



Multilingual Online Translation

Non multa, sed multum

Grammar-Ontology Interoperability - Final Work and Overview

Contract No.:	FP7-ICT-247914
Project full title:	MOLTO - Multilingual Online Translation
Deliverable:	D4.3A Grammar-Ontology Interoperability - Final Work and Overview
Security (distribution level):	Public
Contractual date of delivery:	-
Actual date of delivery:	May, 2013
Type:	Annex to deliverable
Status & version:	Final
Author(s):	Maria Mateva ¹ , Inari Listenmaa ³ , Aarne Ranta ² , Ramona Enache ² , Dana Dannélls ² , Laura Toloşi ¹
Task responsible:	Ontotext ¹
Other contributors:	Ontotext, UGOT ² , UHEL ³

ABSTRACT

D4.3A is an annex to the D4.3 deliverable of WP4 of the MOLTO project. It aims to address the reviewers' remarks and recommendations for D4.3, as well as to present a final overview of the prototypes built in the scope of MOLTO with respect to grammar-ontology interoperability. D4.3A also describes the work after M24 and gives a general overview of the achievements in MOLTO with focus on WP4 - Knowledge Engineering, WP7 - Patents use case, and WP8 - Cultural Heritage use case.

Contents

1	Introduction	4
2	From GF Grammars to SPARQL	7
2.1	Mapping rules and an FSA auto-complete module	7
2.1.1	The mapping rules approach	7
2.1.2	Data lexicons	8
2.1.3	Auto-complete with FSA and GF	8
2.1.4	Advantages and disadvantages	8
2.2	GF as means to generate SPARQL queries - YAQL	9
2.2.1	YAQL	9
2.2.2	SPARQL generation model - Patents use case	10
2.2.3	SPARQL generation model - Cultural Heritage use case	11
2.3	Creation of a query language	11
2.4	Discussion on levels of automation	11
3	GF Grammars Generation from Ontology	12
3.1	The GQHB Tool and the Grammar-Ontology Helper for the GF Eclipse Plugin	12
3.2	Generation of GF grammars from RDF triples	13
3.2.1	Subjects and objects representation	14
3.2.2	Predicates representation and predicates types	15
3.2.3	Software implementation	18
3.3	NL description generation from semantic results. Cultural Heritage use case	19
3.3.1	NL answer generation from semantic results	19
3.3.2	Object description. Cultural Heritage use case	19
3.4	Lexicon generation with TermFactory	20
3.5	Discussion on levels of automation	22
4	Comparison and Integration between KRI and TermFactory	23
4.1	Comparison	23
4.2	Integration	24
5	MOLTO Prototypes	26
5.1	MOLTO KRI	26
5.1.1	Prototype description	26
5.1.2	Grammar-ontology interoperability	26
5.2	MOLTO Patents	29
5.2.1	Prototype description	29
5.2.2	Grammar-ontology interoperability	29
5.3	MOLTO Cultural Heritage	31
5.3.1	Prototype description	31

5.3.2	Grammar-ontology interoperability	31
6	Porting KRI to New Applications	33
6.1	Strengths and limitations of our natural language interface to the semantic repository	33
7	Conclusion	35
8	References	36
9	Appendix A: Groups of RDF predicate types	38

1 Introduction

Within MOLTO we explore the interoperability between Gramatical Framework (GF)¹ and ontologies. Following [Con11] we build three prototypes and surrounding tools, which are described in [MI10], [Dam11], [CEDR12], and in the present document.

Semantic data is usually presented in the RDF standard², in which facts are described as sets of triples. Traditionally semantic data is queried via the SPARQL³ query language for semantic databases. Hence, the gap between the natural language (NL) of a common user and the semantic data remains open. Controlled languages are a possible logical solution, since they provide formal interpretation of a natural language query. In the current document we present our exploration of Gramatical Framework as means to provide query language and response verbalization of the semantic repository data.

Our goal is to build a retrieval system, given a specific ontology, that can be queried with relevant NL questions and can also return NL answers. GF facilitates the process by providing abstract representation of natural language query sentences, which can be processed further. Additionally, it provides multilinguality at a comparatively low cost. Figure 1 demonstrates the workflow of the projected retrieval system.

We explored the following directions in order to gain full interoperability between the grammars and the semantic datastore:

1. Given an ontology (RDF graph), create grammars for queries that can be asked to the retrieval system
2. Given a GF abstract representation of a query sentence, generate a corresponding SPARQL query
3. Given an ontology (RDF graph), create grammars for answers that will be generated from the RDF results. Given an RDF result, generate an NL answer via the GF answer grammars

Mainly within WP4, but also as a continuation in WP7 and WP8, we experiment with different approaches to utilize and possibly automate these steps. We achieve semi-automated SPARQL generation and template-based grammar generation. We build tools and prototypes, which are presented in Section 5 and Section 3.1. Our experience shows that the more GF is directly involved in the interoperability, the more flexible and reliable the technology is.

GF transforms the NL query parsing to abstract representation, which can more easily be converted to other machine and NL languages. Most of the steps are done automatically, see Figure 1. The transition between NL and GF concrete grammar and vice versa

¹<http://www.grammaticalframework.org/>

²<http://www.w3.org/RDF/>

³<http://www.w3.org/TR/rdf-sparql-query/>

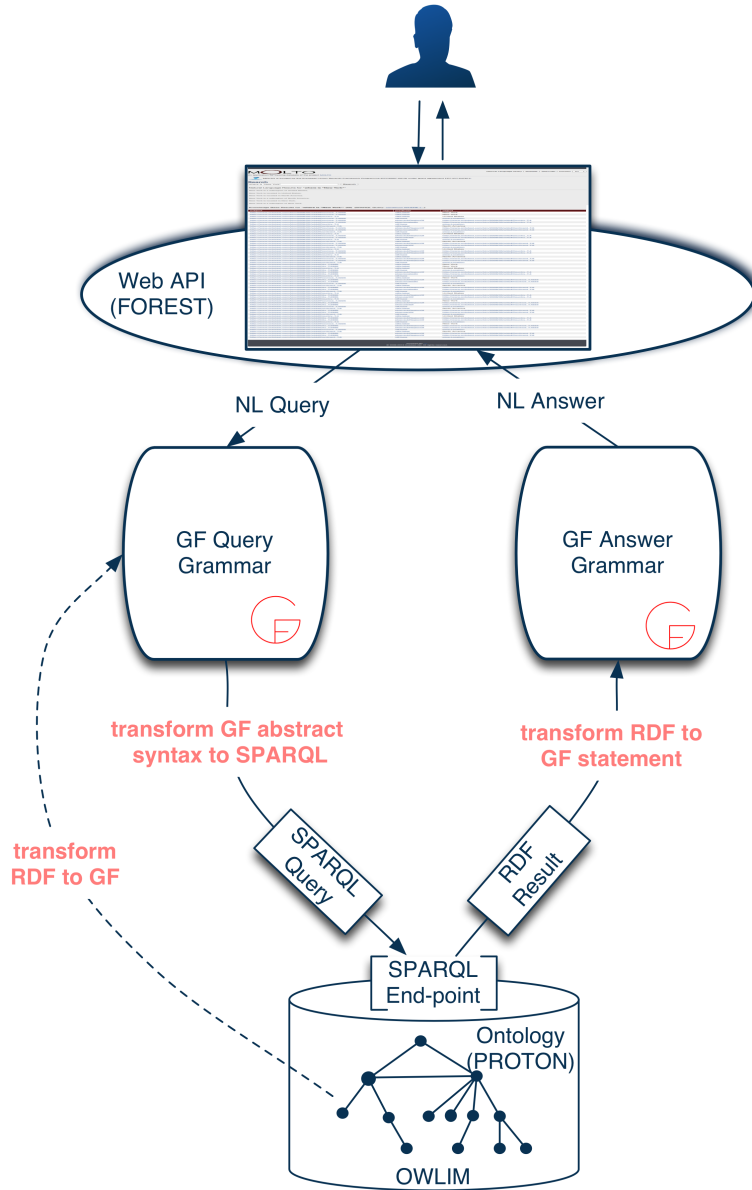


Figure 1: GF-Ontology interoperability

is done directly by the GF parser. The parser transforms the GF concrete representation to GF abstract, as well as from a call to GF concrete grammar to a NL answer. The semantic datastore, which in our case is OWLIM⁴, is responsible for the RDF answer of the SPARQL query.

The first and the third steps are very similar as there an existing RDF graph is mapped

⁴<http://www.ontotext.com/owlim>

to a GF grammar. Our approach is to create grammars that map entities from the ontology with the help of predefined templates. Section 3 explains this in details. The transition from the GF abstract representation to SPARQL query has been explored in different ways, two of which have been chosen and are presented in Section 2. These include mapping rules between GF syntax trees and GF SPARQL grammars, exploring SPARQL as another ‘natural’ language. A third approach, with discursive patterns, is described in [CEDR12].

Based on our approaches for interoperability, we build a retrieval system prototype, called the (MOLTO) Knowledge Representation Infrastructure (KRI)⁵, which uses GF for natural language questions and answers. The KRI prototype is also the basis of two MOLTO use cases, the WP7 Patents prototype⁶ and the WP8 Cultural Heritage(Museum) prototype⁷ (see Sections 5.2 and 5.3), where we develop further its ideas.

At the end of this document, we describe potential usage of lexicon generation, lexicon generation with TermFactory and integration between KRI and TermFactory. The steps to customize the KRI to a new specific domain are given in Section 6, according to the recommendations from the M24 Review Report.

⁵<http://molto.ontotext.com>

⁶<http://molto-patents.ontotext.com/>

⁷<http://museum.ontotext.com/>

2 From GF Grammars to SPARQL

The first direction of interoperability, which we explore, is the generation of SPARQL⁸ queries, given a specific query language and GF grammars that define it. This task has no straightforward solution. We have integrated two different approaches in our prototypes:

- mapping rules that use a predefined domain specific language;
- generation of SPARQL via GF translation, regarding SPARQL as another “natural” language.

In the child projects of MOLTO KRI, the first technology was abandoned and the second was developed further.

