Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Jakub Stejskal**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Data-Oriented Conversational System**

Guidelines:

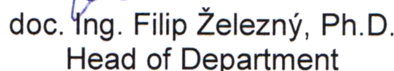
Perform reseach on existing data-oriented dialog systems. Based on the research, identify the basic components, design and implement dialog system answering questions in a natural language. Implement the system as a web service with a simple REST application programming interface. The problem domain is limited to the information included in the university information system KOS. Design and carry out experiments to evaluate quality the solution. Evaluate usability of the dialogue system comparing it with a classical WEB UI design.

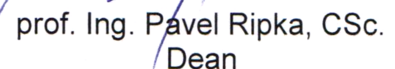
Bibliography/Sources:

LUKE©, Adam - Multilingual multidomain conversational system learning from dialogs. Praha, 2013.
Hardy, Hilda and Strzalkowski, Tomek and Wu, Min and Ursu, Cristian and Webb, Nick and Biermann, Alan and Inouye, R. Bryce and McKenzie, Ashley - Data-Driven Strategies for an Automated Dialogue System - Barcelona, 2004.
Bayan Aref Abu Shawar - A Corpus Based Approach to Generalising a Chatbot System - Leeds, 2005
Verena Rieser, Oliver Lemon - Reinforcement Learning for Adaptive Dialogue Systems - Berlin, 2011

Diploma Thesis Supervisor: Ing. Jan Šedivý, CSc.

Valid until the end of the summer semester of academic year 2014/2015


doc. Ing. Filip Železný, Ph.D.
Head of Department


prof. Ing. Pavel Ripka, CSc.
Dean

Prague, March 3, 2014

Master's thesis

Data-oriented conversational system

Bc. Jakub Stejskal

May 2014

Ing. Jan Šedivý, CSc.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Computer Science and Engineering

Acknowledgement

First of all I would like to thank my supervisor, Ing. Jan Šedivý, CSc., for his patient and constructive guidance. I am indebted to Ing. Jakub Jirůtka for providing insight into KOSapi and to my colleagues who contributed to data collection. My thanks also go to my family and friends for their continued support.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

Abstrakt

Databázová rozhraní v přirozeném jazyce (NLIDB) jsou systémy schopné překladu uživatelských dotazů, napsaných v přirozeném jazyce, do formálních databázových dotazů za účelem získat požadované informace z databáze.

Tato diplomová práce zkoumá možnost využití RESTových webových služeb jakožto báze znalostí pro NLIDB systém, namísto běžně používaných relačních databází. Práce sleduje návrh a implementaci systému NaLIDa, což je NLIDB poskytující rozhraní univerzitního informačního systému prostřednictvím RESTful služby zvané KOSapi.

Systém anotuje vstup v přirozeném jazyce za pomoci nástroje Standford CoreNLP a výsledné anotace využívá ke generování a filtrování hypotéz sémantické interpretace vstupu. Po následné disambiguaci se interpretace mapuje na entity v grafové reprezentaci databáze, ve které je algoritmem pro hledání nejkratších cest nalezeno optimální spojení entit interpretace. Tato cesta je nakonec tranformována do posloupnosti požadavků do RESTového API.

Na testovacích datech posbíraných od členů cílové skupiny byly provedeny experimenty, které ukázaly, že pro platné vstupy systém vykazuje přesnost 96.8 %, úplnost 85.9 %, vůli 88.7 % a průměrnou dobu odezvy 1.6 vteřiny. V porovnání s výsledky ostatních systémů se tyto hodnoty jeví jako uspokojivé. Nicméně experimenty také odhalily příležitosti pro potenciální zlepšení přesnosti, stejně jako pro rozšíření okruhu platných vstupů.

Klíčová slova

přirozený jazyk; zodpovídání otázek; NLIDB; RESTful API

Abstract

Natural language interfaces to databases (NLIDB) are systems that are able to translate natural language queries supplied by users into formal database queries in order to retrieve the requested information from a database.

This thesis explores the possibility to use the RESTful Web services as the underlying knowledge base for a NLIDB system instead of prevalent relational databases. It follows the design and implementation of NaLIDa, a NLIDB that provides an interface to the university information system via intermediate RESTful API service called KOSapi.

The system annotates the natural language input with the help of Standford CoreNLP toolkit and uses the annotations to generate and filter semantic interpretation hypotheses. After a disambiguation, the interpretation is mapped to entities in a graph representation of the underlying database and the shortest path search algorithm finds an optimal join path connecting these entities. In the end, the path is transformed into a sequence of requests to the API.

The experiments performed on the data collected from members of the target audience show that for the valid inputs, the systems exhibits precision 96.8 %, recall 85.9 %, willingness 88.7 % and average response time 1.6 s. These results are considered satisfactory in the comparison with those reported by other systems, but the experiments also revealed opportunities for prospective accuracy improvements and extensions of the valid input range.

Keywords

natural language; question answering; NLIDB; RESTful API

Contents

1. Preface **1**
- 1.1. Structure . . . 1
- 1.2. Typographical note . . . 1

2. Problem definition **2**
- 2.1. Information retrieval . . . 2
- 2.2. Natural language interfaces to databases . . . 3
- 2.3. Natural language interfaces to RESTful Web services . . . 3
- 2.4. KOSapi . . . 3
- 2.5. NaLIDa . . . 3

3. Related work **5**
- 3.1. Natural language interfaces to database . . . 5
  - 3.1.1. Evaluation . . . 5
- 3.2. RESTful Web services . . . 6
  - 3.2.1. API . . . 6
  - 3.2.2. Web service . . . 6
  - 3.2.3. Representational state transfer . . . 7
- 3.3. KOSapi . . . 7
  - 3.3.1. RESTful Service Query Language (RSQL) . . . 8
  - 3.3.2. XPartial . . . 9

4. Design **10**
- 4.1. Scope . . . 10
  - 4.1.1. Natural language . . . 10
  - 4.1.2. Knowledge domain . . . 11
  - 4.1.3. Query utterance types . . . 11
  - 4.1.4. Query constraints . . . 12
  - 4.1.5. Extragrammatical utterances . . . 12
  - 4.1.6. Input modality . . . 13
  - 4.1.7. Output modality . . . 13
- 4.2. Architecture . . . 13
- 4.3. Knowledge domain . . . 14
  - 4.3.1. Schema . . . 14
  - 4.3.2. Lexicon . . . 15
- 4.4. Syntactic analysis . . . 15
  - 4.4.1. Extension of supported expressions . . . 16
- 4.5. Semantic analysis . . . 17
- 4.6. Query generation . . . 19
  - 4.6.1. Projections, constraints and path . . . 19
  - 4.6.2. Entity-relationship graph . . . 20
  - 4.6.3. Shortest path search . . . 20
  - 4.6.4. Query plan . . . 21
  - 4.6.5. SQL query generation . . . 21
  - 4.6.6. REST query generation . . . 22
- 4.7. REST API . . . 26
  - 4.7.1. KOSapi response . . . 26

4.7.2. SQL query 27
4.7.3. Detailed intermediate outputs 27

5. Implementation **28**
5.1. Schema 28
5.1.1. Schema element tree 28
5.1.2. Entity-relationship graph 29
5.1.3. Schema description format 29
5.1.4. Value tokens 31
5.2. Lexicon 31
5.3. Syntactic analysis 31
5.4. Interpreter 32
5.5. Query generator 32
5.6. Command line interface 34
5.7. Web service 34
5.7.1. Home page 35
5.7.2. REST resources 36

6. Evaluation **38**
6.1. Data collection 38
6.2. Evaluation methodology 39
6.3. Results 40

7. Conclusion **45**
7.1. Future work 45

Appendices

A. Contents of the enclosed CD **46**

B. Dependencies **47**
B.1. NaLIDa Core dependencies 47
B.2. NaLIDa Web dependencies 47

C. Entity-relationship model of KOSapi **48**

Bibliography **49**

List of Figures

1. Integration diagram 4

2. Architecture of NaLIDa components 14
3. An example of an *element tree* 15
4. Types of connections on the join path 23

