THE FINAL VERSION APPEARED IN INTERNATIONAL JOURNAL OF
INTELLIGENT INFORMATION AND DATABASE SYSTEMS, VOL. 9, NO. 2, 2016.
HTTPS://DX.DOI.ORG/10.1504/IJIIDS.2016.075419

A Query Language for WordNet-like Lexical Databases

Marek Kubis

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

Abstract: WordNet-like lexical databases are used in many natural language processing tasks, such as word sense disambiguation, information extraction and sentiment analysis. The paper discusses the problem of querying such databases. The types of queries specific to WordNet-like databases are analyzed and previous approaches that were undertaken to query wordnets are discussed. A query language which incorporates data types and syntactic constructs based on concepts that form the core of a WordNet-like database (synsets, word senses, semantic relations, etc.) is proposed as a new solution to the problem of querying wordnets.

Keywords: Wordnet; lexical database; domain-specific language.

Reference

Biographical notes: Marek Kubis received his PhD from Adam Mickiewicz University in 2013. His research interests include lexical semantics, query languages and natural language processing systems.

1 Introduction

Wordnets are used in many natural language processing tasks ranging from word sense disambiguation to information extraction and sentiment analysis. A WordNet-like lexical database[a] is a network of word senses grouped into sets of synonyms that preserves information about semantic relationships that hold among words of a particular language or a group of languages. Although wordnets and the systems to maintain them have been created for more than 30 natural languages, no common tool has emerged that would allow to access the data collected in the different WordNet-like lexical databases in a uniform manner. The data collected in a database management system are usually provided through a dedicated programming language (*query language*). The expressions of this language (*queries*) describe criteria that the data have to satisfy in order to be retrieved from the

[a]The term *wordnet* and *WordNet-like lexical database* are used interchangeably in this paper to refer to any database that employs data organization inspired by that of WordNet (Fellbaum, 1998).

database. A wide variety of query languages of both theoretical and practical importance has been proposed for relational (e.g. Abiteboul et al., 1995), object-oriented (Cattell and Barry, 2000) and semi-structured (Abiteboul et al., 1997; Clark and DeRose, 1999; Boag et al., 2010) database management systems. These languages possess data types based on the concepts that appear in the data models for which they have been developed, and provide dedicated syntactic constructs that allow to formulate concise and easy-to-understand queries. We propose a similar approach to data management with respect to WordNet-like lexical databases by introducing WQuery – a query language that incorporates data types and syntactic constructs based on concepts that form the core of a wordnet.

In regard to the conference paper under the same title (Kubis, 2012), this article contains the following novel elements:

1. Description of the structure of the corpus of wordnet-specific queries gathered for the purpose of developing the language (Section 3).
2. An extended set of sample queries extracted from our corpus that encompasses examples of multilingual queries and queries that involve lexical relations (Section 3).
3. A formal definition of an instance of the WQuery wordnet model (Section 4).
4. An outline of the proof that every query formulable in relational algebra over an instance of the WQuery wordnet model is expressible in the WQuery language (Section 7).
5. A set of rules that can be used to construct a WQuery wordnet model from the LMF-based wordnet models (Section 8.1).
6. Discussion of the advantages of the wordnet-specific data model in comparison to the hierarchical XML-based models (Section 8.2).
7. The formal semantics for the presented language (Appendix A).
8. Analysis of the time complexity of the operations that are executed in the query evaluation process performed by the prototype interpreter (Appendix B).

2 WordNet-like Databases

We begin our description of WordNet-like lexical databases with the definition of a word sense which is a triple that encompasses a:

1. word, e.g. *car*, *auto*, *ambulance*
2. sense number: $1, 2, \ldots$
3. part-of-speech (POS) symbol: *n* for nouns, *v* for verbs, etc.

The *sense number* is a positive integer that distinguishes different senses of a word that share the same POS. In the examples we place ":" to separate the word, sense number and POS symbol of a word sense. Thus, the first noun sense of the word *set* is written as *set:1:n* and its second verb sense as *set:2:v*.

A wordnet consists of *synsets*. A synset is a set of word senses that are synonymous (i.e. they share a meaning). For example, an automobile is represented in WordNet[b] by the set *{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }* accompanied by the gloss *"a motor vehicle with four wheels; usually propelled by an internal combustion engine (...)"*. Since word senses distinguish meanings of a word and synsets group word senses with the same meaning none of the word senses encompassed by the same synset can share the same word. Furthermore, the set of synsets is a partition of the set of word senses with respect to the synonymy relation. Hence, if a word has more than one word sense, then all the word senses belong to different synsets. For instance, the first noun sense of the word *car* is a member of the *{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }* synset mentioned above and the third noun sense of this word belongs to the synset *{ car:3:n gondola:3:n }* which is accompanied in WordNet by the gloss *"the compartment that is suspended from an airship and that carries personnel and the cargo and the power plant"*. Since a word sense identifies its synset, we will usually write only one of them within *{* and *}* and replace the others with "..." (e.g. *{ car:1:n ... }*).

Synsets are connected through *semantic relations*, i.e. binary relations which represent semantic relationships that hold between the meanings of synsets. For instance, the *hypernymy* relation connects synsets representing more general concepts to synsets representing more specific ones (e.g *{ car:1 ... }* is a hypernym of *{ ambulance:1:n }*) and the *meronymy* relation links whole things to their parts (*{ air bag:1:n }* is a meronym of *{ car:1:n ... }*). Word senses are connected through *lexical relations*, i.e. binary relations which represent lexical relationships that hold among words. For example, the *antonymy* relation links word senses that represent opposite concepts such as *good:2:n* and *evil:3:n*. The set of semantic and lexical relations varies among wordnets even if they are created for the same language (cf. Maziarz et al., 2012; Vetulani, 2012). Some relations in a wordnet may be inferred from others. For instance, *hyponymy* connects synset A to synset B if and only if *hypernymy* connects synset B to synset A. The same relationship holds for *meronymy* and *holonymy*.

Besides the data types mentioned above, wordnets also encompass additional data that vary largely among them. For instance, WordNet provides glosses for all synsets, EuroWordNet (Vossen, 2002) stores relations that hold between synsets and Inter-Lingual Index, and PolNet (Vetulani et al., 2010) provides for synsets examples of usage of the words that belong to them. Floating-point numbers are used in SentiWordNet (Baccianella et al., 2010) to represent the notion of synset positivity, integers are used in WordNet to count occurrences of particular senses in semantic concordances and boolean values are used in PolNet to mark not lexicalized synsets.

3 Wordnet-specific Queries

Although WordNet-like lexical databases are used in many natural language processing tasks, the class of queries to be answered by a wordnet-specific query language has not been previously investigated. Therefore, we analyzed papers from major wordnet conferences (Sojka et al., 2003, 2005; Tanács et al., 2007; Bhattacharyya et al., 2010; Fellbaum and Vossen, 2012) and reports from EuroWordNet (Vossen, 2002) and BalkaNet (Tufis, 2004) projects and extracted wordnet-specific queries from them. We extended the collected corpus

[b] Examples of synsets, word senses, glosses and other wordnet-specific data used in the paper are taken from version 3.0 of WordNet (Fellbaum, 1998; Princeton University, 2011a), if not stated otherwise.

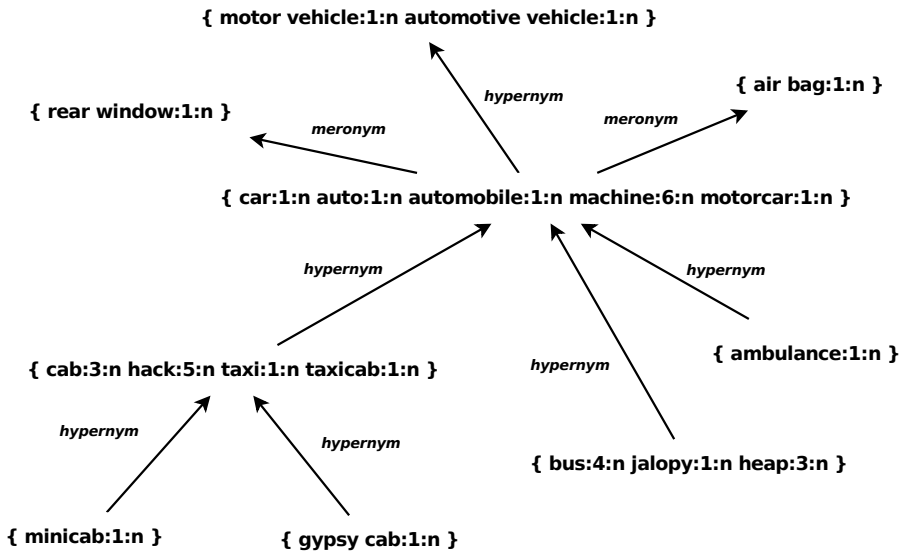

Figure 1 A fragment of WordNet

with queries that we found to be expressible in the wordnet editor DEBVisDic (Horak et al., 2005) and the command-line interface distributed with WordNet (Tengi, 1998). We also incorporated unpublished queries that we found to be useful while validating and evaluating PolNet (Vetulani et al., 2010). The collected corpus contains over eighty wordnet-specific queries. Among them we identified four use-cases that a query language for WordNet-like databases has to fulfil:

1. Navigation – retrieval of synsets, word senses and words together with other data that are interlinked to them through various types of relations.

2. Validation – identification of data that are incorrect such as cycles in the hypernymy relation or redundant meronymy links.

3. Aggregation – computation of quantitative measures that describe the wordnet as a whole such as the number of synsets.

4. Similarly calculation – retrieval of data that are necessary to determine values of semantic similarity measures among words.

The rest of this section discusses examples of queries that represent these use-cases.

