Sorry, I only speak natural language: a pattern-based, data-driven and guided approach to mapping natural language queries to SPARQL

Mariano Rico[1] ⋆, Christina Unger[2], and Philipp Cimiano[2]

[1] Ontology Engineering Group (OEG)
Universidad Politécnica de Madrid
mariano.rico@upm.es
http://oeg-upm.net
[2] Semantic Computing Group
Bielefeld University
{cunger, cimiano}@cit-ec.uni-bielefeld.de
http://www.sc.cit-ec.uni-bielefeld.de/

Abstract. We present a new interface based on natural language to support users in specifying their queries with respect to RDF datasets. The approach relies on a number of predefined patterns that uniquely determine a type of SPARQL query. The approach is incremental and assisted in that it guides a user step by step in specifying a query by incrementally parsing the input and providing suggestions for completion at every stage. The methodology to specify the patterns is informed by empirical distributions of SPARQL query types as found in standard query logs. So far, we have implemented the approach using two patterns only as proof-of-concept. The coverage of the pattern library will be extended in the future. We will also provide an evaluation of the approach on the well-known QALD dataset.

Keywords: SPARQL queries, natural language, pattern based, data driven, guided systems, lemon model

1 Introduction

SPARQL is the query language for the Web of data, but it suffers from a high adoption barrier by end users. Mastering the SPARQL syntax is a problem for users lacking IT skills. But even knowing the SPARQL syntax, a more severe problem remains: the need to know the underlying vocabulary of the queried dataset. Thus, querying RDF data remains a barrier not only for lay people but also for technically skilled users. There are different approaches to alleviate this barrier and to support users in the task of querying RDF data.

One approach consists in guiding users in writing SPARQL queries by some interface, e.g. SindiceTech's Qakis [1] or SparQLed [2], in which you see the

⋆ Supported by LIDER (EU FP7 proj. 610782) and projects JCI-2012-12719, TIN2013-46238-C4-2-R (4V), UNPM13-4E-1814 (INFRA)

SPARQL query but you are assisted to avoid syntactic errors. Such approaches remove the first problem mentioned above.

Another type of approach uses visual metaphors to support writing queries, abstracting from the SPARQL syntax, e.g. Rhizomer [3], GoRelations [4], RelFinder [5] or SPEX [6]. The third type of approach relies on natural, or controlled natural language as query interface. A user writes a query in natural language and the interface systems needs to map this natural language query into SPARQL, thus completely hiding the SPARQL query language from the user. Examples of the latter category are PowerAqua [7] and others.

The systems in the third category are typically not incremental in the sense that they require a user to enter a full query before it is processed. Thus, the user has no guidance on how to write the query. Exceptions are systems such as Attempto-OWL [8] or GINO [9], which rely on controlled natural language. Such approaches define a controlled language in a top-down fashion, without empirically looking at the types of queries users actually make.

We present a novel approach to guided natural language querying that is incremental in the sense that it interprets the question while the user is typing it and can thus provide guidance by proposing possible completions of the query to the user. The approach relies on a number of patterns that have been defined to cover the most frequent types of SPARQL queries. So far, only two patterns have been implemented to provide a proof-of-concept, but the long-term goal is to continue adding natural language patterns iteratively to cover the most frequent types of SPARQL queries sent by users to SPARQL endpoints. In fact, it has been shown that the the SPARQL queries made to SPARQL endpoints follow a power-law distribution [10] (at least for the DBpedia and the Spanish DBpedia).

Figure 1 shows the number of queries for each type (left y-axis) and the percentage of the total number of queries (right y-axis). One can see that the first query type (the most used) was used by 1.2 million queries, corresponding to 42% of the total number of queries. The top-20 queries cover 95% of all queries. This figure was built using 3 months of DBpedia logs (USEWOD 2014 dataset). A specific question of a user might be:

"Give me movies by Tarantino"

The corresponding natural language pattern would be the following:

Give me ***Noun Preposition Instance***

The corresponding SPARQL query type would be the following:

SELECT ?s WHERE {?s *prop instance*}

And the actual SPARQL query would look as follows:

```
SELECT ?s WHERE {?s dbpedia:director dbpedia:Tarantino}
```