5. Class diagram of `nalida.schema` package 29
6. Class diagram of `nalida.interpretation` package 32
7. Class diagram of `nalida.query` package 33
8. NaLIDa RESTful API Home page 35

9. Evaluation measurements for valid inputs 43

10. KOSapi entity-relationship model 48

List of Tables

1. Number and percentage of inputs by validity and correctness of results . 40
2. Precision, recall and willingness for all inputs and the valid ones 41
3. Number and percentage of invalid inputs by violated constraint 42

List of Algorithms

1. Generation of REST query plan 25

Listings

3.1. Snippet of KOSapi resource representation 8
3.2. RSQL grammar in EBNF [1] 9
3.3. XPartial grammar in EBNF [1] 9

5.1. Snippet of the schema description file 30
5.2. Default output of NaLIDa command line interface 34
5.3. XML serialization of KOSapi response 36
5.4. XML serialization of SQL response 36
5.5. XML serialization of interpretations 36
5.6. XML serialization of error message 37

Abbreviations

Throughout the text, following abbreviations are used in the respective meaning.

AI	artificial intelligence
AJAX	Asynchronous JavaScript and XML
API	application programming interface
ASR	automatic speech recognition
DBMS	database management system
DOM	Document Object Model
EBNF	Extended Backus–Naur Form
ERG	entity-relationship graph
GUI	graphical user interface
HTTP	Hypertext Transfer Protocol
KOS	Komponenta Studium
NL	natural language
NLIDB	natural language interface to database
NLP	natural language processing
NP	noun phrase
REST	representational state transfer
RSQL	RESTful Service Query Language
SQL	Structured Query Language
URI	uniform resource identifier
XML	Extensible Markup Language

1. Preface

The topic of this thesis is the development of a natural language interface that allows an untrained user to retrieve structured data from a database without the knowledge of any artificial query language, just by asking a question in a natural language such as English.

The idea of creating NLI for the university information system was conceived by the supervisor of this thesis, as it could represent a natural extension to a mobile application for the information system, that is being developed by other colleagues.

1.1. Structure

The work consists of seven chapters. This chapter introduces the thesis and describes its structure and formal aspects. The solved problem is described and placed into perspective in chapter 2. Chapter 3 presents an overview of the state of the art in the area and of the previous work that provides foundation to this thesis. Chapter 4 concerns the design of the developed system, its scope and its individual architectural components, while chapter 5 describes its implementation details and how it is used. The methodology and the results of evaluation experiments are presented in chapter 6. And finally, the thesis is summarized and concluded by chapter 7.

1.2. Typographical note

Typefaces are used to distinguish symbols of different domains throughout the text:

- Names of projects and technologies are set in SmallCaps.
- Code samples and snippets are set in `teletype`.
- Variable placeholders in code samples are set in *`italics teletype`*.
- Mathematical formulas and important newly defined concepts are set in *italics*.
- Natural language utterances are set in “*`quoted italics teletype`*”.

2. Problem definition

The assignment of this thesis defines its objectives to be a design and an implementation of a dialog system capable of answering questions from the domain of the university information system. The implemented system is required to provide an application programming interface in the form of RESTful Web service. An experimental evaluation of the solution is also demanded in order to assess its quality.

The research revealed that the assignment lies on the boundary of two related but distinct fields, dialog systems and question answering. After consultation with the supervisor, the thesis focuses on the question answering aspect of the problem and it limits the dialog to disambiguation of the questions.

In the context of the evaluation, the assignment mentions usability and comparison with the classic web user interface. However, because the system provides an application programming interface rather than user interface, the criterion of usability is not applicable for its evaluation. A testing data set collected from users is instead employed to evaluate the system by means of methods standard in the question answering field.

The rest of this chapter places the problem at hand into perspective and it further explains it in more detail.

2.1. Information retrieval

We are said to live in an information society. Indeed, an increasing importance of the ability to effectively distribute and manipulate information is a long-term trend. While the recent years show a rising interest in processing and analysis of Internet's unstructured data, a large portion of the useful information is nonetheless stored in a form of a structured databases. In order to extract information from a database, a query must be formulated in such a language that can be directly interpreted and evaluated by a machine.

As computers, and more notably mobile devices such as smartphones and tablets, are growing more accessible and affordable, the information technologies are becoming a domain of the general public. Most people working with these technologies today therefore do not possess a computer background nor training to master an artificial query language.

To reflect this fact, research and development of information systems in last decades has largely focused on design of graphical interfaces and their usability. However, no matter how elaborate these graphical user interfaces (GUI) are, they usually cannot match flexibility and expressiveness of a database query languages.

The most straightforward way for people to inquire and communicate is of course their natural language (NL). A natural language user interface (NLI) has thus the potential to achieve expressiveness comparable to the artificial query languages while exceeding the graphical interfaces in intuitiveness and ease of use.

2.2. Natural language interfaces to databases

Natural language interface to database (NLIDB) is a system that allows user to access (and in some cases manipulate) information stored in a database by providing textual or spoken requests expressed in a natural language. The core functionality of such system is the translation of natural language questions or commands into machine-readable queries executable on the database. However, a full-fledged NLI often has to deal with various other tasks described in chapter 3.1, such as dialog management and response representation.

According to [2], NLIDB has been an open problem since 1960's. Despite 50 years of research and great advancements in fields such as natural language processing (NLP) or automatic speech recognition (ASR) as well as computing in general, there has been no wide spread use and only a limited commercial success of NLIDB systems. This is due to persisting deficiencies and limitations of the existing solutions that are mostly caused by the inherent complexity of natural languages.

2.3. Natural language interfaces to RESTful Web services

It has become a commonplace that technological companies build application programming interfaces (API) to their products and data in form of a Web Service. Service-oriented architecture is used both internally in order to decouple components of a large system and externally to provide a public interface for others to develop applications on.

ProgrammableWeb, a catalog tracking web mash-ups and APIs, currently lists more than 11,000 public APIs and this number is steadily growing [3]. The most prevalent API architecture on ProgrammableWeb is the REST architectural style.

In this setup, it may often be the case that developers of a NLIDB system do not have direct access to the database and they are dependent on a web service that encapsulates the concerned database.

The NLP components of such system does not necessarily differ from those used in the systems backed by a relation database, but the same is not always true for the other components. The components responsible for knowledge-domain configuration, query generation and result processing may work with very different querying mechanisms and data representations.

2.4. KOSapi

KOSapi is an instance of a service described in the previous section. It is a RESTful Web service API to the database of KOS, the study information system of Czech Technical University in Prague. KOSapi serves as a developer-friendly and easy-to-use interface for client applications aiming to utilize the data from KOS.

2.5. NaLIDa

The primary practically-oriented target of this thesis is to implement a RESTful Web service that provides the backend to a natural language interface for KOS. This is achieved by building a NLIDB system using KOSapi as a mediator to the KOS database. On the implementation side, the focus of this thesis is thus divided between

- developing a natural language processing engine exhibiting good performance and accuracy for the KOS knowledge domain,
- wiring this engine to the underlying database via communication principles and technologies used by KOSApi, namely REST, RSQL and XPartial,
- and creating a web application that provides a RESTful API to the engine.

The relations of these components are depicted in Figure 1. Development of a client applications is not aim of this thesis and it is left to others. While the web application includes a simple graphical interface, it's purpose is to be a developer-friendly entry point to the service, rather than an application used by end users.

