Presenting query results in NL for Quelo.

X. Isabel Juárez-Castro

Free University of Bozen-Bolzano

1 Introduction

The Semantic Web has made available a huge amount of information stored in ontologies. The usual method to access this information is through queries written in a query language. These query languages have a specific syntax and have a hard learning curve. Therefore, the non-expert user is unable to access all this information. One of the solutions proposed to solve this problem is with Natural Language Interfaces (NLI). Using NLI the user is able to pose his query in natural language, which the NLI can translate into the query language. This query may later be answered by an ontology reasoner.

Currently several NLI for ontologies have been developed, e.g. GINO [1], Querix [6], PANTO [12], OWLPath [11] and FREyA [2]. However, most of the times the results obtained from the ontology reasoner are returned to the user in a table using the language of the ontology. If NL was necessary to obtain this information, it is questionable that the user will understand the language of the ontology in the results.

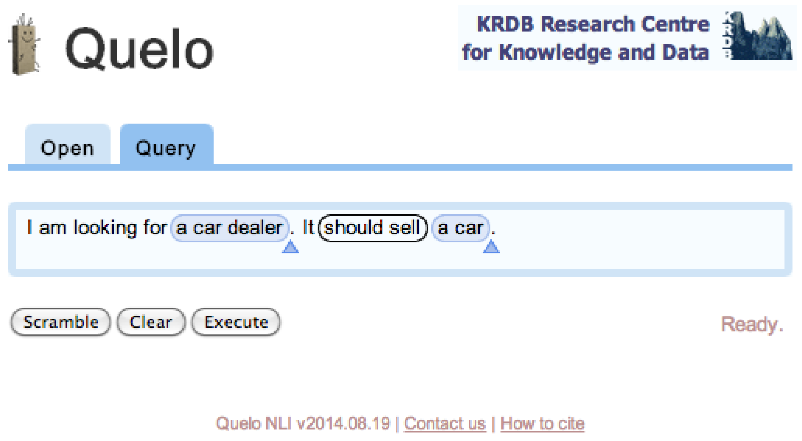
For example, Querix shows the results as a table where the headers of each column are the labels of the concepts in the ontology. They add to the table the concepts present in the query. OWLPath shows just one column with the results, with *Results* as header. FREyA outputs two different type of visualization for the results. If there are few results it displays a graph which represents the result of the query. However, if there are many results it outputs a table with the concepts of the query and generic labels as column headers.

Quelo is a menu-guided NLI for ontologies introduced in [3]. Within Quelo, the user creates a NL query guided by menus. Quelo uses Natural Language Generation(NLG) to produce this NL query; however, as the other NLI previously mentioned, when presenting the results it also uses the language of the ontology instead of a NLG approach.

The goal of this project is to extend Quelo's NLI to produce NL descriptions of query results. More precisely, we aim to automatically generate descriptive headers for the results obtained of the query execution. A descriptive header for the results will help the user to relate the query with the obtained values. This description can also help the user verify whether what he obtained as result is what he intended to get. Additionally, it can give some idea to the user of the relation between the resulting columns.

To produce NL descriptions we use NLG techniques and extend the domain independent [10] grammar based generation approach proposed in [8]. The verbalisation of the headers presents some challenges. It needs to rely on a generation framework that supports in a principled way the syntactic variability required to formulate queries as well as their answer descriptions in NL. For instance, given the NL query description "I am looking for a car sold by a car dealer." where the user asks for instances of the concept *car dealer*, the header description would be a NL expression of the form "A car dealer who sells the car."

Another challenging task is to select the information which should be included in the verbalisation of each concept. We also need to handle the references between the descriptions of the headers. The description of each header should refer correctly to the concept and should relate to the other


Quelo
KRDB Research Centre for Knowledge and Data
Open
Query
Scramble
Clear
Execute
Ready.
Quelo NLI v2014.08.19 | Contact us | How to cite

I am looking for a car dealer. It should sell a car.

Figure 1: *Quelo query example*

descriptions in the best way possible. Finally, there is also the problem of how to integrate the description with the graphical user interface.

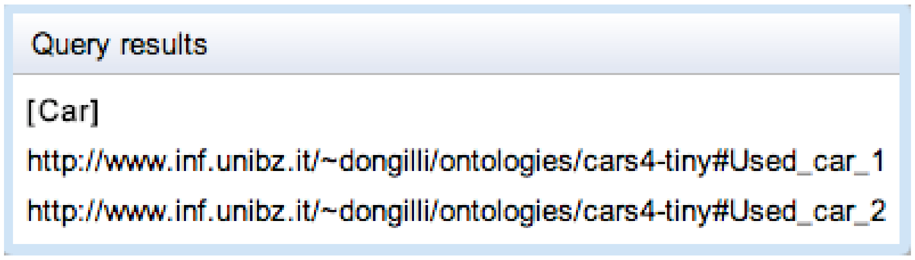
This report is structured as follows. In Section 2 we describe the Quelo framework for which the descriptions will be generated. In Section 3 we describe the NLG process followed to get the header verbalisations. Next we explain how the implementation was done in Section 4. Afterwards, in Section 5, we give the evaluation results. Finally, Section 6 presents some conclusions and future work.

2 Quelo Framework

Quelo is a Natural Language Interface for ontologies which attempts to cover the gap between the information stored in ontologies and the lay user [3]. The main idea is to allow the user to construct a natural language query according to an underlying ontology. The framework guides the user with menus in order to create a query which is consistent with the ontology. Because Quelo is a guided framework the user does not have to have any previous knowledge about the ontology structure or the language used to execute the query. The text shown to the user is automatically generated from the formal query language of the ontology with a Natural Language Generation (NLG) process based on templates.

The query changes according to the user's choices showing the verbalisation of the query in English. Given a query the user can extend it by adding compatible concepts or new properties for an existing concept. The user can also replace part of the query, or all of it, with a related concept; either a super-concept, an equivalent concept or a sub-concept. It is also possible to delete a selection of the query. The user may also select some concepts as the projected concepts for the query [9]. The user interaction we are interested in this project is the selection of the projected concepts. Most of the NLI systems previously mentioned only support the querying of one concept; however, in Quelo is possible to select more than one concept as the answer.

The current Quelo implementation shows the results of the query in a modal window as a table, where each of the columns has as the label of the concept as header. The answers for the query in Figure 1 (i.e. a set of unary tuples) can be seen in Figure 2. In this example the header is just the label Car which is the label of the concept in the ontology. For more complex queries, where there are several projected concepts, a more meaningful header will help the user understand the results presented. That is, to understand the relation among the concepts of the tuple and between these and the query.


Query results

[Car]

http://www.inf.unibz.it/~dongilli/ontologies/cars4-tiny#Used_car_1

http://www.inf.unibz.it/~dongilli/ontologies/cars4-tiny#Used_car_2

Figure 2: *Quelo results example*

The projected concepts are known as sticky nodes, because modifications to the query will not modify these nodes. In the user interface of Quelo the user selects the sticky nodes he is interested in by selecting the edge entering the node. In Figure 1 we can see an example of a query in Quelo. The node "a car" is a sticky node because the incoming edge "should sell" is selected as sticky, shown by the black border surrounding it.

All possible queries obtained with Quelo have a tree shape [5], so a query tree representation is used as middleman between the NL query and the ontology query. This representation allows to reflect back to the ontology query the changes in the NL query. Each node of this tree has a set of concept labels and each edge is also labelled with a relation name. This relation is a binary relation where the starting node is the first argument and the ending node is the second argument. Figure 3 shows some Quelo NL queries and their tree representation. Figure 3a shows the query tree obtained from the query shown in Fig. 1.

A query tree can be described in terms of a complex Description Logic concept which can be transformed into a First Order Logic (FOL) formula [10] where each node label of the query tree becomes a unary predicate and each edge node becomes a binary predicate with the start node as the first argument and the end node as the second argument. For example, the FOL formula for the tree in Fig. 3b would be $Car(x) \wedge run\_on(x, y) \wedge Fuel(y) \wedge sold\_by(x, z) \wedge Car\_dealer(z)$. In the formula all variables are assumed to be existentially quantified, except for the variables of the sticky notes, which remain free as they refer to the projected concepts. More details regarding the query translation can be found in [10].

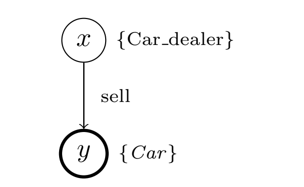
To obtain the query verbalisation the query tree is read in a depth first fashion. Roughly the verbalisation is obtained as follows. For each edge the starting node label acts as the subject of the sentence and the ending node label as the object of the predicate. The verb of the sentence is generated from the edge label. If a node has more than one label, copula sentences are added as in the verbalisation for the tree in Fig. 3c. For a more detailed explanation of the query verbalisation see [10].

