%
verb	1.15%	0.68%
prep+noun	0.33%	0.41%
noun+noun	9.77%	10.40%
verb+noun	0.9%	2.48%
past participle+prop	0.49%	0.5%
all the rest	15.6%	78%

Table 9: Frequency distribution of the classes of the second level of Sun's partition of relation IDs by syntactic pattern

This observation, plus the fact that Sun's rules are not documented in [Sun 2009], led us to define a different set of rules, and hence a different partition of the input set, based on different syntactic patterns. We defined thirteen pairwise-disjoint classes of relation IDs, corresponding to thirteen rules, described in the rest of this chapter. Table 10 shows the frequency distribution of these classes in our corpus.

12 Note that the classes are: for example, relation IDs ending with a proposition are actually relation IDs not starting with "has", not starting with "is", and ending with a proposition.

Some of the classes we defined generalize some of Sun's second level classes. For example, our BEZ classes subsumes all Sun's “is+..”classes. Similarly, Our HVZ classes subsume all Sun's “has+..”classes.

Table 10 already shows the names of the transformation rules, but before we can present the rules one by one in detail, we need to introduce some preliminary, general information about them. Each of our transformation rules is a function with an input, a guard condition, and an output. The input of a rule is a tokenized, tagged and preprocessed relation ID. The guard condition is a condition on the input which must be met for the rule to be applied. The output is a relation template, as we defined it in §2.4.2. In the following sections we will first discuss in detail the input of the rules, in particular the preprocessing operations that come right after the PoS tagging stage and right before the input is fed to the transformation rules. Then, we will briefly explain the pattern matching language we use to describe the guard condition. Next, we will characterize the output of the transformation rules. Finally, we will present the rules one by one.

relation class by rule name	Cumulative Frequency	Mean Frequency
BEZ	0.31%	3.35%
BEZ Passive	0.04%	0.19%
HVZ Range	3.09%	40.73%
HVZ	0.46%	0.67%
VBZ	5.56%	3.55%
VB	0.53%	0.36%
VBN be	3.58%	2.05%
VBN have	6.78%	0.79%
VBG	1.05%	0.06%
No verbs, no nouns	2.43%	3.64%
Simple NP	38.75%	30.27%
Composite NP	20.95%	7.88%
Dangling adjective	0.75%	0.03%
All the rest	15.72%	6.43%

Table 10: Frequency distribution, in our corpus, of the classes of our partition

3.2.4 Input preprocessing

Before a TT-ID can be fed to a rule, it must be preprocessed by assessing some general admission requirements and performing two preliminary transformations. The preprocessing stages are the following, in this sequence:

- discard long input
- discard input with unsupported tags
- remove dangling adverbs

- remove range information

Discarding input

As the length of TT-IDs increases, the syntactic structure becomes more complex, and our approach tends to fail. Instead of returning a useless relation template, we chose to discard long input altogether, to avoid producing useless templates that the knowledge engineers, when they inspect the generated templates, will have to remove nevertheless. Accepted TT-IDs consist of at most eight tagged tokens.

For the same reason, we discard any input TT-ID containing a tag which is not among the supported ones. The set of supported tags is [1]. In the following parts, we will also refer to specific families of tags, namely verbs (elements of [2]), prepositions (elements of [3]), nouns (elements of [4]), adjectives (elements of [5]), and adverbs (elements of [6]).

[1] `Tags := {bez, hvz, vbd, vbz, vb, vbn, vbg, to, in, nn, nns, np, nps, cd, unk, jj, jjs, jjr, od, rb}`

[2] `Verbs := {bez, hvz, vbd, vbz, vb, vbn, vbg}`

[3] `Prepositions := {to, in}`

[4] `Nouns := {nn, nns, np, nps, cd, unk}`

[5] `Adjectives := {jj, jjs, jjr, od}`

[6] `Adverbs := {rb}`

Removing dangling adverbs

When the input TT-ID fulfils the admission criteria, the first preprocessing operation to apply on it is to remove adverb tokens which are not attached to any noun. For example, we want to transform [7] into [8].

[7] `manages:vbz Region:nn Directly:rb`

[8] `manages:vbz Region:nn`

In our system, we assume that this kind of adverbs are to be attached to the main verb of the clause contributed by the relation. To this end, these tokens have to be included in the relation template: more precisely, they should be appended at the beginning of the body of the template, so that they come right after the verb. Later on in the detailed description of each of the transformation rules we will use the function `adverbs()` to retrieve the list of adverbs removed in this way.

The method we designed to collect dangling adverbs is a simple pattern matching on the tagged tokens. An token tagged `rb` is **not** a dangling adverb when it is followed by a token tagged `nn`, `nns`, `np`, `nps`, `cd` or `unk`. Between the two tokens, any number of tokens tagged `rb` or `jj` may appear.

Removing range information

Many relation IDs contain information about the range of the relation, i.e. information regarding the kind of objects appearing on the right side of the connections in-

stantiating the relation[13]. For example, [9] specifies that the range of the relation is an organization.

[9] governor:nn for:in organization:nn

Information about the range of a relation may be useful to knowledge engineers to distinguish relations which would otherwise have the same name. For our purposes this information is useless (in most cases), and it should be removed: we want to transform [9] into [10].

[10] governor:nn for:in

The transformation we designed to this end simply removes all the tagged tokens appearing after the last preposition token or verb token, whichever comes last. This approach is simple and certainly not perfect. Sometimes it may remove a part of the TT-ID which should not be removed. For example, in [11] the whole noun phrase represents the relation, but our transformation turns it in [12], losing the original meaning. Sometimes the transformation may remove only a part of the TT-ID which should be removed. For example, in [13] the noun phrase body of water is characterizing the range of the relation. The purpose of the relation is to model the relation between a region and a body of water the region is the drainage basin of. Our transformation turns it in [14], obfuscating the original meaning.

[11] Country:nn Of:in Nationality:nn

[12] Country:nn Of:in

[13] basin:nn Of:in Body:nn Of:in Water:nn

[14] basin:nn Of:in Body:nn Of:in

As it was the case for the dangling adverbs, the tokens removed as range information are stored for later use, and they are available to the transformation rules through the function `range()`.

As an aside, many relation IDs contain information about the domain of the relation, i.e. information regarding the kind of objects appearing on the left side of the connections instantiating the relation. For example in [15], the domain of the relation is the set of all the warrants, while the range is the set of persons.

[15] warrant:nn Is:bez For:in The:det Arrest:nn Of:nn

In our design, it is up to the transformation rules to detect and discard this kind of information.

3.2.5 Pattern matching on TT-IDs

The description of our transformation rules that will follow relies on pattern matching to describe the guard condition of the rule, as well as to refer to specific tokens or sub-sequences of the input TT-ID to produce the output. Note that these patterns are simply a design tool: the system implementation does not use any dedicated pattern matching language to realize the transformation rules. The syntax of our patterns uses the syntax introduced in §3.1.2 to represent tagged tokens, plus a specific syntax to represent lists. A list of elements is represented using square brackets, with elements separated with commas. The + symbol represents the concatenation operator.