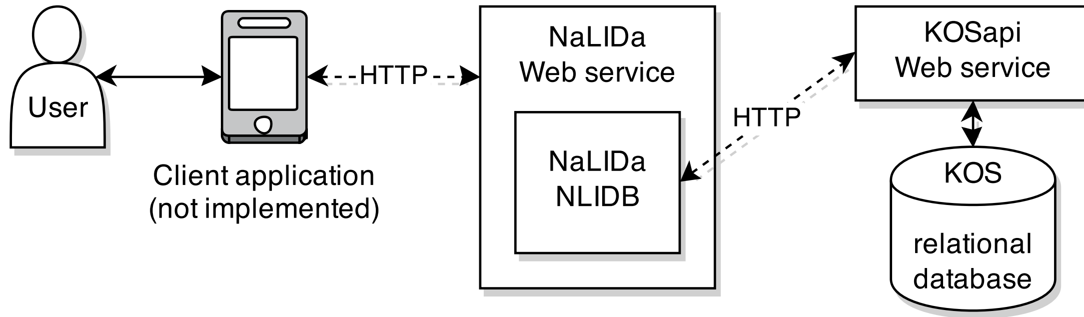


Figure 1. Integration diagram

Successful completion of this task is verified by experimental evaluation.

A secondary, broader goal is to design a versatile NLIDB system that allows for easy extensibility and portability in terms of different knowledge-domains, different database management systems (DBMS) or even different natural language. This is achieved by dividing the system into several components, where each dependency, be it to a domain, a database or a language, is encapsulated in a single extensible and replaceable component.

For easier referencing, the resulting system is call NaLIDa, which is simply short for "Natural Language Interface to Database".

3. Related work

Section 3.1 of this chapter provides an overview of the research in the field of natural language interfaces to databases and section 3.2 defines the important concepts from the areas RESTful Web services. The last section, 3.3, describes KOSAPI and the related technologies.

3.1. Natural language interfaces to database

The discipline of natural language interfaces to databases is a subfield of question answering, information retrieval and natural language processing. NLIDB distinguishes itself from the general question answering in the fact that it focuses on a restricted knowledge domain represented in terms of structured database.

During the fifty years of research in the area, the focus has moved from the implementation of single-purpose systems tied to one database and knowledge domain [4] to the development of universal frameworks that can be easily adapted to a new domain with minimal effort [5]. Portability of the NLIDB systems has been studied on different axes as well, such as independence on the natural language, or on the underlying DBMS. Apart from the actual processing and translation phase, the portable NLIDBs must also focus on the configuration of the system to the given domain or platform. Ideally, this phase should be performed automatically without high demands on the administrator's expertise in database systems, programming or linguistics.

Generally speaking, the main concern of the field is the mapping of the input natural language words to their meaning in the context of the database entities and operations. This task revolves around the concept of ambiguity. According to [6], inputs are considered ambiguous if there are multiple alternative structures that can be built for them. In order to provide answer, the system must first unambiguously determine the single meaning of the input query. That requires analysis of the input on multiple linguistic levels including morphology, syntax and semantics [2]. Some of the research overlap into fields such as dialog systems by studying the conversational discourse [7].

Because the understanding to the natural languages in their entirety is considered to be a AI-complete problem (in sense that it is equal in complexity to the central problem of artificial intelligence - to create computers as intelligent as people) NLIDB system typically limits the supported input utterances to a subset of NL defined for example in terms of syntactic constraints or a limited vocabulary.

3.1.1. Evaluation

The authors of [2] report that there are no standard benchmarks for the evaluation of NLIDB, because it is very complicated to formulate general quantitative measures of a system as complex as NLIDBs. However, in order to have some objectively measurable conclusions, most of the literature on the field presents an experimental evaluation in one form or another.

The most used criterion to determine the quality of a NLIDB system is the translation success, that is, the semantic equivalence between the natural language input and the

resulting database query. To provide an estimation on the success rate of the translation, one typically needs a testing corpus of natural language queries and a method to decide whether the output of the system is the correct translation and provides a relevant answer. Individual publications about NLIDB systems vary in both testing data and the objective function.

Most of the domain-dependent NLIDBs use the database they were build for and an ad-hoc testing corpus either written by the system creators or by independent respondents [8]. The portable NLIDB systems are usually evaluated on multiple benchmark databases such as ATIS [9], Geoquery [5, 10, 9] or Northwind [11, 12]. Because most of the systems consider only relational databases, the benchmark databases are also usually relational (with some exceptions such as Geoquery which is represented as Prolog assertions). Some benchmark databases, such as ATIS or Geoquery are distributed with a corpus of related natural language questions manually annotated with the corresponding translations to database queries. The size of these corpora varies from tens to several hundreds of questions.

The accuracy of a NLIDB system is usually determined by the analysis of the database query produced by the system. In the case of annotated corpora, the system can automatically compare the annotations with the outputs of the translation. However, a single natural language query can be in many cases translated into multiple semantically equivalent database queries. Some systems address this issue by a component that recognizes semantic equivalence of two database queries [5]. The systems whose corpora are not annotated must rely on manual comparison of the input and the output.

3.2. RESTful Web services

3.2.1. API

An application programming interface (API) specifies how different components of a software system interact with each other. Similarly to the way the user interfaces facilitate interaction between humans and computers, API provides the means of communication between different software programs.

3.2.2. Web service

World Wide Web Consortium (W3C) defines Web service as follows:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [13]

Nowadays, the term is however often understood in a broader sense, which does not require a Web service to be defined in terms of WSDL nor to use XML-based SOAP as a communication protocol. A wide range of description formats and several serialization formats (most notably JSON) have been introduced instead and are used along with REST principles.

A new branch of research that combines study of Web services with NLI has emerged in recent years. However, it focuses on the automatic service discovery and composition of Semantic Web services based on ontologies [14, 15]. Even though these studies are somewhat relative to our topic, their orientation on a higher level of composition and on the ontologies render their methods inapplicable for our problem.

3.2.3. Representational state transfer

Representational state transfer (REST) is a software architectural style based on the original notion of the World Wide Web. It is a set of high-level architectural constraints applied the components of a distributed system. These constraints include client-server model, statelessness, cacheability, uniform interface and layered architecture. REST was introduced in 2000 by Roy Fielding in his doctoral dissertation [16]. It is closely related to web standards such as HTTP and Uniform Resource Identifiers (URI).

Although it is widely recognized as a key concept in the area of web services, because of its abstract nature, the term is clouded in misconception and ambiguity. To avoid confusion, we take the definitions of REST terminology from [1].

RESTful services are defined in terms of resources. A *resources* is an arbitrary information or concept that can be named by a unique identifier. A resource is a time-dependent mapping from such identifier to a set of values, where value may be either a resource representation or a resource identifier. While the mapping resource may change in time, it must preserve its semantics.

A *resource identifier* uniquely identifies a resource. The identifiers are represented by Uniform Resource Identifiers, a standard defined in RFC 3986 [17]. A *resource representation* captures an actual state of a resource in a given time in terms of its data and also metadata that describe the data.

3.3. KOSapi

Komponenta studia (KOS) is a component of the information system at the Czech Technical University in Prague. Since its inception at the Faculty of Electrical Engineering in 1992 it has gone through a massive development without any major redesign or modernization and it is thus built upon obsolete technologies and outdated approaches. The symptoms it exhibits are typical: incapability to scale up to the growing user base, to meet new requirements and increasing expectations or even to reliably maintain the system.

Arguably the most pressing deficiency was the absence of an open and standardized communication protocol for accessing its data. This fact represented a major difficulty for development of applications built on top of the KOS platform, be it by the university departments or its students, and effectively stifled innovation and growth of an informational ecosystem.

With these issues in mind, the KOSapi was designed and implemented in 2010 by Jakub Jirůtka as a part of his Bachelor's thesis [18]. Its goal was to create a modern and robust application that would provide an accessible and well-documented API to KOS database. It is a Web service based on the REST architecture, Java Enterprise Edition platform and Spring framework. Jirůtka continued to develop of KOSapi in his Master's thesis [1], where he introduced two features that are crucial for building a NLIDB system for KOSapi, RSQL query language and XPartial projection language.