One of the features of Quelo is its capability to display the query in natural language and interact with the user. The menus enable the user to easily construct a query and the use of natural language allows him to understand better the query. However, this ease of use is not present when the results are displayed.

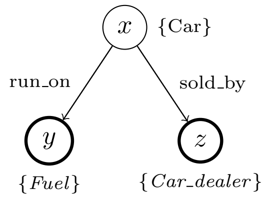
3 Verbalisation of query results

3.1 Proposal TAG

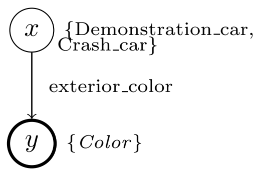
The current implementation of Quelo's NLI relies on templates to generate the NL text. When loading a new ontology for querying with Quelo's NLI, the first step is the automatic construction of a lexicon where all the concepts and relations of the ontology are mapped to a corresponding lexical class. Each lexical entry is associated to a template, with which the verbalisation of the query will



(a) *I am looking for a car dealer. It should sell a car.*



(b) *I am looking for a car. It should run on some fuel and it should be sold by a car dealer.*



(c) *I am looking for a demonstration car. It should be a crash car. Its exterior color should be a color.*

Figure 3: *Query tree examples*

be generated. These templates have fixed components and flags, which indicate if some elements should be added or not to the verbalisation. To obtain the verbalisation of a query, the query tree is traversed depth first from the root, adding to the verbalisation the template associated to each concept or relation.

All relation templates are of the form NP VP, where NP is a noun phrase and VP a verb phrase. Furthermore, this VP is of the form V NP, where V is a verb and it may start with the auxiliary should. The NP's may be just a determiner and a noun or two of such phrases connected with the preposition of. The verb for the VP may come from a relation name or may be the verb be. The NP nouns come either from the concept names or from relation names which have no verb element. As most of existing ontology verbalisers, Quelo Templates [10] assumes a unique mapping from semantics to NL. Given, for instance, a triple Subject Predicate Object it maps this triple to a NL expression of the form $[\mathrm{NP}_{Subject|Predicate}\ [\mathrm{V}_{Predicate|\mathsf{be}}\ \mathrm{NP}_{Object}]]$. Therefore, the verbalisation of a query based on templates is a sequence of NP VP with slots to be filled with concept or relation names. The system also applies ellipsis to remove redundancy composing two adjacent sentences if they have the same subject or the same subject and verb.

Whereas the template-based approach provides a grammatically correct verbalisation, the restrictions of the construction procedure make the verbalisation less fluent. The syntactic structures obtained are limited by the generation rules. Moreover, there is no formalism to describe the semantics of the text, the names of the concepts and relations just fill the slots in the templates without considering other possibilities.

Because of the templates lack of flexibility, it is not straightforward how to extend them to handle the header descriptions. As we will see in Section 3.2, relative clauses (e.g. 1a) and different type of argument realisations (e.g. 1b) are required. Thus, for the verbalisation of the results we adopt the domain independent grammar based approach proposed in [8] for the next update of Quelo. The grammar is a Feature-Based Tree Adjoining Grammar (FB-LTAG) equipped with unification based compositional semantics and the lexicon is automatically extracted from the given ontology.

(1) a. Car_dealer located_in_country Country
The country *in which the car dealer is located.*

b. Car sold_by Car_dealer.
This car dealer *sells* the car.

3.1.1 Grammar

Tree Adjoining Grammar (TAG) is a quintuple $\langle \Sigma, N, I, A, S \rangle$ where Σ is a finite set of terminals, N is a finite set of non-terminals, I a finite set of initial trees, A a finite set of auxiliary trees and S a distinguished non-terminal, the start symbol. Initial trees are trees for which all non-terminal nodes in the frontier are marked for substitution, by convention with a down arrow ($\downarrow$). Auxiliary

```
*ENTRY: located                                   *COANCHORS:
*CAT: v                                           WH ->  wh/np
*SEM: BinaryRel[rel=located_in_country]           V2 ->  be/v
*EQUATIONS:                                       P ->  in/p
anc -> funcrel = +
anc -> pos = ppart
```

Figure 4: *Lexicon entry*

trees are have exactly one non-terminal node in the frontier not marked for substitution, which is called the *foot node* and is marked with an asterisk (*). The label of the foot node must be the same as the label of the root node.

The allowed tree-composition operations between them are substitution and adjunction. Substitution inserts a tree into a frontier node of another tree marked for substitution. Adjunction inserts an auxiliary tree into any node of another with the same label as the root of the auxiliary tree. The TAG used in Quelo is a Feature-Based TAG, where two feature structures are associated to each node, top and bottom. A more detailed review of the implementation of TAG for Quelo is available in [8].

We will derive from corpora the trees which should be added to the grammar in order to obtain the header verbalisation. For each lexical class one or more trees were added to the grammar.

3.1.2 Lexicon

Following [10], the lexicon is automatically created from the ontology and a map of lexical classes to grammar structures. First, each concept or relation name is Part-of-Speech (POS) tagged. For example, for the relation located_in_country the parts of speech would be VBN, IN and NN respectively [1]. Depending on the parts of speech, a lexical class is assigned. Each lexical class is mapped to one or more grammar structures describing its realisation possibilities. Hence, per each concept or relation name there is one or more entries in the lexicon. The lexicon also includes information regarding number, person, animacy, functionality, and tense. The corresponding lexical class for located_in_country is *VBNBe*, a verb in past participle followed by a preposition. This lexical class is related to several grammar structures, some of which are the following:

- W1pnx1nx0VV : [NP_1] in which [NP_0] is located.
- W0nx0VVVpnx1 : [NP_0] which should be located in [NP_1].
- betanx0VPpnx1: [NP_0] located in [NP_1].

These entries will also have a feature to indicate that located_in_country is a functional relation and which is the tense of the verb in the label. The full lexical entry for W1pnx1nx0VV is shown in Fig. 4. The entry points that the entry name is "located", it is associated to the binary relation located_in_country and is of category verb. It also shows it is a functional relation and that the verb is in past participle[2]. All this information will be later used by the Referring Expressions Generator (Section 3.3.3). The coanchors show which type of nodes should be associated in the resulting syntactic tree. In this case there should be a Noun Phrase node for the relative pronoun, a verb node for the verb `to be` and a preposition node with the preposition `in`.

[1]Tag names from the Brown Corpus Tag-set

[2]The possible values for the feature `funcrel` are `+` (functional) and `-` (not functional). For feature `pos` the possible values are `ind` (indicative) and `ppart` (past participle). The absence of this feature indicates a verb in base form.

3.2 Language engineering and corpus analysis

In order to understand better the requirements of the NLG task and to explore the possibilities for displaying the description in Quelo's user interface, first we constructed corpora with examples of both the input and the expected output of the system. In order to construct the corpora nine ontologies with their given lexicographical elements were used [3]. The corpus had two main purposes, to help with the identification of new syntactic constructions for the grammar, and to obtain test cases for the evaluation.

3.2.1 Backward reading and relation names

In the query verbalisation, given a triple `(Subject, Relation, Object)` a set of possible verbalisations is enabled according to the syntactic pattern of the relations [10, 8]. In these verbalisations the logical subject in the triple corresponds to the syntactic subject. In the descriptions generator each verbalisation could be seen as a backwards reading of the relation and its components. The subject of the verbalisation is the range of the relation and the domain of the relation takes the place of the object in the verbalisation.

Depending on the components of the relation name the verbalisations for it may change. For example, in Fig. 3a the relation sell is composed by a transitive verb in simple present. Some possible verbalisations for this edge can be seen in Example (2) where *Car* is chosen as a sticky node. However, if the relation includes a preposition, as in the case of run_on on Fig. 3b, the possible verbalisations change, Example (3), sticky node *Fuel*. Also some verbs do not allow passivisation which reduces the number of possible verbalisations.

Furthermore the tense of the verb and the presence of other words may affect the verbalisation. Example (4) shows possible verbalisations for sold_by (Fig. 3b) with sticky node *Car_dealer*. Here the tense of the verb is past participle, and in this case the preposition by allows more variability in the verbalisations. For the edge exterior_color in Fig. 3c, we have as components an adjective and a noun. If *Color* is our sticky node we can verbalise the edge as in Example (5).

(2) a. Car_dealer sell Car
    b. A car which a car dealer *sells*.
    c. A car which *is sold by* a car dealer.
    d. A car *sold by* a car dealer.

(3) a. Car run_on Fuel
    b. A fuel *on* which a car *runs*.

(4) a. Car sold_by Car_dealer
    b. A car dealer *by* whom a car *is sold*.
    c. A car dealer who *sells* a car.

(5) a. Demonstration_car exterior_color Color
    b. A color which *is the exterior color of* a demonstration car.
    c. A color which a demonstration car *has as exterior color*.
    d. An *exterior color of* a demonstration car.