13 We assume that each connection instantiating a relation is directed from left to right.

The patterns we used in these rules are similar in functionality to regular expressions. Any TT-ID is classified by the pattern as matching or not. If the TT-ID matches the pattern, portions of the TT-ID matching variables in the pattern are accessible using the name of the variable as a handle. The equivalent service in regular expressions is provided by groups defined by parentheses.

The simplest pattern on a TT-ID is a single (list-) variable, like [16]: any TT-ID matches it, and the whole TT-ID will be stored in the only variable `Foo`. All variable names, and only variable names, start with an upper-case letter.

```
[16]    Foo
```

The purpose of our patterns is matching TT-IDs that contain tokens with specific tags in specific position. The next pattern, [17] matches all TT-IDs which start with a plural common noun (`nns`). The token itself will be stored in the (token-) variable `Foo`, while the rest of the TT-ID (i.e. the original TT-ID minus the first token) will be stored in the variable `Bar`.

```
[17]    [Foo:nns] + Bar
```

The next pattern, [18], is matching TT-IDs that contain a token tagged as a comparative adjective (`jjr`) followed by a plural common noun (`nns`). The tokens may appear anywhere in the TT-ID, including at the beginning or at the end. The variables `Foo` and `Bar` may match the empty list. In case the matching TT-ID contains more than a couple of contiguous tokens tagged `jjr` and `nns`, the variables `Adj` and `Noun` will store the elements of the couple whose elements occur in the list before the elements of the other couples.

```
[18]    Foo + [Adj:jjr, Noun:nns] + Bar
```

Patterns may specify that an element in a list is optional. [19] matches all the TT-IDs that start with a token tagged `nns`, and in this sense it is equivalent to [17]. However, if the second token is tagged `in`, it will be stored in the variable `Baz`. Otherwise, `Baz` will be undefined.

```
[19]    [Foo:nns, Baz:in?] + Bar
```

The examples presented so far cover all the details needed to understand the patterns used in the guard conditions of the transformation rules.

3.2.6 Output of the transformation rules

Before we move to present the transformation rules, a note on their output is necessary. Each rule must ultimately transform a TT-ID into a relation template. We introduced relation templates in §2.4.2, where we represented them using tables. In the following, we will use a more compact syntax to describe them. A verb-based relation template will be presented as in [20]: there, each variable represents a lexeme (`Verb` and `Proposition`), or an agreement value (`Voice`, `Tense` and `Aspect`), or a string (`Body`). For example, the verb-based relation template described in table 1 of section §2.4.2 will be represented as [21], where ϵ is the symbol for the empty string. References to specific lexemes will be represented with using an underlined font (e.g. be, have). In case the optional preposition is omitted, we will signal this fact using the symbol $\varnothing$.

```
[20]    should Verb Voice:Tense:Aspect Body Preposition
```

```
[21]    should live active:present:progressive ϵ with
```

We will use a similar syntax for noun-based relation templates, with the difference that the noun lexeme is surrounded by a more evocative context, which gives an idea of the structure of the clause contributed by the template. A noun-based relation template will be presented as in [22]: there, the variable Noun contains a lexeme and the variable Number contains an agreement value.

[22] The Noun Number of the … should be

The transformation rules always assume that the relation is functional. This translates to the fact that the subject of the clause contributed by the template will always be introduced by a definite article, the. In principle it could be possible to retrieve information about the functionality of the relation from the ontology where the relations originate from, but we left this feature aside as an open issue.

3.2.7 Transformation rules

For a rule to be used, its guard condition must be satisfied. The guard conditions are designed to guarantee that no TT-ID will satisfy the guard conditions of two different rules at the same time. Therefore, the template extraction algorithm just considers each of the transformation rules, in their order: if the input TT-ID satisfies the guard condition of a rule, the algorithm applies the rule, and returns the output relation template.

In the following, we describe each of the thirteen transformation rules. For each rule, we will describe its purpose informally using examples, and we will schematically describe the guard condition and the output relation template. The guard condition is represented as a list of distinct constraints, all of which must hold at the same time for the guard condition to be met.

The output relation templates, noun-based or verb-based, contain a reference to a lexical entry, respectively a noun or a verb. These lexical entries are selected, among the ones available in the lexicon, to match a specific token (or list of tokens) of the input TT-ID, designated by the transformation rule. When no lexical entry matches the designated token (or list of tokens), a new lexical entry is automatically created. The new lexical entry, be it a noun or a verb, is built using the token (or list of tokens) itself and its tag (or list of tags). Noun entries created in this way will contain information only about the singular form of the noun, or only about the plural form, depending on whether the token (or list of tokens) is tagged as a plural (nns, nps) or not (nn, np, cd, unk) (in case of a list, the last token is considered). Verb entries created in this way are canned text decorated with inflection information, or, in other words, partially specified lexical entries which can be used to produce only a single inflected form of the verb, as explained in section §2.3.3. A reference to a previously-existing or to a newly-created lexical entry, matching a specific token, will be represented in the output as shown in [23], where NounToken is a variable referring to the token (or list of tokens) used to create the lexical entry through the function lex.

[23] The lex(NounToken) Number of the … should be

As most TT-IDs contain noun phrases without articles, the transformation rules may use a function, named det, to append a definite article at the beginning of an input string, when the string represents a noun phase. [24] shows an application of this function. If the function is applied to a string which does not represent a noun phrase (e.g. a single adjective, or the empty string), it returns the input string unchanged.

[24] should give active:present:progressive det(DirectObject) to

Sometimes it will be necessary to extract a number value (singular or plural) from a list of tokens representing a noun phrase. To this purpose, we will use the function number(X), which, depending on the tag of the last noun token in the input list X, returns plural when the tag is (nns, nps) or singular when it is (nn, np, cd, unk).

Guard condition	The input matches either the pattern Head + [X:bez, Y:P] or the pattern Head + [X:bez, Z:Q] + Tail + [Y:P] Q is not vbn P is a preposition tag Head and Tail do not contain any token tagged with a verb tag different from vbn.
Output	should be active:present:simple adverbs() + det([Z:Q] + Tail) lex([Y:P])

Table 11: Guard and output of the BEZ rule

BEZ rule

The BEZ rule is designed to handle TT-IDs where the verb *to be* is not used as an auxiliary. [25, 26] are sample preprocessed TT-IDs of this kind.

[25] gene:nn product:nn is:bez physical:jj part:nn of:in

[26] is:bez adjacent:jj to:to

The BEZ rule, fed with [25, 26], produces verb-based relation templates contributing clauses like [27, 28].

[27] It should be the physical part of **..**

[28] It should be adjacent to **..**

The implementation of this rule, given in table 11, specifies as a guard condition that no other token may be tagged with a verb tag, except vbn. Tokens tagged vbn are allowed everywhere except right after the is:bez token, and they are considered adjectives. The guard condition also specifies that a preposition should occur at the end of the preprocessed input. The role of this preposition is to introduce the noun phrase associated with the node the edge is entering. The rule assumes that the part before the main verb is information about the domain of the relation, as it occurs to be in [25]. The article of the noun phrase following the verb is always definite, but in principle it should be definite only when the relation is functional. The observations we made in section §3.2.6 about the article in noun-based relation templates apply to this rule: if the rule could be informed that the relation is not functional, it could introduce an indefinite article instead of the definite one. For example, instead of [27] we would have [29].