The API consists of several resources, each providing collection of representations of respective database entity, such as Student, Course or Division. Each resource has element subresources, that provide representation of one concrete entry. Some element subresources have other subresources of their own that represent entities related to the given entry.

The resources in KOSapi are represented in XML format. As the example in Listing 3.1 shows, the actual KOS data are in the `atom:content` element and they are wrapped in Atom metadata elements. The listing contains a snippet of the collection resource

representation of Division entities. It is an excerpt from a list of 10 entries, with only single entry actually shown. The entry metadata include the Atom element `<atom:link rel="self" />` that provides URI of the corresponding element resource.

Listing 3.1 Snippet of KOSapi resource representation

```
<atom:feed xmlns="http://kosapi.feld.cvut.cz/schema/3" ... >
  <atom:id>https://kosapi.feld.cvut.cz/api/divisions</atom:id>
  <atom:updated>2014-05-11T23:57:44.191</atom:updated>
  <atom:link rel="next" href="divisions?offset=10&limit=10"/>
    <atom:entry>
      <atom:title>Faculty of Electrical Engineering</atom:title>
      <atom:link rel="self" href="divisions/13000/"/>
      <atom:content atom:type="xml" xsi:type="division">
        <abbrev xml:lang="en">F3</abbrev>
        <code>13000</code>
        <divisionType>FACULTY</divisionType>
        <name xml:lang="en">Faculty of Electrical Engineering</name>
        <parent xlink:href="divisions/0/"/>
      </atom:content>
    </atom:entry>
    <atom:entry>
      ...
    </atom:entry>
    ...
<osearch:startIndex>0</osearch:startIndex>
<osearch:itemsPerPage>10</osearch:itemsPerPage>
</atom:feed>
```

3.3.1. RESTful Service Query Language (RSQL)

While the support of complex search queries is a fundamental part of most database engines, the majority of REST web services has only minimal query capabilities, such as fulltext search or query string parameters. Because the author of KOSapi was not able to find any relevant solution for complex queries on REST services, he decided to design his own, *RESTful Service Query Language* (*RSQL*).

A syntax of RSQL is a superset of FIQL [19], a filtering language for Atom format. It consists of two semanticaly equal sets of operators: those of the original FIQL and their RSQL counterparts added to provide more intuitive notation. The grammar of RSQL is expressed in Listing 3.2 by means of Extended Backus–Naur Form notation. Several examples are presented and explained in section 4.6.6. Following paragraph briefly describes the syntax and semantics of the RSQL variant.

RSQL query is a boolean expression consisting of one or more criteria connected by logical operators (`and`, `or`). A *criterion* comprises a selector and an argument separated by a *equality operator* (`=`, `!=`). A *selector* is name of an element that serves as a search parameter. Complex entity attributes may be dereferenced using dot notation An *argument* is a sequence of characters optionally enclosed in parentheses. String arguments support wildcards that represent arbitrary characters in a value: `_` for a single character and `*` for multiple characters. The criteria of numerical attributes by use *comparison operators* (`>=`, `<`, etc.) in addition to the equality operators.

Listing 3.2 RSQL grammar in EBNF [1]

```
expression    = [ "(" ],
                ( constraint | expression ),
                [ logical-op, ( constraint | expression ) ],
                [ ")" ];
constraint    = selector, comparison-op, argument;
logical-op    = ";" | " and " | "," | " or ";
comparison-op = "==" | "=" | "!=" | "=lt=" | "<" | "=le=" | "<=" |
                "=gt=" | ">" | "=ge=" | ">=";
selector      = identifier, { ("/" | "."), identifier };
identifier    = ? ["a"-"z", "A"-"Z", "_", "0"-"9", "-"]+ ?
argument      = arg_ws | arg_sq | arg_dq;
argument-ws   = ? ( ~["(", ")", ";", ",", " "] )+ ?;
argument-sq   = ? "'" ~["'"]+ "'" ?;
argument-dq   = ? "\"" ~["\""]+ "\"" ?;
```

3.3.2. XPartial

In many cases, a consumer of a REST resource does not need its whole representation. The ability to specify what data parts the server should send allows for significant reduction in the required network, memory and processing resources.

To address this issue, a concept of *Partial response* in the context of Google Data Protocol [20]. It is a mechanism of specifying a restrictive projection on a resource representation. The required subset of the representation elements is defined by listing their paths into the `field` query string parameter.

The format designed for use in KOSapi, XPartial, is an implementation of a subset of the GDP partial response format. As the name suggests, it is primarily intended for projecting XML representation.

Its grammar in EBNF is shown in Listing 3.3. An XPartial expression is a comma-separated sequence of element paths relative to a root element, e.g. `title,id,content`. A path to a nested element consists of its predecessor hierarchy separated by a slash, e.g. `entry/content/code`. Elements with a common parent can be grouped by parentheses for conciseness, e.g. `entry(title,content(code,range)`. A filtering of elements based on their XML attribute values is denoted in the form `element[@attribute='value']`, such as `name[@lang='en']`.

Listing 3.3 XPartial grammar in EBNF [1]

```
expression  = path,
              { ",", path };
path        = node,
              ( { "/", node } | subselect );
subselect   = "(", expression, ")"
node        = name,
              [ "[", attributes, "]" ];
attributes  = attribute,
              { ",", attribute };
attribute   = "@", att-name, "=", '"', att-value, '"';
att-name    = "a"-"z" | "A"-"Z" | "0"-"9" | "-" | "_" | ":";
att-value   = ? ~["\""]+ ?;
```

4. Design

This chapter describes in detail the approaches and algorithms chosen to fulfill the goals outlined in the chapter 2. The designed system aims to provide a NLIDB system for KOSApi in form of RESTful Web service, while taking into account the perspective of extensibility and portability (in the sense described in section 3.1). The high level design of the system thus emphasizes generality, while the concrete implementation focuses on the context of KOSApi.

The section 4.1 presents the scope of the solved tasks as well as the aspects of the problem, which this thesis does not address. The section 4.2 provides the overall picture of the system architecture and its components. The sections 4.3 to 4.7 then describe each component in detail.

Throughout the chapter, examples and figures are used to demonstrate the discussed topics. For the sake of brevity the examples use only a subset of knowledge domain consisting of three entities and their relations: teachers, divisions and courses. In reality, the system is able to answer queries regarding much larger domain (see appendices A and C).

4.1. Scope

As stated in 3.1, it is not within the power of a contemporary NLIDB system to be able to process a natural language in its entirety. It is therefore an important part of the design process to define a subset of the language that the system will focus on. In short, NaLIDa accepts grammatically correct English non-polar questions and noun phrases in textual form and produces corresponding data in XML format.

This section further describes constraints that were put on a accepted input and provided output and it explains the rationale behind these decisions. Other important non-functional requirements and goals are also outlined here.

4.1.1. Natural language

Data provided by KOSApi come in two parallel language variants - Czech and English. The most reasonable choice for a natural language to which the NLIDB understands is therefore one of these languages. Although the selection of Czech would make more sense from the user perspective, because the vast majority of the CTU students and employees (and hence the users of KOS) are Czech native speakers, English is chosen as the supported natural language for practical reasons.

Firstly, most of the research on NLP and NLIDB, as well as majority of available tools for NLP, focuses on English language. That does not really pose a problem with respect to the research, because its findings are largely transferable to arbitrary language, but the tools such as POS taggers and lexical parser are based on a statistical training from extensive corpora and are therefore language-dependent.

Secondly, if we disregard the concrete use case of KOSApi and consider the portability to other knowledge domains, the English speaking users obviously constitute a much larger potential audience.

4.1.2. Knowledge domain

To acquire the information to produce a response for a submitted query, NaLIDa relies purely on the data provided by the underlying DBMS and the metadata stored in its schema description (see 4.3.1). The system does not possess any external knowledge and it does not perform any post-processing on the extracted data.