In order to explore the verbalisation possibilities, first a corpus of queries with just one edge and one sticky node was created. A total of 56 different queries with their corresponding header verbalisation composed this corpus. The relation names of the ontologies are clustered into 15 different classes according to their syntax pattern. These classes are an extension of the transformation rules found in [10]. These classes are the following:

[3]The domains of the ontologies were cameras, movies, wines, automotive market, e-commerce, tourism, masters organization, abilities and disabilities, and aquatic resource observations

1. *BEZIN* The verb form `is` plus a preposition.
   Example: isInSite: Event $\rightarrow$ Site

2. *BEZJJIN* The verb form `is` plus an adjective plus a preposition.
   Example: isSimilarTo: Thing $\rightarrow$ ProductOrService

3. *BEZNNIN* The verb form `is` plus a noun and a preposition.
   Example: isCoordinatorOf: Coordinator $\rightarrow$ MastersProgram

4. *BEZVBNIN* The verb form `is` plus a verb in past participle and a preposition.
   Example: isAffectedBy: LimbMobility $\rightarrow$ Muscle_Impairment

5. *CompositeNP* Related to the domain of the relation. None of the parts of the relation identifier is a verb.
   Example: equipment_of: Equipment $\rightarrow$ Car

6. *HVZRange* The verb form has and none of the other tokens is a past participle
   Example: has_trailer: Movie $\rightarrow$ Media

7. *NvNn* Predicate identifiers which are neither verbs nor nouns.
   Example: compatibleWith: Lens $\rightarrow$ Body

8. *NvNnIN* A *NvNn* which begins with a preposition.
   Example: ofClimaticZone: AquaticResourceObservation $\rightarrow$ ClimaticZone

9. *SimpleNP* Related to the range of the relation. None of the parts of the relation identifier is a verb.
   Example: exterior_color: Car $\rightarrow$ Color

10. *VB* The token is a verb in present simple.
    Example: sell: Car_dealer $\rightarrow$ Car

11. *VBIN* A *VB* plus a preposition.
    Example: run_on: Car $\rightarrow$ Fuel

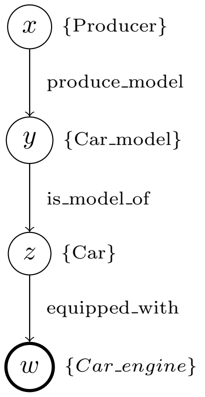
12. *VBNBe* A verb in past participle followed by a preposition. Denotes a fragment of a passive verb form.
    Example: sold_by: Car $\rightarrow$ Car_dealer

13. *VBNHave* A verb in past participle not followed by a preposition. Denotes a fragment of a active verb form.
    Example: approved: Student $\rightarrow$ Course

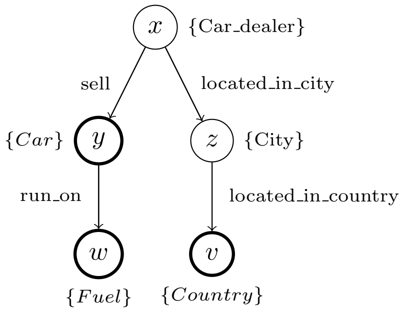
14. *VBZ* The token is a verb in present simple third person singular.
    Example: offers: Faculty $\rightarrow$ MastersProgram

15. *VBZIN* A *VBZ* plus a preposition.
    Example: plays_in: ActorParticipant $\rightarrow$ Movie

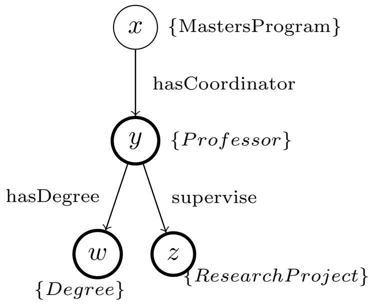
For each class one or more proposals of verbalisation were produced. Some of these verbalisations can be seen in Table 1. For example, for the relation run_on: Car $\rightarrow$ Fuel the verbalisation would read "The *fuel* on which the *car* runs.". Based on the human-authored verbalisations in the corpus we added new syntactic constructions to the existing grammar.



(a) *One sticky node, depth three.*



(b) *Three sticky nodes*



(c) *Three sticky nodes, one is parent of the other two*

Figure 5: *Query examples*

Lexical class	Relation	Verbalisation
BEZIN	isInSite: Event → Site	The site in which an event is.
HVZRange	has_trailer: Movie → Media	The media which is the trailer of a movie.
NvNn	compatibleWith: Lens → Body	A body with which a lens is compatible.
SimpleNP	exterior_color: Car → Color	The color which is the exterior color of a car.
VB	sell: Car_dealer → Car	A car which a car dealer sells.
VBIN	run_on: Car → Fuel	A fuel on which a car runs.
VBNBe	sold_by: Car → Car_dealer	The car dealer by whom a car is sold.
VBNHave	approved: Student → Course	A course which a student has approved.
VBZIN	plays_in: ActorParticipant → Movie	A movie in which an actor participant plays.

Table 1: *Verbalisation examples*

3.2.2 Content of the descriptions

We created a smaller second corpus with queries with only one sticky node but where this sticky node had a depth more than one. An example of this type of query is the query tree in Fig. 5a. The sticky node, marked in italics and with a thicker border, is *Car_engine*. One possible verbalisation is “A car engine with which a car, of which a car model is model, is equipped. This car model is produced by the producer.”. The verbalisations in this corpus read all the relations in the path from the sticky node till the root of the query. However, these verbalisations in most of the cases would be too long. We try to avoid a long verbalisation by inserting reference points and breaking it into smaller segments.

After creating this corpus we decided to break down the verbalisations in two parts. The first phrase would be used as the column header description and the rest of the verbalisation would be shown to the user on demand. In order to provide shorter but meaningful descriptions a maximum length of two relations per sentence was chosen. With more than two relations the sentence becomes too complex to read and it is less fluent. So, for the query in Fig.5a the header description for the sticky node *Car_engine* will be “A car engine with which a car is equipped.” and the long description “A car engine with which a car is equipped. Of this car a car model, which is produced by the producer, is the model.”.

Another corpus with 62 instances was created for queries with more than one sticky node. The presence of other sticky nodes affects the length of the verbalisation, because other sticky nodes can appear in the path till the query root. We decided to cut the description up to the next sticky node found in the path to avoid repetition. Figure 5b shows a query tree with three sticky nodes *Car*, *Fuel* and *Country*. If we assume that the nodes will be described in the given order, the verbalisation of the node *Fuel* will be "A fuel on which a car runs." and not "A fuel on which a car, which the car dealer sells, runs." because the verbalisation for *Car*, which we assumed precedes that of *Fuel*, already includes the information provided by the relative clause.

3.2.3 Ordering and referring expressions

The last example raised the issue of the order of the descriptions. To assign the order of the columns in the results table we take the same order of the nodes in the the query linearisation [3]. This linearisation is based in a depth-first traversal strategy. Therefore, for the query in Fig. 5b the order of the sticky nodes will be *Car*, *Fuel* and *Country* and for Fig. 5c the sticky nodes will be in the order *Professor*, *Degree* and *ResearchProject*.

Another purpose of this last corpus was to analyse referring expressions and, in particular, anaphoric references. We need to find the adequate referring expression for each verbalisation. This corpus helped us identify the rules to assign the best expression in each case. The referring expressions mainly present in this corpus are four classes of determiners; indefinite article, definite article, distal demonstrative and proximal demonstrative; and personal pronouns.

The header descriptions introduce the set of tuples answer to the query and describe these with respect to the query. We use indefinite articles in most of the cases because there may be several instances in the set of answers and we refer to none in particular. Thus, the header description for the answer in Fig. 2 will start with "*A* car sold by . . . ." which is equivalent to "One of the cars sold by . . . .".

The definite article is required when there is a specific instance of the concept which we are referring to. One example of the definite article is shown in Example (7a) where the concept *car dealer* is preceded by the definite article because this concept is a subsequent reference to the topic of the query, i.e. the complex concept described by the whole query (Fig. 5b). Another use of the definite article is in the case of functional relations as in Example (7c) for concept *country* and also for *city*. Although we are introducing the retrieved instances, for this case we know there is only one.

An example of proximal demonstratives is Example (6a). In this example, the concept *car model* is first introduced in a relative clause, where the focus of attention of the sentence is the subject, *car engine*. In order to shift the attention to *car model*, we use the proximal demonstrative determiner. These type of determiners are only present in the longer verbalisations.

Distal determiners are present when we refer to concepts mentioned in previous descriptions. For instance, Examples (7a) and (7b) show the verbalisations for two of the sticky nodes of Fig. 5b. In (7b) the distal determiner antecedes the concept *car* to indicate that this concept refers to the same concept already introduced in the previous verbalisation. In this case, we do not want to change the focus, we just want to point out that this concept was mentioned before.

(6) a. A car engine with which a car, of which *a* car model is model, is equipped.
This car model is produced by the producer.

