The original publication is available at www.springerlink.com
http://link.springer.com/chapter/10.1007/978-3-642-20095-3_41
DOI: 10.1007/978-3-642-20095-3_41

An Access Layer to PolNet – Polish WordNet

Marek Kubis

Adam Mickiewicz University, Faculty of Mathematics and Computer Science, Department of Computer Linguistics and Artificial Intelligence,
ul. Umultowska 87, 61-614 Poznań, Poland
mkubis@amu.edu.pl

Abstract. The paper describes an access layer developed in order to provide access to PolNet (a lexical database developed for the Polish language). The access layer was developed on top of a domain-specific language designed to query WordNet-like lexical databases (WQuery). The paper presents the overall architecture of the access layer and shows typical queries passed by an AI system with NL competence (POLINT-112-SMS) to WQuery. The paper discusses the reasons for integrating an ontology into an NLP system through a domain-specific query language.

Keywords: wordnet, ontology, lexical database, query language, domain specific language

1 Introduction

POLINT-112-SMS is a natural language processing AI system focused on supporting information management in crisis situations [1]. POLINT-112-SMS has to access PolNet (a WordNet-like lexical database developed for the Polish language [2]) to guide such procedures as assignment of possible meanings to words and construction of the internal representation of a whole sentence.

In order to integrate POLINT-112-SMS with PolNet, a software layer has been created that refers to the wordnet stored outside of the system and provides an API used by the system modules to access the database. The layer has been developed on top of WQuery – a system designed to query WordNet-like lexical databases using domain specific artificial language. WQuery is an open source tool. It operates on platforms that provide Java Runtime Environment and is able to work with any wordnet that is stored in an XML file that corresponds to the Global WordNet Grid DTD [3].

The rest of the paper is organized as follows: In the first part the overall architecture of the access layer is described. The second part explains the basic syntactic constructs of the WQuery language used in queries generated by POLINT-112-SMS. The third part presents typical queries passed to WQuery by POLINT-112-SMS. The last part confronts the adopted approach with other possible solutions.

2 Access Layer Architecture

The overall architecture of PolNet Access Layer is shown in Fig. 1. The layer is accessed by POLINT-112-SMS modules, such as NLPM (Natural Language Processing Module), SAM (Situation Analysis Module) and DMM (Dialogue Maintenance Module). POLINT-112-SMS modules access PolNet through API (Application Programming Interface), which consists of procedures responsible for the execution of queries formulated in the WQuery language. An API procedure checks if a query result may be retrieved from PolNet Cache. If the query result is not available or if PolNet Cache has been switched off, then the procedure passes the query directly to WQuery Client, which delegates it further to WQuery Server. PolNet Cache has been introduced into the architecture to reduce query processing time and to make PolNet available to POLINT-112-SMS even if WQuery Server is down.

Query processing takes place on WQuery Server, which is independent of POLINT-112-SMS and may be deployed on a separate physical machine. The server contains WQuery Interpreter, which is responsible for query execution and Wordnets Datastore, which contains one or more (possibly different) PolNet instances. PolNet is loaded into the datastore from an XML file.

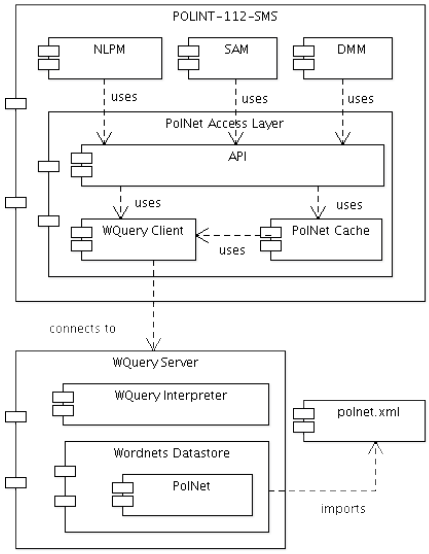

Fig. 1. PolNet Access Layer architecture.

3 WQuery Language

The complete description of the WQuery language is outside the scope of this paper (for details see [4]). The description below concentrates on constructs that are used in POLINT-112-SMS and skips the other ones.