3.1 Navigation

Navigation through synsets, word senses, words and the related data is one of the basic functions of wordnet editors and command-line interfaces (cf. Horak et al., 2005; Tengi, 1998). Ranging from queries that retrieve synsets and word senses that contain particular words

Query 1: Find word senses of the word *car*.

Query 2: Find synsets that contain word senses of the word *car*.

through queries that access their properties such as the following

Query 3: Find glosses of the synsets that contain word senses of the word *car*.

Query 4: Find usage examples of the synsets that contain the first noun sense of *car*.

to that which traverse semantic and lexical relations

Query 5: Find meronyms of the synsets that contain word senses of *car*.

Query 6: Find antonyms of the first adjective sense of the word *good*.

Query 7: Find antonyms of the word *good*.

A more elaborate example of a traversal through a wordnet is a tree query.

Query 8: *(Tree query)* Find all paths such that the first synset of a path belongs to the set S and every other synset of the path is accessible from the previous one through the relation r.

Tree queries may be used to build tree views for semantic relations in wordnet browsers and editors. They may also be used to compute structural measures, such as minimum/maximum depth and height of the hypernymy hierarchy.[c] Devitt and Vogel (2003) present values of such measures obtained for WordNet. In (Kubis, 2011) tree queries are used in the process of building frame-based representations for natural language sentences.

3.2 Validation

The problem of identifying invalid data in a wordnet is discussed in several publications (Grigoriadou et al., 2004; Koeva et al., 2004a,b; Smrž, 2004) within the BalkaNet project. The issue of checking correctness of the wordnet data is one of the arguments given by Koeva et al. (2004a, p. 53) and Rizov (2008, p. 1524) to justify development of wordnet-specific query languages. One third of the queries gathered in our corpus are responsible for finding data that are (or potentially can be) incorrect. We present below three validation tasks that can be realized as queries over a wordnet that identify incorrect data.

3.2.1 Cycle Detection

In the process of extending a wordnet lexicographers responsible for connecting synsets may introduce cycles in semantic relations which by their definition are acyclic. Detection of such cycles is a common procedure performed to control the quality of a wordnet (e.g. Koeva et al., 2004a; Smrž, 2004; Vetulani et al., 2010), therefore we included in our corpus the following query

Query 9: Find synsets that belong to a cycle in the semantic relation r.

[c] By the definition a tree query returns paths in the network of synsets, thus in the case of multiple hypernymy distinct paths within the hypernymy hierarchy are returned.

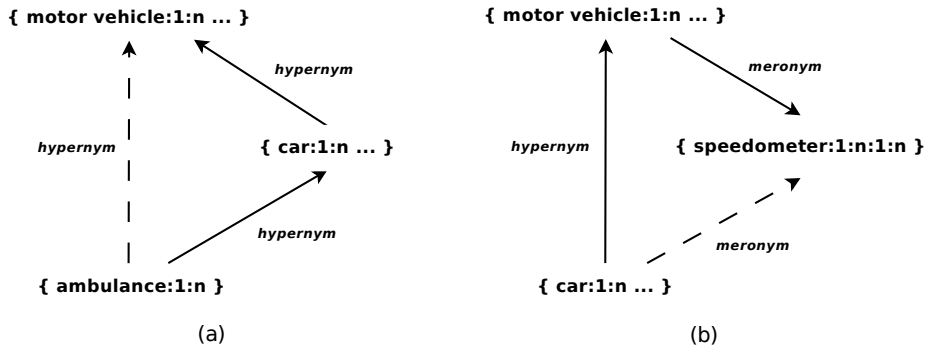

Figure 2 Redundant edges of hypernymy (a) and meronymy (b)

3.2.2 Interlingual Consistency Checking

Koeva et al. (2004a) present several examples of queries that can be expressed in their modal logic language to perform validation of the content and structure of the Bulgarian Wordnet. Particularly interesting are the ones that involve interlingual links between Bulgarian and English wordnets. We refer to two such queries for the purpose of presentation.

Query 10: "Return all ID numbers of Base English synsets which are not included in Bulgarian wordnet" (Koeva et al., 2004a, p. 53)

Query 11: "Return all ID numbers of Base English synsets, which in Bulgarian are marked as not Base." (Koeva et al., 2004a, p. 53)

3.2.3 Redundancy Checking

Let us consider the (artificial) example shown in Figure 2 (a). Since *{ car:1:n ... }* is a hypernym of *{ ambulance:1:n }* and *{ motor vehicle:1:n ... }* is a hypernym of *{ car:1:n ... }* we already know that *{ ambulance:1:n }* represents a concept that is more specific than the one represented by *{ motor vehicle:1:n ... }*. Hence, the hypernymy link between *{ motor vehicle:1:n ... }* and *{ ambulance:1:n }* is redundant. The purpose of the following query which has been incorporated into the validation procedure performed to control the qualify of PolNet (Vetulani et al., 2010) is to identify such redundant links.

Query 12: *(Redundant hypernyms query)* Find pairs of synsets (a, b) such that a is a hypernym of b and a is an indirect transitive hypernym of b.

As shown in Figure 2 (b) the similar problem may occur with respect to meronymy if the creators of a wordnet embraced the rule that hyponyms inherit meronyms from their hypernyms. Thus, if *{ car:1:n ... }* is a hyponym of *{ motor vehicle:1:n ... }* which has a meronym *{ speedometer:1:n }*, then the concept represented by *{ car:1:n ... }* also posses the speedometer. Hence, the meronymy link between *{ car:1:n ... }* and *{ speedometer:1:n }* is redundant. The query below finds such tuples.

Query 13: *(Redundant meronyms query)* Find pairs of synsets (a, b) such that a is a meronym of b and a is a meronym of a transitive hypernym of b.

3.3 Aggregation

Publications that describe wordnets usually provide some quantitative information (e.g. Devitt and Vogel, 2003; Grigoriadou et al., 2004; Koeva et al., 2004a; Vetulani et al., 2010). We identified over twenty different quantities that are used to characterise wordnets in the literature and formulated queries that compute them. For example, the following queries should be answered in order to calculate some basic quantities that are given in the literature for BulNet (Koeva et al., 2004a) and PolNet (Vetulani et al., 2010).

Query 14: How many synsets/word senses/words there are in the wordnet?

Query 15: How many tuples do the hypernymy relation have?

Query 16: How many senses do words have on average?

Query 17: How many synsets do not have hypernyms?

3.4 Similarity Computation

Computation of a semantic distance among words and concepts is one of the basic applications of lexical databases. WordNet-based semantic similarity measures are exploited in such natural language processing tasks like word sense disambiguation (Patwardhan et al., 2003). The path-based similarity measures such as the Leacock and Chodorow (1998) measure[d] involve computation of paths that connect synsets containing particular words. Hence, our corpus includes the following query

Query 18: *(Reachability query)* Find all paths that connect the synset α to the synset β through the semantic relation r.

For example, if one uses Query 18 to search for paths that connect *{ minicab:1:n ... }* to *{ car:1:n ... }* through hypernymy, the result will contain the following paths

```
{ minicab:1:n } hypernym { cab:3:n ... } hypernym { car:1:n ... }
{ minicab:1:n } hypernym { minicar:1:n } hypernym { car:1:n ... }
```

Beside paths connecting synsets through semantic relations, similarity measures exploit also the notion of the least common subsumer (e.g. Patwardhan et al. (2003) – the Resnik, Jiang-Conrath and Lin measures) which is defined as a common hypernym γ of two synsets α and β such that no other hypernym of α and β is placed lower in the hypernymy hierarchy than γ. Thus, the following query has been added to the corpus.

Query 19: Find the least common subsumers of the synsets α and β.

For instance, the synset *{ car:1:n ... }* will be indicated by Query 19 as the least common subsumer of the synsets *{ bus:4:n ... }* and *{ minicab:1:n ... }* shown in Figure 2.

[d] Defined by the formula $sim_{ab} = max[-log(Np/2D)]$, where Np means the number of nodes of a path p between words a and b, and D means the depth of hyponymy.

4 Data Model

According to the description of a WordNet-like lexical database presented in Section 2, a wordnet query language has to operate on such data types as synsets, word senses, words, glosses, POS symbols and sense numbers. Particular wordnets also include data that are floating-point numbers, booleans, integer numbers and arbitrary character strings (cf. Section 2). Thus, all the data types mentioned above are taken into account in the WQuery wordnet model. Furthermore, sense numbers are treated as integers and words, glosses and POS symbols as members of the generic character string data type.[e] One may notice that the relationships which define the structure of a wordnet may be modeled as a set of binary relations that link word senses to their synsets, words, sense numbers and part of speech symbols. Thus, the WQuery wordnet model can be interpreted as a special case of the relational model (Abiteboul et al., 1995, ch. 3) that requires inclusion of particular relations.[f] For the purpose of this interpretation we assume that every relation name R has associated a list $attr(R)$ of names denoting consecutive arguments of R. Under this assumption we define domain, schema and instance of the WQuery wordnet model as follows

Definition 4.1: A domain of the WQuery wordnet model consists of synsets, word senses, character strings, boolean values, integers and real numbers.