(7) a. A car sold by *the* car dealer.
b. A fuel on which *that* car runs.
c. *The* country in which *the* city, in which the car dealer is located, is located.

In this corpus we use the personal pronouns `he` and `him` to replace NPs for animate concepts which have already been referred to. The verbalisations in (8) show the descriptions for the sticky nodes of the query in Fig. 5c, where both pronouns appear. For (8b) we replace the NP for *professor* with he because the concept is in the nominative case, whereas we replace it with `him` in (8c) because here the concept is the object of the phrase.

(8) a. A *professor*.
   b. A degree *he* has.
   c. A research project which is supervised by *him*.

3.3 Generation architecture

Natural Language Generation (NLG) is the task of obtaining a human readable text from an input which can be another text or an abstract representation of the information. In this case we want to generate the description of a projected concept with respect to a given query and the other projected concepts. Given a ontology K, a query Q, and a set of projected concepts S our task is to produce natural language descriptions for each $s_i \in S$ which is consistent with Q.

We use a pipeline architecture to divide the generation process into sequential tasks. Our pipeline consists of three major tasks organized a follows. The first task is document planning which determines the elements of the query to be used, linearises it, and divides it into sentences, generating a text specification. Then, we intersperse the surface realisation task with the REG task. The surface realisation generates a phrase structure tree from the text specification. The REG verbalises the remaining underspecified NPs.

3.3.1 Document Planning

The document planner input is a query and a set of projected concepts called sticky nodes. We use the tree representation of the query introduced in Section 2. The document planner obtains a linearisation of the path from the selected sticky node to the root of the query tree, selects a section as the header description, and divides it into sentences.

Inverse Path The linearisation of the path from a sticky node to the root is straightforward because each node has a reference to its parent. We limit this path to be until the root or another sticky node. The obtained linearisation will be a sequence alternating concepts and relations $c_0, r_0, c_1, r_1, \ldots c_k$, where c_0 is a label of a sticky node and c_k is the label of other sticky node or of the query root. The order of this sequence is given by the bottom-up reading of the tree. For the query tree in Figure 5b the linearisation for the sticky node *Country* will be Country, located_in_country, City, located_in_city, Car_dealer.

Content Selection The content selection module selects the content, i.e. list of messages, to be conveyed by the generated text. In our case, given a query and a set of sticky nodes, it needs to choose which subset of concepts and relations from the query should form part of the verbalisation of each sticky node.

As we discussed in Section 3.2.2 there will be two verbalisations for each sticky node, we will focus on the column header description verbalisation. The corpus created already gives one limitation regarding the number of relations, which should be maximum of two. We developed an algorithm to choose how many relations to include in each header description avoiding repetition.

The implemented algorithm maximizes the interrelation between the sticky nodes but minimizes the repetition. The input to the algorithm is the query tree Q and the set S of sticky nodes. The output will be, for each sticky node s_i, a list of tuples which will be encoded as discourse messages. The main purpose of the algorithm is to remove all those tuples which are not linked to any other tuple.

```
Message-id: msg01                 Message-id: invmsg01              Message-id: invmsg02
Relation: sell                    Relation: sell                    Relation: Country
Arguments:                        Arguments:                        Arguments: []
  arg1: Car_dealer [order=1]        arg1: Car_dealer [order=2]
  arg2: Car        [order=2]        arg2: Car        [order=1]
```

(a) *Message* **(b)** *Inverse message* **(c)** *Atomic message*

Figure 6: *Different type of messages*

Let s_i be a sticky node, s'_i be the parent of s_i and s''_i the parent of s'_i. There exists relations r and q so that $r(s'_i, s_i)$ and $q(s''_i, s'_i)$ are part of the query tree Q. For each node s_i let $\tau_i = \{(s_i, s'_i), \ldots\}$ be a list of tuples corresponding to the section of the path from the sticky node s_i to the root of Q or to another sticky node including a maximum of two relations and three different nodes, the initial text plan.

The algorithm is composed of three parts. First we create a set Σ with all the tuples of the initial text plans τ_i. For the query in Figure 5a Σ contains two tuples which are $\Sigma = \{(\mathsf{Car\_engine}, \mathsf{Car}), (\mathsf{Car}, \mathsf{Car\_model})\}$. For the query in Figure 5b, $\Sigma = \{(\mathsf{Country}, \mathsf{City}), (\mathsf{City}, \mathsf{Car\_dealer}), (\mathsf{Car}, \mathsf{Car\_dealer}), (\mathsf{Fuel}, \mathsf{Car})\}$.

The second step is to remove all the tuples from Σ for which the second element is not a stick node and it does not appear in another tuple. For the query in Figure 5a the method would delete the tuple $(\mathsf{Car}, \mathsf{Car\_model})$ because Car_model is not present in any other tuple and is not a sticky node, $(\mathsf{Car\_engine}, \mathsf{Car})$ would be also removed for the same reasons. However, for the query in Figure 5b the method would not delete the tuple $(\mathsf{Car}, \mathsf{Car\_dealer})$ because Car_dealer is an element of the tuple $(\mathsf{City}, \mathsf{Car\_dealer})$. The tuple $(\mathsf{Fuel}, \mathsf{Car})$ would also remain because Car is a sticky node and it also appears in the tuple $(\mathsf{Car}, \mathsf{Car\_dealer})$. This will make the verbalisations more cohesive because it avoids verbalising concepts which appear just once.

The last part of the algorithm is the reconstruction of the text plans for each node. For each sticky node s_i we add to the text plan τ_i the tuples (s_i, s'_i) and (s'_i, s''_i) if they exist in Σ and we remove them from Σ. However, if the resulting τ_i is empty there are two cases to consider. If s_i is referenced in any other tuple as the second element, τ_i will remain empty. But if the sticky node is not referenced in any other tuple still in Σ, the tuple (s_i, s'_i) will be added to τ_i. We add this tuple because we require that each projected concept has a context with respect to the query. This will add to the description of this sticky node the label of the incoming edge, which is the one used to mark as sticky nodes the concepts in the GUI as seen in Section 2.

The output of the module is a list of lists of discourse messages. The discourse messages of the query verbalisation are of the type shown in Figure 6a, where the arguments of the relation are in the same order as in the query. This message could be roughly translated into the NL query "A car dealer sells a car.". However, for the header descriptions new types of discourse messages are needed. Each tuple in τ_i represents an inverse message of the form shown in Fig. 6b. Because we traverse the query tree bottom-up the messages are inverted. A possible verbalisation for this inverse message would be "A car which is sold by a car dealer.". If τ_i is empty, the list of discourse messages for sticky node s_i will contain a single atomic message with the label of the node as relation name, as in Fig. 6c. A possible verbalisation of this message would be "A country.".

Content Segmentation In this module we group the messages into sentence sized chunks. We aggregate at most two messages per sentence, making the second message a relative clause. We decided to use at most two messages per sentence because sentences with nested relative clauses are more difficult to understand. For the header descriptions we have at most two messages, so there is no need to separate the messages as each description will have exactly one sentence. For the longer descriptions, we adopt a fixed size of two messages per sentence and divide the messages accordingly. The output of this module is the text specification.

3.3.2 Surface realisation

Before calling the Surface Realiser, the text specification should be translated into the semantics accepted by it, an *ordered* flat semantics formula. An ordered flat semantics formula is a flat FOL formula where each atomic predicate is annotated with order information. The order of the predicates is given by the text specification and it affects the result given by the surface realiser[8]. In a similar way than a query is translated into a FOL formula in Section 2 and [10] we translate the selected subparts of the query of each sticky node. The translation of the concepts is an unary predicate as well, but the translation of the relations differs slightly. Instead of a binary predicate we need a ternary predicate because we add a third argument for the variable of the event.

For the query tree in Fig. 3a, shown here for ease of reference, the text specification would have the inverse message shown in Fig. 6b. For the inverse messages the order of the arguments, and therefore the order of the predicates in the formula, is inverted. The order of the predicates of the inverse message would be Car sell Car_dealer(9a). The concept Car will be translated to $Car(y)$, sell to $sell(e_1, x, y)$ and Car_dealer to $Car\_dealer(x)$. The resulting ordered formula is shown in (9b) where each predicate has an ordering index.

The input for the Surface Realiser is the flat semantics (obtained from the text specification) plus the TAG grammar (Section 3.1.1) and the ontology lexicon (Section 3.1.2). The Surface Realiser will produce a realisation with syntactic structure and morpho-syntactic information. The Referring Expressions Generator adds features to this realisation to complete underspecified referring expressions. Finally, the Morphological Realiser assigns the correct inflections to each lexeme in the realisation to obtain a sentence.

Example (9) show all the steps involved in this module. The flat semantics formula (9b) is given to the Surface Realiser to obtain the realisation with underspecified NPs and morpho-syntactic information in (9c). The Referring Expressions Generator (described below, Section 3.3.3) will then add features, for instance to the determiners of each underspecified NP, to obtain the complete lemmatised phrase in (9d). In the example, the NP *car* gets assigned a indefinite article as determiner and the NP *car dealer* a definite article. At last, the morphological realiser replaces lemmas, according to the specified features, by the correct word forms. So, the indefinite determiner specification is realized by `A`, the definite by `the`, and *sell* is inflected to `sells`.