[29] It should be a physical part of **..**

Assuming that the previous stages of the elaboration didn't introduce errors, the BEZ rule fails when one of its unchecked assumptions is not met, that is:

- when the content of Tail is not suitable for direct insertion in the target context;
- when the content of Head is not information about the domain;

- when the relation is not functional (e.g. [25]).

BEZ Passive rule

The BEZ Passive rule is designed to handle TT-IDs where the verb *to be* is an auxiliary in a passive verb. [30, 31] are sample preprocessed TT-IDs of this kind.

[30] is:bez composed:vbn of:in

[31] meat:nn type:nn is:bez made:vbn from:in

The BEZ Passive rule, fed with [30, 31], produces verb-based relation templates contributing clauses like [32, 33]

[32] It should be composed of ..

[33] It should be made from ..

The BEZ Passive rule accepts preprocessed TT-IDs that contain the token is:bez, followed by a token tagged vbn, and ending with a preposition. The implementation of the rule is very similar to the previous one, only the verb is processed differently. Like BEZ, this rule assumes that the part before the first verb is information about the domain of the relation, as it occurs to be in [31]. The article added to the tail is always definite, but there is no guarantee that this choice is the correct one. Assuming that the previous stages of the elaboration didn't introduce errors, the BEZ Passive rule fails when one of its unchecked assumptions is not met, that is:

- when the content of Tail is not suitable for direct insertion in the target context;
- when the content of Head is not information about the domain.

Guard condition	The input matches the pattern Head + [X:bez, W:vbn] + Tail + [Y:P] P is a preposition tag Head and Tail do not contain any token tagged with a verb tag different from vbn
Output	should lex([W:vbn]) passive:present:simple adverbs() + det(Tail) lex([Y:P])

Table 12: Guard and output of the BEZ-passive rule

HVZ-Range rule

The HVZ-Range rule is designed to handle TT-IDs where the object of the verb *to have* is a characterization of the role of the range in the relation. [34, 35] are sample non-preprocessed TT-IDs of this kind. We show non-preprocessed TT-IDs because to produce the relation template, the HVZ-Range rule uses the range information, which is removed in the preprocessing stage.

[34] has:hvz author:nn

[35] airport:nn has:hvz IATA:unk code:nn

The HVZ-Range rule, fed with [34, 35], produces noun-based relation templates contributing clauses like [36, 37].

[36] Its author should be ..

[37] Its IATA code should be ..

For this kind of input, we want to produce noun-based relation templates using the range information. In [34, 35] the noun phrase following the verb has does not contain prepositions. This is not always the case: there are some non-preprocessed IDs where the information about the range after the token has:hvz is a complex noun phrase containing one or more prepositions. This is the case for example in [38]. However, there is another class of TT-IDs similar in syntax to [38] where the noun phrase following has does not contain only information about the range but also information characterizing the relation itself. [39] is a non-preprocessed example of this kind. The range information in [39] is organization:nn, and [39] should contribute a clause structure like [40].

Guard condition	The input matches the pattern `Head + [X:hvz]` `Head` does not contain any token tagged with a verb tag different from `vbn` `range()` is not empty
Output	The `lex(range())` `number(range())` of the … should be

Table 13: Guard and output of the HVZ-Range rule

[38] has:hvz number:nn of:in columns:nns

[39] has:hvz voting:nn rights:nns in:in organization:nn

[40] It should have voting rights in ..

Unfortunately, the syntactic structure of [41] is the same of [40], and therefore we have no way to distinguish the two. If we transformed them both in the same way, in the way we treated [36, 37], we would transform [41] in [43], which is plain wrong.

[41] Its voting rights in organization should be ..

Ideally, the last three tokens of [38] would have been removed at the last stage of preprocessing, when the information about the range is removed. In practice, our range removal algorithm is very simple, and it proceeds backwards, removing tokens until a verb or a preposition is found. Therefore, it will handle [38] and [39] in the same way. Since there is no way to distinguish [38] from [39] using the information we possess, and since TT-IDs like [39] are more frequent in our corpus than TT-IDs like [38], we chose to forget about TT-IDs like [38]. To do so, we specify in the guard condition of the HVZ-Range rule that the input preprocessed TT-ID contains the verb in last position. This is equivalent to specify that the tokens following the verb do not contain any preposition, and therefore they are removed during preprocessing, as range information. Like BEZ, this rule assumes that the part before the main verb is information about the domain of the relation, as it occurs to be in [35].

Assuming that the previous stages of the elaboration didn't introduce errors, the HVZ-Range rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `range()` is not suitable for direct insertion in the target context;
- when the relation is not functional (e.g. [42]);
- when the content of `Head` is not information about the domain.

[42] family:nn has:hvz member:nn

HVZ rule

The HVZ rule is designed to handle TT-IDs where the verb *to have* is not used as an auxiliary, and its object, if any, does not characterize the role of the range in the relation. [43, 44] are sample non-preprocessed TT-IDs of this kind. We show non-preprocessed TT-IDs because we want to highlight that, unlike the HVZ-Range rule, the HVZ rule discards range information.

[43] has:hvz voting:nn rights:nns in:in organization:nn

[44] has:hvz access:nn to:in account:nn

The HVZ rule, fed with [43, 44], produces verb-based relation templates contributing clauses like [45, 46].

[45] It should have voting rights in **..**

[46] It should have access to **..**

For this kind of input, we want to discard range information and keep the rest as either the direct object or a complement to a form of the verb *to have*. The implementation of this rule assumes that the range information has been already removed during preprocessing, and therefore that the tokens appearing after the has:hvz token are not range information. The rule therefore specifies, in the guard condition, that there should be a preposition at the end of the last token of the input. This preposition is used to introduce the noun phrase associated with the node the edge is entering. We have already seen that our range extraction method is not reliable, and this causes this rule to process incorrectly input like [38]; however we accepted this limitation. Like BEZ, this rule assumes that the part before the main verb is information about the domain of the relation.

Assuming that the previous stages of the elaboration didn't introduce errors, the HVZ rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `Tail` is not suitable for direct insertion in the target context;
- when the content of `Head` is not information about the domain.

Guard condition	The input matches the pattern `Head + [X:hvz] + Tail + [Y:P]` `P` is a preposition tag `Head` and `Tail` do not contain any token tagged with a verb tag different from `vbn`
Output	should have active:present:simple adverbs() + Tail lex([Y:P])

Table 14: Guard and output of the HVZ rule

VBZ rule

The VBZ rule is designed to handle TT-IDs that contain a simple present, third person singular active verb. [47, 48, 49] are sample preprocessed TT-IDs of this kind.

[47] drug:nn interacts:vbz with:in

[48] flows:vbz into:in

[49] graph:nn depicts:vbz

The VBZ rule, fed with [47, 48, 49], produces verb-based relation templates contributing clauses like [50, 51, 52].

[50] It should interact with **..**

[51] It should flow into **..**

[52] It should depict **..**

We want to process this input in the same way the HVZ rule processes its input. The VBZ rule accepts preprocessed TT-IDs that contain exactly one token tagged `vbz`. The tokens following the verb are used as the body of the output relation template, possibly with a definite article in front of it. Like BEZ, this rule assumes that the part before the main verb is information about the domain of the relation.