Definition 4.2: A schema of the WQuery wordnet model is a finite set S of relation names that satisfies the following conditions:

1. S contains at least the following relation names: $synsets$, $wordsenses$, $words$, $possyms$, $synset$, $word$, $sensenum$, pos and $senses$.
2. $attr(synsets) = attr(wordsenses) = attr(words) = attr(possyms) = [src]$
3. $attr(synset) = attr(word) = attr(sensenum) = attr(pos) = attr(senses) = [src, dst]$

Definition 4.3: An instance of the WQuery wordnet model with domain D and schema S is a mapping I that assigns to every relation name $R \in S$ a finite relation $I(R) \subset D^{|attr(R)|}$ and satisfies the following conditions:

1. $I(synsets)$, $I(wordsenses)$, $I(words)$, $I(possyms)$ are sets (unary relations) of synsets, word senses, words and POS symbols respectively
2. $I(synset)(x, y) \iff y$ is a synset containing word sense x
3. $I(word)(x, y) \iff y$ is a word of word sense x
4. $I(sensenum)(x, y) \iff y$ is a sense number of word sense x
5. $I(pos)(x, y) \iff y$ is a POS symbol of word sense x
6. $I(senses)(x, y) \iff x$ is a word, synset, POS symbol or sense number of word sense y

[e] We call all the aforementioned data types together *WQuery data types* in the rest of the paper.

[f] This interpretation is convenient for the purpose of constructing formal semantics of the WQuery language presented in Appendix A.

We assume that positions of relation arguments correspond to the positions of their names on the $attr$ list. Thus, if $t \in I(pos)$, then we will refer to the first element of t either by $t(1)$ or $t(src)$.

As is evident from the definition above, the WQuery wordnet model permits inclusion of arbitrary relations defined among values of WQuery data types. Therefore, semantic relations can be represented in the model as binary relations that link synsets to synsets, lexical relations as binary relations that link word senes to word senses. Extensions specific to particular existing wordnets can be modeled using the same approach. For instance, usage examples may become the *usage* relation that links synsets to character strings and sense occurrence counts may be stored as a relation *tag_count* that links word senses to the appropriate integer numbers.

5 Syntax and Semantics

5.1 Path Expressions

The syntax and semantics of the WQuery language are focused on searching for paths in instances of the data model defined in the previous section. Paths are represented at the syntactic level by *path expressions*, and at the semantic level by tuples that consist of values of data types defined in Section 4, interleaved with the names of the relations that connect the consecutive values. A path expression consists of one or more *steps*, such that the first one (*generator*) describes a set of tuples and the following ones (*transformations*) extend the tuples defined by the preceding expressions by attaching edge names and nodes to them. For example, the expression

```
{car}.meronym
```

consists of a generator `{car}` which represents the set of synsets that contain the word *car*, the transformation `.meronym` which extends one-element paths generated by `{car}` with edges reachable through the `meronym` relation. The result of the sample path expression contains the following tuples

```
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
 ↪ meronym { air bag:1:n }
{ car:1:n auto:1:n automobile:1:n machine:6:n motorcar:1:n }
 ↪ meronym { reverse:2:n reverse gear:1:n }
```

The path expression above and the queries in the rest of the paper are evaluated against WordNet 3.0 if not stated otherwise. The ↪ symbol in the example represents line continuation.

Transformations may involve conventional regular operators, such as `*` (zero or more), + (one or more), `|` (alternative) and parentheses. For instance, the following query

```
{car:1:n}.hypernym+
```

returns paths that connect the *{ car:1:n ... }* synset to its transitive hypernyms

```
{ car:1:n ... } hypernym { motor vehicle:1:n ... }
{ car:1:n ... } hypernym { motor vehicle:1:n ... }
 ↪ hypernym { self-propelled vehicle:1:n }
```

Table 1 Sample Generators

Generator	Value
`{car:2:n}`	The synset that contains the second noun word sense of the word *car* if the word sense *car:2:n* exists, otherwise an empty set
`{car}`	Synsets that contain the word *car*
`{}`	All synsets stored in the wordnet
`car:2:n`	The second noun word sense of the word *car* or an empty set if the word *car* does not have the second sense as a noun
`::`	All word senses stored in the wordnet
`'car'`	The string *car* if the word exists in the wordnet, otherwise an empty set
`car`	Same as above
`` `car` ``	The string *car*
`"`	All words stored in the wordnet
`123`	The integer *123*
`1.0`	The float *1.0*
`true`	The boolean value *true*
`!hypernym`	Tuples of the `hypernym` relation
`(q)`	Tuples found by the query `q`

```
{ car:1:n ... } hypernym { motor vehicle:1:n ... }
 ↪ hypernym { self-propelled vehicle:1:n }
 ↪ hypernym { wheeled vehicle:1:n }
...
```

One may also use the ^ operator to traverse relations in the opposite direction.[g] For example, to search for hyponyms instead of hypernyms one may prepend the ^ symbol to the `hypernym` relation in the query above. Thus, the result of the expression

```
{car:1:n}.^hypernym+
```

will contain paths that connect the *{ car:1:n ... }* synset to its transitive hyponyms, e.g.

```
{ car:1:n ... } ^hypernym { bus:4:n jalopy:1:n heap:3:n }
{ car:1:n ... } ^hypernym { cab:3:n ... }
{ car:1:n ... } ^hypernym { cab:3:n ... } ^hypernym { minicab:1:n }
```

The arguments of relations may be referenced by their names. For instance, to traverse the relation `hypernym` from the `src` argument to the `author` argument (cf. Table 3) one may formulate the following query

```
{car:1:n}.src^hypernym^author
```

The underscore symbol can be used instead of a relation name to traverse any eligible relation (cf. Appendix A.3). Therefore, to find all direct connections of the *{ car:1:n ... }* synset one may issue the query

```
{car:1:n}._
```

[g] By the opposite direction we mean the traversal from the argument named dst to the argument denoted by src.

5.2 Variable Bindings

In order to select data from paths, one may augment a path expression with variables placed after a step. For instance, the path expression

```
{car}$a.hypernym+@P$b
```

will bind the variable `$b` to the last transitive hypernym on the path, the path variable `@P` to all elements of the path generated by the transformation `.hypernym+` that precede `$b` and the variable `$a` to a node generated by `{car}`. Hence, in the case of the path

```
{ car:1:n ... } hypernym { motor vehicle:1:n ... }
 ↪ hypernym { self-propelled vehicle:1:n }
```

the variables have the following values

```
@P = hypernym { motor vehicle:1:n ... } hypernym
$a = { car:1:n ... }
$b = { self-propelled vehicle:1:n }
```

Values bound to variables may be referenced in *filters* and *projections* – two constructs that can be placed after a step of a path expression in order to modify the result. A filter consists of a logical condition enclosed in square brackets. If the condition inside the filter holds for a path, then the path is added to the resulting multiset, otherwise it is eliminated. For example, the query

```
{car}.hypernym+$a[$a = {vehicle:1:n}]
```

finds paths that connect synsets that contain the word *car* to the synset that contains the word sense *vehicle:1:n* through the `hypernym` relation, e.g.

```
{ car:1:n ... } hypernym { motor vehicle:1:n ... }
 ↪ hypernym { self-propelled vehicle:1:n }
 ↪ hypernym { wheeled vehicle:1:n } hypernym { vehicle:1:n }
```

Filters that compare a value bound to the last step of the path with a value of a particular generator may also be abbreviated by placing the generator after the dot instead of the bracketed expression. Thus, the query above may be reformulated as

```
{car}.hypernym+.{vehicle:1:n}
```

A projection consists of a path expression enclosed in < and >. The result of the evaluation of the enclosed expression replaces the path for which the projection is evaluated. For example, the query

```
{car}$a.hypernym.meronym$b<$a,$b>
```

finds all pairs that consist of a synset that contains the word *car* and a meronym of its hypernym

```
{ car:1:n ... } { power brake:1:n }
{ car:1:n ... } { speedometer:1:n speed indicator:1:n }
{ car:1:n ... } { suspension:5:n suspension system:1:n }
...
```

5.3 Functions

WQuery provides a set of built-in functions which take as arguments and return as values arbitrary lists of tuples and their variable bindings. Among them there are conventional aggregate operators, such as `count`, `max` and `avg`, mathematical functions, such as `log` and `exp`, and special path-oriented operators, such as `shortest` and `longest`, which return the shortest and longest tuple of a multiset, respectively. What differs WQuery functions from their counterparts in other query languages is that the functions that do not modify, remove or replace elements of their arguments preserve variable bindings. Thus, the query

```
longest({car:5:n}.hypernym+$a)
```

not only returns the path

```
{ car:5:n ... } hypernym { compartment:2:n } hypernym { room:1:n }
 ↪ hypernym { area:5:n } hypernym { structure:1:n ... }
 ↪ hypernym { artifact:1:n ... } hypernym { whole:2:n unit:6:n }
 ↪ hypernym { object:1:n ... } hypernym { physical entity:1:n }
 ↪ hypernym { entity:1:n }
```

which is the longest path from *{ car:5:n ... }* to the top of the hypernymy hierarchy, but also binds the *{ entity:1:n }* synset to the variable `$a`.

5.4 Other Constructs

In order to make the language robust we have introduced additional constructs. Since we noticed multiple attempts to use relational databases to store and query wordnets (e.g. Princeton University, 2011b; Yablonsky and Sukhonogov, 2005; Rouhizadeh et al., 2010), we assumed that the minimal usable wordnet query language should express at least the queries that are representable in relational algebra. Thus, we extended the language with multipath operators that are counterparts of relational algebra operators. The `union`, `intersect`, `except` and cross product (`,`) operators treat paths as tuples and (as shown in Section 7) make it possible to translate queries formulated in relationally complete languages to WQuery. We also added conventional arithmetic operators to make it easier to compute similarity measures mentioned in Section 3 directly in WQuery and introduced imperative constructs such as `if-else` statements, `while` loops and sequential execution blocks.

6 Wordnet-specific Queries in WQuery

We will now show how the wordnet-specific queries described in Section 3 may be expressed in WQuery.