That obviously means that it answers only questions regarding the database knowledge domain. In order to produce useful and up-to-date answers, the knowledge base is further limited only to data concerning the current and next semester.

There is however also a number of less obvious implications. In the first place, NaLIDa uses only the current query to determine what data the user requests. It does not keep track of dialog history and it does not use precedent queries and responses to determine the context of the current query. For example, if user asks two consecutive questions "*Who is the lecturer of the KO course?*" and "*What other courses does he teach?*", the system is not able to correctly answer the second question, because it does not connect the anaphoric reference "*he*" to its meaning in context of previous question, "*the lecturer of KO course*", and the respective response data, the concrete Teacher entity.

Secondly, it does not use other potentially available context information, such as the identity of the user or the time of the request. The questions referring to its sender (e.g. "*my schedule*", "*What exams do I take?*") or a relative time (e.g. "*tomorrow's TAL parallels*") are therefore out of scope.

Lastly, the system is limited by the KOSapi query mechanisms. It is fairly common for NLIDB systems to support *aggregate functions*, such as average, count, maximum, minimum or sum. However KOSapi does not have such a capability and NaLIDa is thus not able to correctly answer questions such as "*How many students attend PAL course?*" or "*the course with the most credits*".

Many NLIDBs also support joining data from multiple entities into a single result via JOIN operations [11]. The query parameters and the results of a a request to KOSapi is however bound to a single resource. NaLIDa is able to decompose a submitted query into multiple consecutive KOSapi requests so as to allow query criteria on multiple entities. The final results of the query are nonetheless data of a single entity type.

4.1.3. Query utterance types

The types of the supported input utterances are limited to non-polar questions and noun phrases. *Non-polar questions* (or wh-questions) are interrogative sentences that use an interrogative word such as "*when*", "*which*", "*who*" or "*how*", to indicate the information that the user desires. A *noun phrase* is a phrase which has a noun as its head word, or which performs the same grammatical function as such a phrase.

Other types of utterances, such as polar questions (i.e. yes-no questions) or declarative and imperative sentences usually do not produce sensible results or any results at all. The main causes for this limitation are that for such sentences

- the answer is not directly extractable from KOSapi without a further reasoning (e.g. yes-no answer for the polar questions),
- it is complicated to tell what information does such a sentence ask for (e.g. for declarative sentence) or
- the syntactic parser is not able to reliably process such a sentence (e.g. for imperative sentence, as explained in section 5.3).

Some possible ways to overcome these issues and extend the range of supported inputs are outlined in section 4.4.1. The examples 1 and 2 show the types of the supported and the unsupported utterances respectively.

(1) Examples of supported utterances

a. "*What is the capacity of the BDT tutorial?*" *wh-question*

b. "*the name of the PAH course*" *noun phrase*

(2) Examples of unsupported utterances

a. "*I want to know who teaches BDT course.*" *declarative s.*

b. "*Give me the emails of machine learning teachers.*" *imperative s.*

c. "*Are there any exam terms for AU course?*" *yes-no question*

4.1.4. Query constraints

Query constraints in KOSAPI are realized by RSQL expressions. As shown in section 3.3.1, RSQL is quite an expressive language that allows connecting a variety of comparison criteria into complex logical structure. For sake of simplicity, NALIDA uses only subset of this expressivity.

The constructed constraints have a form of a conjunction of attribute-value equality criteria:

$$att_1 = val_1 \wedge att_2 = val_2 \wedge \cdots \wedge att_n = val_n$$

This limitation allows for wide range of useful queries while significantly reducing the complexity of the syntactic and semantic analysis as well as the query generation. The limitation was establish based on the assumption that only a small fraction of input queries does not comply with it in practice. The assumption was confirmed by the results of the evaluation experiments presented in section 6.3.

Another limitation is that date and time criteria are not supported, because although they are possible in KOSAPI, their value must be in format `yyyy-MM-dd-Thh:mm:ss`, such as `2014-01-01T12:00:00`. As the wildcards are not supported for date attributes, their criteria must always specify exact second of an exact date. Such constraints do not have much practical use, even if we consider preprocessing of time phrases (such as "*noon on the first of January 2014*") into the given format. The date attributes are therefore regarded as string attributes without any recognized values.

4.1.5. Extragrammatical utterances

The system does not actively attempt to detect or correct erroneous inputs.

It is nonetheless tolerant to a wide range of syntactic errors in practice, such as dropped articles or prepositions, subject-object disagreement, capitalization errors and punctuation errors shown in the Example 3. This is possible due to fact that the semantic analysis uses only a small portion of the syntactic information contained in the utterance (see the sections 4.4 and 4.5) and some changes in the information thus do not affect its outcome.

Missing diacritics and spelling errors such as those in the Example 4, on the other hand, cause utterance interpretation to fail in many cases, because they prevent a successful lemmatization. The misspelled words are then ignored by the semantic analysis, which may not matter for entity or attribute words, but is almost always fatal for value words.

(3) Examples of supported extragrammatical utterances

a. "`courses Department Cybernetics`" *dropped articles/prepositions*

b. "`What do Jan Šedivý teach?`" *subject-object disagreement*

c. "`STUDENTS of bdt course`" *capitalization errors*

d. "`What courses are taught at the 13133 deptment?`" *misspelling*

(4) Examples of unsupported extragrammatical utterances

a. "`What does Jan Sedivy teach?`" *misspelling*

b. "`cuorses at the Department of Cybernetics`" *misspelling*

4.1.6. Input modality

Although the ability to process a spoken input is considered a prerequisite for a practical success of NLIDBs, majority of research in the field is limited to textual requests [2]. Historically, this was probably caused by the poor performance of ASR system at the time, but also by relative separability of the tasks - output of an ASR can be simply used as input of a NLIDB.

In any case, this thesis honors the tradition of text-based NLIDBs and leaves the handling of other modalities to client applications. There are general-purpose ASR tools available on most of the thinkable platforms, but it is possible that these recognizers would not perform sufficiently well for some specialized knowledge domains and that it would be necessary to train a domain-specific language models.

4.1.7. Output modality

The output of the system is also textual. As already mentioned, the data are provided in the format in which they are retrieved from KOSApi. That means that they have a form of XML documents. For consistency, other responses, such as error messages and disambiguation messages, are also returned in XML.

4.2. Architecture

The system architecture as well as terminology used in this chapter is inspired by Precise system described in [10, 9]. In taxonomy of [2], it falls into the category of Intermediate representation languages. with the interpretations in the role of the intermediate representation.

NaLIDa is divided into several components depicted in Figure 2. The components are chained together to interpret and evaluate the natural language input. The *schema* and *lexicon* components hold metadata about the underlying database and the knowledge domain data. The *syntactic analysis* component accepts the text of the user input and performs its parsing and annotation using the information in Lexicon. The *interpreter* uses the annotations to semantically interpret the input. If the interpreter finds multiple valid interpretation, the user is asked to disambiguate them. The *query generator* then translates the interpretation into a database query that is executed to produce the final results. This functionality can be used directly or via the last component of the system, the REST API.