Assuming that the previous stages of the elaboration didn't introduce errors, the HVZ rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `Tail` is not grammatical for our purposes, e.g. [53];
- when the article introduced in the tail is the wrong one, e.g. [54] which becomes [55];
- when the content of `Head` is not information about the domain.

[53] Holds:vbz underspecified:jj

[54] possesses:vbz copy:nn of:in

[55] it should possess the copy of **..**

Guard condition	The input matches the pattern `Head + [X:vbz] + Tail + [Y:P?]` If `P` is defined, `P` is a preposition tag `Head` and `Tail` do not contain any token tagged with a verb tag different from `vbn`
Output	should lex([X:vbz]) active:present:simple adverbs() + det(Tail) lex([Y:P])

Table 15: Guard and output of the VBZ rule

VB rule

The VB rule is designed to handle TT-IDs that contain a simple present, non-third person singular verb used transitively. [56] is a sample preprocessed TT-ID of this kind.

[56] generate:vb phrase:nn

The VB rule, fed with [56], produces verb-based relation templates contributing clauses like [57].

[57] It should generate ..

The implementation of the VB rule discards all TT-IDs which, after preprocessing, contain something besides the verb. Assuming that the previous stages of the elaboration didn't introduce errors, the VB rule will not fail.

Guard condition	The input matches the pattern `[(X:vb)]`
Output	should `lex(X:vb) active:present:simple adverbs() ∈ ∅`

Table 16: Guard and output of the VB rule

VBN-Be rule

The VBN-Be rule is designed to handle TT-IDs that contain a past participle which can be interpreted as a fragment of a passive verb form. [58, 59] are sample preprocessed TT-IDs of this kind.

[58] ordered:vbn by:in

[59] intended:vbn for:in consumption:nn by:in

The VBN-Be rule, fed with [58, 59], produces verb-based relation templates contributing clauses like [60, 61].

[60] It should be ordered by ..

[61] It should be intended for consumption by ..

Unfortunately, there is another class of TT-IDs that contain a token tagged `vbn` which cannot be interpreted in this way, e.g. [62] (not preprocessed). [62] should be transformed into a verb-based relation template to contribute a clause like [63].

[62] targeted:vbn industry:nn

[63] It should have targeted ..

In order to capture only the occurrences of tokens tagged `vbn` which are part of a passive verb form, we require that a preposition follows the `vbn` token immediately. Passive verb forms do not take direct objects, therefore TT-IDs like [62] are not accepted by the rule. However, this does not exclude the case that the past participle is an abbreviation of an active past verb form. For example [64], cannot be considered a passive form because the verb is intransitive. Verbs which can be used both in transitive and intransitive form like [65] may also be problematic in that the knowledge engineer may have intended them as active instead of passive. So, while [64] must eventually contribute a clause like [66], [65] is ambiguous, and may contribute [67] or [68].

[64] travelled:vbn to:in

[65] killed:vbn with:in

[66] It should have travelled to ..

[67] It should have killed with a knife

[68] It should be killed with a knife

Regarding input like [64], we must say that in principle it would be possible to look up linguistic resources to find whether a verb can only be used intransitively. However, while in our corpus such TT-IDs do appear, they are much less frequent than TT-IDs which can be interpreted as fragments of passive verb forms. For what concerns [65], even if we had information about the transitivity of the verb in general, that information would not be enough to solve the ambiguity. Our decision regarding this issue is: we interpret all of them as fragments of passive verb forms, and accept the fact that the rule will fail on certain input.

The VBN-Be rule accepts preprocessed TT-IDs that contain exactly one token tagged `vbn`, followed by a token tagged with a preposition tag. Like BEZ, this rule assumes that the part before the main verb is information about the domain of the relation. This information is always discarded. The output we want to produce for this input is similar to the one produced by the BEZ-Passive rule.

Assuming that the previous stages of the elaboration didn't introduce errors, the VBN-Be rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `Tail` is not suitable for direct insertion in the target context, e.g. [69];
- when the content of `Head` is not information about the domain;
- when the verb is intransitive or when it was intended as used intransitively, e.g. [64, 65].

[69] connected:vbn along:in inside:in

Guard condition	The input matches the pattern `Head + [X:vbn, Y:P] + Tail` `P` is a preposition tag `Head` and `Tail` do not contain any token tagged with a verb tag
Output	should lex([X:vbn]) passive:present:simple Tail + adverbs() lex([P:Y])

Table 17: Guard and output of the VBN-Be rule

VBN-Have rule

The VBN-Have rule is designed to handle TT-IDs that contain a past participle which can be interpreted as a fragment of an active verb form. [70, 71, 72] are sample non-preprocessed TT-IDs of this kind. We show non-preprocessed TT-IDs because the preprocessed versions of these TT-IDs are harder to understand out of context.

[70] lock:nn requested:vbn

[71] supervised:vbn action:nn

[72] sent:vbn text:nn message:nn to:in

The VBN-Have rule, fed with [70, 71, 72], produces verb-based relation templates contributing clauses like [73, 74, 75].

[73] It should have requested ..

[74] It should have supervised ..

[75] It should have sent a message to ..

We implemented the VBN-Have rule following the observations we reported in the description of the VBN-Be rule. While some TT-IDs of this class are being incorrectly captured by the VBN-Be rule, e.g. [64, 65], the VBN-Have rule handles all the remaining TT-IDs. These are all the TT-IDs containing a single verb token, tagged `vbn`, such that the token following it, if any, is not a preposition. Unlike BEZ and many other rules, this rule assumes that the part before the main verb is information about the range of the relation, but only when `range()` is empty. Range information is discarded, as usual.

Assuming that the previous stages of the elaboration didn't introduce errors, the VBN-Have rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `Tail` is not suitable for direct insertion in the target context, e.g. [76];
- when the content of `Head` is not information about the domain;
- when the verb is not a part of an active verb form but a simple verbal adjective, e.g. [77].

[76] connected:vbn along:in inside:in

[77] correlated:vbn structures:nns

Guard condition	The input matches the pattern `Head + [X:vbn, Y:P?] + Tail` If `P` is defined, `P` is not a preposition tag `Head` and `Tail` do not contain any token tagged with a verb tag
Output	should `lex([X:vbn])` active:past:simple `adverbs() + det([Y:P] + Tail)` ∅

Table 18: Guard and output of the VBN-Have rule

VBG rule

The VBG rule is designed to handle TT-IDs that contain a present participle which can be interpreted as a fragment of a present progressive verb. [78, 79] are sample non-preprocessed TT-IDs of this kind. We show non-preprocessed TT-IDs because to produce the relation template, the VBG rule uses the range information, which is removed in the preprocessing stage.

[78] process:nn running:vbg on:in

[79] communicating:vbg by:in means:nn of:in

The VBG rule, fed with [78, 79], produces verb-based relation templates contributing clauses like [80, 81].

[80] It should be running on **..**

[81] It should be communicating by means of **..**

The VBG rule accepts preprocessed TT-IDs that contain exactly one token tagged `vbg` which is not followed by any noun. The condition is set to prevent interpreting adjectival participles like [82] as incomplete verbs, yielding [83], which would be wrong. This comes at the cost of losing TT-IDs like [84], which has the same structure as [82] but which can still be handled in the same way [78, 79] are handled.