2.1 Mapping rules and an FSA auto-complete module

Our initial approach for NL-to-SPARQL generation was to build a model of explicit mappings between GF abstract trees and SPARQL queries. This was achieved with mapping rules, for which we built a domain specific language and parser. We didn’t use any direct communication with the GF service or process. Instead, we applied automatically generated trees and hand-written rules. Additionally, we built a custom auto-complete mechanism - based on a finite state automaton (FSA), which loaded all previously generated trees and their linearizations as strings. It used lexicons for different types (e.g. *Person*, *Location*, *Organization*, *Drug*, etc.).

As a result, we found that the mapping rules were a redundant level of abstraction and did not explore GF in the best possible way. However, the auto-complete based on FSA proved to be useful and fast enough, so it remained part of our MOLTO prototypes (See Section 5).

2.1.1 The mapping rules approach

We built a small domain specific language and parser for the mapping rules. The language aimed to provide syntactic sugar for some generalized mappings between GF and SPARQL, as briefly described in [CEDR12]. The following rule is an example for such a mapping rule from the KRI prototype:

```
//all people with their aliases
//all organizations with their aliases
(QSet ?X) | single(X) && type(X) == "" && name(X) != Location ->
construct WHERE {
  sparqlVar(name(X)) rdftype() class(name(X)) .
  sparqlVar(name(X)) property(hasAlias) sparqlVar(name(X)) ## ".alias". };
```

Here the left-hand side is the GF parsed tree syntax (QSet ?X). name(X) gives the type of the entity X, e.g. *Location*, *Organization*, *Person*, *Job Title*. On the right-hand side of

⁸<http://www.w3.org/TR/rdf-sparql-query/>

the rule we declare a “construct” query that corresponds to the `QSet` sentences. We use `sparqlVar` to get the name of the variable (e.g. `?name`), the `class` function - to return its RDF type, and the `property` function - to return the full name of the “hasAlias” predicate.

The example demonstrates how we created dynamic generation of SPARQL queries, depending on their type, which is inferred from the data lexicon type we allow for this abstract GF tree.

2.1.2 Data lexicons

Data lexicons are typed lexicons that are previously extracted from the semantic repository. They can be used both for the auto-complete module (see Section 2.1.3) and the concrete GF grammars. A good example of this scenario is the patents prototype ([MGE⁺13]), where we have 8 dictionaries: Drugs, Active Ingredients, Patent Numbers, Application Numbers, Routes Of Administration, Dosage Forms, TE Codes and Markets. The list of entities is extracted from aligned ontologies in the biomedical domain. They are integrated both in the grammars and in the auto-complete module. We translate the lexicons from English into French and German with statistical machine translation (SMT) by UPC. It has been possible due to the translation of the semantic annotations as described in [MGE⁺13]. This way we achieve completeness of the French and German query languages. In the cultural heritage use-case acquiring such data is more challenging since we need multilingual values in 15 languages. Our progress is reported in [MGE⁺13] and [DDE⁺13]. We provide more information on lexicon extraction in Section 3.4.

2.1.3 Auto-complete with FSA and GF

The three Ontotext prototypes use a custom auto-complete module, which is based on a minimalistic acyclic FSA fed with linearized GF grammar trees and patterns. The resources are preliminary generated from the GF grammar of the target language(s) and mainly consist of generated sentences from the grammar and data lexicons for each data entity type. The latter are data types that are retrieved from the RDF database. The data lexicons are useful for the generation of both the FSA resources and the GF grammar itself. In fact, the initial motivation for using FSA, instead of the GF auto-complete, is to avoid getting heavy lexicons from a large semantic repository, whose entities would be too many to be efficiently supported by GF. As a result, we found FSA very useful because of the control it gives to the displayed queries and also the control of suggested auto-completion types.

2.1.4 Advantages and disadvantages

The mapping rules allow certain dynamic generation of SPARQL. However, they are difficult to debug and maintain when there are changes in the grammar or in the desired SPARQL syntax. The parser for the rules also requires support. Moreover, this approach does not fully benefit from GF. For example, GF can be used for direct translation to SPARQL so the mapping rules and the invocation of their parser become redundant.

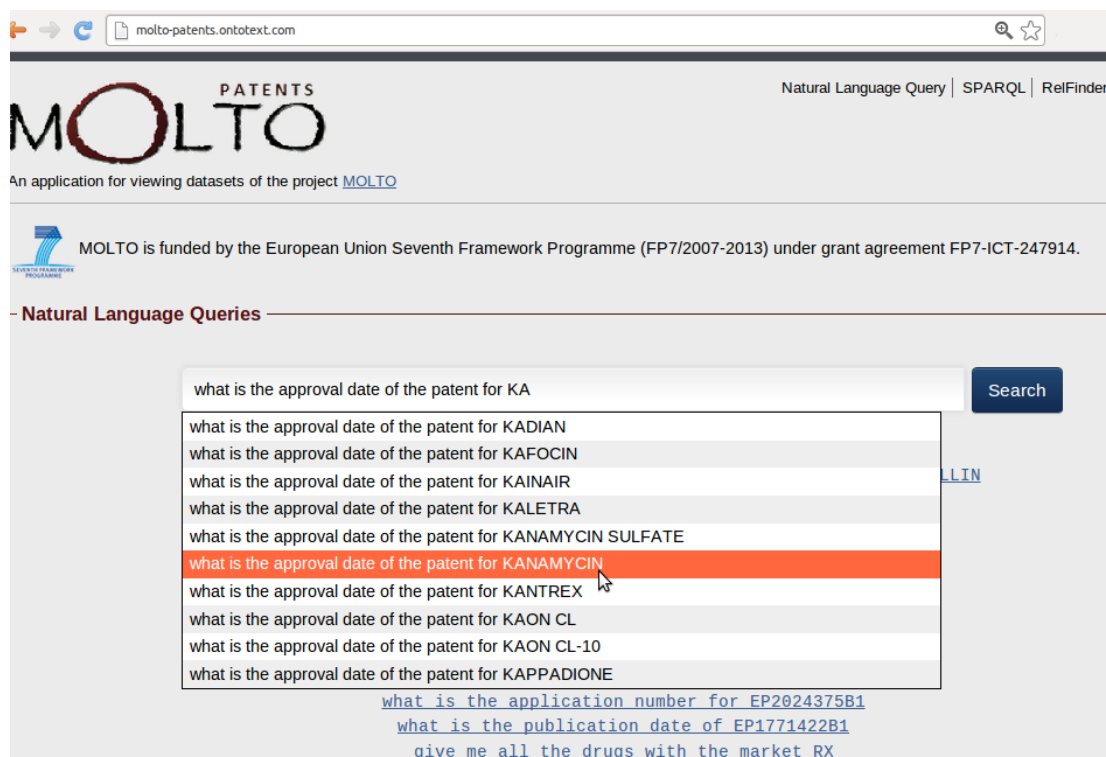


Figure 2: MOLTO KRI autocomplete in the MOLTO-patents prototype

2.2 GF as means to generate SPARQL queries - YAQL

Within MOLTO, the consortium partners came up with the idea to review SPARQL as a natural language from a GF point of view. This means that, in addition to abstract and concrete grammars for all natural languages, we create a SPARQL concrete grammar and we observe SPARQL as a natural language into which we need to translate.

2.2.1 YAQL

In the final year of MOLTO, UGOT contributed to WP4 with the creation of the YAQL (“yet another query language”) grammar module, which provided the basis for domain specific SPARQL generation. The implementation of a query language in YAQL is explained in [Ran12]. YAQL has straightforward abstract syntax generation from ontology, with just the minimum of lexical types:

```
Kind : usually CN
Entity : usually NP
Property : can be VP, AP, ClSlash
Relation : VPSlash built from V2, AP, comparatives
```

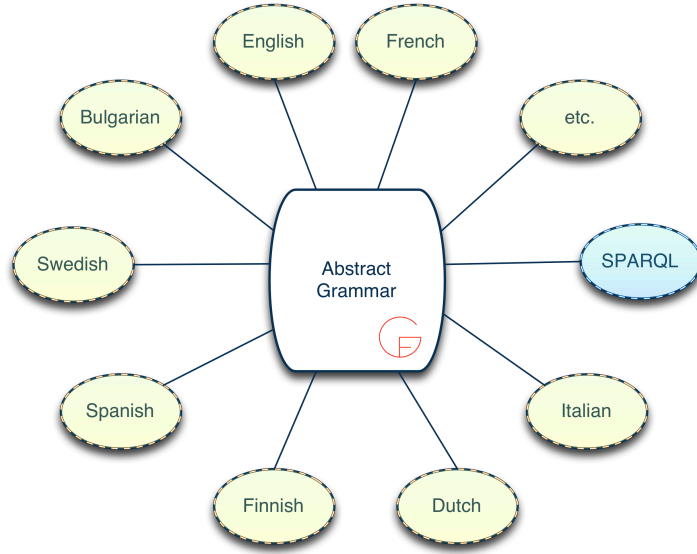


Figure 3: SPARQL as another GF language

2.2.2 SPARQL generation model - Patents use case

The SPARQL generation model of WP7 was described in [MGE⁺13]. It is the first implementation of SPARQL generation via GF that we applied in a KRI-based prototype. The model uses YAQL. It allows the mapping a GF function (abstract query expression) to a SPARQL query. The example below shows the abstract, NL concrete and SPARQL concrete representations of the “show patents that mention X and Y” query as developed for the *PatentQuery*⁹ grammar.

Abstract syntax:

```
QShowConceptXY : Concept -> Concept -> Query ;
```

Concrete natural language interface:

```
QShowConceptXY c1 c2 = mentionP (mkNP and_Conj c1 c2) ;
```

Concrete SPARQL syntax:

```
QShowConceptXY c1 c2 =
{s = "PREFIX pkm: <http://proton.semanticweb.org/protonkm#> $n
PREFIX psys: <http://proton.semanticweb.org/protonsys#> $n
CONSTRUCT { $n ?doc pkm:mentions ?x . $n ?doc pkm:mentions ?y } $n
WHERE { $n ?x psys:mainLabel " ++ c1.s ++ ". $n
?doc pkm:mentions ?x . $n ?y psys:mainLabel " ++ c2.s ++ ". $n
?doc pkm:mentions ?y . $n } " } ;
```

⁹svn://molto-project.eu/wp7/query/grammars

The SPARQL generation in this case is similar to the mapping rules but it relies on GF for translation and does not involve additional layers of pre-defined rules.