6.1 Navigation

Results of Queries 1-7 can be determined by the following WQuery expressions

```
car.senses
{car}
```

```
{car}.gloss
{car:1:n}.example
{car}.meronyn
good:1:a.antonym
good.senses.antonym
```

Let `r` be the name of a semantic relation r. Let `g` be a generator that represents the set of synsets S. The tree query of S under r (Query 8) is defined by the expression

```
g.r+
```

Note that `g` always exists since we can generate particular synsets from S using the `{word:num:pos}` generators, join these generators with the `union` operator and enclose the resulting expression with `()`. In some cases one broader generator such as `{}` or `{word}` may be used instead of joining several simpler generators with `union`. For instance, to search for paths that connect synsets containing word senses of the *artifact* word to their transitive hypernyms one may formulate the expression

```
{artifact}.hypernym+
```

6.2 Validation

Synsets that participate in cycles in the semantic relation `r` (Query 9) can be detected by the query

```
{}$a[$a in $a.r+]
```

Assuming that Bulgarian and English synsets are connected through the `ili` relation, the `lang` relation assigns language codes to synsets and the `bcs` relation holds for synsets that are base concepts (cf. wordnet structure defined in Koeva et al., 2004b) Query 10 and 11 can be expressed in WQuery as follows

```
{}[lang = 'en' and bcs and empty(ili[lang = 'bg'])].id
{}[lang = 'en' and bcs and not ili[lang = 'bg'].bcs].id
```

In order to find redundant hypernyms (Query 12) one may issue the following WQuery query

```
{}$b.hypernym$a[$a in $b.hypernym.hypernym+]<$a, $b>
```

Redundant meronyms (Query 13) are located by the query

```
{}$b.meronym$a[$a in $b.hypernym+.meronym]<$a, $b>
```

Note that except for the relation names we do not assume existence of any particular data in the wordnet. Hence, both Query 12 and 13 may be executed against any wordnet that contains the hypernymy and meronymy relations named as above.

6.3 Aggregation

The numbers of synsets, word senses and words (Query 14) are given by the following queries

```
count({})
count(::)
count('')
```

The number of hypernymy tuples (Query 15) is given by

```
count({}.hypernym)
```

The number of senses per word (Query 16) is calculated by

```
count(::)/count('')
```

The number of synsets without hypernyms (Query 17) is determined by the query

```
count({}[count(hypernym) = 0])
```

6.4 Similarity Computation

Let `g` and `h` be generators for synsets α and β, respectively. The reachability query from α to β with respect to the relation `r` (Query 18) is fulfilled by the expression

```
g.r*.h
```

For example, to find paths that connect the synset that contains the word sense *minicab:1:n* with the synset that contains the word sense *car:1:n* through hypernymy one may formulate the expression

```
{minicab:1:n}.hypernym*.{car:1:n}
```

Let `g` and `h` be generators for synsets α and β, respectively. The least common subsumers of α and β (Query 19) are determined by the query

```
longest((g.hypernym*$a<$a>
    intersect h.hypernym*$b<$b>)$s.hypernym*)<$s>
```

For instance, the least common subsumers of the synsets that contain the senses *minicab:1:n* and *bus:4:n* are computed by the query

```
longest(({minicab:1:n}.hypernym*$a<$a>
    intersect {bus:4:n}.hypernym*$b<$b>)$s.hypernym*)<$s>
```

7 Remarks on Expressive Power

The data model presented in Section 4 is a special case of the relational model which restricts the domain of the database to synsets, word senses, strings, booleans and numbers and imposes existence of wordnet-specific relations. Hence, we may compare the expressive power of the languages considered in theory of relational databases and the WQuery language with respect to the databases that are valid instances of the data model described in Section 4. As mentioned in Section 5.4 WQuery provides direct counterparts for four operators of relational algebra – the union, intersection, difference and cross product. We will now show that these operators together with the constructs described in Sections 5.1-5.3 express all queries that are formulable in relational algebra over instances of the WQuery wordnet model. We will consider the so called unnamed perspective of relational algebra described in (Abiteboul et al., 1995, p. 54) which references the relation arguments by their positions. Let D be the domain of a WQuery wordnet model. WQuery provides a generator for every element of the domain (cf. Appendix A.2). Let γ be a mapping that assigns generators to the elements of the domain. The mapping T that translates queries formulated in relational algebra to the WQuery expressions is defined inductively with the following rules:

1. $T(\{(d)\}) = \gamma(d)$, where $d \in D$ and $\{(d)\}$ is a query that returns a single unary tuple that contains d
2. $T(R) = !\,R$, where R is a name of a relation.
3. $T(\pi_{i_1,\ldots,i_k}(q)) =$ `(`$T(q)$`) $1`$\cdots$`$n <$i_1`,...,`$i_k`>`
   where q is a query with arity equal to n and $i_1, \ldots, i_k \leq n$ and $\pi_{i_1,\ldots,i_k}(q)$ stands for the projection of the result of q to its $i_1, \ldots, i_k$-th arguments.
4. $T(\sigma_{i=d}(q)) =$ `(`$T(q)$`) $1`$\cdots$`$n [$i = `$\gamma(d)$`]<$1,...,`$n>`,
   where q is a query with arity equal to n and $i \leq n$, $d \in D$ and $\sigma_{i=d}(q)$ stands for the selection of tuples from the result of q which have their i-th argument equal to d.
5. $T(\sigma_{i=j}(q)) =$ `(`$T(q)$`) $1`$\cdots$`$n [$i = `$j`]<$1,...,`$n>`,
   where q is a query with arity equal to n and $i, j \leq n$ and $\sigma_{i=j}(q)$ stands for the selection of tuples from the result of q which have their i-th argument equal to the j-th argument.
6. $T(q \times r) = T(q)$`,`$T(r)$, where q and r are queries.
7. $T(q \cup r) =$ `(`$T(q)$ `union` $T(r)$`)`, where q and r are queries.
8. $T(q - r) =$ `(`$T(q)$ `except` $T(r)$`)`, where q and r are queries.

One can verify by induction that regardless of an instance of the WQuery wordnet data model queries q and $T(q)$ return the same result.

In Section 5.4 we also mentioned that WQuery has imperative constructs such as conditional statements, loops and sequential execution blocks. These constructs make it possible to prove that the WQuery language can express any query computable over an instance of the WQuery wordnet model. The proof can be derived by constructing a mapping from the QL language (Chandra and Harel, 1980) to WQuery. We do not present it in the paper due to its length. It can be found in (Kubis, 2013, pp. 126–133).

8 Related Work

8.1 Wordnet-specific tools and models

A tool called Hydra that provides a modal logic based language for querying wordnets has been described in (Rizov, 2008). The Hydra language may be translated to relational calculus by adjusting the standard translation from the modal language to the first-order language defined in (Blackburn and van Benthem, 2007). Therefore, one cannot express in Hydra Query 8 and 18 because these queries involve computation of the transitive closures of relations, which are not computable in relational calculus. The *WN* language (Koeva et al., 2004b) is another interesting modal logic based formalism designed to query WordNet-like databases. *WN* represents the transitive closure of hypernymy as a separate relation symbol. However, it does not provide operators that would allow to compute the entire paths that transitively connect synsets through arbitrary chosen semantic relations. Hence, one cannot formulate in *WN* Query 18. Neither *WN* nor Hydra encompass aggregate functions and arithmetic operators. Therefore, they are unable to answer aggregate queries described in Section 3.3. DEBVisDic (Horak et al., 2005) – a tool for browsing and editing wordnets

Table 2 Mappings of attributes of LMF elements to mandatory relations of the WQuery model

relation name	argument name	XPath expression over LMF
words	src	Lemma/@writtenForm
wordsenses	src	Sense/@id
possyms	src	Lemma/@partOfSpeech
synsets	src	Synset/@id
synset	src	Sense/@id
	dst	Sense/synset
word	src	LexicalEntry/Sense/@id
	dst	LexicalEntry/Lemma/@writtenForm
sensenum	src	Sense/@id
	dst	position of Sense element within LexicalEntry
pos	src	LexicalEntry/Sense/@id
	dst	LexicalEntry/Lemma/@partOfSpeech

– provides an option to search for synsets by specifying the word form, word sense and several other properties that conform to the underlying XML representation of a wordnet. These search criteria may be combined together by using logic operators. Such queries may be expressed in WQuery using the synset generator followed by a filter with a suitable condition (Section 5.2). Queries 12, 13, 18 and 19, are not directly expressible using the search form of DEBVisDic.

The WQuery wordnet model with its uniform treatment of wordnet data as relations among domain-specific data types is sufficiently general to encompass wordnets stored using LMF-based formats such as Wordnet-LMF (Soria et al., 2009) and its extensions (Henrich and Hinrichs, 2010). The mandatory relations of the WQuery wordnet model can be populated from XML elements and attributes of a Wordnet-LMF file in accordance with Table 2. The elements and attributes that are not employed to populate mandatory relations can be introduced into an instance of the WQuery wordnet model by including additional relations. Example mappings from Wordnet-LMF elements to WQuery relations are presented in the first part of Table 3. The second part of Table 3 presents mappings for extensions postulated in (Henrich and Hinrichs, 2010). Generally, we add new relations for all attributes and subelements of *Synset* and *Sense* elements (e.g. *baseConcept*, *Definition*, *SenseExample*). We also create new relations for distinct values of *relType* attribute of *SynsetRelation* and *SenseRelation* elements and for distinct values of *externalReference* attribute of *MonolingualExternalRef*. *Meta* attributes of *Synset*, *Sense* and *LexicalEntry* elements are mapped to new relations (cf. *author* relation in Table 3). *Meta* attributes of LMF elements that are not mapped to mandatory relations of the WQuery model (e.g. *SynsetRelation*, *SenseRelation*, *SubcategorizationFrame*) are included as additional arguments of their relations (cf. *author* attribute of *hyponym* relation and *date* attribute of *antonym*).