[82] founding:vbg member:nn

[83] It should be founding ..

[84] using:vbg program:nn

The guard condition reported in table 19 forbids a token tagged as noun to follow the token tagged vbg. The guard is complicated because it also forbids that a noun in the range follows the vbg token the end of the preprocessed TT-ID. Unlike BEZ, this rule assumes that the part before the main verb is information either about the domain or the range of the relation. This information is always discarded.

Assuming that the previous stages of the elaboration didn't introduce errors, the VBG rule fails when one of its unchecked assumptions is not met, that is:

- when the content of Tail is not suitable for direct insertion in the target context;
- when the content of Head is not information about the domain, e.g. [85];

[85] attempt:nn at:in performing:vbg

Guard condition	The input matches the pattern Head + [X:vbg, Y:G] + Tail + [Z:P?] or the pattern Head + [X:vbg, Z:P?] If the input matches the second pattern and P is undefined, either range() is empty or it matches [W:G] + Rest G is not a noun tag, nor it is unk If P is defined, P is a preposition tag Head and Tail do not contain any token tagged with a verb tag
Output	should lex([X:vbg]) active:present:progressive adverbs() + det([Y:G] + Tail) lex([Z:P])

Table 19: Guard and output of the VBG rule

No verbs, no nouns rule

The No-Verbs-No-Nouns rule is designed to handle TT-IDs which do not contain verbs or nouns. [86, 87] are sample preprocessed TT-IDs of this kind.

[86] adjacent:jj to:in

[87] before:in

The No-Verbs-No-Nouns rule, fed with [86, 87], produces verb-based relation templates contributing clauses like [88, 89].

[88] It should be adjacent to ..

[89] It should be before ..

In preprocessed TT-IDs like [86, 87] the contribution of the relation we plan to extract is either a preposition or an adjective. Among the TT-IDs in this class, only those which end with a preposition token can be used to build a sentence. This token should be a preposition for the same reason explained in the BEZ rule: to introduce the noun phrase associated with the node the edge is entering. Our rule further constrains the tokens which have been removed during preprocessing.

Assuming that the previous stages of the elaboration didn't introduce errors, the No-Verbs-No-Nouns rule may fail when an adjective in the input referred to a noun which was removed during preprocessing.

Guard condition	The input matches the pattern `Head + [Y:P]` `P` is a preposition tag `Head` does not contain any token tagged with a verb or noun tag
Output	should be active:present:simple det(Head) + adverbs() lex([X:P])

Table 20: Guard and output of the No-Verbs-No-Nouns rule

Simple NP rule

The Simple-NP rule is designed to handle TT-IDs which consist of a characterization of the role of the range in the relation. [90, 91] are sample preprocessed TT-IDs of this kind.

[90] scientific:jj name:nn

[91] uml:unk specification:nn

The Simple-NP rule, fed with [90, 91], produces noun-based relation templates contributing clauses like [92, 93].

[92] Its scientific name should be **..**

[93] Its UML specification should be **..**

The implementation of this rule simply accepts input all TT-IDs which contain a noun, no verbs and no prepositions to produce a noun-based relation template. If the input contains a dangling adjective, the input is discarded. A dangling adjective is a token tagged with an adjective tag which is not followed by any token tagged with a noun tag. Between the adjective and the noun any number of adjectives and adverbs may appear.

Assuming that the previous stages of the elaboration didn't introduce errors, the Simple-NP rule fails when the relation is not functional, e.g. [94].

[94] brother:nn

Guard condition	The input matches the patterns `Head + [X:N] + Tail` `N` is a noun tag `Head` and `Tail` do not contain any token tagged with a verb tag or with a preposition tag, and they do not contain dangling adjectives
Output	The lex(Head + [X:N] + Tail) number(Head + [X:N] + Tail) of the … should be

Table 21: Guard and output of the Simple-NP rule

Composite NP rule

The Composite-NP rule is designed to handle TT-IDs which consist of a characterization of the role of the domain in the relation. [95, 96] are sample preprocessed TT-IDs of this kind.

[95] product:nn of:in

[96] coastal:jj zone:nn of:nn

The Composite-NP rule, fed with [95, 96], produces verb-based relation templates contributing clauses like [97, 98].

[97] It should be the product of **..**

[98] It should be the coastal zone of **..**

The implementation of this rule simply accepts input all TT-IDs which contain no verbs but contain a noun and a preposition, and uses this input as the predicate nominal of a verb-based relation template. If the input contains a dangling adjective, the input is discarded. As explained earlier, a dangling adjective is a token tagged with an adjective tag which is not followed by any token tagged with a noun tag.

Assuming that the previous stages of the elaboration didn't introduce errors, the Composite-NP rule fails when the relation is not functional, e.g. [99].

[99] biography:nn of:in

For this kind of TT-ID the range removal method is subject of the same sort of errors that we encountered in HVZ-Range, that is, it may remove only part of the range. For example, from [100] the preprocessing stages produce [101] which the Composite-NP transforms into a verb-based template that contributes the clause [102], which does not convey the original meaning of the relation.

[100] point:nn of:in sale:nn

[101] point:nn of:in

[102] It should be the point of **..**

Guard condition	The input matches the patterns `Head + [X:N] + Tail + [Y:P]` `Head` and `Tail` do not contain any token tagged with a verb tag, and they do not contain dangling adjectives `N` is a noun tag `P` is a preposition tag
Output	should be `active:present:simple det(Head + [X:N] + Tail)+adverbs() lex([Y:P])`

Table 22: Guard and output of the Composite-NP rule

Dangling Adjective rule

The Dangling-Adjective rule is designed to handle TT-IDs that contain a noun and a dangling adjective. [103, 104] are sample non-preprocessed TT-IDs of this kind. We show non-preprocessed TT-IDs because the preprocessed versions of these TT-IDs are harder interpret out of context.

[103] background:nn relevant:jj to:in conclusion:nn

[104] vulnerable:jj to:in attack:nn with:in type:nn

The aim of the rule is to use the adjective and anything following it to build a predicate nominal. The tokens before the adjective are always considered information about the domain of the relation, like in [105]. The Dangling-Adjective rule, fed with [103, 104], produces verb-based relation templates contributing clauses like [105, 106].

[105] It should be relevant to **..**

[106] It should be vulnerable to attack with **..**

The implementation of this rule simply accepts all preprocessed TT-IDs containing a noun and a dangling adjective and ending with a preposition. The preposition is required for the same reason explained in the BEZ rule: to introduce the noun phrase associated with the node the edge is entering. TT-IDs containing dangling adjectives but not containing nouns are handled by the No-Verbs-No-Nouns rule.

Assuming that the previous stages of the elaboration didn't introduce errors, the Dangling-Adjective rule fails when one of its unchecked assumptions is not met, that is:

- when the content of `Tail` is not suitable for direct insertion in the target context;
- when the content of `Head` is not information about the domain.