(9) a. Car sell Car_dealer
b. $car(y)[0]$ $sell(e_1, x, y)[1]$ $car\_dealer(x)[2]$
c. $[_{\text{NP}}$ car$]$ which $[_{\text{NP}}$ car dealer$]$ $[_{\text{VBZ}}$ sell$]$.
d. $Det_{[indef]}$ car which $Det_{[def]}$ car dealer $sell_{[ind]}$.
e. A car which the car dealer sells.

The surface realiser used is the one implemented on [8]. Given an ordered flat semantics formula, a TAG grammar and the lexicon, the surface realiser obtains several realisations sorted by *cost*,

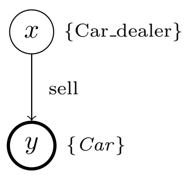


Figure 3a: *I am looking for a car dealer. It should sell a car.*

where the first realisation has the lowest cost. A realisation with a cost zero is one whose surface syntactic elements preserve the order given by the flat semantics elements. The realiser proceeds in four steps.

- **Lexical selection**: The SR selects from the grammar the elementary trees which have some part of the semantics of the flat semantics formula.
- **Tree combination**: These trees are combined via substitution and adjunction to get trees which cover the input formula.
- **Yield extraction**: All syntactically complete trees rooted at S (NP) and associated exactly with the input semantics are retrieved. Their yield provides the set of lemmatised sentences (noun phrases).
- **Morphological realisation**: The lemmas associated with morpho-syntactic features are replaced by the appropriate word forms.

Regarding the word forms for the verbs, the verb in the edge label may not be in the required verbal tense by the grammar structure. Therefore, a lemmatiser is needed to obtain the lemma of the verb and a reverse lemmatiser to inflect the verb into the adequate verb form. Both were obtained based on the software provided by [7].

3.3.3 Referring Expression Generator

The REG completes underspecified NPs of the syntactic realisation. For instance, it will specify the adequate determiner. The determiner could be an article, definite or indefinite, or a demonstrative, distal or proximal. The underspecification of some features in the nodes of the grammar causes that these determiners are not instantiated. The REG will add to the realisation the appropriate features and values for these nodes.

From the corpus analysis in Section 3.2.3 we derive the following rules for completing the REs. Each query has a topic determined by the root node of the query tree. If the concept is the topic, or was previously referred in an atomic message, e.g. "A car.", we assign a definite article. Concepts referred in previous verbalisations in the header will use the distal demonstrative `that`. For the longer verbalisation the determiner will be the proximal demonstrative `this`. Non previous referred concepts will be assigned an indefinite article, unless the concept is the topic of the query. Example (10) shows instances of each of these cases.

(10) a. *A* car dealer.
  b. A car which *the* car dealer sells.
  c. An equipment of *that* car. *This* equipment is ...
  d. *The* country in which the car dealer is located.

Example (10d) shows a special case when we also employ the determinative article, for functional relations. Given a relation r, r is said to be functional if $\forall x, r(x, y) \wedge r(x, y') \rightarrow y = y'$. That is, for each x there is at most one y in relation r with x. Therefore, when we refer to a y in relation r^- with x we are referring to the single y which is in relation r with x and we use the determinative article. If the ontology relations are annotated with functionality information, this will be present in the relations in the corresponding lexical entries and propagated by the features in the nodes of the grammar trees to the corresponding argument NPs through feature structure unification.

The pseudocode for the REG is given in Algorithm 1. For the realisation in (9c) the call will be have $rIdx$ set to zero, $refs$ and $atomicRefs$ as empty sets, *topic* set to *car dealer*, and *anim* to `NULL`. The last argument, *anim*, is a structure in where we will store the id of the animate concept to pronominalise and the index of the last description where this concept was mentioned. This structure is shared among the verbalisations for all the sticky nodes.

For the first NP of concept *car* as it is not an animate concept, neither the topic, nor part of a functional relation and $refs$ and $atomicRefs$ are empty the REG will give it a indefinite article as determiner (line 10) and $refs$ will be *car*. For the next NP, *car dealer* is the topic, so we will obtain a definite article (line 6). The procedure for the longer descriptions REG is similar and it is shown in Algorithm 2. The main difference is that we maintain an internal list for the references in the same description $sameRefs$ and the use of the proximal demonstrative in line 9.

Algorithm 1 Obtain the correct determiner for each NP in the realisation.

```
1:  procedure REG($r, rIdx, refs, atomicRefs, topic, anim$)
2:      for $[_{\mathrm{NP}}\ c] \in r$ do
3:          if usePronoun($c, rIdx, anim$) then                         ▷ see Algorithm 3
4:              realise $[_{\mathrm{NP}}\ c]$ with $Prn_{[sub|obj]}$              ▷ he or him
5:          else if $c$ is functional or $c \in topic \cup atomicRefs$ then
6:              realise $[_{\mathrm{NP}}\ c]$ with $Det_{[def]}\ c$                ▷ the
7:          else if $c \in refs$ then
8:              realise $[_{\mathrm{NP}}\ c]$ with $Det_{[dist]}\ c$               ▷ that
9:          else
10:             realise $[_{\mathrm{NP}}\ c]$ with $Det_{[indef]}\ c$              ▷ a or an
11:         $refs \leftarrow refs \cup c$
12:     if $r$ is atomic NP then
13:         $atomicRefs \leftarrow atomicRefs \cup c$
```

Algorithm 2 Obtain the correct determiner for each NP in the long description realisation.

```
1:  procedure longDescREG($longR, topic, otherRefs$)
2:      $sameRefs \leftarrow \emptyset$
3:      for $[_{\mathrm{NP}}\ c] \in longR$ do
4:          if $c$ is functional or $c = topic$ then
5:              realise $[_{\mathrm{NP}}\ c]$ with $Det_{[def]}\ c$                ▷ the
6:          else if $c \in otherRefs$ then
7:              realise $[_{\mathrm{NP}}\ c]$ with $Det_{[dist]}\ c$               ▷ than
8:          else if $c \in sameRefs$ then
9:              realise $[_{\mathrm{NP}}\ c]$ with $Det_{[prox]}\ c$               ▷ this
10:         else
11:             realise $[_{\mathrm{NP}}\ c]$ with $Det_{[indef]}\ c$              ▷ a or an
12:         $sameRefs \leftarrow sameRefs \cup c$
```

We use pronouns to obtain more concise descriptions. This type of references may add ambiguity; therefore, we use them only for one animate concept, if it exists. The realisation of the pronouns depends on the grammatical case of the concept (i.e. nominative or accusative). Each lexical entry has information in its features regarding animacy.

Based on the Centering Theory [4] we limit the use of pronouns to references which appear in consecutive descriptions. The Centering Theory proposed by Grosz *et al.* [4] provides two rules to obtain a highly coherent discourse depending on the changes of center of the utterance (the focus of attention) and the choice of referring expressions. Rule 1 limits the presence of pronouns in the utterances. It states that no NP can be realised by a pronoun unless the center is also realised by a pronoun. Rule 2 states that to achieve higher coherence it is preferred to retain the center through the utterances than to shift it. Therefore, concepts referred in *the previous* utterance are preferred to be the center than new ones. They also indicate that concepts in the subject position are more likely to be the center.

If we consider the set of header descriptions as a discourse, where each description is an utterance, these rules will limit the pronominalisation of concepts. Because we will pronominalise just one concept, by Rule 1 this concept should be the center of the description. By Rule 2 and the order of the descriptions, we conclude that a concept will continue to be the center only for consecutive descriptions.

In a description where none of the present concepts is referred in the preceding description, the preferred center will be the subject of it. A concept will only take the subject place in the verbalisation where this concept is the main concept described, when the description is of this concept. By the depth-first traversal order (Section 3.2.3), this will be the first reference of this concept, that is, there are no previous references to this concept. Therefore, any concept not referred in the preceding description and not in the subject position will not be the center of the description and cannot be pronominalised.

Examples (11) - (13) show the realisations obtained for three queries with the same projected concepts, but different structure, which gives them a different headers order. The concept *professor* is an animate concept indicated by the superscript plus sign (+). The underlined NPs are the centers of attention for each description. The changes of center follow the Centering Theory rules. The concept *professor* will be pronominalised in instances (11b), (12b), (12c) and (13b). For Example (13d) we realise the NP with a definite description instead of a pronoun because the center in this description is *research project*, not *professor*.