8.2 General purpose tools

As for general purpose tools adapted to query wordnets, relational databases are reported to have been used in several projects (e.g. Princeton University, 2011b; Yablonsky and Sukhonogov, 2005; Rouhizadeh et al., 2010). Since Query 8 and 18 involve computation of

Table 3 Example mappings from attributes of LMF elements to WQuery relations

relation name	argument name	XPath expression over LMF
bcs	src	Synset/@id
	dst	Synset/@baseConcept
gloss	src	Synset/@id
	dst	Synset/Definition/gloss
hyponym	src	Synset/@id
	dst	Synset/SynsetRelations/SynsetRelation [@relType = "has_hyponym"]/@target
	author	Synset/SynsetRelations/SynsetRelation [@relType = "has_hyponym"]/Meta/@author
mon_ref_x	src	Synset/@id
	dst	Synset/*/MonolingualExternalRef/[@relType = "x"]/ @externalReference
	system	Synset/*/MonolingualExternalRef/[@relType = "x"]/ @externalSystem
author	src	LexicalEntry/Lemma/@writtenForm or Synset/@id or Sense/@id
	dst	LexicalEntry/Meta/@author or Synset/Meta/@author or Sense/Meta/@author
antonym	src	Sense/@id
	dst	Sense/*/SenseRelation[@relType = "antonym"]/@target
	date	Sense/*/SenseRelation[@relType = "antonym"]/Meta/@date
frames	src	Sense/@id
	dst	Sense/*/SubcategorizationFrame/@frame
examples	src	Sense/@id
	dst	Sense/SenseExamples/SenseExample/@text
	frame	Sense/SenseExamples/SenseExample/@frame

transitive closures, they cannot be formulated in relationally complete languages. Besides the use of relational databases, there have been attempts to store wordnets in RDF (Graves and Gutierrez, 2005; Gangemi et al., 2006) files. Thus, one may consider querying a wordnet using the RDF query language SPARQL (Prud'hommeaux and Seaborne, 2008; Harris and Seaborne, 2013). In the RDF representations of WordNet semantic relations among synsets become RDF properties that connect URIs representing synsets. The first version of the SPARQL language specification (Prud'hommeaux and Seaborne, 2008) provided only constructs for traversing paths of fixed length in a RDF graph. Therefore, as in the case of relational databases, Query 8 and 18 cannot be computed in SPARQL 1.0. The revised specification of SPARQL contains the property path construct (Harris and Seaborne, 2013, chapter 9) that can be used to traverse property paths of arbitrary length, thus one can search for transitive hypernyms of synsets. Nevertheless, the result set of a SPARQL query is still by definition a mapping from variables specified in query to RDF terms (cf. Harris and Seaborne, 2013, Sec. 18.1.8), hence nodes on the path in the hypernymy tree that are not individually bound to variables within the query have no direct representation in the result set. In contrast, the WQuery result set is a list of pairs that consist of both a set of variable bindings and a tuple of an arbitrary length, thus hypernymy paths can be included in the result regardless of the variables declared in the query.

XQuery (Boag et al., 2010), a Turing-complete XML query language (Kepser, 2004), can be used to formulate any queries computable over XML documents. Thus, one may consider using it instead of WQuery to query wordnets stored in XML documents conforming to wordnet representation schemes such as Wordnet-LMF discussed in Section 8.1. However, in contrast to WQuery which provides wordnet-specific data types, wordnet data are interpreted in XQuery as a hierarchical structure built from XML elements. This generic, hierarchical representation of wordnet data is less convenient than wordnet-specific, graph-based interpretation of wordnet structure provided by WQuery when we consider queries that traverse semantic relations. For instance, the query that searches for hypernyms of synsets containing the word *auto* can be formulated in WQuery as

```
{auto}.hypernym
```

The same data can be retrieved by the following XQuery query

```
let $sid := doc("wordnet.xml")//LexicalEntry
    [Lemma/@writtenForm = 'auto']/Sense/@synset
let $synsets := doc("wordnet.xml")//Synset[@id = $sid]
return doc("wordnet.xml")//Synset
    [@id = $synsets//SynsetRelation[@relType = 'hypernym']/@target]
```

The XQuery query is much longer since it has to incorporate XML-specific details of the document structure defined by Wordnet-LMF that are not related to the structure of a wordnet as defined in Section 2. In order to formulate the query above, one has to take into account that links of semantic relations are defined by `SynsetRelation` elements enclosed in the `Synset` elements representing synsets which are sources for the links. It is also necessary to take into consideration that the target of the link and the name of the semantic relation are indicated by XML attributes,[h] hence they have to be prefixed with `@` signs. The impedance mismatch between the wordnet data and its hierarchical representation within a Wordnet-LMF document becomes even more visible when we consider a query that determines transitive hypernyms of *auto* synsets. Both WQuery and XQuery can traverse

[h] `target` and `relType` respectively.

the underlying data models with path expressions that involve regular operators. In the case of WQuery, the transitive hypernyms can be determined by the query

```
{auto}.hypernym*
```

which involves the `*` operator. The `//` operator of XQuery cannot be applied to find the transitive hypernyms in a Wordnet-LMF document in a similar manner because by its definition `//` retrieves XML elements that are below the given element in the XML document hierarchy and all the links that constitute the hypernymy hierarchy in a Wordnet-LMF conformant document are at the same level. Therefore, to traverse a semantic relation using an XQuery query over a Wordnet-LMF model one has to implement a custom function. By adapting the transitive closure function presented in (Boncz et al., 2007, p. 11), we attained the following solution

```
declare namespace wf = 'wordnet.functions';
declare function wf:tc($seed as element()*, $visited as element()*,
  $relation as xs:string) as element()* {
  let $fringe := (doc("wordnet.xml")//Synset
    [@id = $seed//SynsetRelation[@relType = $relation]/@target])
    except $visited
  return  if (exists($fringe))
    then ($fringe,wf:tc($fringe,($visited,$fringe),$relation))
    else ()
};

let $sid := doc("wordnet.xml")//LexicalEntry
  [Lemma/@writtenForm = 'auto']/Sense/@synset
let $synset :=  doc("wordnet.xml")//Synset[@id=$sid]
return wf:tc($synset,$synset,'hypernym')
```

which is far more verbose than the WQuery expression presented above.

9 Conclusion

We collected a corpus of wordnet-specific queries that arise in various use-cases ranging from navigation through the data and validation of the wordnet content to computation of the semantic distance between words. We found that none of the existing wordnet-specific tools is capable of expressing all of the analyzed queries. Therefore, we proposed a new solution to the problem of querying WordNet-like lexical databases. The solution is based on the data model that incorporates wordnet-specific data types and a query language that utilizes the concept of a path. We have shown that by using the WQuery language one can formulate all wordnet-specific queries described in Section 3. We also shown that the wordnet-specific data model of WQuery leads to queries that are far more concise than their counterparts formulated in a general purpose tool used to query XML documents that conform to the LMF-based representation standards of wordnets. We have also investigated the expressive power of the presented language. In the future we would like to introduce optimization techniques for query evaluation based on the structure of a WordNet-like database.

A Formal Semantics

We describe in this section the formal semantics of the fragment of the WQuery language presented in Section 5. The interpretation is given within $[\![$ and $]\!]$ brackets placed after every alternative expansion of the grammar rules which are specified using the BNF notation. The interpretation is not specified for a rule if it is the same as the interpretation of its expansion. The interpretation of an expression $expr$ over an instance I of WQuery wordnet model with domain D and schema S (cf. Definition 4.3) and a set of variable bindings[i] B is denoted by $[\![expr]\!]^{I,B}$. Initially the set of variable bindings is empty, hence the result of evaluating the WQuery query q over an instance of the WQuery wordnet model I is given by the formula $[\![q]\!]^{I,\emptyset}$. As can be deduced from the rules presented below the result of a query (i.e. the expression $expr$) is a list of pairs that consist of a list of domain elements and a set of variable bindings. We call such a structure a *dataset*. This data structure is also used to represent the semantics of the subexpressions defined in Sections A.2, A.3 and A.6. The $\oplus$ and $\subset$ operators are used in the interpretation rules given below to denote concatenation and inclusion testing for lists. The $\in$ sign is either list or set inclusion depending on the context. We use $[\varphi(x_1,\ldots,x_n)|\ \psi(x_1,\ldots,x_n)]$ notation to formulate list comprehensions. We also use the $L(\cdot)$ operator to denote a dataset that consists of the last elements of the lists belonging to its argument, i.e. $L(D) = [([l],\emptyset)\mid (t,v)\in D, t = f\oplus[l]]$.