2.2.3 SPARQL generation model - Cultural Heritage use case

The SPARQL generation model of WP8 is described in [DRE+13] and [DDE+13]. It uses YAQL as well but it is more dynamic than the one in WP7. The approach allows parametrized building of SPARQL queries. The example below is from the *QueryPainting* grammar¹⁰, *QueryPaintingSPARQL.gf*:

```
QPainter p = {wh1 = "?painter"; prop = p ; wh2 = " painting:createdBy ?painter ." } ;
QYear p = {wh1 = "?date"; prop = p ; wh2 = " painting:hasCreationDate ?date ." } ;
QMuseum p = {wh1 = "?museum"; prop = p ; wh2 = " painting:hasCurrentLocation ?museum ." } ;
QColour p = {wh1 = "?color"; prop = p ; wh2 = " painting:hasColor ?color ." } ;
QSize p = {wh1 = "?dim"; prop = p ; wh2 = " painting:hasDimension ?dim ." } ;
QMaterial p = {wh1 = "?material"; prop = p ; wh2 = " painting:hasMaterial ?material ." } ;
```

We believe that this is the direction in which SPARQL generation from GF grammars should be developed when it needs to be applied in different domains. Similar ideas are presented also in WP6 - Sage syntax generation, WP11 - ACE syntax generation, and WP12 - the business modeling domain grammars in the Be Informed Studio.

2.3 Creation of a query language

Our experience in the KRI-based prototypes shows that the better the allowed questions cover actual relations in the ontology graph, the better the query language is. This gives us motivation to search for query patterns that adequately cover the predicates in the RDF representation of the ontology. The verbalization methodology suggested in Section 3.2 can utilize automatic example-based creation of queries for the ontology by selecting probable GF (query) patterns for the different types of RDF predicates. But, due to the variety of possible question types (“what”, “which”, “who”, “how many”, etc.), this method cannot be highly precise and the resulting grammars would need in-depth expert revision. Hence, the semi-automatic approach remains the optimal solution to the creation of a query language.

2.4 Discussion on levels of automation

Adopting GF as described in Section 2.2 allows direct translation from natural language to SPARQL (or any other machine language, such as ACE or Sage languages). The level of automation depends on the complexity of the required query language and on the implementation of the SPARQL generation grammars. Hence, a more dynamic approach requires more efforts of GF experts. We also observe the need of a (simple) testing framework that can validate the soundness of the generated machine language. In any case, this model is preferable to the earlier approach of mapping rules.

¹⁰svn://molto-project.eu/wp8/d8.3/grammars

3 GF Grammars Generation from Ontology

In the scope of WP4, we have chosen to represent ontologies in RDF because each fact in the RDF graph has its own semantics. It is defined by a triple of **subject**–**predicate**–**object**, which we aim to verbalize in the following ways:

- **SUBJECT is PREDICATE PREP OBJECT**. Example: “Ontotext is located in Bulgaria.”
- **SUBJECT HAS-PREDICATE OBJECT**. Example: “Sweden has capital Stockholm.”
- **SUBJECT VERB-PREDICATE OBJECT**. Example: “Hilton Group PLC owns Livingwell.”

GF is very suitable for verbalizing such sentences as well as for providing their multilingual representation. Moreover, GF allows multiple linearizations of the same grammatical entity (e.g. “all patents”, “show all patents”, “what is the information about all patents”, etc.). For a better perspective on this, first we provide additional information on our previous work and its development throughout MOLTO.

3.1 The GQHB Tool and the Grammar-Ontology Helper for the GF Eclipse Plugin

The GQHB tool, described in [CEDR12], is extended to a wizard as part of the GF Eclipse plugin([Cam12]). This work logically belongs to WP2 but due to its late completion it could not be included in the respective deliverables. The wizard manual is publicly available at Github¹¹.

The wizard functionality is almost identical to the GHQB tool. It connects to a SPARQL endpoint, inspects the ontology and provides lists of classes as well as their respective instances. It also allows the user to create GF categories from the selected classes and GF entities from these categories. These entities are verbalized with the RDF labels of the instances from the ontology. This generation is based on a template defined by a GF expert. The structure of the xml template is:

```
- "templates" - root element
  - "template" - template for a single query
    - "pattern" - the query displayed to the user
    - "fun" - abstract grammar's initial functions (without the generated ones)
    - "lincat" - concrete (English) grammar's initial linearization categories
    - "lin" - concrete (English) grammar's initial linearizations
    - "sparql" - SPARQL query, usually with pattern
    - "sparql-lin" - concrete SPARQL grammar's initial linearizations
```

An example of the template can be found at [svn://molto-project.eu/wp4/projects/molto-core/molto-repository-helper/resources/template.xml](https://molto-project.eu/wp4/projects/molto-core/molto-repository-helper/resources/template.xml). As a result, three concrete grammars are generated: English concrete (or any other language), SPARQL concrete and an abstract grammar.

¹¹https://github.com/GrammaticalFramework/gf-eclipse-plugin/blob/master/README_ontology-grammar.md

A screenshot of the wizard GUI is shown on Figure 4.

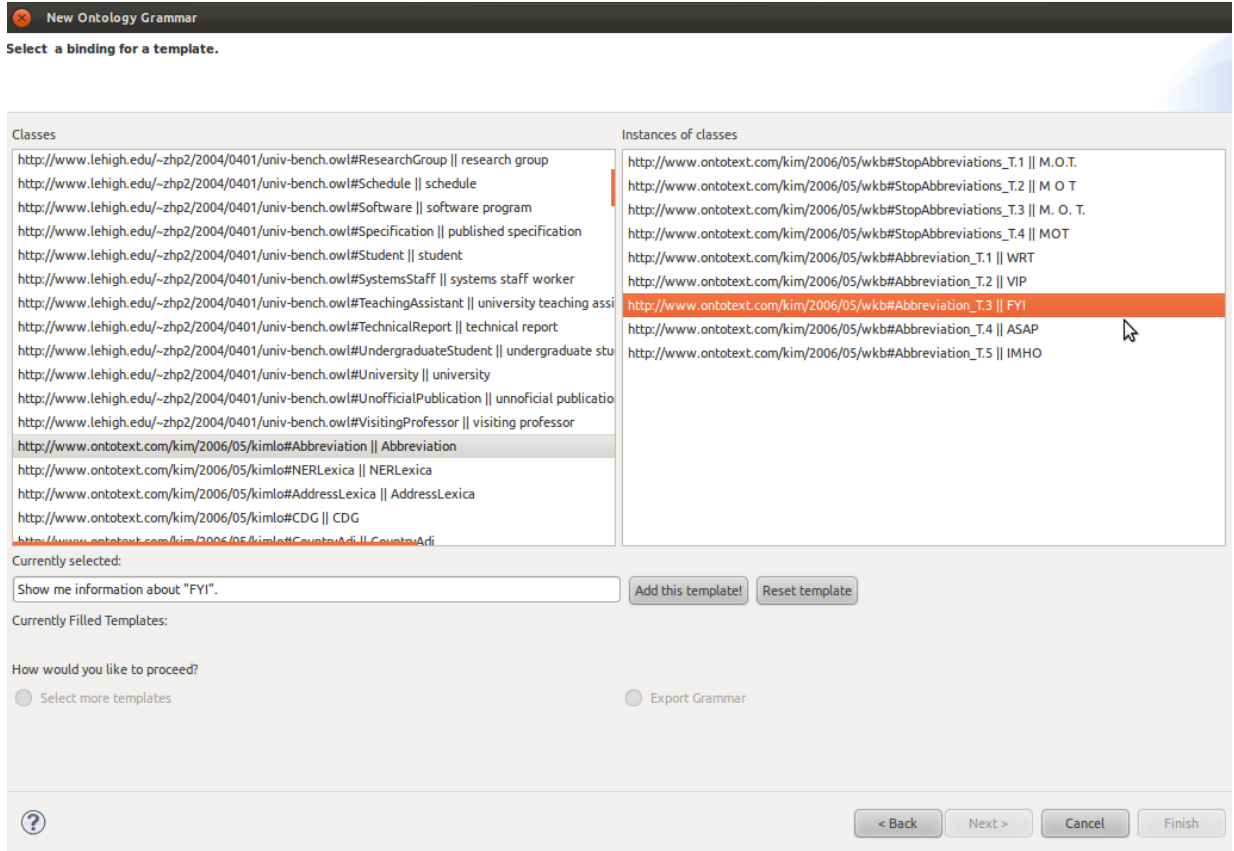


Figure 4: Grammar-Ontology Helper - selection of instances of classes to generate GF grammars

The motivation for such a tool and wizard is to enable GF non-experts to create GF grammars on-line. In fact, this method is closely related to lexicon generation. However, the creation of the templates requires GF expertise in the generic grammar model creation.

3.2 Generation of GF grammars from RDF triples

Throughout the work on the MOLTO KRI prototype (see Section 5.1), and inspired by the idea of a generic grammar, described in [CEDR12], we created a generator of generic grammar that explores ontology facts and generates (data) grammars for them. The automatic generation of grammar entities and semi-automatic generation of grammars for the RDF predicates (GF expressions) comprise our model for verbalization of RDF triples.

The idea continues from earlier work within the MOLTO project. [Ena10] describes the translation of the SUMO¹² axioms to GF, the linearizations written mostly manually. Our approach automatizes some of the work, taking advantage of the usually consistent naming

¹²Suggested Upper Merged Ontology

conventions of the RDF predicates. [Lis12] is a feasibility study, which describes different uses of ontologies in lexicon management. As a conclusion, generation of GF grammars from RDF data turned out to be promising. Other directions taken by [Lis12] and the University of Helsinki(UHEL) include pure lexicon generation (see Section 3.4).