3.1 Data Types

Basic Data Types. There are six basic data types in WQuery. Two of them – the word sense data type and the synset data type – represent concepts around which WordNet-like lexical databases are organized [5]. The other basic data types – strings, integers, floats and booleans – have been introduced in order to support general purpose computations. The string data type represents character data. The integer and float data types represent numbers. The boolean data type represents logical values – `true` and `false`.

Word senses are presented in this paper as triples that consist of a word form, a sense number and a part of speech symbol (*n* for nouns, *v* for verbs, etc.) separated by colons as in the following example

`przedmiot:1:n` *(an object)*

Synsets are presented as lists of word senses surrounded with "{" and "}".

```
{ obiekt:1:n przedmiot:1:n rzecz:1:n }
```

Relations. Let *X* mean a set of values of the basic data type *T (predecessor set)*, Let *Y* mean a set of values of the basic data type *U (successor set)*. A relation is a subset of the Cartesian product of X and Y.

WQuery uses relations to transform members of predecessor sets into members of successor sets. For example, `hyponyms` is a relation that transforms synsets into their hyponyms and `gloss` is a relation that transforms synsets into strings that represent their descriptions.

Tuples. Tuples are finite, ordered collections. Each element of a tuple must belong to a basic data type. Tuples in WQuery are flat. It is impossible to construct a tuple that contains another tuple as its element.

3.2 Basic Syntax

Generators. Generators are expressions that represent[1] multisets of objects which share the same (basic) data type. Generators are the simplest valid WQuery queries.

Boolean, integer and floating point constants generate multisets that contain exactly one element defined by the expressed constant.

```
wquery> true
true
wquery> 123
123
wquery> 1.2e2
120.0
```

1 Generate in WQuery terminology.

The content of the multiset generated from a string depends on the surrounding characters:

- If the string has been surrounded with single quotes, then the generated multiset contains that string if and only if it is a valid word form in the given wordnet; otherwise, the generated multiset is empty.
- If the string has been surrounded with back quotes, then the generated multiset contains that string.
- If the string has not been surrounded with any quotation marks, then it is treated as if it has been surrounded with single quotes.

```
wquery> 'przedmiot'
przedmiot
wquery> `a1b2c3d4`
a1b2c3d4
wquery> 'a1b2c3d4'
(no result)
wquery> przedmiot
przedmiot
```

A multiset that contains at most one word sense may be generated from a string, a sense number and a part of speech symbol joined together with colons. If the constructed triple does not point to a correct word sense in the wordnet, then the generated multiset will be empty.

```
wquery> przedmiot:1:n
przedmiot:1:n
wquery> a1b2c3d4:1:n
(no result)
```

A multiset that contains synsets may be generated from string and word sense generators by surrounding them with “{” and “}” signs. The multiset will contain every synset that includes at least one word form (or word sense) represented by the generator found between “{” and “}”.

```
wquery> {przedmiot}
{ obiekt:1:n przedmiot:1:n rzecz:1:n }
{ obiekt:2:n przedmiot:4:n cel:4:n przedmiot:4:n }
{ przedmiot:3:n temat:1:n }
{ przedmiot:2:n przedmiot nauczania:1:n }
{ przedmiot:5:n }
wquery> {przedmiot:1:n}
{ obiekt:1:n przedmiot:1:n rzecz:1:n }
```

Curly brackets “{}” without a generator inside represent all synsets found in the given wordnet.

Paths. A path consists of a generator followed by zero or more transformations.[2] Each transformation begins with a dot followed by the name of the relation that should be applied to the expression stated before the dot. The result of transforming an expression that generates the multiset *M* using the relation *R* is a multiset

$$\{\ b \mid a \text{ belongs to } M \text{ and } (a, b) \text{ belongs to } R\ \}.$$

For example, to find all hyponyms of synsets that contain the second noun sense of the word form *spoiwo (a binder)* one may write

```
wquery> {spoiwo:2:n}.hyponyms
{ gips:1:n }          (gypsum)
{ klej:1:n }          (glue)
{ wapno:1:n }         (lime)
```

and to find all words that belong to the hyponyms mentioned above one may transform the former expression using the `words` relation

```
wquery> {spoiwo:2:n}.hyponyms.words
gips
klej
wapno
```

If the result of a transformation has the same data type as the source, then it is possible to compute transitive closure of the transforming relation by placing the "!" operator after the relation name. For example, to find all transitive hyponyms of synsets that contain the second noun sense of the word form *spoiwo* one may write

```
wquery> {spoiwo:2:n}.hyponyms!
{ gips:1:n }
{ klej:1:n }
{ lepik:1:n }
{ wapno:1:n }
```

Filters. A filter is an expression that may be placed after any step of a path in order to remove some elements from the multiset generated by that step. A filter consists of a condition surrounded with "[" and "]". A condition is an expression that involves such operators as the equality check "=", the inequality check "!=", the multiset membership check "in", etc. Conditions may be combined together using logical operators "and", "or" and "not".