Our long-term goal is to have an approach that covers 90% of all queries. By guiding the user, one ensures that only such queries are typed in that can actually

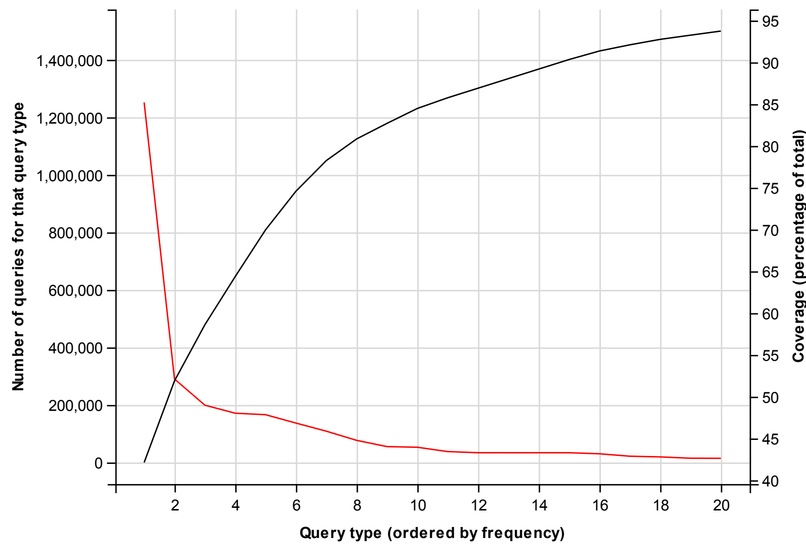

Fig. 1. Pareto distribution of query types for DBpedia (USEWOD 2014 queries dataset).

be processed by the system, thus reducing errors and increasing robustness of the system. With this method we can cover quickly the most frequent query types, but covering the long tail would require implementing hundreds of query types. Our system is intended to allow an incremental way of adding new query types.

An important challenge is to have a system with high lexical coverage that covers as many alternative ways of referring to one given property, class or individual as possible. We rely on ontology lexica as modeled by the lemon model [11] to make such lexicalizations explicit. We exploit existing lexicalizations of DBpedia which relate vocabulary elements in DBpedia to the lexical entries that verbalize these via the lemon model [12]. In this way, for instance, we can look up in the lexicon that `dbpedia:starring` can be verbalized by `''movie with''` and that the property `dbpedia:producer` is verbalized by `''movie by''`, etc. The lexicalization relation is clearly not 1:1, as one vocabulary element can be verbalized by different lexical entries, and one lexical entry can be ambiguous and potentially refer to different vocabulary elements. A possible methodology here would be to prioritize the creation of lexical entries in a way that is informed by frequency of use of types (classes) and properties in the given dataset.

A further challenge is to have a real-time response so that query completion is immediate from the users' point of view. This can be achieved by appropriate inverted indexes that return for a class which properties are associated to this class, which instances stand in the subject or object of a particular property, etc.

We rely on an index [13][3] that provides a quick response to intensive SPARQL queries like `select ?type where {?s dbpedia:starring ?v . ?s a ?type}` (type of the subject in triples with property `dbpedia:starring`). Table 1 shows the result of the previous query, but ordered by usage (triples with subjects of that type). With this information we provide to the user a list of options starting with `work`, followed by `film` and `television show`, etc.

Table 1. Subject types for triples with property `dbpedia:starring` in the Spanish DBpedia SPARQL endpoint, ordered by usage.

Subject class	**Count**	**%triples**
http://dbpedia.org/ontology/Work	155,508	100.0%
http://dbpedia.org/ontology/Film	118,887	76.45%
http://dbpedia.org/ontology/TelevisionShow	36,621	23.55%

2 Approach by example

We illustrate our approach by one example shown in Figure 2. We distinguish between *query pattern types* and *NL patterns*. One query pattern type can be expressed by different NL patterns. We show two query patterns with different NL patterns below:

- **Query Pattern Type 1.** This pattern corresponds to the query type

  SELECT ?s WHERE {?s *prop instance*}

  NL patterns that can be used to express this query pattern type are:
  *What is the **Noun** of **Instance**?*
  e.g. *What is the capital of England?*, mapping to the SPARQL query

  ```
  SELECT ?s WHERE {dbpedia:England dbpedia:capital  ?s}
  ```

  or *Give me **Noun Preposition Instance***
  e.g. *Give me movies by Tarantino*, mapping to the SPARQL query

  ```
  SELECT ?s WHERE {?s dbpedia:director dbpedia:Tarantino}
  ```