A.1 Literals

$\langle bqstring\rangle$ ::= character string s encompassed in back quotes $[\![s]\!]^{I,B}$

$\langle qstring\rangle$ ::= character string s encompassed in single quotes $[\![s]\!]^{I,B}$

$\langle nqstring\rangle$ ::= character string s without spaces beginning with a letter $[\![s]\!]^{I,B}$

$\langle word\rangle$ ::= $\langle qstring\rangle$ | $\langle nqstring\rangle$

$\langle alpha\rangle$ ::= $\langle bqstring\rangle$ | $\langle word\rangle$

$\langle float\rangle$ ::= fixed point constant denoting number f $[\![f]\!]^{I,B}$

$\langle integer\rangle$ ::= integer constant denoting number i $[\![i]\!]^{I,B}$

$\langle boolean\rangle$ ::= 'true' $[\![true]\!]^{I,B}$ | 'false' $[\![false]\!]^{I,B}$

$\langle const\rangle$ ::= $\langle bqstring\rangle$ | $\langle float\rangle$ | $\langle integer\rangle$ | $\langle boolean\rangle$

A.2 Generators

$\langle const\_gen\rangle$ ::= $\langle const\rangle$ $[\![[([\,[\![const]\!]^{I,B}],\emptyset)]]\!]^{I,B}$

$\langle sense\_gen\rangle$ ::= '::' $[\![[([m],\emptyset)|\ m\in I(wordsenses)]]\!]^{I,B}$

| $\langle alpha\rangle$ ':' $\langle integer\rangle$ ':' $\langle alpha'\rangle$

$$\left[\!\!\left[\begin{array}{l}[([m],\emptyset)|\ m\in I(wordsenses)\wedge(m,\ [\![alpha]\!]^{I,B})\in I(word)\ \wedge\\ (m,\ [\![integer]\!]^{I,B})\in I(sensenum)\wedge(m,\ [\![alpha']\!]^{I,B})\in I(pos)]\end{array}\right]\!\!\right]^{I,B}$$

$\langle synset\_gen\rangle$ ::= '{}' $[\![[([s],\emptyset)|\ s\in I(synsets)]]\!]^{I,B}$

| '{' $\langle expr\rangle$ '}'

[i] The set of variable bindings is a set of pairs that consist of a variable name and its value.

$$\left[\!\!\left[\begin{array}{l}[([s],\emptyset)|\ (m,s) \in I(synset) \wedge m \in L(\ [\![expr]\!]^{I,B}) \vee \\ \ (m,w) \in I(word) \wedge (m,s) \in I(synset) \wedge w \in L(\ [\![expr]\!]^{I,B})]\end{array}\right]\!\!\right]^{I,B}$$

$\langle fun\_call\_gen \rangle$ ::= $\langle nqstring \rangle$ '(' $\langle expr \rangle$ ')'

$[\![f(\ [\![expr]\!]^{I,B})$, where f is a function denoted by $[\![nqstring]\!]^{I,B}]\!]^{I,B}$

$$\langle word\_gen \rangle ::= \langle word \rangle \left[\!\!\left[\begin{array}{ll}[([\ [\![word]\!]^{I,B}],\emptyset)], & \text{if } [\![word]\!]^{I,B} \in I(words) \\ [], & \text{otherwise}\end{array}\right]\!\!\right]^{I,B}$$

$\langle relation\_gen \rangle$ ::= '!' $\langle nqstring \rangle$ $[\![[(t,\emptyset)|\ t \in I(\ [\![nqstring]\!]^{I,B})]]\!]^{I,B}$

$\langle adom\_gen \rangle$ ::= '__' $[\![[([v],\emptyset)|\ \exists_{R \in S}(t \in R \wedge \exists_{1 \leq n \leq |t|} v = t(n))]]\!]^{I,B}$

$\langle par\_gen \rangle$ ::= '(' $\langle expr \rangle$ ')' $[\![expr]\!]^{I,B}$

$\langle var\_gen \rangle$ ::= $\langle step\_var \rangle$ $[\![[([v],\emptyset)|\ (\ [\![step\_var]\!]^{I,B}, v) \in B]]\!]^{I,B}$

| $\langle tuple\_var \rangle$ $[\![[(v,\emptyset)|\ (\ [\![tuple\_var]\!]^{I,B}, v) \in B]]\!]^{I,B}$

$\langle generator \rangle$ ::= $\langle const\_gen \rangle$ | $\langle synset\_gen \rangle$ | $\langle sense\_gen \rangle$ | $\langle fun\_call\_gen \rangle$

| $\langle par\_gen \rangle$ | $\langle var\_gen \rangle$ | $\langle adom\_gen \rangle$ | $\langle word\_gen \rangle$ | $\langle relation\_gen \rangle$

A.3 Relational Expressions

For the purpose of defining relational expressions, we introduce the following helper operations:

1. $L \bowtie R = [(l \oplus r, p_l \cup p_r)|\ (l,p_l) \in L, ([h] \oplus r, p_r) \in R, l(|l|) = h]$
2. $C(L) = [(p,v) \in L|\ p = l \oplus [x] \oplus r \oplus [x]]$

The $\bowtie$ and C take datasets as arguments. $L \bowtie R$ concatenates the lists and sums the bindings belonging to its arguments if and only if the tail of the list that belongs to its left hand side argument is equal to the head of the list belonging to its right hand side argument. This is a WQuery counterpart of a natural join restricted to the last argument of the left hand side relation and the first argument of the right hand side one. The C operator selects from its argument all tuples that end with an element that also occurs on one of the previous positions. It is used to remove cycles while computing values of relational expressions defined using the $\{\cdot,\cdot\}$ operator. Beside $\bowtie$ and C we also use the $\overline{n}$ notation in the rules below to denote a character string that consists of digits of a decimal representation of the integer n.

$\langle rexpr \rangle$ ::= $\langle nqstring \rangle$ $[\![[([t(src), r, t(dst)],\emptyset)|\ t \in I(r), [src,dst] \subset attr(r)]]\!]^{I,B}$

where $r = [\![nqstring]\!]^{I,B}$

| '_' $[\![[([t(src), r, t(dst)],\emptyset)|\ r \in S, t \in I(r), [src,dst] \subset attr(r)]]\!]^{I,B}$

| '^' $\langle nqstring \rangle$ $[\![[([t(dst), [\ \text{'^'}\ ] \oplus r, t(src)],\emptyset)|\ t \in I(r), [src,dst] \subset attr(r)]]\!]^{I,B}$

where $r = [\![nqstring]\!]^{I,B}$

| '^' '_' $[\![[([t(dst), [\ \text{'^'}\ ] \oplus r, t(src)],\emptyset)|\ r \in S, t \in I(r), [src,dst] \subset attr(r)]]\!]^{I,B}$

| $\langle nqstring \rangle$ '^' $\langle nqstring' \rangle$ '^' $\langle nqstring'' \rangle$

$[\![[([t(s), s \oplus [\ \text{'^'}\ ] \oplus r \oplus [\ \text{'^'}\ ] \oplus d, t(d)],\emptyset)|\ t \in I(r), [s,d] \subset attr(r)]]\!]^{I,B}$

where $s = [\![nqstring]\!]^{I,B}, r = [\![nqstring']\!]^{I,B}, d = [\![nqstring'']\!]^{I,B}$

| $\langle nqstring \rangle$ '^' '_' '^' $\langle nqstring' \rangle$

$[\![([t(s), s \oplus [\text{ '^' }] \oplus r \oplus [\text{ '^' }] \oplus d, t(d)], \emptyset) |\ r \in S, t \in I(r), [s, d] \subset attr(r)]\!]^{I,B}$

where $s = [\![nqstring]\!]^{I,B}, d = [\![nqstring']\!]^{I,B}$

| $\langle rexpr' \rangle$ 'I' $\langle rexpr'' \rangle$ $[\![rexpr']\!]^{I,B} \oplus [\![rexpr'']\!]^{I,B}$

| $\langle rexpr' \rangle$ '.' $\langle rexpr'' \rangle$ $[\![rexpr']\!]^{I,B} \bowtie [\![rexpr'']\!]^{I,B}$

| $\langle rexpr' \rangle$ '{' $\langle integer \rangle$ '}'

$$\left[\!\!\left[\begin{array}{ll} (\ [\![rexpr'\{\overline{n-1}\}]\!]^{I,B} - C(\ [\![rexpr'\{\overline{n-1}\}]\!]^{I,B})) \bowtie [\![rexpr']\!]^{I,B} & \text{if } n > 0 \\ [\![adom\_gen]\!]^{I,B} & \text{if } n = 0 \end{array} \right]\!\!\right]^{I,B}$$

where $n = [\![integer]\!]^{I,B}$

| $\langle rexpr' \rangle$ '{' $\langle integer \rangle$ ',' $\langle integer' \rangle$ '}'

$[\![rexpr'\{\overline{m}\}]\!]^{I,B} \oplus [\![rexpr'\{\overline{m+1}\}]\!]^{I,B} \oplus \cdots \oplus [\![rexpr'\{\overline{n}\}]\!]^{I,B}$

where $m = [\![integer]\!]^{I,B}, n = [\![integer']\!]^{I,B}$

| $\langle rexpr' \rangle$ '{' $\langle integer \rangle$ ',' '}'

$[\![rexpr'\{\overline{m}, \overline{n}\}]\!]^{I,B}$

where $n =$ smallest natural number such that $[\![rexpr'\{\overline{n}\}]\!]^{I,B} - C(\ [\![rexpr'\{\overline{n}\}]\!]^{I,B}) = []$

$m = [\![integer]\!]^{I,B}$

| $\langle rexpr' \rangle$ '*' $[\![rexpr'\{0,\}]\!]^{I,B}$

| $\langle rexpr' \rangle$ '+' $[\![rexpr'\{1,\}]\!]^{I,B}$

A.4 Conditions

$\langle condition \rangle$ ::= $\langle expr' \rangle$ 'in' $\langle expr'' \rangle$

$$\left[\!\!\left[\begin{array}{ll} true, & \text{if } L(\ [\![expr']\!]^{I,B}) \subset L(\ [\![expr'']\!]^{I,B}) \\ false, & \text{otherwise} \end{array} \right]\!\!\right]^{I,B}$$

| $\langle expr' \rangle$ '=' $\langle expr'' \rangle$

$$\left[\!\!\left[\begin{array}{ll} true, & \text{if } [\![expr' \text{ 'in' } expr'']\!]^{I,B} = true \text{ and } [\![expr'' \text{ 'in' } expr']\!]^{I,B} = true \\ false, & \text{otherwise} \end{array} \right]\!\!\right]^{I,B}$$

| 'not' $\langle condition' \rangle$

$$\left[\!\!\left[\begin{array}{ll} true, & \text{if } [\![condition']\!]^{I,B} = false \\ false, & \text{otherwise} \end{array} \right]\!\!\right]^{I,B}$$

| $\langle condition' \rangle$ 'or' $\langle condition'' \rangle$

$$\left[\!\!\left[\begin{array}{ll} true, & \text{if } [\![condition']\!]^{I,B} = true \text{ or } [\![condition'']\!]^{I,B} = true \\ false, & \text{otherwise} \end{array} \right]\!\!\right]^{I,B}$$

| $\langle condition' \rangle$ 'and' $\langle condition'' \rangle$

$$\left[\!\!\left[\begin{array}{ll} true, & \text{if } [\![condition']\!]^{I,B} = true \text{ and } [\![condition'']\!]^{I,B} = true \\ false, & \text{otherwise} \end{array} \right]\!\!\right]^{I,B}$$