Each element generated by the filtered step is passed separately to the filter. The element being analyzed in the current pass may be referenced in the condition using "#" operator. For example, to find all synsets that contain the word form *przedmiot*, except the one that contains the first noun sense of the word form *przedmiot*, the following expression may be written

---

2 The generator and the following transformations are called (path) steps.

```
wquery> {przedmiot}[# != {przedmiot:1:n}]
{ obiekt:2:n przedmiot:4:n cel:4:n przedmiot:4:n }
{ przedmiot:3:n temat:1:n }
{ przedmiot:2:n przedmiot nauczania:1:n }
{ przedmiot:5:n }
```

A reference and the following dot may be omitted if they are followed by at least one transformation, so the expression

```
wquery> {przedmiot}[rzecz in #.words]
```

may be reformulated as

```
wquery> {przedmiot}[rzecz in words]
```

Filters may also be used as generators. A filter placed as the first step of a path generates a multiset that consists of exactly one boolean value (a result of the condition inside the filter).

```
wquery> [rzecz in {przedmiot}.words]
true
```

Selectors. A selector is an expression that makes it possible to retrieve data from the multiset generated by the chosen step of a path. The selector consists of "<" and ">" signs that surround the chosen step. For example, to find hyponyms of hyponyms of synsets that contain the word form *izba* in its second noun sense (*a chamber of parliament*) together with hypernyms of their hypernyms, one may write

```
wquery> <{izba:2:n}>.hyponyms.<hyponyms>
{ izba:2:n }
        { sejm:1:n }
        { senat:1:n }
```

Path Expressions. Paths may be combined together using `union`, `intersect`, `except` and "," operators. They are responsible for creating unions, intersections, differences and Cartesian products of multisets generated by paths passed to them as arguments.

```
wquery> {spoiwo:2:n}.hyponyms union {przedmiot}
{ obiekt:1:n przedmiot:1:n rzecz:1:n }
{ obiekt:2:n przedmiot:4:n cel:4:n }
{ gips:1:n }
{ cement:1:n }
{ przedmiot:3:n temat:1:n }
{ przedmiot:2:n przedmiot nauczania:1:n }
{ przedmiot:5:n }
{ klej:1:n }
```

```
{ wapno:1:n }
wquery> {spoiwo:2:n}.hyponyms intersect {gips}
{ gips:1:n }
wquery> {spoiwo:2:n}.hyponyms except {gips}
{ klej:1:n }
{ wapno:1:n }
```

Imperative Expressions. WQuery possesses several expressions that support the imperative programming paradigm.

- An *emission* is an expression of the form

  ```
  emit path_expr
  ```

  which passes tuples generated by the path expression `path_expr` to the output.
- A *conditional*[3] is an expression of the form

  ```
  if path_expr then imp_expr_a else imp_expr_b
  ```

  which executes the imperative expression `imp_expr_a` if the path expression `path_expr` is true and executes `imp_expr_b` otherwise. The subexpression `else imp_expr_b` is optional. The path expression `path_expr` is assumed to be false if and only if it generates an empty multiset or the generated multiset includes exactly one tuple that consists of boolean values and at least one of those values is false. The second case makes it possible to use filter generators as conditionals in a convenient way, as shown in the example at the end of this section.
- A *block* is an expression of the form

  ```
  do imp_expr_1 imp_expr_2 … imp_expr_n end
  ```

  which executes sequentially imperative expressions `imp_expr_1`, `imp_expr_2`, …, `imp_expr_n`.
- An *assignment* is an expression of the form

  ```
  var_decls = path_expr
  ```

  where the path expression `path_expr` has to generate exactly one tuple. The assignment binds variable names from the comma separated list `var_decls` to consecutive elements of the tuple generated by the path expression `path_expr`.
- An *iterator* is an expression of the form

  ```
  from var_decls in path_expr imp_expr
  ```

3 Conditionals shall not be confused with expressions called conditions, which are placed inside filters.

which for every tuple generated by the path expression `path_expr` executes the imperative expression `imp_expr` with variable names from the comma separated list `var_decls` bound to consecutive elements of the tuple being processed in the current step.