(11) a. $\underline{[_{\text{NP}} \text{ masters program}]}$ → A masters program.
b. $[_{\text{NP}} \text{ professor}^{+}]$ who is coordinator of $\underline{[_{\text{NP}} \text{ masters program}]}$. → A professor who is coordinator of the masters program.
c. $[_{\text{NP}} \text{ research project}]$ which $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ $[_{\text{VBZ}} \text{ supervise}]$ → A research project which *he* supervises.
d. $\underline{[_{\text{NP}} \text{ faculty}]}$ in which $[_{\text{NP}} \text{ masters program}]$ is $[_{\text{VBN}} \text{ given}]$. → A faculty in which the masters program is given.

(12) a. $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ → A professor.
b. $[_{\text{NP}} \text{ research project}]$ which $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ $[_{\text{VBZ}} \text{ supervise}]$. → A research project which *he* supervises.
c. $[_{\text{NP}} \text{ masters program}]$ of which $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ is coordinator . → A masters program of which *he* is coordinator.
d. $[_{\text{NP}} \text{ faculty}]$ in which $\underline{[_{\text{NP}} \text{ masters program}]}$ is $[_{\text{VBN}} \text{ given}]$. → A faculty in which that masters program is given.

(13) a. $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ → A professor.
b. $[_{\text{NP}} \text{ masters program}]$ of which $\underline{[_{\text{NP}} \text{ professor}^{+}]}$ is coordinator. → A masters program of which *he* is coordinator.
c. $[_{\text{NP}} \text{ faculty}]$ in which $\underline{[_{\text{NP}} \text{ masters program}]}$ is $[_{\text{VBN}} \text{ given}]$. → A faculty in which that masters program is given.
d. $\underline{[_{\text{NP}} \text{ research project}]}$ which $[_{\text{NP}} \text{ professor}^{+}]$ $[_{\text{VBZ}} \text{ supervise}]$. → A research project which *the professor* supervises.

The function in Algorithm 3 (which is called in Algorithm 1 line 3) shows how we encode these restrictions. We keep track of the last description index in which the animate concept was referred in *anim.idx* and of this concept id in *anim.cp*. The only case we decide to pronominalise is when the referred concept is the same as the animate concept in *anim.cp*, and the index of the description is the consecutive index to that in *anim.idx* (line 3).

Algorithm 3 Decide whether pronominalise or not.

```
1: function usePronoun($c, descIdx, anim$)    ▷ $anim$ stores the animate concept id and last reference index
2:     if $c$ is not animate then return FALSE
3:     if $anim.cp = c$ and $descIdx = anim.idx + 1$ then
4:         increment $anim.idx$
5:         return TRUE
6:     else if $anim.cp =$ NULL then
7:         $anim.cp \leftarrow c$
8:         $anim.idx \leftarrow descIdx$
9:     return FALSE
```

3.3.4 NLG Functional API

The available operations for the descriptions of the headers of the results table are two, `getDescriptions` and `getLongDescriptions`.

- `getDescriptions` Given a list of sticky nodes, it will return a NL description for each one of them in a list. The query to which these sticky nodes belong is implicitly passed to the method as it can be reconstructed from the sticky nodes.

- `getLongDescriptions` This operation also accepts as parameter a list of sticky nodes and returns the long descriptions for each of them.

These operations will call the NLG pipeline to produce the descriptions required. Both operations can be called by the Web app to obtain the verbalisations for the headers and integrate them to the results table.

4 Implementation

4.1 From Query Generation to Description Generation

We based the implementation in the existent structure for the query verbalisation. For the verbalisation of the query, an incremental generator was developed in [8]. Even though the NLG pipeline used in both instances is the same, there are differences in the implementation of each phase.

Document planning We added new types of messages to support inverse messages and atomic messages. The aggregation method was changed to reflect a maximum of two messages per sentence.

Surface realisation Although we use the same surface realiser, the parameters to call differ, as for the short description the expected result is a noun phrase, e.g. "A car.", not a sentence as in the query verbalisation and the long description. Also we added support for the lemmatisation and inflection of verbs required to obtain a description verbalisation, e.g. change from `sold_by` to "A car dealer which *sells* the car.".

Referring expressions generation We had to implement the new algorithm for REG, as the query generation does not include the use of demonstratives or pronouns. The use of functional roles to assign a determinative article is also a new part of this module.

4.1.1 Structure and design

The implementation of the description generator was coded in Java. The main package of the implementation is the package `it.unibz.quelo.nlg.dscgen` which contains all of the new classes created for the headers verbalisation. Figure 7 shows the packages and classes added to generate the description. The packages are organized as follows:

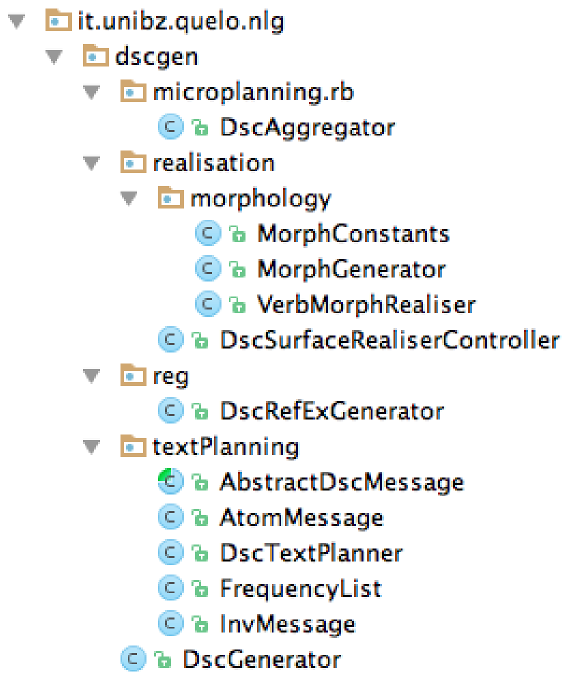
- `it.unibz.quelo.nlg.dscgen.textPlanning` provides all the classes needed to generate the text plan.
- `it.unibz.quelo.nlg.dscgen.microplanning.rb` includes the aggregator class needed to obtain the text specification.
- `it.unibz.quelo.nlg.dscgen.realisation` contains the class used to call the Surface Realiser with the appropriate arguments depending if we ask for the short or long description.
- `it.unibz.quelo.nlg.dscgen.realisation.morphology` includes the classes necessary for the verb morphology.
- `it.unibz.quelo.nlg.dscgen.reg` has the implementation of the Referring Expressions Generator.
- `it.unibz.quelo.nlg.dscgen` contains the class which calls all the other classes to generate the verbalisations.

Figure 8 shows the class diagram for the new classes. The main class is `DscGenerator` where we call the text planner `DscTextPlanner`, which will produce per each node the `DscTextPlan` conformed of `InvMessages` or a `AtomMessage`, both subclasses of the abstract class `AbstractDescMessage`. `DscTextPlanner` uses the auxiliary class `FrequencyList` to construct the text plan for each sticky node as explained in Section 3.3.1. Later these messages are aggregated into sentences by the `DscAggregator` to produce the text specification.

Each message of the text specification has the method `loadTo` which returns the flat semantics corresponding to it. The flat semantics will be used by `DscSurfaceRealiserController` to call the surface realiser and obtain the realisations. The surface realiser used is `IncChartRealiser` form the package `it.unibz.quelo.nlg.icgen` which is the incremental surface realiser used to obtain the query. The realisation obtained from the surface realiser is decorated by the class `DscRefExGenerator` with more features for the determiners and underspecified noun phrases according to the algorithm described in Section 3.3.3.

Finally, the `VerbMorpRealiser` will be called to change the tense of the verbs into the correct ones with the help of the morphological generator `MorphGenerator`, which is based in the code developed by K. Humphreys, J. Carroll and G. Minnen [7]. While they included the inflection and lemmatisation of nouns, we removed this part as it is not necessary for our implementation. The core of their code is a lexer written in Flex which we translated into JFlex to obtain Java code.