A.5 Variable Bindings

$$\begin{array}{l} \langle step\_var \rangle \ ::= \ \text{'\$'} \ \langle nqstring \rangle \ [\![[\text{'\$'}] \oplus [\![nqstring]\!]^{I,B}]\!]^{I,B} \\ \langle tuple\_var \rangle \ ::= \ \text{'@'} \ \langle nqstring \rangle \ [\![[\text{'@'}] \oplus [\![nqstring]\!]^{I,B}]\!]^{I,B} \\ \langle step\_vars \rangle \ ::= \ \langle step\_var \rangle \ [\![step\_var]\!]^{I,B} \\ \ \ | \ \ \langle step\_vars' \rangle \ \langle step\_var \rangle \ [\![[\ [\![step\_var]\!]^{I,B}] \oplus [\![step\_vars']\!]^{I,B}]\!]^{I,B} \\ \langle vars \rangle \ ::= \ \langle step\_vars \rangle \ [\![step\_vars]\!]^{I,B} \\ \ \ | \ \ \langle step\_vars \rangle \ \langle tuple\_var \rangle \ \langle step\_vars' \rangle \\ \qquad [\![\ [\![step\_vars]\!]^{I,B} \oplus [\ [\![tuple\_var]\!]^{I,B}] \oplus [\![step\_vars']\!]^{I,B}]\!]^{I,B} \end{array}$$

A.6 Path Expressions

In order to specify the semantics of path expressions, we define the $A(D, v)$ operator that assigns elements of tuples from the dataset D to the variables from the list v in accordance with the behaviour presented in Section 5.2.

$$A(D, v) = [(t, B \cup Bind(t, v))|\ (t, B) \in D]$$

where

$$Bind(t, v) = \begin{cases} RBind(t, \$v_1, \ldots, \$v_n) & \text{if } v = (\$v_1, \ldots, \$v_n) \\ (LBind(t, \$v_1, \ldots, \$v_m) & \\ \cup\ CBind(t, @p, m, n) & \text{if } v = (\$v_1, \ldots, \$v_m, @p, \$w_1, \ldots, \$w_n) \\ \cup\ RBind(t, \$w_1, \ldots, \$w_n)) & \end{cases}$$

and

$$\begin{array}{rl} LBind(t, \$v_1, \ldots, \$v_n) & = \{(\$v_1, t(1)), \ldots, (\$v_n, t(n))\} \\ CBind(t, @p, m, n) & = \{(@p, (t(m+1), \ldots, t(|t| - n)))\} \\ RBind(t, \$v_1, \ldots, \$v_m) & = \{(\$v_1, t(|t| - m + 1)), \ldots, (\$v_m, t(|t|))\}. \end{array}$$

We assume above that $LBind$ and $RBind$ are not defined[j] if $n, m > |t|$ and that $(t(m + 1), \ldots, t(|t| - n)) = ()$ if $m + n > |t|$. On the basis of the A operator, we can define the path expression as follows:

$$\begin{array}{l} \langle path \rangle \ ::= \ \langle generator \rangle \ \langle vars \rangle \ [\![A(\ [\![generator]\!]^{I,B},\ [\![vars]\!]^{I,B})]\!]^{I,B} \\ \ \ | \ \ \langle path' \rangle \ \text{'.'} \ \langle rexpr \rangle \ \langle vars \rangle \ [\![\ [\![path']\!]^{I,B} \bowtie A(\ [\![rexpr]\!]^{I,B},\ [\![vars]\!]^{I,B})]\!]^{I,B} \\ \ \ | \ \ \langle path' \rangle \ \text{'<'} \ \langle expr \rangle \ \text{'>'} \ [\![\oplus[\ [\![expr]\!]^{I,B \cup B'} \mid (t, B') \in [\![path']\!]^{I,B}]]\!]^{I,B} \\ \ \ | \ \ \langle path' \rangle \ \text{'['} \ \langle condition \rangle \ \text{']'} \ [\![[\ (t, B') \in [\![path']\!]^{I,B} \mid [\![condition]\!]^{I,B \cup B'} = true]]\!]^{I,B} \\ \ \ | \ \ \langle path' \rangle \ \text{'.'} \ \langle generator \rangle \ \langle vars \rangle \\ \qquad [\![A([\ (t, B') \in [\![path']\!]^{I,B} \mid t(|t|) \in [\![generator]\!]^{I,B}],\ [\![vars]\!]^{I,B})]\!]^{I,B} \end{array}$$

[j]The prototype interpreter discussed in Appendix B rises an error in this case.

A.7 Multi-path Expressions

$\langle unary\_expr \rangle$ ::= '-' $\langle path \rangle$ $[\![([-e], \emptyset) \mid e \in L(\,[\![path]\!]^{I,B})]\!]^{I,B}$ | $\langle path \rangle$

$\langle mul\_expr \rangle$ ::= $\langle unary\_expr \rangle$ '*' $\langle mul\_expr' \rangle$

$[\![([x * y], \emptyset) \mid x \in L(\,[\![unary\_expr]\!]^{I,B}), y \in L(\,[\![mul\_expr']\!]^{I,B})]\!]^{I,B}$

| $\langle unary\_expr \rangle$ '/' $\langle mul\_expr' \rangle$

$[\![([x/y], \emptyset) \mid x \in L(\,[\![unary\_expr]\!]^{I,B}), y \in L(\,[\![mul\_expr']\!]^{I,B}), y \neq 0]\!]^{I,B}$

| $\langle unary\_expr \rangle$

$\langle add\_expr \rangle$ ::= $\langle mul\_expr \rangle$ '+' $\langle add\_expr' \rangle$

$[\![([x + y], \emptyset) \mid x \in L(\,[\![mul\_expr]\!]^{I,B}), y \in L(\,[\![add\_expr']\!]^{I,B})]\!]^{I,B}$

| $\langle mul\_expr \rangle$ '-' $\langle add\_expr' \rangle$

$[\![([x - y], \emptyset) \mid x \in L(\,[\![mul\_expr]\!]^{I,B}), y \in L(\,[\![add\_expr']\!]^{I,B})]\!]^{I,B}$

| $\langle mul\_expr \rangle$

$\langle with\_expr \rangle$ ::= $\langle add\_expr \rangle$ 'with' $\langle with\_expr' \rangle$

$[\![(l \oplus r, B_l \cup B_r) \mid (l, B_l) \in [\![add\_expr]\!]^{I,B}, (r, B_r) \in [\![with\_expr']\!]^{I,B}]\!]^{I,B}$

| $\langle add\_expr \rangle$

$\langle isect\_expr \rangle$ ::= $\langle with\_expr \rangle$ 'intersect' $\langle isect\_expr' \rangle$

$[\![(t, \emptyset) \mid (t, B_l) \in [\![with\_expr]\!]^{I,B}$ and $(t, B_r) \in [\![isect\_expr']\!]^{I,B}]\!]^{I,B}$

| $\langle with\_expr \rangle$

$\langle expr \rangle$ ::= $\langle isect\_expr \rangle$ 'union' $\langle expr' \rangle$

$[\![(t, \emptyset) \mid (t, B_l) \in [\![isect\_expr]\!]^{I,B}$ or $(t, B_r) \in [\![expr']\!]^{I,B}]\!]^{I,B}$

| $\langle isect\_expr \rangle$ 'except' $\langle expr' \rangle$

$[\![(t, \emptyset) \mid (t, B_l) \in [\![isect\_expr]\!]^{I,B}$ and $(t, B_r) \notin [\![expr']\!]^{I,B}]\!]^{I,B}$

| $\langle isect\_expr \rangle$

B Complexity of Query Evaluation in the Prototype Interpreter

A prototype interpreter of the WQuery language has been implemented in Scala. The WQuery wordnet model described in Section 4 is realized in the interpreter as an in-memory database. Relations are stored as lists of lists. Hash indexes are created for every argument of a relation. The WQuery interpreter operates on this database in a manner that corresponds mostly to the interpretation rules of our formal model described in Appendix A. Therefore, to analyze the complexity of the query evaluation process executed by the interpreter, we will analyze the complexity of the primitive operations used to define the interpretation rules.

According to our formal model, the dataset which is a result of evaluating a query is a list of pairs that consist of a list (of domain elements) and a set of variable bindings (cf. Sec. A.2, A.6 and A.7). The same data structure is maintained in the interpreter with sets of variable bindings realized as hash tables. Therefore, the predominant operations that contribute to the efficiency of the query evaluation are the ones that process lists. The list operators that are used both for specifying the WQuery semantics and implementing the prototype

interpreter are: concatenation $\oplus$, difference $-$, membership testing $\in$, inclusion testing $\subset$, and various forms of list comprehension $[\psi|\varphi]$. The time complexity of membership testing and concatenation is proportional to the list sizes. Inclusion testing and difference are quadratic to the list sizes. List comprehensions can be interpreted as nested loops, hence their complexity depends on the sizes of the arguments used for iteration, the complexity of the condition ψ being checked and the expression φ being evaluated for every step of the loop.