- **Query Pattern Type 2.** Corresponds to the query type:

  SELECT ?s WHERE {?s a *class* .
  ?s *property instance*}

  NL patterns that express this query pattern type are for example:
  *What **Noun Verb Instance**?*
  e.g. *What movies starred John Travolta?*, mapping to the SPARQL query

[3] See `http://loupe.linkeddata.es/loupe/`

```
SELECT ?s WHERE {?s a dbpedia:Film .
                 ?s dbpedia:starred dbpedia:John_Travolta}
```

or *Which* ***Noun Verb Preposition Instance****?*
e.g. *Which river passes through London?*, mapping to the SPARQL query

```
SELECT ?s WHERE {?s a dbpedia:River .
                 ?s dbpedia:crosses dbpedia:London}
```

Overall, NL patterns thus represent slotted and typed patterns into which elements can be inserted to be completed. Once all elements have been inserted into an NL pattern, the SPARQL query is determined.

Figure 2 shows two NL patterns. Each NL pattern represents a sequence of so called *Parseable Elements*. Each element parses or recognizes one part of the input. A question is parsed by applying all NL patterns available and determining which ones match or recognize the input. This is determined iteratively by checking for each parseable element whether it recognizes or matches the corresponding part of the question. In general, multiple patterns will match a question at any time. We distinguish the following types of parseable elements:

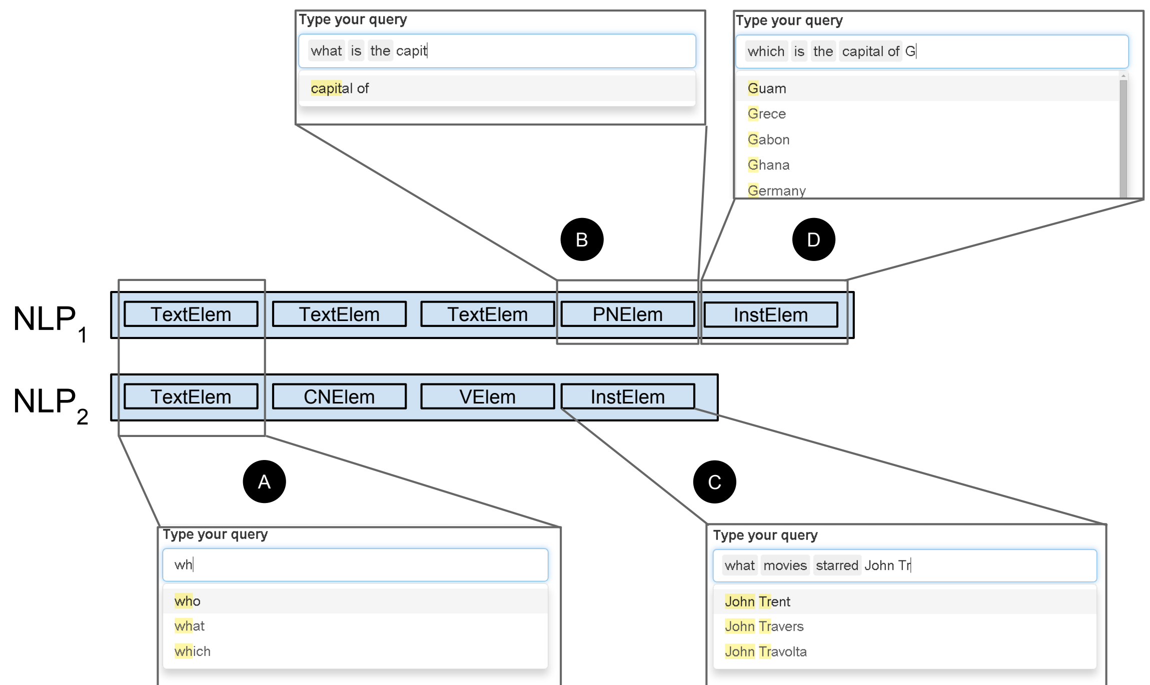

Fig. 2. Use case for two NL query patterns NLP_1 and NLP_2

– **TextElements:** These elements are responsible for recognizing constant filler elements (e.g. *'what', 'who'*, etc.)
– **PropertyNounElements:** These elements are responsible for recognizing a noun that denotes a property (e.g. *'capital'*)
– **ClassNounElements:** These elements are responsible for recognizing a noun that denotes a class (e.g. *'river'*)
– **VerbElements:** These elements are responsible for parsing a verb that denotes a property (e.g. *'passes'*)
– **InstanceElements:** These elements are responsible for parsing an instance (e.g. *'London', 'John Travolta'*), etc.

For each pattern, the system checks whether it accepts the input query. It iterates through the sequence of parseable elements that constitute an NL pattern and verifies for each parseable element whether it recognizes the corresponding part of the sentence. A simple lookahead method is used to compute possible completions of the question by showing the user the inputs that the following parseable element will accept. Choices made by the user are propagated to future elements of the sequence as needed and modeled by corresponding dependencies between the parseable elements in the sequence. In the example depicted in Figure 2), the user has typed in `what`, which matches both patterns. Then, the system proposes completions for both patterns. The user can delete any number of previous selections to return to a previous point and the system continues the process from that point.

3 Architecture of the system

The architecture of the system is depicted in Figure 3. The main components of the architecture are the following ones:

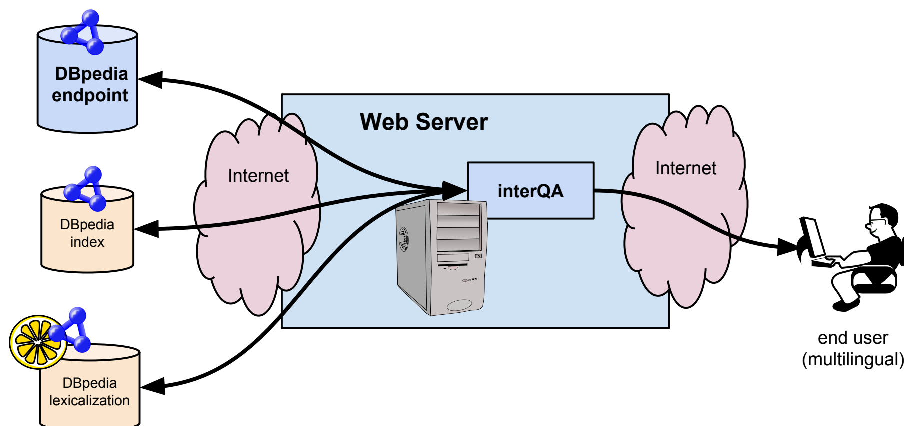

Fig. 3. Key components in the interQA system for the DBpedia use case.

1. **SPARQL endpoint:** As any user interface, a quick response is a fundamental issue. In order to avoid an excessive number of requests to the endpoint, we create an index with the results of a specific set of extractive queries. Therefore, we need a privileged access to the endpoint. In our experiments we have used the Spanish DBpedia. Other endpoints could run the indexer in low-demand periods (by night) or with small pagination (the smaller the pagination size, the longer time to create the index).
2. **SPARQL endpoint index:** This index is available online as a REST service. For the case of the Spanish DBpedia see (`http://loupe.linkeddata.es/loupe`).
3. **Question Interpretation component:** the component[4] that incrementally interprets the user's input by matching it to the patterns and returns possible completions. It is described in the next section.
4. **Frontend:** A web server running a web application to interact with the user.
5. **Lexicon:** a lexicon that contains information about how the vocabulary elements of the dataset are lexicalized in natural language. The system currently uses the lexicon for DBpedia in three languages (Spanish, English, German).

4 Question Interpretation component

Figure 4 shows the java classes diagram of the component (and their methods) as well as class dependencies. In short, there are two basic interfaces (`ParseableElement` and `QAPattern`). There are 4 classes that implement the methods defined by `ParseableElement`: `StringElement`, `InstanceElement`, `PropertyNoun` and `ClassNoun`. In this exampe we define two classes implementing the `QAPattern` interface: `QueryPattern1` and `QueryPattern2`. The `QueryPatternManager` class manages the query patterns and sequentially requests the possible values to show (as a list) to the user (method `getNext()`). If several query patterns are available at that point, they will be merged and shown to the user. Once the user selects a list item, the whole string is parsed (method `parse()`). Depending on the user selection some patterns will be discarded. At the end of the process at least one query pattern will be available.