One of the aspects of the implementation worth mentioning is that the flat semantics formula implementation allows for refinements regarding the type of grammar structures to use in the verbalisation. The grammar structures are associated with syntactic classes and it is possible to indicate in the formula which syntactic class is preferred for each predicate. This was specially useful to cut out undesired realisations. Example (14) show an example of such formula where the preferred syntactic class for the node is indicated in the second pair of square brackets.



```
it.unibz.quelo.nlg
  dscgen
    microplanning.rb
      DscAggregator
    realisation
      morphology
        MorphConstants
        MorphGenerator
        VerbMorphRealiser
      DscSurfaceRealiserController
    reg
      DscRefExGenerator
    textPlanning
      AbstractDscMessage
      AtomMessage
      DscTextPlanner
      FrequencyList
      InvMessage
    DscGenerator
```

Figure 7: *Description generator packages and classes*

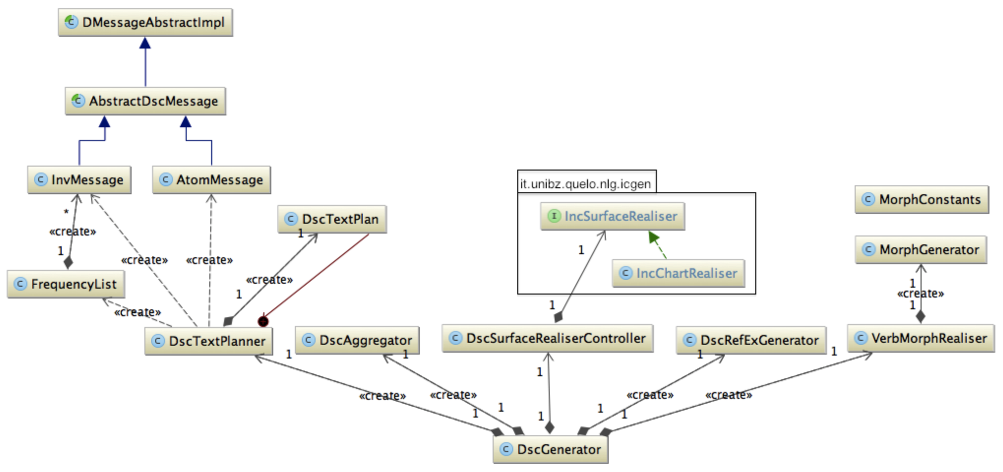


Figure 8: *Package dscgen class diagram*

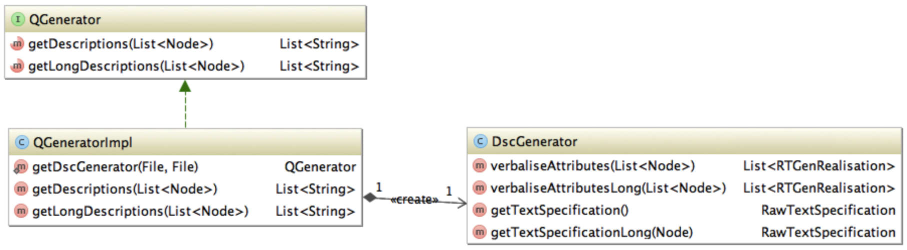

QGenerator
getDescriptions(List<Node>) List<String>
getLongDescriptions(List<Node>) List<String>
QGeneratorImpl
getDscGenerator(File, File) QGenerator
getDescriptions(List<Node>) List<String>
getLongDescriptions(List<Node>) List<String>
1
«create»
1
DscGenerator
verbaliseAttributes(List<Node>) List<RTGenRealisation>
verbaliseAttributesLong(List<Node>) List<RTGenRealisation>
getTextSpecification() RawTextSpecification
getTextSpecificationLong(Node) RawTextSpecification

Figure 9: *Description generator interface class diagram*

(14) $car(y)$[0][Unary-argument] $sell(e_1, x, y)$[1] $car\_dealer(x)$[2][Unary-argument]

The interface `QGenerator` of the package `it.unibz.quelo.nlg` is the main interface to access the methods provided by the Description API. This interface is implemented by the class `QGeneratorImpl`. The methods exposed by this interface are:

1. `getDescriptions(List<Node> stickyList): List<String> descriptions`

2. `getLongDescriptions(List<Node> stickyList): List<String> longDescriptions`

Figure 9 shows the class diagram for this interface, its implementation and how it is related to `DscGenerator`.

4.1.2 Sample code

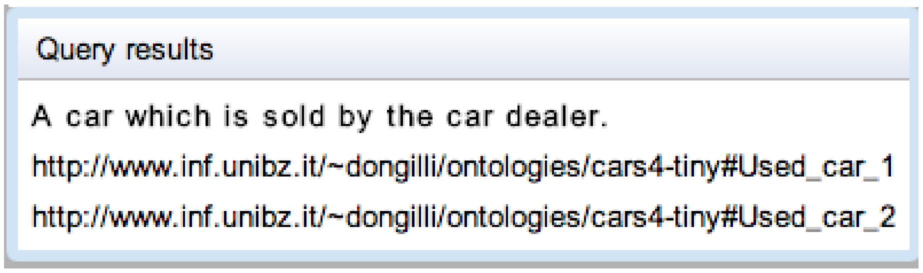
Here we will present some examples of use for the Description API in the integration to the Quelo Web app.

Initialising the generator In order to get an instance of `QGenerator` we call the static method `getDscGenerator(File resourcesBasepath, File lexiconFile)` of the class `QGeneratorImpl`. The first argument indicates where the resources used by the generator can be stored or found. The generator will try, for each resource, to read it from `resourcesBasepath`, if it is not found it will extract it from the library and store it there. The second argument `lexiconFile` is the lexicon file of the ontology for which we want to generate the descriptions. The grammar used in the generation is domain independent and therefore it is given by the API; however, it is necessary to assign the lexicon of the ontology [4]. Listing 1 shows an example of how to invoke this method.

```java
try{
  File resBasepath = new File("full/path/to/resources/root");
  File lexiconFile = new File("full/path/lexicon_file.xml");
  QGenerator qg = QGeneratorImpl.getDscGenerator(resBasepath,
      lexiconFile);
}catch(IOException e){
  e.printStackTrace();
}
```

Listing 1: *Initialise the Description Generator for an ontology.*

[4]This lexicon can be obtained automatically from the ontology


Query results

A car which is sold by the car dealer.

http://www.inf.unibz.it/~dongilli/ontologies/cars4-tiny#Used_car_1

http://www.inf.unibz.it/~dongilli/ontologies/cars4-tiny#Used_car_2

Figure 10: *Web-app integration*

Obtaining the descriptions The descriptions for a list of sticky nodes are obtained calling the method `getDescriptions(List<Node> stickyNodes)`, as in Listing 2 line 4. The arguments `stickyNodes` should be a list of `Nodes` from the package `it.unibz.quelo`, which are obtained from the query. This method returns a list of strings, where each string is the description of the `Node` in the same position. To obtain the long descriptions the call to the method `getLongDescriptions(List<Node> stickyNodes)` is analogous, Listing 2 line 5.

```
1 List<Node> stickyNodes = ...; // Get sticky nodes from query
2
3 QGenerator gen = ...; // Get QGenerator
4 List<String> descriptions = gen.getDescriptions(stickyNodes);
5 List<String> lDescriptions = gen.getLongDescriptions(
      stickyNodes);
```

Listing 2: *Obtain descriptions for a sticky node list.*

4.2 Web-app integration

The Quelo architecture is a Client-Server application based in GWT (Google Web Toolkit). The client is a Web-application which displays the NL query. The user interacts with this Web-app to construct the query using menus.

On the server side there are two main subsystems: the Query Logic (QL) subsystem and the NLG subsystem. The server must integrate the GUI with these two subsystems. The QL subsystem, developed in [5] , is where all the logic operations take place. The NLG subsystem currently implemented in Quelo is the one developed in [10] which is based in templates.

In order to integrate the header verbalisations into the Quelo Web-app we modified both the server and client code. In the server side, we added the descriptions to the results object and on the client side we display these descriptions as the header of the results table. The description was added to each header of the column results of Quelo. In Figure 10 we can see new results table for the query on Fig. 1. This table now has a header which describes the obtained results with respect to the query asked.

5 Evaluation

In order to evaluate the performance of our implementation, we compared the descriptions obtained with our module with the manually written descriptions for a query corpus. First we tested a corpus of 123 queries with just one sticky node and one relation. For each query more than one verbalisation was produced. Table 2 shows a summary of the results for this test.

As mentioned in Section 3.3.2 the surface realiser sorts the obtained realisations by cost. The first verbalisation is the one with the lowest cost and the one that will be used in the Web-app. For

	Correct	Incorrect	Blank	Total	Coverage	Correctness
First verbalisation	102	19	2	123	98.37%	84.30%
Second verbalisation	11	101	11	123	91.06%	9.82%

Table 2: *Results for CL1*

this test we obtained 98.37% of coverage, where only for two queries no verbalisation was generated. Of the obtained verbalisations, the first verbalisation was correct on 84.3% of the cases. For the second verbalisation, the correctness decreased drastically to 9.82%. This decrease in correctness shows that the cost function used, which is based in the order of the flat semantics, usually selects a correct verbalisation.

In Table 3 the results for the first verbalisation are divided into the lexical classes described in Section 3.2.1. The classes with a higher number of incorrect cases are *SimpleNP*, *NvNnIN* and *CompositeNP*. Examples (15) - (17) show some generated verbalisations where the first verbalisation is in (a.) and the second one is in (b.). Examples (15) and (16) have the same generated verbalisation for different relations, because the grammar structure used removes the nouns present in the relation. Whereas the noun `movie` is present in the range of both relations, the noun `photo` (`trailer`) is not present in neither the domain nor the range and it is necessary to keep the sense of the verbalisation. Example (17) shows a case where this construction is used successfully. The second verbalisation is too verbose in this case.

(15) *CompositeNP* photo_of_movie: Media $\rightarrow$ Movie
  a. The movie of the media.
  b. The movie of which the media *is the photo*.