The following expression iterates through all senses of synsets that contain the word form *przedmiot* and, depending on the sense number, returns the word form or the part of speech symbol of the word sense processed in the current step.

```
wquery> from $a in {przedmiot}.senses
wquery> do
wquery>     if [$a.sensenum <= 2]
wquery>         emit $a.word
wquery>     else
wquery>         emit $a.pos
wquery> end
n
przedmiot nauczania
przedmiot
temat
n
n
n
obiekt
rzecz
przedmiot
obiekt
```

4 Typical Queries Appearing in POLINT-112-SMS

The following subsections present three tasks performed by modules of POLINT-112-SMS that require access to PolNet to be fulfilled. The first and second task play an important role in the process of analyzing and interpreting sentences in the NLP module. The third one is executed by the PolNet Access Layer itself. Besides the tasks described below, all other procedures of POLINT-112-SMS that require access to PolNet obtain it through the layer presented in this paper.

4.1 Obtaining Word Meanings

Queries that retrieve all word senses of a particular word form are executed for all lemmas of words of every sentence passed to the POLINT-112-SMS NLP module in order to link words to their possible meanings stored in PolNet. For example, to find the possible meanings of words in the sentence “Kibic ma czapkę.” (“A team supporter has a cap.”) the following three queries shall be executed

```
wquery> kibic.senses
wquery> mieć.senses
wquery> czapka.senses
```

A word sense that belongs to a synset which is marked as *not lexicalized* should not be accepted as a possible meaning of a word passed to the NLP module. In order to exclude such senses, the NLP module extends the queries presented above with filters that check whether the senses belong to synsets that are transformable using the relation `nl` (not lexicalized) to the boolean value `false`

```
wquery> kibic.senses[synset.nl = false]
wquery> mieć.senses[synset.nl = false]
wquery> czapka.senses[synset.nl = false]
```

4.2 Creating and Composing Frames

POLINT-112-SMS internal knowledge representation is based on the idea of frames [6]. The construction of frames is guided by queries that check synset collection membership.

Firstly, queries of this kind are used to choose frames appropriate to represent the meanings of particular words. For example, the system represents articles by the frame `article` and the NLP module assumes that every article is a transitive hyponym of a synset that contains the word form *przedmiot* in its first noun sense, so if the sentence “Kibic ma czapkę.” is analyzed, then for every sense *w:n:p* of the word form *czapkę (a cap)* found by the last query in Section 4.1, the NLP module executes a query

```
wquery> [{w:n:p} in {przedmiot:1:n}.hyponyms!]
```

to decide whether the word form *czapkę* may be mapped to the frame `article`. In this stage the majority of possible meanings found by the queries in Section 4.1. is eliminated because no corresponding frames exist.

Secondly, queries that check synset collection membership are used to determine if one frame may be nested as a slot of the other one. The following query checks whether a synset that contains the sense *w:n:p* is equal to or is a transitive hyponym of a synset that contains the word form *nakrycie głowy* in its first noun sense (*headgear*). This query is executed for every sense *w:n:p* of the word form *czapkę* to check if the frame that represents *czapkę* may be put in the slot `head` of the frame `appearance`.

```
wquery> [{w:n:p} in
wquery>     ({'nakrycie głowy':1:n}
wquery>         union
wquery>     {'nakrycie głowy':1:n}.hyponyms!)]
```

This stage eliminates meanings for which frame nesting has not led to a representation of the whole sentence as a single frame. All single frame based representations of the whole sentence created during this stage are passed to the Dialogue Maintenance Module as alternative meanings of the sentence.

4.3 Refreshing PolNet Cache

PolNet Cache stores links between synsets and their transitive hyponyms. All pairs that consist of a synset followed by its transitive hyponym are obtained by the query

```
wquery> <{}>.<hyponyms!>
```

The cache also stores links between synsets and their senses represented as triples that consist of a word, sense number and synset. The triples are generated by the query below

```
wquery> from $s, $w in <{}>.<senses>
wquery> emit $w.word, $w.sensenum, $s
```

5 Discussion

The simplest method of integrating a wordnet with an NLP system is to represent the wordnet directly by using data structures of the system (for example, to store the wordnet as a list of terms in a system that is implemented in Prolog). This approach was rejected because it introduces high coupling between the system and the wordnet. Every time a new version of PolNet was integrated the code of the system would have to be updated, thus resulting in a new version of the system.