5 Conclusions

We have presented a preliminary system that interprets natural language questions with respect to SPARQL and has three key features: i) it is *pattern-based* in the sense that it exploits a number of patterns that cover the most frequent types of SPARQL queries, ii) it is *data-driven* in the sense that it is grounded in an analysis of frequent query types and in that it uses indices to anticipate possible completions of a query in real time, and iii) it is *guided* in the sense that it computes potential completions and displays these to the user. In this

[4] See `http://github.com/ag-sc/InterQA`

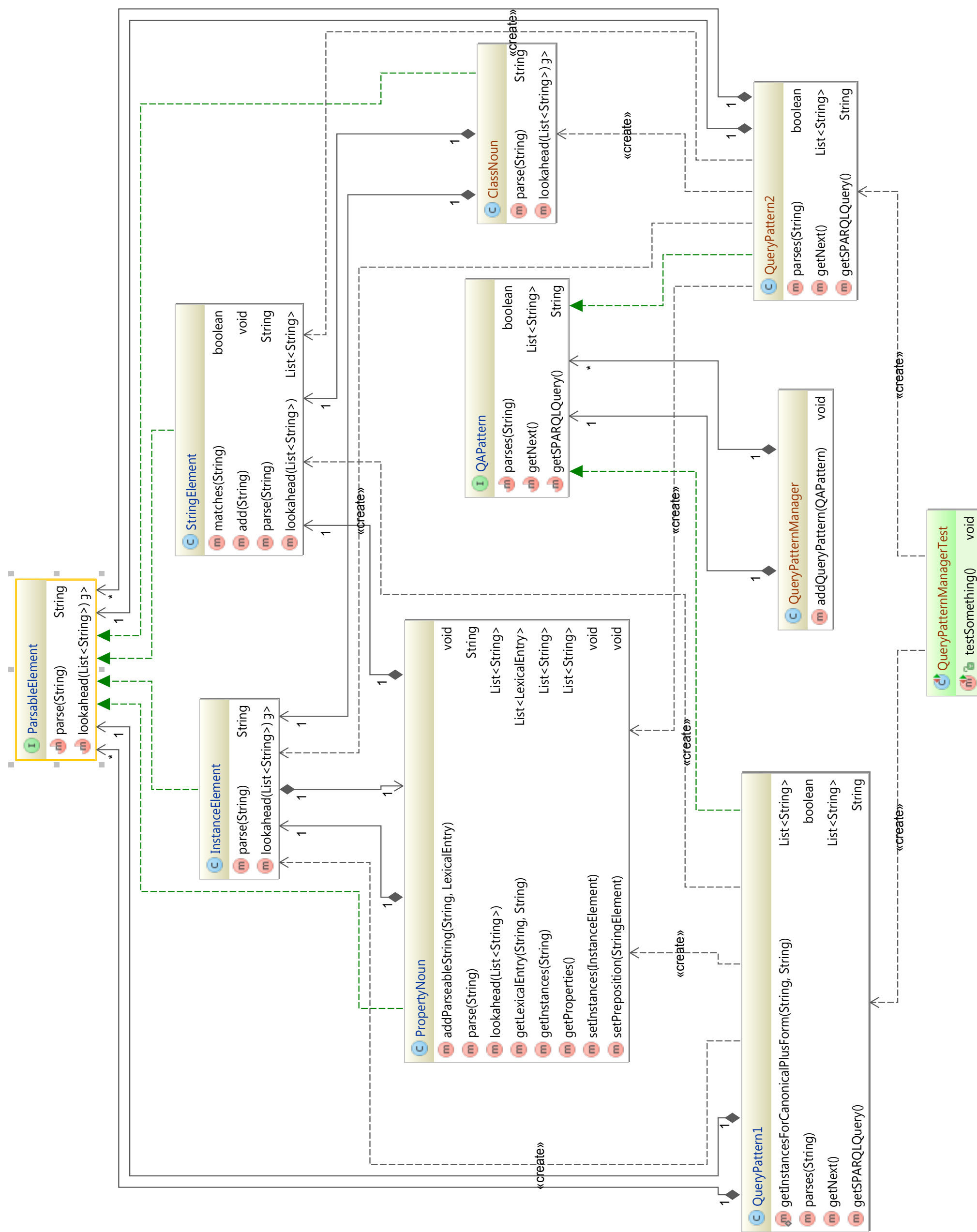

Fig. 4. Class diagram of the Question Interpretation component.