Guard condition	The input matches the pattern `Head + [X:J] + Tail + [Z:P]` `P` is a preposition tag `Head` contains a token tagged with a noun tag `J` is an adjective tag, and `X` is a dangling adjective – i.e. `Tail` does not contain any token tagged with a noun tag `Head` and `Tail` do not contain any token tagged with a verb tag
Output	should be active:present:simple be `[X:J] + Tail lex([Z:P])`

Table 23: Guard and output of the Dangling-Adjective rule

3.2.8 Evaluation

In order to evaluate the system, we collected some relation identifiers from a few ontologies that were not included in our corpus, we ran our system and we assessed how many of the generated templates would be suitable for direct use in the Query Tool NLI. Evaluating whether a template matches the intended meaning of the relation is in general a hard task. While several templates can be easily classified as correct or wrong models of the textual representation of a relation, in many other situations it is not straightforward to do so, especially when the relation ID contains abbreviations and unknown words, or anyhow when the meaning of the relation is not clear to the evaluator. A small number of specialistic terms and abbreviations may be reasonable in the textual representation of highly domain-specific relations. However, with no further information available, it is hard to evaluate whether a domain expert would find the generated text acceptable. The following examples, taken from our

corpus, illustrate this point. While [107-110] are reasonable representations of the corresponding relations (not shown here), [111] and [112] are probably not.

[107] Its address text nc should be ..

[108] It should be an agreement feature for ..

[109] Its airline iata carrier code should be ..

[110] It should be a host db name for ..

[111] Its altitude angle predicate fn nad 83 should be ..

[112] Its launch afld afsd should be ..

In order to avoid this problem, we chose the relations to use for evaluation from ontologies dealing with relatively simple domains: more precisely, we used some of the ontologies developed by OrdnanceSurvey, the national mapping agency of Great Britain, plus the ontology of the Pizza Tutorial by Manchester University. These ontologies in total contribute 64 unique relations.

The evaluation procedure was the following: since some relation IDs contained an additional namespace encoded right before the relation, we preprocessed the relations to remove this information. Next, we automatically generated relation templates from these relation IDs. Finally we inspected the generated templates to evaluate whether they would be usable to generate text or whether they need some further care. Each template was given a positive or negative mark. The result of this evaluation is that for 48 out of 64 relations (75%) the generated template is suitable for direct use with the Query Tool NLI. Table 24 shows the 64 relation IDs together with a sample text generated using the templates produced by our system.

This specific test sample highlights that the syntactic structure of the relation IDs is quite simple, but the part of speech tagging task is challenging. For example, flows, gushes, feeds are interpreted as nouns instead of verbs. Another kind of frequent error is the misinterpretation of a relation as functional. For example, isPartOf is represented by the system with [115], whereas [116] would be the correct way. The same happens for hasStorey, isIngredientOf, hasTopping.

[113] It should be the part of ..

[114] It should be a part of ..

Another notable error is the failed representation of hasCountryOfOrigin, where the preprocessing wrongly interpreted origin as range information, a problem that we described in §3.2.4. In two instances the procedure failed to produce any template because the part of speech tagger assigned a tag which our approach does not handle (rp, for particle). This evaluation did not produce exciting results, but the errors reported are encouraging in that they hint at the areas in need of an improvement.

Relation ID	Realized template	Evaluation
by	It should be by ..	+
contains	It should contain ..	+
directlyContains	It should contain directly ..	+
drains	One of its drains should be ..	-
emits	One of its emits should be ..	-

enables	It should enable ..	+
feeds	One of its feeds should be ..	-
flows	It should be flows ..	-
flowsDuring	It should flow during ..	+
flowsFrom	It should flow from ..	+
flowsIn	It should flow in ..	+
flowsInto	It should flow into ..	+
flowsOver	It should flow over ..	+
flowsUnder	It should flow under ..	+
for	template generation failed	-
from	It should be from ..	+
gushes	Its gushes should be ..	-
hasAddress	Its address should be ..	+
hasBase	Its base should be ..	+
hasCNSCode	Its cns code should be ..	+
hasCUKCode	Its cuk code should be ..	+
hasCircularCurrent	Its circular current should be ..	+
hasCountryOfOrigin	It should have country of ..	-
hasCurrent	Its current should be ..	+
hasFlow	Its flow should be ..	+
hasFootprint	Its footprint should be ..	+
hasGORCode	Its gor code should be ..	+
hasHistoricInterest	Its historic interest should be ..	+
hasHistoricPurpose	Its historic purpose should be ..	+
hasIngredient	Its ingredient should be ..	-
hasLinearForm	Its linear form should be ..	+
hasName	Its name should be ..	+
hasPart	Its part should be ..	-
hasPurpose	Its purpose should be ..	+
hasQUACode	Its qua code should be ..	+
hasSpiciness	Its spiciness should be ..	+
hasStorey	Its storey should be ..	-
hasTopping	Its topping should be ..	-
isAdjacentTo	It should be adjacent to ..	+
isBaseOf	It should be the base of ..	+

isCarriedOutBy	template generation failed	-
isConnectedTo	It should be connected to ..	+
isContainedIn	It should be connected in ..	+
isControlledBy	It should be controlled by ..	+
isEdgeOf	It should be the edge of ..	+
isFedBy	It should be fed by ..	+
isIngredientOf	It should be the ingredient of ..	-
isLedBy	It should be led by ..	+
isLocatedAt	It should be located at ..	+
isLocatedBehind	It should be located behind	+
isLocatedIn	It should be located in ..	+
isLocatedNear	It should be located near ..	+
isLocatedOn	It should be located on ..	+
isLocatedUnder	It should be located under ..	+
isOperatedBy	It should be operated by ..	+
isOwnedBy	It should be owned by ..	+
isPartOf	It should be the part of ..	-
isSaturatedBy	It should be saturated by ..	+
isSubjectTo	It should be the subject to ..	-
isToppingOf	It should be the topping of ..	+
of	It should be of ..	+
operatesDuring	It should operate during ..	+
requires	It should require ..	+
spans	Its spans should be ..	-

Table 24: Evaluation data

4 Conclusion

In this thesis, we have been discussing the development of a natural language interface to knowledge bases on NLG techniques. We have presented Query Tool, a query interface framework relying on description logics reasoners. The Query Tool interaction model prevents the user from building unsatisfiable queries and at the same time allows the users to explore the content and the capabilities of the KB.

We described the development of the natural language interface of Query Tool, which relies on a NLG system to produce the textual representation of the query. Our system uses NLG to represent the whole query, along with all the elements which 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 English text.

Before discussing the design of the interface itself, we reviewed the requirements and we compared our requirements with related work in the literature. Then, we presented the design of the syntax of the generation language. We selected the syntactic features of the language on the basis of the data available in corpus of relation ID. Our goal was to keep the system simple while still being expressive enough to allow the representation in natural language of most of the relations in the corpus.

Next, we discussed the templates mechanism. Our generator is not tailored to any specific knowledge domain. On one hand, this means that it can be used to interface any KB. On the other hand, this also means that it is simple enough to be useful in any context, and it does not come with all the resources needed to generate text out-of-the-box. To use it with a specific KB, the system must be provided with a lexicon and a template map. The lexicon contains the words to be used in the generated text. The template map is the bridge between the natural language and the knowledge representation language: it associates each atomic concept and each relation 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 relation. System engineers deploying the Query Tool NLI must craft both the lexicon and the template, and this task with an understanding of the target KB and of basic notions of linguistics such as verb tenses, noun genders and countability. In order to ease the burden of developing these resources, we experimented with a rule-based technique to have the system generate them automatically. In addition, we experimented with a data-based approach to gather additional linguistic information regarding a relation. This information can be fed to the system based on the previous technique when the relation ID is not informative or however not useful.