We use the RDF objects and subjects with their URIs and labels to create respectively abstract and English concrete GF representations. The verbalization of predicates is further investigated in the final year of MOLTO. It has also been of scientific interest to external research groups ([AGL13]).

The motivation for this is that RDF predicates are usually meaningful and follow a strict naming convention, especially in a single ontology. This is not always true and sometimes RDF predicates names are in different languages (e.g. in DBpedia¹³).

Hereby we present a method to verbalize RDF predicates, in case that they are in English and follow a naming convention. It is based on in-depth analysis of 1700+ predicate names from 10 random public repositories. It is worth pointing that the repositories are not pre-selected in any manner. A number of observations and heuristics lead to the separation of the predicates into groups that allow easier verbalization via example-based GF grammars. We describe the results in 3.2.2.

For predicates in other languages we can utilize similar algorithm but for some of them it requires additional morphological analysis.

3.2.1 Subjects and objects representation

For RDF **subjects** and **objects**, GF representation is straightforward - all are presented in GF by the **Entity** category. This model can be further extended, for example by defining more specific type (GF category) of the Entity and thus have lots of categories (*Person*, *Location*, *Organization*, etc.). These categories usually are named after the respective class in the ontology, from which we take instances for the grammar. We tried this differentiation in the MOLTO KRI prototype but we found no practical benefit in splitting the categories into the grammar, except for easier lexicon types splitting, which facilitates the auto-complete technology.

The corresponding abstract and concrete grammar representation for the first five PROTON ontology¹⁴ class instances in alphabetical order is as follows:

Abstract syntax:

```
cat Entity;

fun
  Airline_T_1 : Entity;
  Airline_T_10 : Entity;
  Airline_T_10_0 : Entity;
  Airline_T_10_1 : Entity;
  Airline_T_10_2 : Entity;
  ...
```

¹³<http://dbpedia.org/>

¹⁴<http://www.ontotext.com/proton-ontology>

Concrete syntax:

```
lincat
  Entity = Str ;

lin
  Airline_T_1 = "Japan Airlines System Corporation" ;
  Airline_T_10 = "Cathay Pacific Airways, Ltd." ;
  Airline_T_10_0 = "Cathay Pacific Airways, Ltd." ;
  Airline_T_10_1 = "Cathay Pacific Airways, Limited" ;
  Airline_T_10_2 = "Cathay Pacific Airways" ;
  ...
```

The linearization data is taken from the English RDF labels.

The Entity category serves to represent ontology subjects and predicates in GF. To achieve complete automation of the GF verbalization of an RDF triple, we need to focus on the predicates. In the case of MOLTO KRI we provide representation for a number of useful predicates. They are chosen after observation of the graph results to each `construct` query¹⁵.

Normally the user does not need to verbalize all predicates. For example, `rdf:type` is one we preferred to skip, as “John Smith has type Person.” is useless information in our case.

3.2.2 Predicates representation and predicates types

The abstract representation clearly has automatable parts - e.g. the one that defines the predicates. The next example is a shortened version of the GF abstract grammar for the PROTON predicates verbalized in the MOLTO KRI prototype.

```
cat
  Phrase;
  Property;
  [Property]{2};

fun
  text: Entity -> Property -> Phrase;
  and: [Property] -> Property;

  activeInSector: Entity -> Property ;
  childOrganizationOf : Entity -> Property ;
  hasCapital : Entity -> Property ;
  holder : Entity -> Property ;
  locatedIn : Entity -> Property ;
  owns : Entity -> Property ;
  ...
```

The corresponding concrete English syntax also reveals repetitive behaviour. This is proved by the GF patterns for `activeInSector` and `locatedIn`, as well as the one for `hasCapital` and `holder`. The latter is verbalized as “has holder”. Certainly, GF allows

¹⁵`construct` queries return graph over certain description. Hence, they always return RDF triples.

multiple verbalizations of a single pattern and this flexibility can contribute to more diverse expression of the produced answer language.

```

lincat
  Phrase = S;
  Property = VPS;
  [Property] = [VPS];

oper s2e : Str -> NP = \s -> symb (mkSymb s) ;

lin
  BaseProperty = BaseVPS;
  ConsProperty = ConsVPS;
  and ps = ConjVPS and_Conj ps;
  text x y = PredVPS (s2e x) y;

  activeInSector x = MkVPS (mkTemp presentTense simultaneousAnt)
    positivePol (mkVP (mkA2 (mkA "active") (mkPrep "in")) (s2e x)) ;

  hasCapital x = MkVPS (mkTemp presentTense simultaneousAnt)
    positivePol (mkVP (mkVPSlash have_V2) (mkNP (mkCN (mkCN (mkN "capital")) (s2e x)))) ;

  holder x = MkVPS (mkTemp presentTense simultaneousAnt)
    positivePol (mkVP (mkVPSlash have_V2) (mkNP (mkCN (mkCN (mkN "holder")) (s2e x)))) ;

  locatedIn x = MkVPS (mkTemp presentTense simultaneousAnt)
    positivePol (mkVP (mkA2 (mkA "located") (mkPrep "in")) (s2e x)) ;

  owns x = MkVPS (mkTemp presentTense simultaneousAnt)
    positivePol (mkVP (mkV2 "own") (s2e x)) ;
  ...

```

Having the answer grammar generated, what remains is to have it verified by an expert. Afterwards, a call to it is made for each RDF triple from a result set that has to be verbalized. Our MOLTO KRI prototype uses the below linearization call to the GF engine:

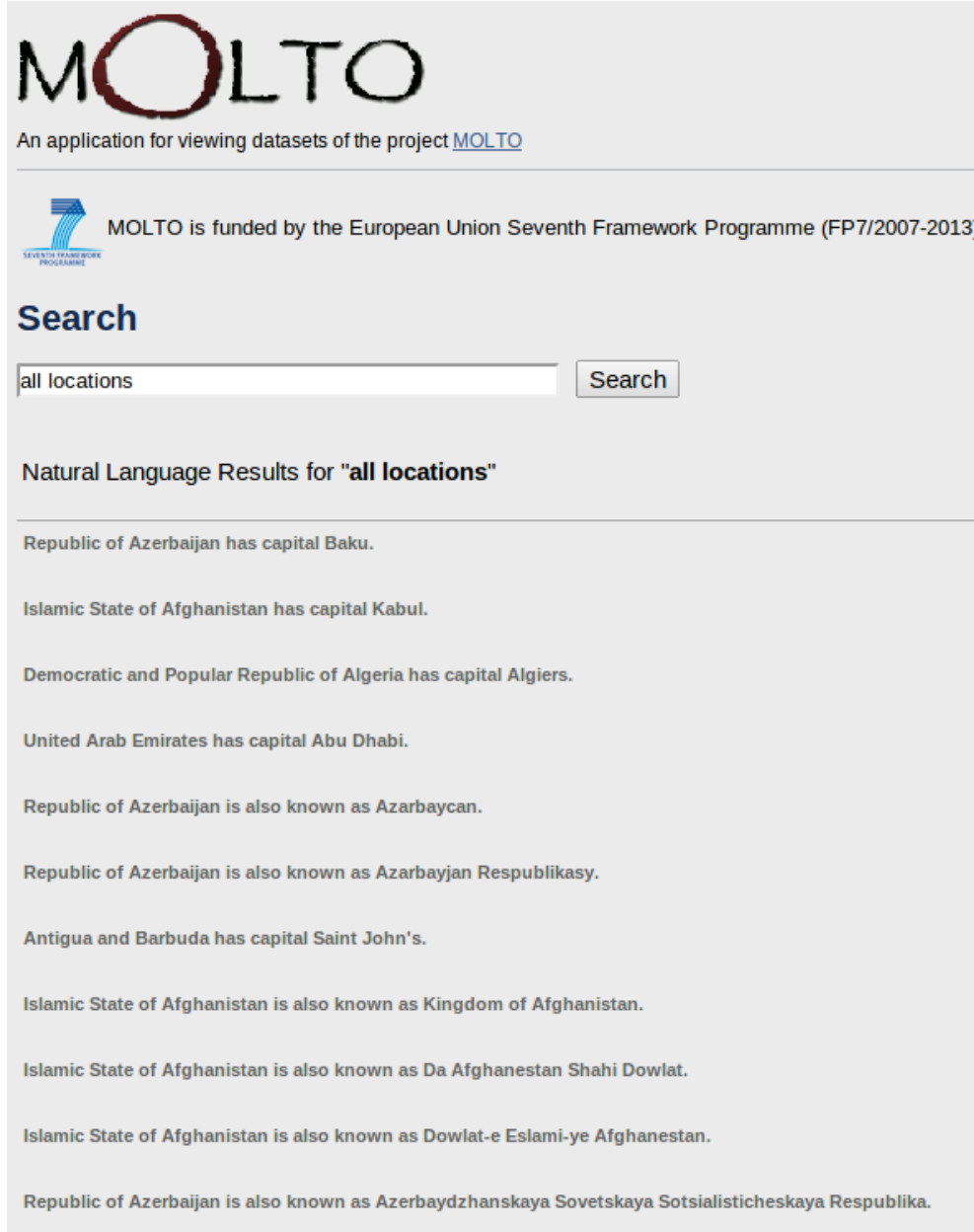
```
1 text <subject> (<predicate> <object>)
```

<subject> is of type Entity and predicate and object define the Property. The text function, defined as `text x y = PredVPS (s2e x) y`, takes both as a parameter and formalizes the expression with GF means. The final answers are passed to the Forest UI. Examples are shown in Figure 5.

In order to have a reliable verbalization of RDF predicates, we had to find patterns that we could apply to predicate names. Hence, it was necessary to divide the predicates, according to a GF pattern compatible type. We explored 10 public repositories with above 1700 unique predicates, made analysis of the common predicates patterns and extracted several most common patterns of predicate names. The initial research showed that predicates could be divided into 3 general groups and about 20 smaller ones.