Besides the list operations, the evaluation of a query involves also application of the dataset operators L, C, A and $\bowtie$ introduced in Appendix A. The tail taking operation in Scala is $O(1)$, thus the time complexity of L is proportional to the number of elements of the dataset passed as its argument. The C operator traverses every tuple of every pair belonging to the passed argument, hence its complexity is proportional to the number of elements in the dataset times the maximum size of the tuple within it. The A operator applies the $Bind$ function once for every tuple within its first argument. The worst case of application of $Bind$ occurs when the numbers of step variables on the left and right hand side of the path variable is equal to the tuple size. Even in that case, the time complexity of $Bind$ is proportional to the size of the variable list. Thus, the complexity of A is limited by the number of elements in the dataset times the number of variables being bound. The $\bowtie$ operator implementation realized in the interpreter iterates through the tuples of one of its arguments and looks up the matching tuples from the second argument in the hash index. The worst case scenario for $D \bowtie E$ occurs when all the tuples from D end with the same value which is also the first element of every tuple from E. In that case, we concatenate every tuple from D with every tuple from E, therefore the time complexity of this operation is proportional to the size of the Cartesian product of D and E.

Table 4 Time complexity of list operations. L, R, L_1, L_n denote lists, e, e_1 and e_n denote list elements.

Operation	**Time Complexity**
$e \in L$	$O(\|L\|)$
$L \oplus R$	$O(\|L\| + \|R\|)$
$L \subset R$	$O(\|L\|\|R\|)$
$L - R$	$O(\|L\|\|R\|)$
$[\varphi(e_1, \ldots, e_n) \| e_1 \in L_1, \ldots, e_n \in L_n, \psi(e_1, \ldots, e_n)]$	$O(\|L_1\| \cdots \|L_n\| O(\varphi) O(\psi))$

Table 5 Time complexity of dataset operations. D and E denote datasets, v denotes a list of variables.

Operation	**Time Complexity**
$L(D)$	$O(\|D\|)$
$C(D)$	$O(\|D\| max_{t \in D}(\|t\|))$
$A(D, v)$	$O(\|D\|\|v\|)$
$D \bowtie E$	$O(\|D\| max_{t \in D}(\|t\|) \|E\| max_{t \in D}(\|t\|))$

References

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-53771-0.

Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1): 68–88, 1997.

Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. SentiWordNet 3.0: An Enhanced Lexical Resource for Sentiment Analysis and Opinion Mining. In Calzolari et al. (2010). ISBN 2-9517408-6-7.

Pushpak Bhattacharyya, Christiane Fellbaum, and Piek Vossen, editors. *Proceedings of the Fifth Global WordNet Conference – GWC 2010*, Mumbai, India, 2010. Narosa Publishing.

Patrick Blackburn and Johan van Benthem. Modal Logic: a Semantic Perspective. In Patrick Blackburn et al., editors, *Handbook of Modal Logic*, pages 1–84. Elsevier, 2007.

Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language (Second Edition). W3C recommendation, W3C, December 2010. http://www.w3.org/TR/2010/REC-xquery-20101214/ (Access date: 2013-03-14).

P. A. Boncz, T. Grust, J. Siméon, and M. van Keulen. 06472 Abstracts Collection – XQuery Implementation Paradigms. In P. A. Boncz, T. Grust, J. Siméon, and M. van Keulen, editors, *XQuery Implementation Paradigms*, number 06472 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. `http://drops.dagstuhl.de/opus/volltexte/2007/893` (Access date: 2015-03-21).

Nicoletta Calzolari et al., editors. *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, May 2010. ELRA. ISBN 2-9517408-6-7.

R. G. G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco, California, 2000. ISBN 1-55860-647-5.

Ashok K. Chandra and David Harel. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116/ (Access date: 2013-03-14).

Ann Devitt and Carl Vogel. The Topology of WordNet: Some Metrics. In Sojka et al. (2003), pages 106–111.

Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA, 1998. ISBN 978-0-262-06197-1.

Christiane Fellbaum and Piek Vossen, editors. *Proceedings of the Sixth International Wordnet Conference – GWC 2012*, Matsue, Japan, January 2012.

Aldo Gangemi, Guus Schreiber, and Mark van Assem. RDF/owl representation of wordnet. W3C working draft, W3C, June 2006. http://www.w3.org/TR/2006/WD-wordnet-rdf-20060619/ (Access date: 2013-03-21).

Alvaro Graves and Claudio Gutierrez. Data Representations for WordNet: A Case for RDF. In Sojka et al. (2005), pages 165–169.

Maria Grigoriadou, Harry Kornilakis, Eleni Galiotou, Sofia Stamou, and Evangelos Papakitsos. The Software Infrastructure For The Development And Validation Of The Greek Wordnet. Tufis (2004), pages 89–105.

Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

Verena Henrich and Erhard Hinrichs. Standardizing Wordnets in the ISO Standard LMF: Wordnet-LMF for GermaNet. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 456–464, Beijing, China, August 2010. Coling 2010 Organizing Committee.

Ales Horak, Karel Pala, Adam Rambousek, and Martin Povolny. DEBVisDic - First Version of New Client-Server Wordnet Browsing and Editing Tool. In Sojka et al. (2005).

Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Proceedings of the Extreme Markup Languages® 2004 Conference, 2-6 August 2004, Montréal, Quebec, Canada*, 2004.

Svetla Koeva, Angel Genov, and Georgi Totkov. Towards Bulgarian Wordnet. Tufis (2004), pages 45–60.

Svetla Koeva, Stoyan Mihov, and Tinko Tinchev. Bulgarian Wordnet – Structure and Validation. Tufis (2004), pages 61–78.

Marek Kubis. An Access Layer to PolNet – Polish WordNet. In Zygmunt Vetulani, editor, *Human Language Technology. Challenges for Computer Science and Linguistics*, volume 6562 of *Lecture Notes in Computer Science*, pages 444–455. Springer Berlin / Heidelberg, 2011.

Marek Kubis. A Query Language for WordNet-like Lexical Databases. In Jeng-Shyang Pan, Shyi-Ming Chen, and Ngoc-Thanh Nguyen, editors, *Intelligent Information and Database Systems*, volume 7198 of *Lecture Notes in Artificial Intelligence*, pages 436–445. Springer Heidelberg, 2012.

Marek Kubis. *Język zapytań dla leksykalnej bazy danych typu WordNet*. PhD thesis, Uniwersytet im. Adama Mickiewicza w Poznaniu, June 2013.

Claudia Leacock and Martin Chodorow. *Combining Local Context and WordNet Similarity for Word Sense Identification*, chapter 11, pages 265–283. In Fellbaum (1998), 1998. ISBN 978-0-262-06197-1.

Marek Maziarz, Maciej Piasecki, and Stanisław Szpakowicz. Approaching plWordNet 2.0. In Fellbaum and Vossen (2012).

Siddharth Patwardhan, Satanjeev Banerjee, and Ted Pedersen. Using Measures of Semantic Relatedness for Word Sense Disambiguation. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 2588 of *Lecture Notes in Computer Science*, pages 241–257. Springer Berlin / Heidelberg, 2003.

Princeton University. About Wordet. http://wordnet.princeton.edu, 2011a. (Accessed 25 March 2013).

Princeton University. WordNet - Related Projects. http://wordnet.princeton.edu/wordnet/related-projects/, 2011b. (Accessed 25 March 2013).

Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, W3C, jan 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (Access date: 2013-03-14).

Borislav Rizov. Hydra: a Modal Logic Tool for Wordnet Development, Validation and Exploration. In Nicoletta Calzolari et al., editors, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, 2008.

Masoud Rouhizadeh, Mahsa A. Yarmohammadi, and Mehrnoush Shamsfard. Developing the Persian WordNet of Verbs; Issues of Compound Verbs and Building the Editor. In Bhattacharyya et al. (2010).

Pavel Smrž. Quality Control and Checking for Wordnet Development: A Case Study of BalkaNet. Tufis (2004), pages 173–182.

Petr Sojka et al., editors. *Proceedings of the Second International WordNet Conference – GWC 2004*, Brno, Czech Republic, 2003. Masaryk University.

Petr Sojka et al., editors. *Proceedings of the Third International WordNet Conference – GWC 2006*, Brno, Czech Republic, 2005. Masaryk University.

Claudia Soria, Monica Monachini, and Piek Vossen. Wordnet-LMF: Fleshing out a Standardized Format for Wordnet Interoperability. In *Proceeding of the 2009 international workshop on Intercultural collaboration*, pages 139–146, New York, USA, 2009. ACM.

Attila Tanács, Dóra Csendes, Veronika Vincze, Christiane Fellbaum, and Piek Vossen, editors. *Proceedings of the Fourth International WordNet Conference – GWC 2008*, Szeged, Hungary, 2007. University of Szeged, Department of Informatics.

Randee I. Tengi. *Design and Implementation of the WordNet Lexical Database and Searching Software*, chapter 4, pages 105–127. In Fellbaum (1998), 1998. ISBN 978-0-262-06197-1.

Dan Tufis, editor. *Romanian Journal of Information Science and Technology*, volume 7(1-2). 2004.

Zygmunt Vetulani. Wordnet Based Lexicon Grammar for Polish. In Nicoletta Calzolari et al., editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*. European Language Resources Association (ELRA), may 2012.

Zygmunt Vetulani, Marek Kubis, and Tomasz Obrebski. PolNet - Polish WordNet: Data and Tools. In Calzolari et al. (2010). ISBN 2-9517408-6-7.

Piek Vossen. *EuroWordNet: general document*. dare.ubvu.vu.nl, 2002. http://dare.ubvu.vu.nl/handle/1871/11116.

Sergey Yablonsky and Andrey Sukhonogov. Semi-Automated English-Russian WordNet Construction: Initial Resources, Software and Methods of Translation. In Sojka et al. (2005), pages 345–347.