The second method is to store the wordnet outside the system and to provide a set of basic access procedures, such as “get a synset by its word form and sense number” or “get hyponyms of a synset”. This approach was rejected because the queries responsible for creating and nesting frames (like those shown in Section 4.2.) combine calls to such procedures together with the other ones (like multiset union, transitive closure computation and multiset filtering). These combinations vary between frames and if they were hard-coded into the system it would be hard to understand and modify them.

The third method is to integrate a wordnet through a tool that provides a versatile query language. However, query languages (besides WQuery) do not operate on wordnet related data structures, such as synsets, word senses and relations, but rather enforce the adoption of their own data model (e.g. tables and columns in SQL, XML elements and attributes in XQuery, RDF triples in SPARQL). This makes queries more verbose and harder to understand. For example, to formulate an SQL query analogous to those presented at the bottom of Section 4.1 one would have to write

```
select distinct w.id from wsenses w
```

```
where w.word = 'czapka' and (
(select nl.value from wsenses s
inner join nl
on s.synset_id = nl.synset_id
where w.id = s.id) = false)
order by w.id
```

where `wsenses(id, word, sensenum, synset_id)` is a table that stores word sense identifiers, words, sense numbers and synset identifiers, and `nl(synset_id, value)` is a table that stores synset identifiers together with boolean values that indicate whether a synset is lexicalized.

6 Conclusion

This paper describes how to integrate an AI system with NL competence (POLINT-112-SMS) with a WordNet-like lexical database (PolNet). The approach is based on a domain-specific language WQuery used to query the wordnet. Usage of WQuery as a part of the layer developed to access PolNet has two main advantages. Firstly, it decouples the system from the wordnet, thus permitting to switch between different versions of PolNet without the need to create a new version of the system. Secondly, it provides straightforward API that allows to create concise queries that involve such operations as computing transitive closures and filtering multisets.

Although the paper describes integration between POLINT-112-SMS and PolNet, the approach based on WQuery is so generic that it can be easily adopted in other systems that have to integrate a wordnet. The only requirement is to provide a copy of the wordnet that is compatible with the Global WordNet Grid XML DTD.

Acknowledgements

This work has been partially supported by the Polish Ministry of Science and Higher Education, grant R00 028 02 (within the Polish Platform for Homeland Security).

References

1. Vetulani, Z., Marciniak, J., Konieczka, P., Walkowska, J. An SMS-based System Architecture (Logical Model) to Support Management of Information Exchange in Emergency Stuations. POLINT-112-SMS. In: Zhongzhi Shi, Mercier-Laurent. E., Leake D. (eds.) Intelligent Information Processing IV (Book Series: IFIP International Federation for Information Processing, Subject collection: Computer Science), Volume 288/2009, pp. 240-253., Springer Boston, (2008)
2. Vetulani Z., Walkowska, J., Obrębski, T., Marciniak, J., Konieczka, P., Rzepecki, P. An Algorithm for Building Lexical Semantic Network and Its Application to PolNet - Polish WordNet Project. In: Vetulani, Z., Uszkoreit, H. (eds.) Human Language Technology. Challanges of the Information Society. Third Language and Technology Conference, LTC

2007, Poznan, Poland, October 5-7, 2007, Revised Selected Papers. LNAI, vol. 5603, pp. 369-381, Springer, Heidelberg (2009)
3. Global WordNet Grid XML DTD, http://www.globalwordnet.org/gwa/grid/bwn2.dtd, Access date: September 23, 2010
4. Kubis, M., WQuery User Guide, http://wquery.org/user-guide.pdf, Access date: September 23, 2010
5. Fellbaum, C.: WordNet: An Electronic Lexical Database (Language, Speech, and Communication), MIT Press (1998)
6. Minsky, M.: A framework for representing knowledge, MIT-AI Laboratory Memo 306 (1974), http://web.media.mit.edu/~minsky/papers/Frames/frames.html, Access date: October 10, 2009.