For those that did not fit in any specific group, we suggested a compromise generic pattern that would work in general, but would not benefit from the grammatical flexibility of GF. Predicates that fall in the general pattern category (see types A0, B0, and C0 in Appendix A) are the best candidates for revision by GF experts and for being assigned custom GF patterns.

Figure 5: MOLTO KRI: verbalization of RDF triples



The screenshot displays the MOLTO KRI application interface. At the top, the MOLTO logo is shown, followed by the text "An application for viewing datasets of the project [MOLTO](#)". Below this, a banner indicates funding by the European Union Seventh Framework Programme (FP7/2007-2013). The "Search" section features a text input field containing "all locations" and a "Search" button. The results, titled "Natural Language Results for 'all locations'", list various entities and their attributes in a normalized format. The results are as follows:

- Republic of Azerbaijan has capital Baku.
- Islamic State of Afghanistan has capital Kabul.
- Democratic and Popular Republic of Algeria has capital Algiers.
- United Arab Emirates has capital Abu Dhabi.
- Republic of Azerbaijan is also known as Azarbaycan.
- Republic of Azerbaijan is also known as Azarbayjan Respublikasy.
- Antigua and Barbuda has capital Saint John's.
- Islamic State of Afghanistan is also known as Kingdom of Afghanistan.
- Islamic State of Afghanistan is also known as Da Afghanistan Shahi Dowlat.
- Islamic State of Afghanistan is also known as Dowlat-e Eslami-ye Afghanistan.
- Republic of Azerbaijan is also known as Azerbaydzhanskaya Sovetskaya Sotsialisticheskaya Respublika.

In Table 1 we list the three general types with their GF generalizations.

After a stable initial pre-processing, every predicate is normalized to one of the above classes. For example, all noun-prepositions, e.g. “parentOf” are normalized to “is parent of”, all noun-nouns, e.g. “systemProperty” are normalized to “has system property”.

The English GF patterns for the three major types are proposed on Figure 6.

All patterns we have found are listed in Appendix A. E1 and E2 stand for entities as described in the previous section (3.2.1). The X keyword stands for both nouns and

Group name	Summary	Description
A	“HAS” predicates	Predicates that are most easily described by “has quality ” phrase.
B	“IS” predicates	Predicates that are most easily described by “is quality preposition ” phrase.
C	“Other verbs” group	Predicates that start with verbs other than forms of “to be” and “to have”.

Table 1: General groups of RDF predicates

Figure 6: Patterns to verbalize GF types

Type Name	Pattern	Examples	GF Pattern
TYPE A1	E1 has X E2.	has president, has profession	mkCl e1 have_V2 (mkNP (mkCN (mkN "president") e2))
TYPE B1	E1 is X Prep E2.	is similar to is complement of	mkCl e1 (mkA2 "similar" "to") e2
TYPE C1	E1 Verb E2.	contains permits	mkCl e1 (mkV2 "contain") e2

adjectives. The motivation for this assumption is that in English the noun-noun phrases and adjective-noun phrases behave similarly. For example “system properties” and “useful properties” behave in the same way in a sentence. Unfortunately, this assumption is incorrect for languages like German, French, Finnish, Bulgarian and many others, where one can expect significant differences such as reverse word order or need of gender agreement. Hence, if this direction of pre-processing is kept in the future, some grammatical analysis will be necessary for languages other than English. It will split the example types into a larger number.

Nevertheless, our observations are that the majority of predicates fit in few of the types described in Appendix A, Table 2. For example, A1, A2, A4, B1 and C1 comprise about 88% of all predicates.

The obvious problems with this approach are that we cannot guarantee complete correctness, as ontologies are designed by humans and can have many name convention floats, or just new predicate structures. Also, predicate semantics is sometimes ambiguous. For example, *A-contains-B* can be a design to both “A contains B.” and “B contains A.” as explained in ([MS06]).

3.2.3 Software implementation

We have implemented a utility in java that takes a SPARQL endpoint as a parameter and defines the types for its RDF predicates. When we have GF examples for all types, it is trivial to generate the GF files, which represent automatically generated English grammar. This architecture is derived from the *Wkb* grammar([MRM13]), created in collaboration by

Ontotext and the University of Gothenburg. The grammar is mostly automatically generated and manually edited by GF experts. The source of the grammar is available at [svn://molto-project.eu/wp4/projects/molto-kri/natural-language-queries/resources/grammars/answer](https://molto-project.eu/wp4/projects/molto-kri/natural-language-queries/resources/grammars/answer) and consists of:

- Abstract GF grammar for the predicates
- Concrete English GF grammar for the predicates
- Abstract GF grammar for the data (entities from the ontologies - both subjects and objects)
- Concrete English GF grammar for the data - with the English labels from the ontology

3.3 NL description generation from semantic results. Cultural Heritage use case

Internally we have drawn the conclusion that verbalization of simple facts is not quite useful if we cannot generate a whole object description.

3.3.1 NL answer generation from semantic results

One option for an object description is to verbalize all facts in the molecule of a concrete ontology instance to a certain depth. For example, on Figure 5 the query results

- *Islamic State of Afghanistan has capital Kabul.*
- *Islamic State of Afghanistan is also known as Kingdom of Afghanistan.*

are the verbalized triples returned to the query:

```
construct where {  
  ?subject ?predicate ?object .  
  ?subject rdfs:label "Islamic State of Afghanistan" .  
}
```

where we have a description of the molecule “Islamic State of Afghanistan” with depth 1.

3.3.2 Object description. Cultural Heritage use case

In the WP8 prototype on cultural heritage, we generate paintings descriptions ([DRE⁺13]). For each painting, we execute a single call to the GF description function, which takes the following types of parameters: `painting(URI)`, `painter`, `title`, `length`, `height`, `year`, `material` and `museum`.

The NL queries are translated to SPARQL queries against a single ontology - *painting.owl*. Each query that generates description is a `select` query with (a subset of) the above list of parameters. The base query, which answers “show everything about all paintings”, is:

```

PREFIX painting: <http://spraakbanken.gu.se/rdf/owl/painting.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

#common parameters list for description generating queries
SELECT DISTINCT ?painting ?title ?material ?author ?year ?length ?height ?museum
WHERE {
  ?painting rdf:type painting:Painting ;
    rdfs:label ?title ;
    painting:createdBy ?author ;
    painting:hasCurrentLocation ?museum ;
    painting:hasCreationDate ?date;
    painting:hasDimension ?dim .
  ?author rdfs:label ?painter .
  ?date painting:toTimePeriodValue ?year .
  ?dim painting:lengthValue ?length ;
    painting:heightValue ?height .
  ?museum rdfs:label ?loc .
} LIMIT 200

```

The answer generating call to the GF DPainting is of the form:

```

1 -unlextext -lang="TextPaintingEng" DPainting (PTitle
  TAdoration_of_the_Magi__28Vel_C3_A1zquez_29)
  PDiego_Velazquez NoPaintingType NoColours (MkSize (SIntInt 127 204)) (MkMaterial MCanvas) (
    MkYear (YInt 1619)) (MkMuseum MMuseo_del_Prado)

```

In our method we follow the “convention over configuration” paradigm.

TAdoration_of_the_Magi__28Vel_C3_A1zquez_29, PDiego_Velazquez, MkMuseum MMuseo_del_Prado are preliminary generated in the GF grammar from the ontology, using the instances’ URIs also for their representation in the grammar. (The first letter(s) of the instance anme in the grammar are used to define its type, mostly for user’s better orientation). The number parameters are directly passed to the GF engine as integers.

This GF request generates the following English description.

```

"The Adoration of the Magi was painted on canvas by Diego Velazquez in 1619. It measures 127 by
  204 cm. This work is displayed at the Museo del Prado."

```

Except for English, we also support 14 other natural languages, whose grammars are created manually. This model allows direct translation of the queries and the returned answers, which is easily demonstrated with Museum prototype user interface.

3.4 Lexicon generation with TermFactory

As a complementary strategy to grammar creation, University of Helsinki has focused on lexicon generation. Lexicons translation can very well enforce the automatic grammar generation for different languages, when we have a base for English (or other languages). UHEL has defined a format from which there is a conversion to GF lexicon. This format is used in TermFactory (TF)[Car13], a platform for multilingual terminology management created in UHEL.

The structure of the TF top ontology schema is shown in Figure 7. Base forms (yellow rectangles) are linked to expressions (yellow ovals), and expressions are linked to concepts (blue ovals) with relations in green. The green relation is represented by triples, whose subject is of type `term1:lang-EXP-POS-_ont-TERM`, for instance `term1:en-dog-N-_ont-Dog`; the predicate `term:hasDesignation` marks the expression, and `term:hasReferent` - the concept.