The first technique mines a corpus for linguistic expressions that describe a given relation. To this purpose, the it requires some initial data regarding the elements which are connected by this relation. The retrieved expressions can either be suggested to the user, which will likely find an appropriate pick to hand-craft a new template, or used to feed the next technique. This solution is especially useful when the second technique fails to process the relation ID.

The second technique allows to produce the resources necessary to configure our NLI for use with new relations of a KB, using as a source of data the relation IDs of

KB's ontology. However, the process is not completely reliable, and therefore systems engineers must review the result and make the necessary corrections. We described the technique to handle relation IDs, but the technique in principle could be adapted to handle concept IDs as well.

We evaluated this second technique against 64 unique relations collected from real ontologies, with the result that for 48 out of 64 relations (75%) the technique generates a template suitable for direct use with the Query Tool NLI. This evaluation suggests that while the generation of the template map is far from being reliable, it can speed up the work of systems engineers, who do not need to create the whole map from scratch, but only have to review the generated map and repair any error. This improves the portability of the Query Tool NLI, as it is faster to configure to work with a new KB. Our evaluation suggests that, when used to retrieve a linguistic expressions for a target relation among those our system can handle, in 22% of the cases, our system will return one or more correct results, out of five. In 55% of the cases, it will return no expressions, and in the remaining 27% of the cases, it will return a list of useless expressions. While the scope of our experiment was limited to a certain subset of relations, our evaluation suggests that this techniques can, in a limited way, support the process of configuration of the Query Tool NLI.

4.1 Open issues and future work

In general, this work has suggested that recent advances in description logics allow for more intelligent menu-based natural language interfaces. Moreover, our experiments have shown that there are untapped resources which can be used to improve the ease of deployment of natural language interfaces to databases.

An interesting but challenging open question which this experiment indirectly suggests is whether the same ideas could be applied to a controlled natural language editor: in particular, whether it could be possible to introduce reasoning capabilities to controlled natural language editors, which by norm use syntax to restrict the possible completions of sentences and queries. Also, it would be interesting to know whether the techniques that we designed to improve the portability of our system could be applied to improve the portability of a system based on a controlled natural language.

4.1.1 Natural Language Generator

A missing feature of the NLG system is the lack of support for adjectives in nominal predicates. This leads to sentences like [1] in place of [2].

[1] The tea is a black thing.

[2] The tea is black.

Another elegant addition would be a better support for pronouns, to realize sentences like [3] in place of [4].

[3] I am looking for something black.

[4] I am looking for a black thing.

Late in the design stage, a problem arose regarding certain sentences which appear as a result of the addition of a new node to a query. The problem involves the concept selected as label of the new node, and the relation used as label of the edge. When the concept happens to be the natural range of the relation, i.e. the concept which

describes exactly all the elements that can take part to the relation, the generator will produce sentences like [5] and [6]. Both these sentences are not informative, in that they do not convey meaning when interpreted literally. This is strikingly obvious in [6]. The role of these sentences in the Query Tool interaction model is on one hand to allow the user to specify, in a later stage, further information about the new node inserted, and on the other hand, to state that the element represented by the node actually exists. For example, omitting [6] in a query specification may eventually retrieve cars which do not have any engine, if such cars were represented in the KB.

[5] The mother of the girl should be a person.

[6] The engine of the car should be an engine.

Last but not least, while not necessarily an issue, the referring expressions generator is rather naïve, and there is certainly room for improvement.

4.1.2 Template generator

The only issue with the rule-based template generator we plan to address involves the distinction of functional relations when generating noun-based templates. Information regarding functionality of relations is not available to the rule-based technique, but it may be possible to obtain it. Currently, the technique assumes that all relations are functional, and therefore the article produced for noun-based templates is always definite even in situations where it should not be, e.g. in [7].

[7] The parent of the boy should be a doctor.

An area of improvement could be an effort to modify the generator to reuse existing lexical resources.

Acknowledgements

I would like to thank all the people who contributed to this work with ideas, criticisms, experience, support, encouragement, and trust: my supervisors Gertjan van Noord and Enrico Franconi, but also Gosse Bouma, Xiantang Sun, Chris Mellish, Sergio Tessaris, Paolo Dongilli, Paolo Guagliardo, Laura Perez, and to the participants of the CNL 2009 workshop. A special thanks to Richard Power for giving the talk which eventually led me to join to this project – even if he doesn't know that.

I would like to thank the EM-LCT team, in particular Gisela Redeker and Raffaella Bernardi, and with them the University of Groningen and the University of Bolzano. They took care of all the details and arranged everything to make all this not only possible but even *pleasurable*.

Finally, I would like to thank my family, my girlfriend and my friends. It is because they took care of me that I could spend my time in working on what I like.

Thank you!

Appendix 1: Graphical User Interface

As part of a different project ([Trevisan 2009]), we developed a Web-based graphical user interface for Query Tool that showcases the NLI we described in this thesis. Illustration 19 shows a screenshot of the software. The shot was taken while a menu containing compatibles and properties which are available for addition. The underlying ontology was developed internally by a different team, and the linguistic resources have been automatically generated from the ontology using the techniques discussed in this thesis. One may notice that the template generator failed to pick the appropriate indefinite article for `used car`.