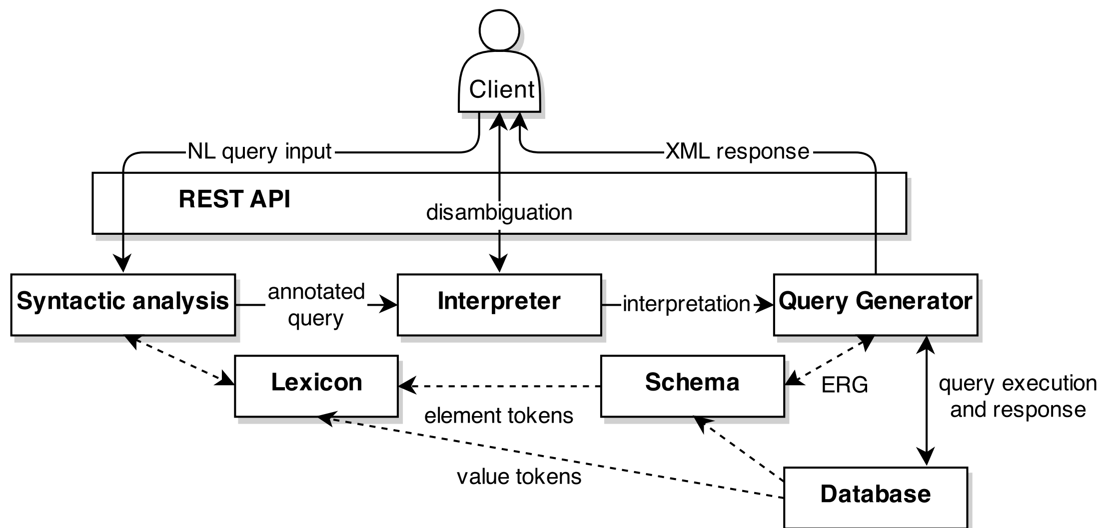


Figure 2. Architecture of NaLIDa components

4.3. Knowledge domain

The schema and lexicon components encapsulate the data specific for the underlying database and its knowledge domain. The other components use them to acquire information about the structure of the database elements and their mapping to the natural language words.

4.3.1. Schema

A *schema* is a structure representing the mutual relationships between elements E. The schema contains three types of elements: entities E_e, attributes E_a and values E_v. The attribute elements are further divided into primitive attributes E_{ap}, reference attributes E_{ar} and subresources E_{as}. A special singleton *wh-word element* e_w denoting an interrogative word is also present in the schema. Figure 3 shows a structure of a reduced schema.

An *entity element* (also called "relation" or "table" in the context of the relational databases) represents a knowledge base domain concept (such as Teacher, Course or Division).

An *attribute element* (also called "column" in the relational databases) belongs to a unique entity element and it represents a property of its entity (such as code, name or subdivisions). Because the name of an attribute can be ambiguous (e.g. code of Course and code of Division), attributes are identified also by their entity name (e.g. Course.code, Division.code). Each attribute of an entity may or may not be assigned a value or a collection of values.

A *primitive attribute* is an attribute whose value is of a primitive type, i.e. numerical ("integer"), textual ("string") or enumerative ("enum"). For each primitive attribute there is a corresponding *value element*.

On the other hand, a value of a *reference attribute* or a *subresource* is a reference to an entity. References are realized by foreign keys in relational databases and by URIs in REST services. Subresource is a concept specific to REST APIs. From the schema point of view, it is simply a different type of reference attribute with a collection of

values.

Two elements that are semantically related to each other are said to be *compatible*. A value elements are compatible with a corresponding attribute elements and also with the entity element that the attribute is assigned to. An attribute element is compatible with the its entity element. A reference attribute or a subresource is furthermore compatible with the entity element that it refers to. The wh-word element is compatible with every element.

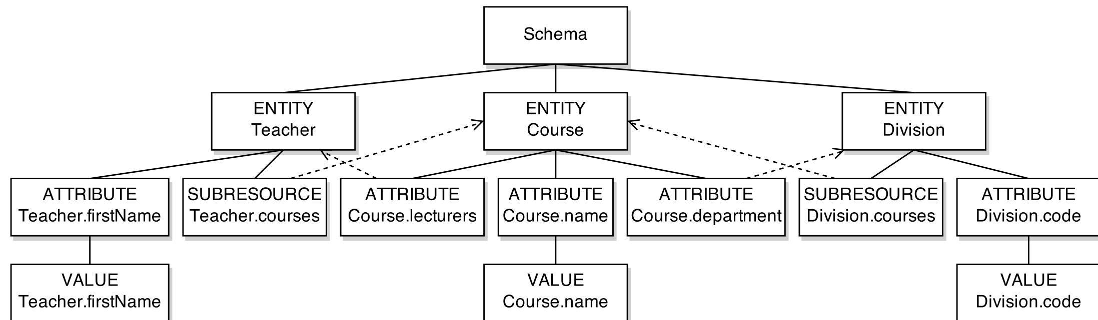


Figure 3. An example of an *element tree*

4.3.2. Lexicon

A lexicon is a structure that provides possible meanings for a given natural language word. It is a dictionary that maps word lemmata to tokens and tokens to elements.

A *lemma* is a canonical form of a lexeme, i.e. of a set of words forms with the same meaning. For instance, words `teach`, `teaches`, `taught` are forms of the same lexeme with lemma `teach`.

A token is a set of lemmata that *matches* an element. That means that words whose lemmatization correspond to the token may together represent the matching element. Multiple tokens may match the same element, and conversely, a token may match several different elements. For example, a token {`division`} matches entity element Division as well as attribute element Teacher.division and both tokens {`last, name`} and {`surname`} match a single attribute token Teacher.lastName.

More formally, given a set of all lemmata L, a set of all tokens $T \subset \mathcal{P}(L)$ [1] and a set of elements E, a lexicon provides two functions: [2]

$$tokens(l) : L \to \mathcal{P}(T)$$
$$tokens(l) = \{t \in T : l \in t\}$$

$$elements(t) : T \to \mathcal{P}(E)$$
$$elements(t) = \{e : t \text{ matches } e\}$$

4.4. Syntactic analysis

The syntactic analysis component takes the natural language input, and assuming that the input conforms to the constraints described in section 4.1, it annotates the input

[1] $\mathcal{P}(X)$ denotes the power set of a set X

[2] $f : I \to O$ denotes that f is a function of l with domain I and codomain O

with the syntactic information required for the semantic analysis, namely a parse tree and word token sets.

The first steps of the syntactic analysis are word segmentation, lemmatization and part-of-speech tagging. These annotations are not directly used by the other components, but they are prerequisites for the following parsing and tokenization. An example of the results of the individual steps of the analysis is shown in Example 5.

The *word segmentation* breaks the input stream of text into words using word delimiters such as whitespaces and punctuation, so that the individual words may be analyzed separately. Note that this process is also called tokenization in literature, but the term is used with a slightly different meaning in the context of this thesis. The *lemmatizer* then assigns a lemma $lemma(w_i)$ to each word w_i of the n-tuple w produced by word segmentation. A lemmatization of a word n-tuple w is therefore a n-tuple of lemmata of individual words: $lemma(w) = (lemma(w_1), \ldots, lemma(w_n))$

The *tokenizer* uses the lemmata and the lexicon to annotate each word with tokens that it might represent. Because tokens may consist of multiple lemmata, only those that are completely covered by the input utterance w are assigned to the words. The token set $T(w_i)$ of each word w_i is thus constructed as follows:

$$T(w_i) = \left\{ t \in T : t \in tokens(lemma(w_i)) \wedge t \subset \bigcup_{j=1}^{n} lemma(w_j) \right\}$$

The *part-of-speech* tagger assigns a parts of speech to each word, such as noun, verb, adjective, etc. These tags are used by the *parser* to create a parse tree that represents syntactic structure of the input utterance and provides the grammatical relations between its words.

The parser represents the parse tree by means of the Standard Stanford dependencies, which are collapsed dependencies with propagation of conjunct dependencies [21]. In the collapsed representation, dependencies involving prepositions, conjuncts, as well as information about the referent of relative clauses are collapsed to get direct dependencies between content words.