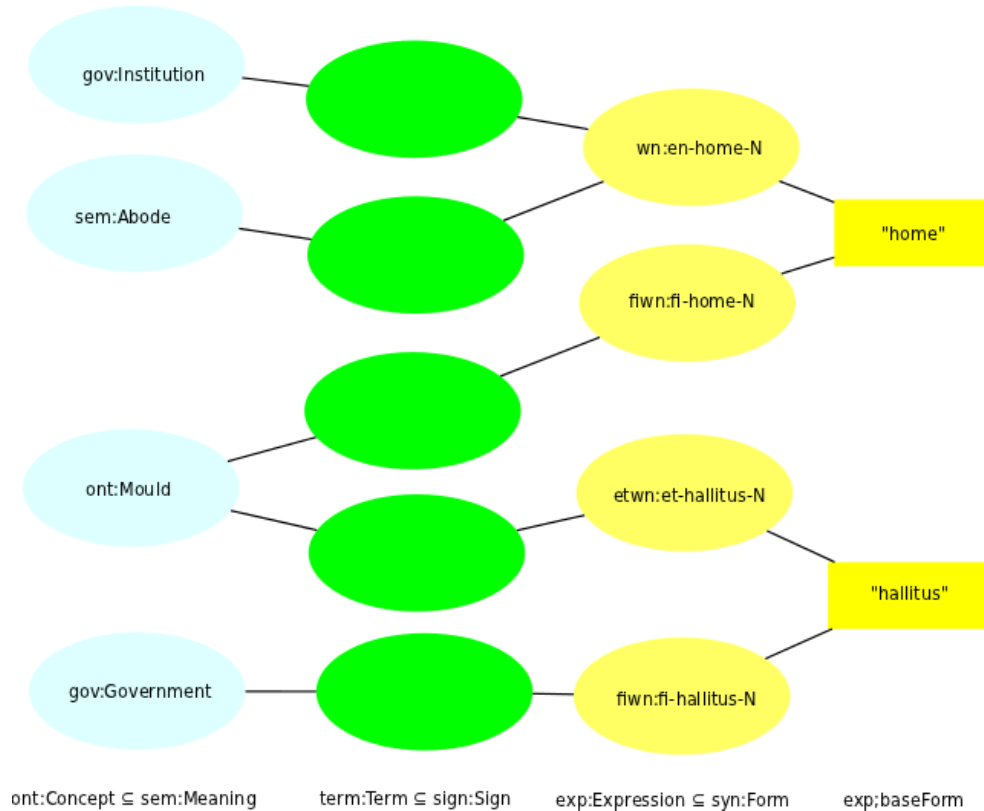


Figure 7: TermFactory top ontology schema

Below is an example showing the mapping between the Czech expression for “Patagonian toothfish” to its concept in DBPedia:

```
term1:cs-ledovka_patagonská-N-_dbp-Patagonian_toothfish
a      term:Term ;
term:hasDesignation exp1:cs-ledovka_patagonská-N ;
term:hasReferent dbpedia:Patagonian_toothfish .

exp1:cs-ledovka_patagonská-N
a      exp:Designation ;
exp;baseForm "ledovka patagonská" ;
exp:catCode "N" ;
exp:langCode "cs"
```

The format includes optional properties that are needed in the GF grammar. To handle relational concepts intelligently, GF requires entries with morphological and syntactic

valency information. The following example shows how a GF feature is added to the TF entry default grammatical features. There are two properties, a TF native feature `syn:frame` and a GF native feature `gf:lin`. A snippet of the result is shown below. Expert users can code the GF frame formula directly in the `gf:lin` feature. For non-experts, the `syn:frame` feature takes as values the traditional style frame descriptions of type “V something to something” from which the `gf:lin` feature value can be generated by rule or just listed in the GF mapping file.

```
term1:en-forbid-V-_sem-Forbid
  a      term:Term ;
  syn:frame "V someone from being" ;
  gf:lin "ingV2V (mkV str) noPrep from_Prep" ;
  term:hasDesignation exp1:en-forbid-V ;
```

The details of the search and the final conversion to a GF grammar are explained in the TermFactory User Manual¹⁶, Section 6.3. The TermFactory user interface comes with predefined query aliases, which can be pipelined to harvest terms and convert the results to GF, and with a TF XHTML editor, which can be used to add grammatical information to the entries manually.

3.5 Discussion on levels of automation

Verbalizing ontology data can be utilized by GF very successfully, combined with the following human expertise:

- Ontologies/Semantic data expert - who examines the domain ontologies and suggests generic SPARQL model(s) for the domain;
- GF expert - who implements a dynamic SPARQL model and the GF answer generation model.

The two kinds of experts need to cooperate actively, especially for refining the model.

In case we want to verbalize simple facts, the answer generation grammar can be done semi-automatically, which consists of three major steps: ontology predicates analysis (automatic), GF pattern assignment (automatic, straightforward for English), and final GF expert edit. The answer sentences are result from the GF linearization of facts with the specific grammar.

¹⁶http://tfs.ling.helsinki.fi/doc/TFGuide_en.xhtml#s6.3

4 Comparison and Integration between KRI and TermFactory

In compliance with the reviewers' recommendations, we compared KRI to TF and proposed steps for their integration.

4.1 Comparison

KRI is an information retrieval system over a semantic repository. It allows the retrieval of RDF facts and their representation in natural language, as well as coverage in several languages.

TermFactory is a higher-level system for management of term ontologies, a type of storage for storages. TF does not describe real events or relations. It maps terms to concepts, and focuses on multilinguality and the syntactic and morphological completeness of the terms.

Both KRI and TF use the Ontotext RDF databases in FactForge¹⁷. With KRI, the user can make queries to the databases in natural language. With TF, the user can make the databases better suited for the GF grammar and lexicon generation. When these data collections are ported to a TermFactory site, one can add grammatical properties to the expressions that do not have such properties yet, as well as convert that representation format to GF grammars.

The information in FactForge is usually of the following type:

```
dbpedia:Salmon
  rdfs:label      "Salmon@en"
  dbp-ont:family  dbpedia:Salmonidae
  dbp-ont:class   dbpedia:Actinopterygii
```

All properties of the resource `dbpedia:Salmon` are useful for querying. Once in TF, the resource is linked to a linguistically rich expression. For many of the common concepts there are already existing terms and expressions in some of the TermFactory repositories. Therefore it is enough to add a predicate that links `dbpedia:Salmon` to `exp1:en-salmon-N` or for another language, such as `exp1:sv-lax-N` for the Swedish expression for salmon. The latter resource looks like the following:

```
exp1:en-salmon-N
  a          exp:Designation ;
  exp:baseForm "salmon" ;
  exp:catCode "N" ;
  exp:langCode "en" .
```

As a result, the concept `dbpedia:Salmon` will also be linked to the following concept in TF top ontology.

¹⁷<http://www.ontotext.com/factforge>, <http://factforge.net/>

```
term1:en-salmon-N-_dbp-Salmon
  a      term:Term ;
  term:hasDesignation exp1:en-salmon-N ;
  term:hasReferent dbpedia:Salmon .
```

4.2 Integration

Prerequisites:

1. Ontotext has a TF site. We name it *OntoTF*.
2. Ontotext has a large semantic repository, such as FactForge. We name it *OntoFF*.

Procedure:

1. *OntoTF* connects to a mirror instance of *OntoFF*, we name it *OntoTF_FF*, where editing is allowed.
2. The new semantic data (from *OntoTF*) is aligned with the current one (in *OntoFF*). Linkage is provided through the *owl:sameAs* predicate between, for example `dbpedia:Salmon` and `exp1:en-salmon-N`. `exp1:en-salmon-N` has a lot of additional properties in contrast to `dbpedia:Salmon`.
3. Querying *OntoTF_FF* returns TF enriched results. Also, it can be used for grammatically rich lexicon extraction for GF grammars.

For example, if our query against *OntoFF* returns results such as `dbpedia:Dog`, the same query against *OntoTF_FF* will return the equivalent `term1:en-dog-N-_dbp-dog` and `textexp1:en-dog-N` (details are listed below).

```
term1:en-dog-N-_dbp-Dog
  a      term:Term ;
  rdfs:comment "an English term for the property ont0:Dog"^^xsd:string ;
  term:hasDesignation exp1:en-dog-N ;
  term:hasReferent dbp:Dog .

exp1:en-dog-N
  a      exp:Designation ;
  rdfs:comment "an English noun"@en ;
  exp:baseForm "dog" ;
```

The example below shows a generated GF lexicon:

```
—This is a GF file produced automatically from a TermFactory lexicon.
—# -path=../prelude:../abstract:../common
instance LexAnimalsEng of LexAnimals = CatEng ** open ParadigmsEng,SyntaxEng in {
  flags coding=utf8 ;

  oper
    dbp-Dog_N = mkN "dog" ;
```



```
dbp-Cat_N = mkN "cat" ;  
dbp-Horse_N = mkN "horse" ;  
...  
}
```

5 MOLTO Prototypes

The main prototype of MOLTO project is the KRI prototype, developed by Ontotext and UGOT, which serves as a "core technology" for the other two prototypes - MOLTO Patents (Section 5.2) and MOLTO Cultural Heritage (Section 5.3). The KRI prototype is presented in [MI10], [Dam11], [CEDR12] and in Section 5.1 of the present document.

The technologies used in all prototypes are:

- OWLIM 5 ([BKO⁺11]) - Ontotext's semantic repository
- Grammatical Framework for query language definition, NL query parsing, and/or NL answers generating
- Forest 1.4. as a UI for information retrieval and browsing over semantic repository. (Forest is an internal software product of Ontotext, based on the Spring Framework¹⁸)

In the following sections we provide details on each particular prototype and how they compare to the final version with focus on grammar-ontology interoperability.

5.1 MOLTO KRI

The MOLTO KRI prototype is built as part of WP4 and is publicly accessible at <http://molto.ontotext.com>.

5.1.1 Prototype description

This prototype is based on PROTON, which is a light-weight upper-level ontology that defines types such as **Person**, **Location**, **Organization** and **Job Title**. It provides both queries and answers in natural languages using GF (examples on Figures 8 and 9). The NL query to SPARQL translation is done via the mapping rules module (Section 2.1). The RDF statement to NL answer translation follows the approach of verbalizing predicates with semi-automatically generated grammars (Section 3.2).

5.1.2 Grammar-ontology interoperability

The query grammar was designed by Ontotext and created by experts from UGOT, in 6 different languages. The grammars can be found at <svn://molto-project.eu/wp4/projects/molto-kri/natural-language-queries/resources/grammars/>. The NL queries are parsed to GF abstract representation, which is mapped to SPARQL queries via mapping rules (Section 3.2).