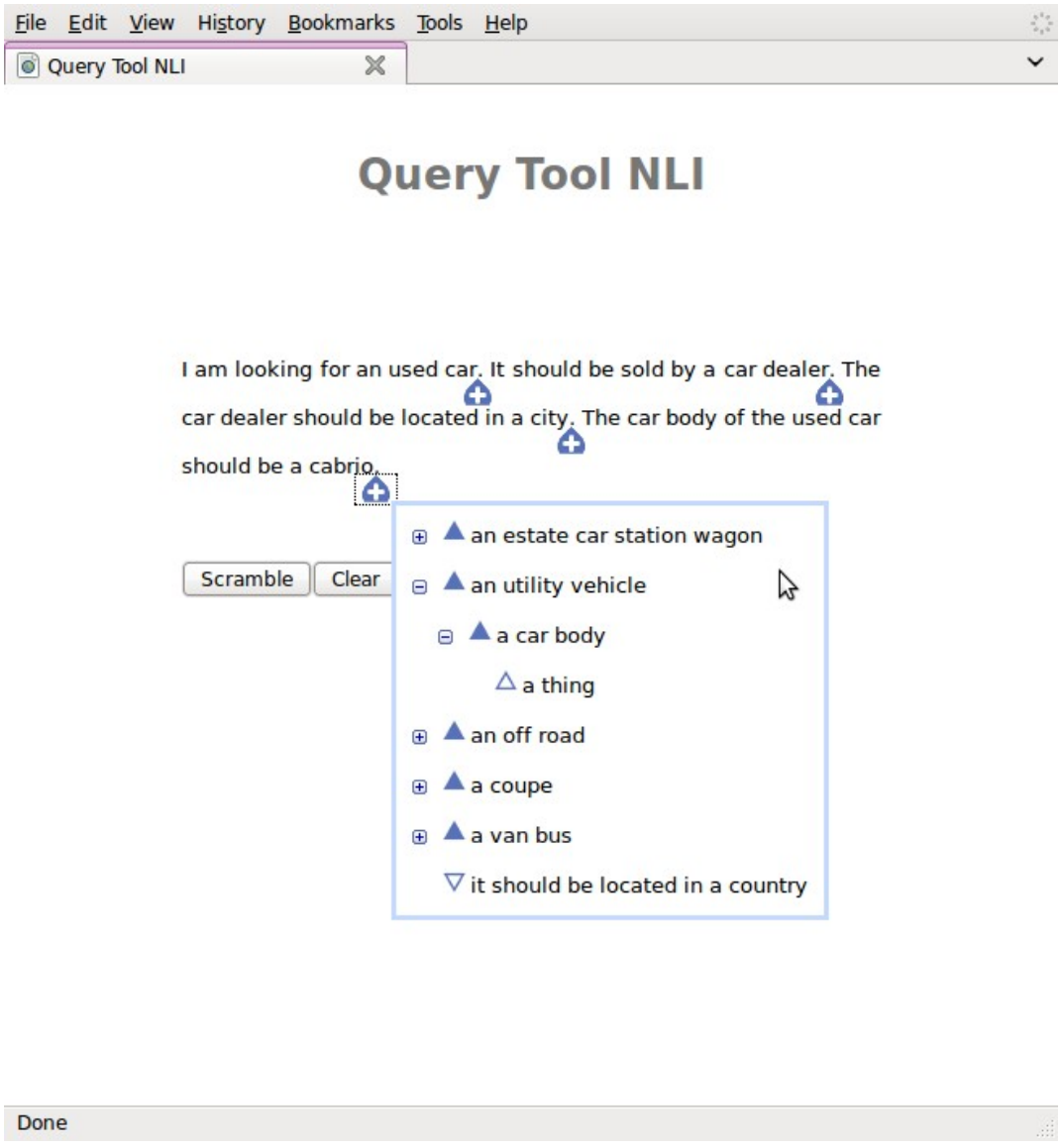

Illustration 19: Screenshot of the graphical user interface of Query Tool

References

Androutsopoulos, I., Ritchie, G. D., & Thanisch, P. (1995). Natural language interfaces to databases - An introduction. *Journal of Natural Language Engineering, 1*, 29-81.

Bernstein, A. & Kaufmann E. (2006). GINO - A guided input natural language ontology editor. In proceedings of ISWC 2006: *The 5th International Semantic Web Conference*, 144-157.

Bernstein, A., Kaufmann, E., Göhring, A., & Kiefer. C. (2005). Querying ontologies: A controlled English interface for end-users. In proceedings of ISWC 2005: *The 4th International Semantic Web Conference*, 112-126.

Dongilli, P. (2008). Natural language rendering of a conjunctive query. (KRDB Research Centre Technical Report No. KRDB08-3). Bozen, IT: Free University of Bozen-Bolzano.

Dongilli, P., Franconi, E., & Tessaris, S. (2004). Semantics driven support for query formulation. In proceedings of DL 2004: *The 2004 International Workshop on Description Logics*, 112-122.

Guagliardo, P. (2008). An ontology based visual tool for query formulation support: theoretical foundations. (KRDB Research Centre Technical Report). Bozen, IT: Free University of Bozen-Bolzano.

Guagliardo, P. (2009). Theoretical foundations of an ontology-based visual tool for query formulation support. (KRDB Research Centre Technical Report No. KRDB09-5). Bozen, IT: Free University of Bozen-Bolzano.

Hallett, C., Scott, D., & Power, R. (2007). Composing questions through conceptual authoring. *Computational Linguistics, 33*(1), 105-133.

Ittoo, A, & Bouma, G. (2009) Semantic selectional restrictions for disambiguating meronymy relations. In proceedings of CLIN09: *The 19th Computational Linguistics in the Netherlands meeting*, to appear.

Kuhn, T. (2009). How controlled English can improve semantic wikis. In proceedings of SemWiki2009: *The Fourth Workshop on Semantic Wikis*. Retrieved from http://ceur-ws.org/Vol-464/paper-03.pdf.

Lin, D. & Pantel, P. (2001). Discovery of inference rules for question answering. *Natural Language Engineering, 7*(4), 343-360.

de Marneffe, M. & Manning, C. D. (2008) The Stanford typed dependencies representation. In proceedings of Coling 2008: *The workshop on Cross-framework and Cross-domain Parser Evaluation*, 1-8.

Mellish, C. & Sun, X. (2005). The Semantic Web as a linguistic resource: opportunities for natural language generation. *Knowledge Based Systems 19*(5), 298–303.

Mueckstein, E. M. (1985). Controlled natural language interfaces (extended abstract): the best of three worlds. In proceedings of: *The 1985*

ACM thirteenth annual conference on Computer Science, 176-178.

Nguyen, D. P. T., Matsuo, Y., & Ishizuka, M. (2007). Relation extraction from wikipedia using subtree mining. In proceedings of AAAI-07: *The Twenty-Second Conference on Artificial Intelligence, 22*(2), 1414-1420.

Perez, L. H. (2009). Intelligent query interface: adding natural language support. (KRDB Research Centre Technical Report). Bozen, IT: Free University of Bozen-Bolzano.

Power, R., Scott, D., & Evans, R. (1998). What you see is what you meant: direct knowledge editing with natural language feedback. In proceedings of ECAI 98: *the 13th Biennial European Conference on Artificial Intelligence*, 675-681.

Ranta, A. (2004). Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming, 14*(2), 145-189.

Reiter, E., & Dale, R. (1997). Building Applied Natural Language Generation Systems. Journal of Natural Language Engineering

Reiter, E., & Dale, R. (2000). Building Natural Language Generation Systems. Cambridge University Press.

Sun, X. (2009). Unpublished doctoral dissertation draft. University of Aberdeen.

Sun, X., & Mellish C. (2007). Domain independent sentence generation from RDF representations for the Semantic Web. In proceedings of ECAI'06: *Proceedings of the Eleventh European Workshop on Natural Language Generation,* 105-108.

Thompson, C. W., Pazandak, P., & Tennant, H. R. (2005). Talk to your semantic web. *IEEE Internet Computing, 9*(6), 75-78.

Trevisan, M. (2008). A natural language generator for QueryTool. (KRDB Research Centre Technical Report). Bozen, IT: Free University of Bozen-Bolzano.

Trevisan, M. (2009). A natural language interface for QueryTool. (KRDB Research Centre Technical Report No. KRDB09-6). Bozen, IT: Free University of Bozen-Bolzano.

Tennant, H. R., Ross, K. M., Saenz, R. M., Thompson, C. W., & Miller, J. R. (1983). Menu-based natural language understanding. In proceedings of: *the 21st annual meeting of the Association for Computational Linguistics*, 151-158.

Tufis, D. & Mason O. (1998). Tagging romanian texts: a case study for QTAG, a language independent probabilistic tagger. In proceedings of LREC'98: *The First International Conference on Language Resources & Evaluation*, 589-596.

Wu, F., & Weld, D. S. (2008). Automatically refining the wikipedia infobox ontology. Proceeding of WWW2008: *The 17th international conference on World Wide Web*, 635-644.