(5) Example of the syntactic analysis

a. "`What courses are taught by Šedivý?`" *input*

b. "`What`", "`courses`", "`are`", "`taught`", "`by`", "`Šedivý`" *segmentation*

c. "`what`", "`course`", "`is`", "`teach`", "`by`", "`Šedivý`" *lemmatization*

d. "`what`"/$WhWord$, "`courses`"/*Entity*:Course, "`are`"/∅, "`taught`"/*Subresource*:Teacher.courses, *Attribute*:Course.department, . . . , "`by`"/∅, "`Šedivý`"/$Value$:Teacher.lastName *tokenization*

e. "`What`"/WDT, "`courses`"/NNS, "`are`"/VBP, "`taught`"/VBN, "`by`"/IN, "`Šedivý`"/NNP *POS tagging*

f. "`What`" "`courses`" "`taught`" "`Šedivý`" *parse tree*

4.4.1. Extension of supported expressions

While the parsing of imperative sentences is usually not successful in the current configuration, there is an easily implementable workaround that would allow the translation of both imperative and declarative sentences. It lies in the detection of the imperative or the declarative part of the sentence using a rule-based pattern-matching. As shown

in the examples 6 and 7, removal of the detected part (in **bold**) produces a noun phrase or a wh-question that can be processed by the current system.

(6) Declarative sentences (DS) to noun phrases (NP) and wh-questions (WQ)
  a. **I need** the emails of the machine learning teachers. *DS → NP*
  b. **I want to know** who teaches the BDT course. *DS → WQ*

(7) Imperative sentences (IS) to noun phrases (NP) and wh-questions (WQ)
  a. **Give me** the emails of the machine learning teachers. *IS → NP*
  b. **Show me** who teaches the BDT course. *IS → WQ*

4.5. Semantic analysis

The purpose of the semantic analysis component called *interpreter* is to assign a meaning to the annotated input produced by the syntactic analysis. A valid interpretation of the input is found by producing all complete tokenizations of input words, then producing all interpretations hypotheses of these tokenizations and finally using the attachment function to filter the meaningless hypotheses out.

A *complete tokenization* is a set of tokens such that every word lemma of input appears in exactly one token. In another words, a complete tokenization is an *exact cover* of word lemmatization by tokens. The exact cover problem is known to be NP-complete [22], but the average size of our instances is so small that they can be solved by exhaustive search in a practically imperceivable time.

The construction of the interpretations and their filtering is performed in several stages. Firstly, only the *tokenizable words*, the words whose lemmata correspond to some token (i.e. $tokens(lemma(w_i)) \neq \emptyset$), are considered in this phase. This filters out the words to which the system is not able to assign a meaning, leaving m-tuple w' of tokenizable words. If no tokenizable words are left after this step (i.e. $m = 0$), the computation ends and a failure is reported to the user.

Tokenization hypotheses set T^H is a set of m-tuples of tokens constructed as m-ary Cartesian product of the token sets of the tokenizable words:

$$T^H : \mathcal{P}(T^m)$$

$$T^H = T(w'_1) \times \cdots \times T(w'_m)$$

An element of this set is a tokenization hypothesis t:

$$t : T^m$$

$$t = (t_1, \ldots, t_m) \in T^H$$

However, a tokenization hypothesis t created in this way is not guaranteed to represent a complete tokenization of w'. A set of unique tokens $\{t_1, \ldots, t_m\}$ is thus created from the interpretation hypothesis and if this set does not exactly cover w', the corresponding hypothesis is rejected. A set of all valid tokenization hypotheses T^V contains all tokenization hypotheses that represent a complete tokenization and are thus not rejected.

Next stage is the construction of interpretation hypotheses and their filtering by an attachment function. The *attachment function* A is a binary relation on the set of tokens derived from the parse tree as follows: $(t_i, t_j) \in A$ if and only if a corresponding

word w'_i is dependent on w'_j or vice versa, i.e. there is at least one of edges (w'_i, w'_j) and (w'_j, w'_i) in the parse tree.

An *interpretation hypothesis* is a possible assignment of a single element to each token in a valid tokenization hypothesis. It represents a way how to interpret an input utterance in terms of database elements. The set I of all interpretation hypotheses consists of all possible combinations of token-element matches of each valid tokenization hypothesis:

$$I : \mathcal{P}((T \times E)^m)$$

$$I = \bigcup_{t \in T^V} (\{t_1\} \times elements(t_1)) \times \cdots \times (\{t_m\} \times elements(t_m))$$

The intuition behind the filtering of interpretation hypotheses by the attachment function is that the words that are syntactically dependent are also related semantically and their semantic interpretations (i.e. their assigned elements) should therefore be compatible (in terms of compatibility defined in section 4.3.1).

An interpretation hypothesis $((t_1, e_1), \ldots, (t_m, e_m)) \in I$ complies with a semantic constraint $(t_i, t_j) \in A$ if elements e_i and e_j are compatible. An interpretation hypothesis is a *valid interpretation* if it complies with all semantic constraints in A.

The last phase of the hypotheses filtering is the most discriminative one. It discards all hypotheses that have higher than minimal interpretation size. The *interpretation size* is a number of unique entity elements featured in the hypothesis including those that are represented by its attribute or value elements and those that are referenced by attributes. This step can be regarded as an application of Occam's razor principle, the simplest interpretations are considered to be the best ones. The power of this rule is demonstrated in the Example 8. When more tokens matching multiple elements occur in an utterance, the hypotheses generation is affected by combinatorial explosion. While some of these hypotheses may be filtered out in previous phases, it often up to interpretation size filtering to do most of the work.

(8) Example of interpretation hypotheses filtering by interpretation size.
Interpretation hypotheses for tokens "*`username`*", "*`Jakub`*", "*`Stejskal`*"

a. Student.username, Student.firstName, Student.lastName — *1 entity,* ✓
b. Student.username, Student.firstName, Person.lastName — *2 entities,* ✗
c. Student.username, Teacher.firstName, Student.lastName — *2 entities,* ✗
d. Teacher.username, Student.firstName, Student.lastName — *2 entities,* ✗
e. Teacher.username, Teacher.firstName, Student.lastName — *2 entities,* ✗
f. Person.username, Person.firstName, Person.lastName — *1 entity,* ✓
g. Person.username, Teacher.firstName, Person.lastName — *2 entities,* ✗
...

The output of the interpreter is a set of the valid interpretations. If this set is empty, it means that the system was not able to interpret the input and a failure is reported to the user. If the set contains a single interpretation, the system proceeds to the query generation. If the set contains multiple valid interpretations, the user is asked to disambiguate them manually by choosing the correct one. The selected interpretation is then passed to the query generator.

4.6. Query generation

The *query generator* is a component that takes a single valid interpretation and translates it into a database query plan. The translation consists of two main phases. The first one is identifying the shortest path in a ER graph that connects the relevant entities. The second is a generation of a query plan using the identified path. While the first is general and independent on type of the underlying database, the second is DBMS-specific.

Two implementations of the query generators are presented in this section. The focus is primarily on the RSQL query generator described in section 4.6.6. A RSQL query plan devised by the generator can be executed against KOSApi to retrieve queried data. The SQL generator described in section 4.6.5 generates a plan comprising a single SQL SELECT statement. Because the system does not have means to directly access the relational database of KOS, a SQL query plan cannot be executed and it serves only as a demonstration of portability of NaLIDa to relational databases.

4.6.1. Projections, constraints and path

A query that corresponds to a valid interpretation $q = ((t_1, e_1), \dots, (t_m, e_m)) \in I$ can be described by a triple $(proj_q, constr_q, path_q)$, where the elements of the triple represent the projections, the constraints and the join path of the query.

The $proj_q$ is a set of entity or attribute elements e_i whose corresponding tokens t_i are attached to a token t_w corresponding to the wh-word element e_w:

$$proj_q = \{e_i \in E_e \cup E_a : (t_i, e_i), (t_w, e_w) \in q \wedge (t_i, t_w) \in A\}$$

If q does not contain a token corresponding to the wh-word element, i.e. input utterance is a noun phrase, $proj_q$ is set to the element corresponding to the root of the parse tree instead. The $proj_q$ must contain a single entity e_{proj} (called *projection entity*) or attributes belonging to a single entity e_{proj}. The projection entity element represents the database entity whose data (or their subset specified by the attributes in $proj_q$) is extracted from database as the final result of the query. The elements in e_{proj} are translated into the SELECT clause in the SQL generator or into XPartial parameter in the RSQL generator.