The SPARQL queries are "construct" queries, which return RDF graph that consists of "subject-predicate-objects" triples. Our task is to verbalize these triples, the focus being on predicates. To achieve this, we use a separate module of the answer grammar. Originally,

¹⁸<http://www.springsource.org/spring-framework>

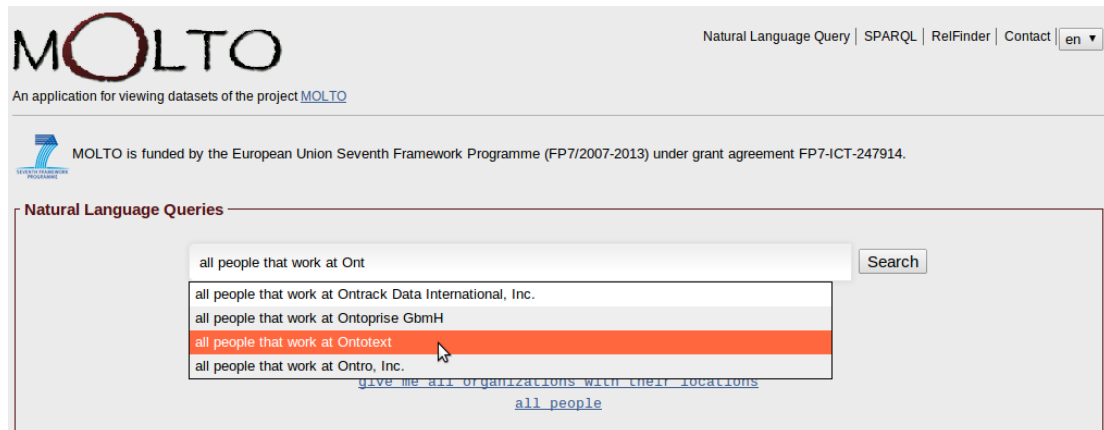


Figure 8: MOLTO KRI queries

it was semi-automatically generated, and only in English. In the final year of MOLTO it was refactored and other answer languages were added by the UGOT experts.

The whole answer grammar has two main parts - data and expressions (based on RDF predicates). The data grammar is automatically generated from the ontology. It contains over 135000 entities from specific classes, selected for the purpose of the prototype. They are extracted from the ontology and placed in the GF general **Entity** category. Each subject/object is described by the meaningful part of their URI (the one without the domain) and the English RDF label from the ontology. The resulting abstract GF representation is demonstrated in Section 3.2.1. The GF grammars for verbalization of predicates are semi-automatically generated and manually translated to the different languages.

Search

Natural Language Results for "all people that work at 'Ontotext'"

Dimitar Manov has position Project Manager of Ontotext.

Damyan Ognyanoff has position Senior software engineer of Ontotext.

Borislav Popov has position Project Manager of Ontotext.

Marin Dimitrov has position Project Manager of Ontotext.

Angel Kirilov has position Senior software engineer of Ontotext.

Miroslav Goranov has position Quality Assurance Engineer of Ontotext.

Atanas Kiryakov has position head of Ontotext Lab.

Rossen Marinov has position Software engineer of Ontotext.

Knowledge Base Results for "all people that work at 'Ontotext'" (65) (SPARQL Query: [construct WHERE {...](#))

subject	predicate	object
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdf:type	ptop:Person
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	ptop:hasPosition	http://www.ontotext.com/kim/2006/05/wkb#Position.1
http://www.ontotext.com/kim/2006/05/wkb#Position. ONTO_AK	ptop:withinOrganization	http://www.ontotext.com/kim/2006/05/wkb#Division.1
http://www.ontotext.com/kim/2006/05/wkb#Division.Ontotext	rdfs:label	Ontotext
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdfs:label	Atanas Kiryakov
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdf:type	psys:Entity
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdf:type	ptop:Agent
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdf:type	ptop:Object
http://www.ontotext.com/kim/2006/05/wkb#Person.AtanasKiryakov	rdf:type	ptop:Person
http://www.ontotext.com/kim/2006/05/wkb#Person.BorislavPopov	rdf:type	ptop:Person
http://www.ontotext.com/kim/2006/05/wkb#Person.BorislavPopov	ptop:hasPosition	http://www.ontotext.com/kim/2006/05/wkb#Position.1

Figure 9: MOLTO KRI results

5.2 MOLTO Patents

MOLTO Patents is built as part of WP7 and uses MOLTO KRI as a basis. The public prototype can be found at <http://molto-patents.ontotext.com>.

5.2.1 Prototype description

This prototype allows querying in three natural languages (English, French and German) against documents-and-RDF-facts retrieval system. The system contains collections of biomedical patents from the EPO¹⁹ corpus, that are semantically annotated using the domain ontologies e.g. an internally developed FDA²⁰ data ontology, which is aligned with other applicable ones(details are given in [MGE⁺13]). The query grammar is significantly improved for the current version of the prototype. It has been rewritten by GF experts from UGOT. The current query language allows asking over 20 million semantically different queries, each with several natural language versions. (This large number comes mostly from the “show patents that mention X and Y”, where X and Y are among several lexicon lists - drugs, active ingredients, etc.)

In this use case we do not provide NL answers. Instead, we retrieve ontology data and biomedical patent documents, translated with statistical machine translation techniques by UPC.

5.2.2 Grammar-ontology interoperability

Here, in MOLTO patents, for the first time we applied the GF generating SPARQL approach, defined in Section 2.2.2. We explored its advantages over the mapping rules, which are now permanently substituted. It is interesting to point out that the query language is based strictly on ontology relations (e.g. “what are the active ingredients of BACLOFEN”) and also on semantic annotations over the documents (e.g. “patents that mention PENICILLIN and AMPICILLIN”), where the results are documents that contain annotations of both drugs. In general, the query language is based on ontology relations and data, and on semantic data that we add through our GATE²¹ annotation pipeline. Annotations were transmitted from English to French and German during the translation process. This process allowed to extract the lexicons from the annotations in these target languages, to complete the query language.

¹⁹European Patents Office

²⁰Food and Drug Administration Agency, USA

²¹<http://gate.ac.uk/>

5.3 MOLTO Cultural Heritage

MOLTO Cultural Heritage is built as part of WP8 and also uses MOLTO KRI as a basis. The public prototype is located at <http://museum.ontotext.com>.

5.3.1 Prototype description

The specific part of the prototype is an aligned model of ontologies in the cultural heritage domain, which provides data for the retrieval system. The system generates natural language descriptions of museum paintings objects. This prototype provides NL queries on the topic of “museum paintings” in 15 languages.

The user interface is shown on Figure 12.

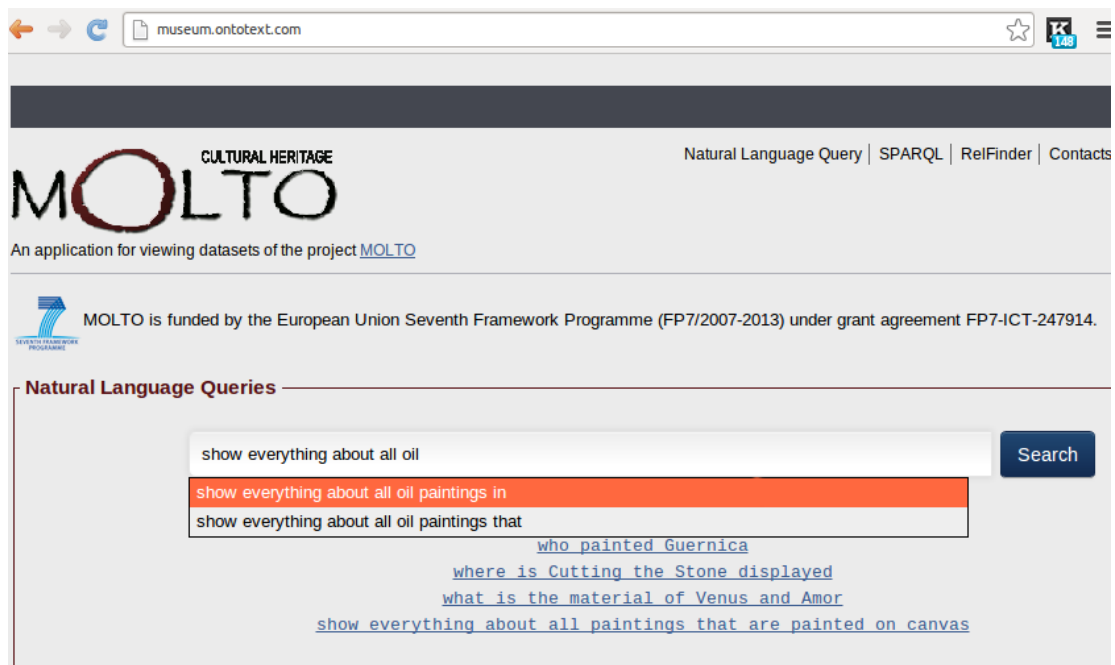


Figure 12: MOLTO cultural heritage queries

The corresponding result (truncated version) is displayed on Figure 13.

5.3.2 Grammar-ontology interoperability

The query language is designed to cooperate with the *painting.owl* ontology, which is mapped to other painting data, such as DBpedia, GIM, GCM, and CIDOC-CRM as linking ontology. In this use case the focus is on the generation of paintings descriptions. Automatically generated grammar is not applicable. Data dictionaries for the grammars are partially provided from the ontology data in the available languages. We use “select” queries to retrieve specific paintings details and pass them to a GF function, which