(16) *CompositeNP* trailer_of_movie: Media $\rightarrow$ Movie
  a. The movie of the media.
  b. The movie of which the media *is the trailer*.

(17) *CompositeNP* soundtrack_of_movie: Soundtrack $\rightarrow$ Movie
  a. The movie of the soundtrack.
  b. The movie of which the soundtrack *is the soundtrack*.

The two cases where there are no verbalisations are due to missing definitions in the lexicon, therefore the surface realiser cannot do the lexical selection to produce a verbalisation[5]

A second corpus of 40 queries, each with one or more sticky nodes was used to test the Referring Expressions module. This corpus has 115 verbalisations. Table 4 show the results obtained for this corpus first realisation, aggregated by the lexical class of the relation. For the verbalisations with more than one relation, the first relation determined the lexical class. For this test suite we had a coverage of 99.13%. The only blank verbalisation was also because of a missing definition in the lexicon. The correctness for this test suite was of 97.82%. For this corpus, none of the verbalisations for the lexical classes of *SimpleNP* and *CompositeNP* is correct. However, non of the errors where due to incorrect assignment of determinative or pronouns. The errors found were similar to the ones found in the previous corpus[6].

[5]These two relations are directed: Director $\rightarrow$ Movie and isToSite: Distance $\rightarrow$ Site

[6]The results summary files for the tests can be found in the project repository under doc/results-verbalisation/evaluation

Lexical class	Correct	Incorrect	Blank	Total	Coverage	Correctness
SimpleNP	3	5	0	8	100%	38%
NvNnIN	6	5	0	11	100%	55%
CompositeNP	13	4	0	17	100%	76%
BEZNNIN	4	1	0	5	100%	80%
HVZRange	36	4	0	40	100%	90%
VBNHave	1	0	1	2	50%	100%
BEZIN	3	0	1	4	75%	100%
NvNn	2	0	0	2	100%	100%
VB	2	0	0	2	100%	100%
VBIN	1	0	0	1	100%	100%
VBNBe	19	0	0	19	100%	100%
VBZ	6	0	0	6	100%	100%
VBZIN	6	0	0	6	100%	100%
TOTAL	102	19	2	123	98.37%	84.30%

Table 3: *Detailed results for CL1*

Lexical class	Correct	Incorrect	Blank	Total	Coverage	Correctness
CompositeNP	0	11	0	11	100%	0%
SimpleNP	0	2	0	2	100%	0%
VBZIN	4	1	0	5	100%	80%
VBNBe	23	4	1	28	96%	85%
VBIN	6	1	0	7	100%	86%
HVZRange	29	3	0	32	100%	91%
VBZ	10	1	0	11	100%	91%
BEZNNIN	1	0	0	1	100%	100%
NvNn	2	0	0	2	100%	100%
VB	10	0	0	10	100%	100%
Atomic	6	0	0	6	100%	100%
TOTAL	91	23	1	115	99.13%	79.82%

Table 4: *Detailed results for CL2*

6 Conclusion and Future Work

In this project we implemented the whole NLG pipeline for the verbalisation of the headers. Because it was the implementation of the full NLG pipeline, the project included references to several other areas of Natural Language Processing, as the centering theory, theories for the use of demonstratives and the reverse lemmatisation of verbs.

One of the most important parts was the content selection. The developed algorithm provides a verbalisation which allows for each sticky node to have some context with respect to the query, without it being too long. For the surface realisation we used a surface realiser for TAG grammars. TAG grammars provide a linguistically principled mechanism for NLG. One of its main features is that allows to model different syntactic contexts for a given ontology concept or relation. Furthermore, the grammar combined with an automatic lexicon extraction approach provides a domain independent generation framework, which for the implementation for Quelo is essential.

The input for the SR is an ordered flat semantics formula. The order of the formula provides a sorting mechanism for the obtained verbalisations. With this mechanism we obtained a correct verbalisation in 80% of test cases evaluated. However, there are some cases where the verbalisation with the lowest cost was not the best verbalisation. For example "An equipment of a car" is an acceptable verbalisation for equipment_of:Equipment, Car but the verbalisation "The color of the car." loses some information when verbalising exterior_color: Car, Color. A better verbalisation would be "An exterior color of a car.". Future work could include the implementation of a better selection algorithm. We would like to add some ranking mechanism to choose which is the best verbalisation in each case. One possibility is the training of a classifier as the one used for the query verbalisation.

We could add support for anaphoric references for inanimate objects. For example, we can use the pronoun `it` in (18b) because the relation between both concepts is semantically strongly related to the concept we are referring to. However, for the Example (18c), it is not the case and the pronoun would make the description ambiguous. (Would `it` refer to the car or to the fuel?). However, in Example (19c) though there are two antecedents (*country* and *car*) the relation run_on is strongly related to *car*, not to *country*, and the reference is unambiguous. We could improve the current REG by detecting selectional restrictions; however, to learn how to detect them is beyond the scope of this project.

(18) a. A car.

b. The fuel on which *it* runs.

c. The country in which *the car* is located.

(19) a. A car.

b. The country in which *it* is located.

c. The fuel on which *it* runs.

Additional future work would be the user-based usability evaluation of the approach. As we mentioned in the introduction, the goal of the NLIs is to provide a user friendly interface based in NL and in Quelo's case an interface for ontologies. Therefore we need to do a usability test with real users to test the real impact of this tool. With this evaluation we could get feedback regarding the ease of use of Quelo.

A major area of improvement is the presentation of the verbalisation in the Web-app. The longer verbalisation could be included in the interface via a button and another popup window. In order to make the results more clear to the user, we could also include a verbalisation of the tuples of the query answers. That is, for the first tuple in Figure 2 we could add a verbalisation such as "The used car 1 is sold by the car dealer Auto Center Wetzikon".

Also a lexicon editor should be added to allow the correction of the lexicon automatically generated. There are some ambiguous cases where human feedback is needed, for example, for the relation offer if it is considered a verb one verbalisation could be "...which uses ..." whereas if it is considered a noun it could be "...which is an offer of ...".

For the next iteration of Quelo, we will add constants to the query. The current Quelo covers a wide range of queries; however, the results obtained may be further restricted with the use of constants.

The development of the project allowed the author to study in depth the NLG pipeline and its phases. Also a better understanding of the natural language module of Quelo was achieved. TAG grammars offer a solid solution for NLG which support several forms of syntactical manipulation, as the one needed for the header verbalisation.

References

[1] Abraham Bernstein and Esther Kaufmann. GINO-a guided input natural language ontology editor. *The Semantic Web-ISWC 2006*, LNCS 4273:144–157, 2006.

[2] Danica Damljanovic, Milan Agatonovic, and Hamish Cunningham. FREyA : an Interactive Way of Querying Linked Data using Natural Language. In *Proceedings of the 8th international conference on The Semantic Web*, pages 125–138, Crete, Greece, 2012. Springer-Verlag.

[3] Enrico Franconi, Paolo Guagliardo, Sergio Tessaris, and Marco Trevisan. A natural language ontology-driven query interface. In *9th International Conference on Terminology and Artificial Intelligence*, page 43, 2011.

[4] Barbara J Grosz. Centering : A Framework for Modeling the Local Coherence of Discourse. *Computational Linguistics*, 21(2):203–225, 1995.

[5] Paolo Guagliardo. *Theoretical Foundations of an Ontology-Based Visual Tool for Query Formulation Support.* Masters thesis, Free University of Bozen-Bolzano, December 2009.

[6] Esther Kaufmann, Abraham Bernstein, and Renato Zumstein. Querix : A Natural Language Interface to Query Ontologies Based on Clarification Dialogs. In *In 5th International Semantic Web Conference*, number November, pages 980–981, Athens, G, 2006.

[7] Guido Minnen, John Carroll, and Darren Pearce. Applied Morphological Processing of English. *Natural Language Engineering*, 7(3):207–223, 2001.

[8] Laura Perez-Beltrachini, Claire Gardent, and Enrico Franconi. Incremental Query Generation. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 183–191, Gothenburg, Sweden, 2014.

[9] Marco Trevisan. A Graphical User Interface for Querytool. Technical report, KRDB, Bolzano, Italy, 2009.

[10] Marco Trevisan. *A Portable Menu-Guided Natural Language Interface to Knowledge Bases.* Masters thesis, Free University of Bozen-Bolzano, December 2009.

[11] R. Valencia-García, F. Garcia-Sanchez, D. Castellanos-Nieves, and J.T. Fernandez-Breis. OWLPath: An OWL ontology-guided query editor. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 41(1):121–136, 2011.

[12] Chong Wang, Miao Xiong, Qi Zhou, and Yong Yu. PANTO: A portable natural language interface to ontologies. In *Proceedings of the 4th European conference on The Semantic Web: Research and Applications*, pages 473–487, Innsbruck, Austria, 2007. Springer-Verlag.