way, one ensures that only such queries are entered into the system that are also valid and can be processed and interpreted by the system, thus reducing errors and increasing robustness. Future work includes increasing the coverage of the system with more patterns, improving the indexes and covering languages other than English. An evaluation of the approach remains to be done.

References

1. E. Cabrio, J. Cojan, A. P. Aprosio, B. Magnini, A. Lavelli, and F. Gandon, "QAKiS: an Open Domain QA System based on Relational Patterns," in *Proc. of 11th International Semantic Web Conference. Poster & Demostrations Track*, 2012.
2. S. Campinas, T. E. Perry, D. Ceccarelli, R. Delbru, and G. Tummarello, "Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation," in *Proc. of 23rd International Workshop on Database and Expert Systems Applications (DEXA)*, pp. 261–266, IEEE, 2012.
3. R. García, J. M. Gimeno, F. Perdrix, R. Gil, and M. Oliva, "The Rhizomer Semantic Content Management System," in *Proc. of First World Summit on the Knowledge Society. Emerging Technologies and Information Systems for the Knowledge Society. LNCS 5288*, pp. 385–394, Springer, 2008.
4. L. Han, T. Finin, and A. Joshi, "GoRelations: An Intuitive Query System for DBpedia," in *Proc. of Joint International Semantic Technology Conference (JIST). LNCS 7185*, pp. 334–341, Springer, 2011.
5. S. Lohmann, P. Heim, T. Stegemann, and J. Ziegler, "The RelFinder User Interface: Interactive Exploration of Relationships between Objects of Interest," in *Proc. of 15th International Conference on Intelligent User Interfaces*, pp. 421–422, ACM, 2010.
6. S. Scheider, A. Degbelo, R. Lemmens, C. van Elzakker, P. Zimmerhof, N. Kostic, J. Jones, and G. Banhatti, "Exploratory querying of SPARQL endpoints in space and time," *Semantic Web Journal (to appear)*, 2015.
7. V. Lopez, M. Fernández, E. Motta, and N. Stieler, "PowerAqua: Supporting users in querying and exploring the Semantic Web," *Semantic Web Journal*, vol. 3, no. 3, pp. 249–265, 2011.
8. N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Attempto Controlled English for Knowledge Representation," in *Proc. of 4th International Summer School. Reasoning Web. LNCS 5524*, pp. 104–124, Springer, 2008.
9. A. Bernstein and E. Kaufmann, "GINO–A Guided Input Natural Language Ontology Editor," in *Proc. of 5th International Semantic Web Conference (ISWC). LNCS 4273*, pp. 144–157, Springer, 2006.
10. M. Rico and A. Gómez-Pérez, "The Pareto principle also rules SPARQL queries," *Journal of Web Semantics (in preparation)*, 2015.
11. J. McCrae, D. Spohr, and P. Cimiano, "Linking Lexical Resources and Ontologies on the Semantic Web with Lemon," in *Proc. of 8th Extended Semantic Web Conference (ESWC). Part 1. The Semantic Web: Research and Applications. LNCS 6643*, pp. 245–259, Springer, 2011.
12. C. Unger, J. McCrae, S. Walter, S. Winter, and P. Cimiano, "A lemon lexicon for DBpedia," in *Proc. of 1st International Workshop on NLP and DBpedia, co-located with the 12th International Semantic Web Conference (ISWC). CEUR Vol-1064*, 2013.

13. N. Mihindukulasooriya, M. Rico, and R. Garcia-Castro, “An Analysis of Quality Issues of the Properties Available in the Spanish DBpedia,” in *Proc. of Conference of the Spanish Association for Artificial Intelligence (CAEPIA), LNCS (to appear)*, Springer, 2015.