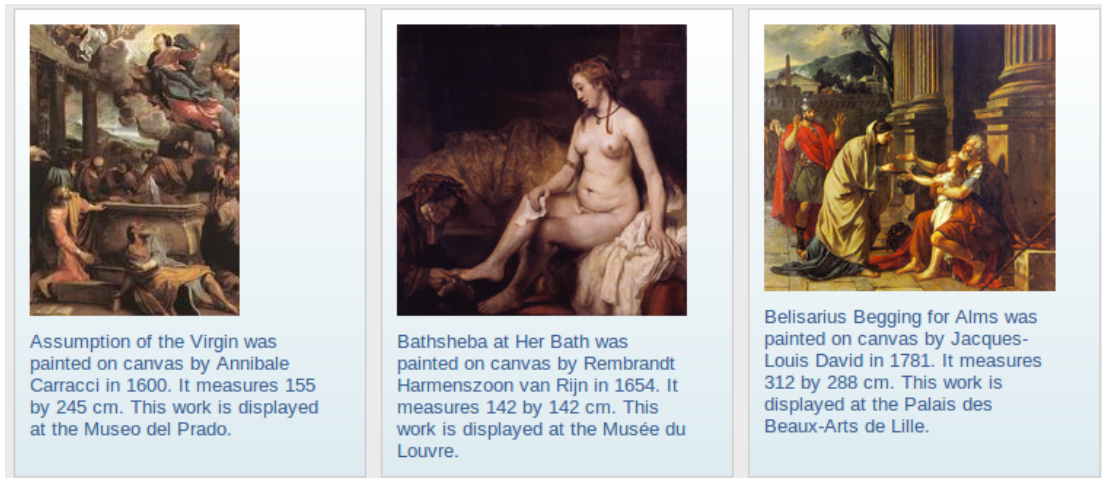


Figure 13: MOLTO cultural heritage results - paintings descriptions, English

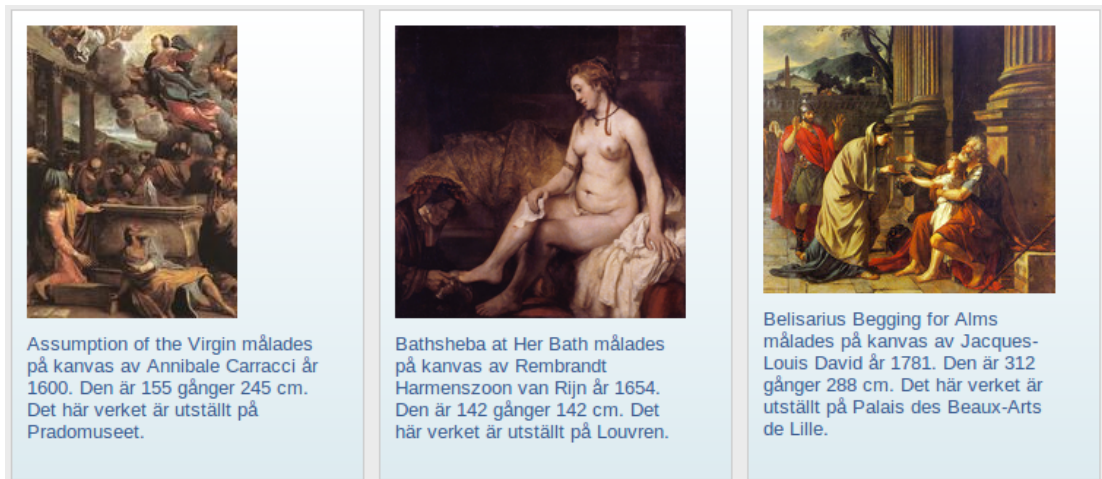


Figure 14: MOLTO cultural heritage results - paintings descriptions, Swedish

generates paintings descriptions (Section 3.3.2). For the GF grammars-to-SPARQL transformation, we directly use the new approach described in Section 2.2.

6 Porting KRI to New Applications

In this section, we present the steps for adapting KRI to new domains. Some of the steps are optional, depending on the task and the initial resources for it.

1. Examine the domain. Align semantic data and ontologies, if needed.
2. Suggest applicable query language and create query grammars (either manually or extract them from the ontology).
3. Define “variable” types - for GF data grammars and for the data lexicons for the FSA auto-complete. Extract the corresponding lexicons.
4. Create exemplary SPARQL queries for the GF model. A SPARQL expert is required.
5. Create dynamical model for the domain specific SPARQL generation. A SPARQL expert is required.
6. Create/correct manually the GF grammars for query and/or answer. This process may require a few iterations and co-ordination between the experts.
7. Translate/update the query grammars in other natural languages.
8. Generate auto-complete resources from the produced grammars.
9. Add the grammars to the prototype and suggest example queries for the UI.
10. Test the resulting system. If not satisfied, return to step 6.

The process of refining the GF SPARQL grammar usually includes a few iterations, while we change the NL queries and the expected result.

6.1 Strengths and limitations of our natural language interface to the semantic repository

A grammar-based interface from NL queries to ontologies allows the maximum accuracy of the results returned to the user. The NL query is translated into SPARQL without ambiguities, which ensures the correctness of the triples. In fact, as far as it concerns evaluation measures, a system of this kind can claim to have 100% precision and recall as there is no variance in the NL-to-SPARQL direction. Here it is important to note, that this result can be decreased if there are errors in the data, or, for example, in the semantic annotation pipeline, when queries are mostly against the annotations(as in the Patents use case). However, in order to achieve maximum accuracy, the MOLTO interface requires involvement of experts, which might be a limitation.

Usually a predefined list of NL queries needs to be created by experts in the specific domain of the application. They should reflect the information most frequently searched

by users. Next, a GF expert needs to implement(or correct) the query grammar that can generate all predefined queries. The grammar has an abstract core and several concrete grammars, if multiple language support is required by the application. A special concrete grammar must be designed to offer immediate translation to SPARQL queries, which is key for the grammar-ontology interoperability. Finally, a GF answer grammar needs to be implemented in order to process the triples that result from running the SPARQL query to returning the NL answers. We suggest partially automated ways of verbalizing RDF triples (see Section 3.2), which can facilitate the automatization of the answer grammar, but cannot substitute the GF expert participation.

7 Conclusion

Throughout MOLTO we have explored a few different directions of grammar-ontology interoperability. Our experience has shown that Grammatical Framework is suitable for creation of grammars that define a query language, including corresponding machine language, that to be used for query execution against the retrieval system. Verbalization of response is easily supported as well. It can be focused on only the simple facts verbalization, or it could be used for parametrized descriptions, which we both demonstrated in the related work packages. The degree of automation of the whole interoperability cycle can be very high, but this requires high expertise from GF point of view, good understanding of the domain, and good planning of the verbalization model.

8 References

References

- [AGL13] Alessio Palmero Aprosio, Claudio Giuliano, and Alberto Lavelli. *Automatic Expansion of DBpedia Exploiting Wikipedia Cross-Language Information*. Montpellier, France, May 2013. ESWC.
- [BKO⁺11] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. OWLIM: A family of scalable semantic repositories. *Semantic Web Journal*, 2:33–42, June 2011.
- [Cam12] John Camilleri. Gf eclipse plugin, 2012.
- [Car13] Lauri Carlson. *TermFactory User Guide*, 2007–2013.
- [CEDR12] Milen Chechev, Ramona Enache, Mariana Damova, and Aarne Ranta. *D4.3 Grammar-Ontology Interoperability*, May 2012. Deliverable 4.3. MOLTO FP7-ICT-247914.
- [Con11] MOLTO Consortium. Molto enlarged eu annex i - description of work, 2011.
- [Dam11] Mariana Damova. *D4.2 Data Models and Alignment*, May 2011. Deliverable 4.2. MOLTO FP7-ICT-247914.
- [DDE⁺13] Mariana Damova, Dana Dannells, Ramona Enache, Maria Mateva, and Aarne Ranta. Natural language interaction with semantic web knowledge bases and lod. In Paul Buitelaar and Philip Cimiano, editors, *Towards the Multilingual Semantic Web*. Springer, Heidelberg, Germany, 2013.
- [DRE⁺13] Dana Dannells, Aarne Ranta, Ramona Enache, Mariana Damova, and Maria Mateva. *D8.3 Translation and retrieval system for museum object descriptions*, 2013. Deliverable 8.3. MOLTO FP7-ICT-247914.
- [Ena10] Ramona Enache. Reasoning and language generation in the sumo ontology. Master’s thesis, Chalmers University of Technology, 2010.
- [Lis12] Inari Listenmaa. Ontology-based lexicon management in a multilingual translation system – a survey of use cases. Master’s thesis, University of Helsinki, November 2012.
- [MGE⁺13] Maria Mateva, Meritxell Gonzàlez, Ramona Enache, Cristina España-Bonet, Lluís Màrquez, Borislav Popov, and Aarne Ranta. *D7.3 Patent MT and Retrieval. Final Report.*, April 2013. Deliverable 7.3. MOLTO FP7-ICT-247914.
- [MI10] Peter Mitankin and Atanas Ilchev. *D4.1 Knowledge Representation Infrastructure*, November 2010. Deliverable 4.1. MOLTO FP7-ICT-247914.

- [MRM13] Peter Mitankin, Aarne Ranta, and Maria Mateva. Wkb answer grammar. *Wkb Grammar*, 2013.
- [MS06] Chris Mellish and Xiantang Sun. The semantic web as a linguistic resource: opportunities for natural language generation. knowledge based systems. In *Presented at the Twenty-sixth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, 2006.
- [Ran12] A. Ranta. *Implementing Programming Languages. An Introduction to Compilers and Interpreters, with an appendix coauthored by Markus Forsberg*. College Publications, London, 2012.

9 Appendix A: Groups of RDF predicate types

Type Name	Pattern	Examples	# of pred.
TYPE A0	GENERAL A Group type	-	11
TYPE A1	E1 has X E2.	has president, has profession	533
TYPE A2	E1 has X X E2.	has subsequent work has general manager	643
TYPE A3	E1 has X Prep E2.	has route of administration has distance to london	53
TYPE A4	E1 has X X X E2.	has free flight time has first driver team	179
TYPE A5	E1 has X Prep X X E2.	has distance to charing cross has number of doctoral students	17
TYPE A6	E1 has X X Prep X E2.	has end year of insertion has ethnic groups in year	8
TYPE A7	E1 has X X X X E2.	has port 1 docking date has original maximum boat beam	30
TYPE A8	E1 has X Prep X X X E2.	has rank in final medal count has percentage of area water round	6
TYPE A9	E1 has X X Prep X X E2.	has qmf language of short tmpl has qmf strsqlval of short tmpl	18
TYPE A10	E1 has X X X Prep X E2.	has qmf is subformat of long	1
TYPE A11	E1 has X X X X X E2.	has national topographic system map number	8
TYPE B0	GENERAL B Group type	-	8
TYPE B1	E1 is X Prep E2.	is similar to is complement of	87
TYPE B2	E1 is X X Prep E2.	is doing business as is child organization of	27
TYPE B3	E1 is Prep E2.	is from is within	4
TYPE C0	GENERAL C Group type	-	6
TYPE C1	E1 Verb E2.	contains permits	59
TYPE C3	E1 Verb Prep E2.	conforms to refers to	11
TYPE C4	E1 Verb Prep X E2.	wins at asia wins at challenges	13
TYPE C5	E1 Verb X Prep E2.	was intended for shows features of	2

Table 2: RDF predicates types classification with respect ot GF patterns