The $constr_q$ is a set of all token-value element pairs in q:

$$constr_q = \{(t_i, e_i) \in q : e_i \in E_v\}$$

They represent conditions that each database record must fulfill in order to be included in the results. They are translated into the WHERE clause in the SQL generator or into RSQL parameter in the RSQL generator.

And finally, $path_q$ is the shortest path in the entity-relationship graph that connects all entity elements belonging to the value elements in $constr_q$ (called *constraint entities*) and that ends in the projection entity element. The path describes how the individual entities of the query are connected with each other. In the SQL generator, the path is translated into the JOIN constraints in WHERE clause. In the RSQL generator, it is a basis for decomposition of the constraints into a query plan comprising individual consecutive REST requests.

(9) Example of an input, its interpretation and $(proj_q, constr_q, path_q)$ triple

a. "`What are the surnames and phones of the teachers lecturing KO course?`" *input query*

b. "*`what`*"/$WhWord$, "*`name`*"/*Attribute*:Teacher.lastName,
"*`phone`*"/*Attribute*:Teacher.phone, "*`teacher`*"/*Entity*:Teacher,
"*`lecture`*"/*Attribute*:Course.lecturers, "*`ko`*"/$Value$:Course.code,
"*`course`*"/*Entity*:Course *interpretation*

c. *Attribute*:Teacher.lastName, *Attribute*:Teacher.phone $proj_q$

d. "*`ko`*"/$Value$:Course.code $constr_q$

e. *Entity*:Course → *Attribute*:Course.lecturers → *Entity*:Teacher $path_q$

4.6.2. Entity-relationship graph

The *entity-relationship graph* (ERG) is a weighted directed graph representing mutual interconnectivity of the database entities through their reference attributes and subresources. The ERG is used in the query generation phase for finding an optimal path connecting the constraint elements and ending in the projection entity.

The ERG vertices $V(ERG)$ are all entity, reference attribute and subresource elements of the schema. There are two types of ERG edges $A(ERG)$ with different weights. A *direct edge* with weight w_D connects an entity and an attribute if a constraint on these elements can be realized within a single REST request. This includes connections between entity and its attribute, but also connections from a referred entity to a referring non-collection attribute (thanks to the dereferencing).

On the other hand, an *indirect edge* with weight w_I connects an entity and an attribute if a constraint on these elements requires two consecutive REST request to be performed. Such connections are between a collection reference attributes (including all subresources) and the referred entity.

The weights $w_U, w_I \in \mathbb{R}_+$ where w_U is slightly less than w_I to ensure that when two paths with equal number of edges are found, the one that can be realized in less REST request is considered to be shorter.

Given that:

- $ent(e_a)$ is the entity element to which an attribute element e_a belongs,
- $ref(e_a)$ is the entity element to which an attribute element e_a refers,
- $col(e_a)$ denotes that the attribute element e_a refers to a collection.

the vertices $V(ERG)$ and the weighted edges $A(ERG)$ are defined as follows:

$$V(ERG) = E_e \cup E_{ar} \cup E_{as}$$

For $\forall e_a \in E_{ar} \cup E_{as}$:

$$\left.\begin{matrix}(ent(e_a), e_a, w_D)\\(e_a, ref(e_a), w_I)\end{matrix}\right\} \in A(ERG)$$

$$\left.\begin{matrix}(e_a, ent(e_a), w_D)\\(ref(e_a), e_a, w_D)\end{matrix}\right\} \in A(ERG) \text{ unless } col(e_a)$$

4.6.3. Shortest path search

To find the $path_q$, a search is performed for the shortest path in the ERG that connects all constraint entities and end in the projection entity. Note that the path may contain entities that are neither constraint entities nor projection entity. For instance, the utterance "*`teachers of Jakub Stejskal`*" asks for teachers that teach courses attended by Jakub Stejskal. Even though that the entity Course is not explicitly mentioned in

the utterance, it must appear in the join path, as it connects the constraint entity Student with the projection entity Teacher.

Firstly, the ERG is reweighted is such way that an edge has weight 0 if it connects two elements present in the interpretation. This ensures that the paths covering the interpretation are preferred.

The Floyd–Warshall algorithm for finding all shortest paths in a weighted directed graph [23] is then used on the reweighted ERG to determine the shortest pairwise distance of the elements.

All possible orderings of the query entities are constructed by appending the projection entity to each permutation of the constraint entities. A cost of each ordering is computed by summing up the pairwise distances of its consecutive elements. The ordering with the minimal cost is then used to construct the join path $path_q$ by appending pairwise shortest paths between consecutive elements of the ordering.

Given the nature of the ERG edges, it holds that the vertices along the path $path_q$ are an alternating sequence of entity elements and attribute elements beginning with some constraint entity and ending with the projection entity, and that for each attribute element e_a on the path its neighboring elements are its $ent(e_a)$ and $ref(e_a)$ (in any order).

4.6.4. Query plan

A query plan is a sequence of consecutive queries, each taking the results of the previous one as an argument. The first query is executed without any argument and the output of the last query is the final result of the query plan. In this way, the system can handle a natural language query that cannot be carried out as a single database query.

Moreover, a query plan is agnostic to implementation of its individual queries, which means that queries of different types (e.g. using a different database or even DBMS) could be combined within one query plan. Such queries would only need to be compatible in terms of their inputs and outputs, which are represented as a collection of strings.

4.6.5. SQL query generation

The translation of a triple $(proj_q, constr_q, path_q)$ to a SQL query plan is rather straightforward. The query plan comprise a single query that represents a simple SELECT statement. A select statement consists of three clauses:

`SELECT` <*SELECT clause*> `FROM` <*FROM clause*> `WHERE` <*WHERE clause*>

The *SELECT clause* contains comma separated identifiers of the projections attribute elements from $proj_q$. If a projection is an entity element, a wildcard `.*` is appended to its identifier to denote that all its attributes are to be returned.

The *FROM clause* lists all tables that the query uses. It contains a comma separated list of all entity elements in $path_q$.

The *WHERE clause* lists of equality query constraints separated by conjunction literal `AND`. Apart from the constraints specified in $constr_q$, it also contains the constraints representing the join path $path_q$.

For each token-value pair $(t, e) \in constr_q$ an equality constraint is constructed that assigns the value of token t to the attribute identifier of the value element e. The form of the constraint depends on a type of value of the element. The constraints of string type use the pattern matching operator `LIKE`:

$$<name\ of\ e>$$ `LIKE '%`*<words of t separated by %>*`%'`

The constraints of other types, i.e. integer or enum, use equality operator:

<name of e>`=`*<word if t>*

For each edge $(e_i, e_j) \in path_q$ where $ent(e_i) \neq ent(e_j)$ a join constraint is constructed as follows, while appending a primary key identifier `.id` to the entity element names:

<name of e_i*>*`=`*<name of* e_j*>*

(10) Example of SQL query corresponding to interpretation in Example 9

```
SELECT Teacher.lastName, Teacher.phone
FROM Teacher, Course
WHERE Course.code LIKE '%ko%'
AND Course.lecturers=Teacher.id;
```

SQL query - SELECT statement

4.6.6. REST query generation

The generation of a REST query plan is more complicated due to fact that a request to a REST API is always bound to a single resource. A resource provides a representation of a single entity or a collection of entities of the same type and it can only be constraint by attributes of this entity or by attributes of the entities that are referenced by an attribute of this entity. If the join path includes multiple entities that are not directly connected, the corresponding query plan consists of multiple queries, each representing a requests to different resource of the underlying REST API. An intermediate query yields a set of result URIs that are used by subsequent query to produce its requests.

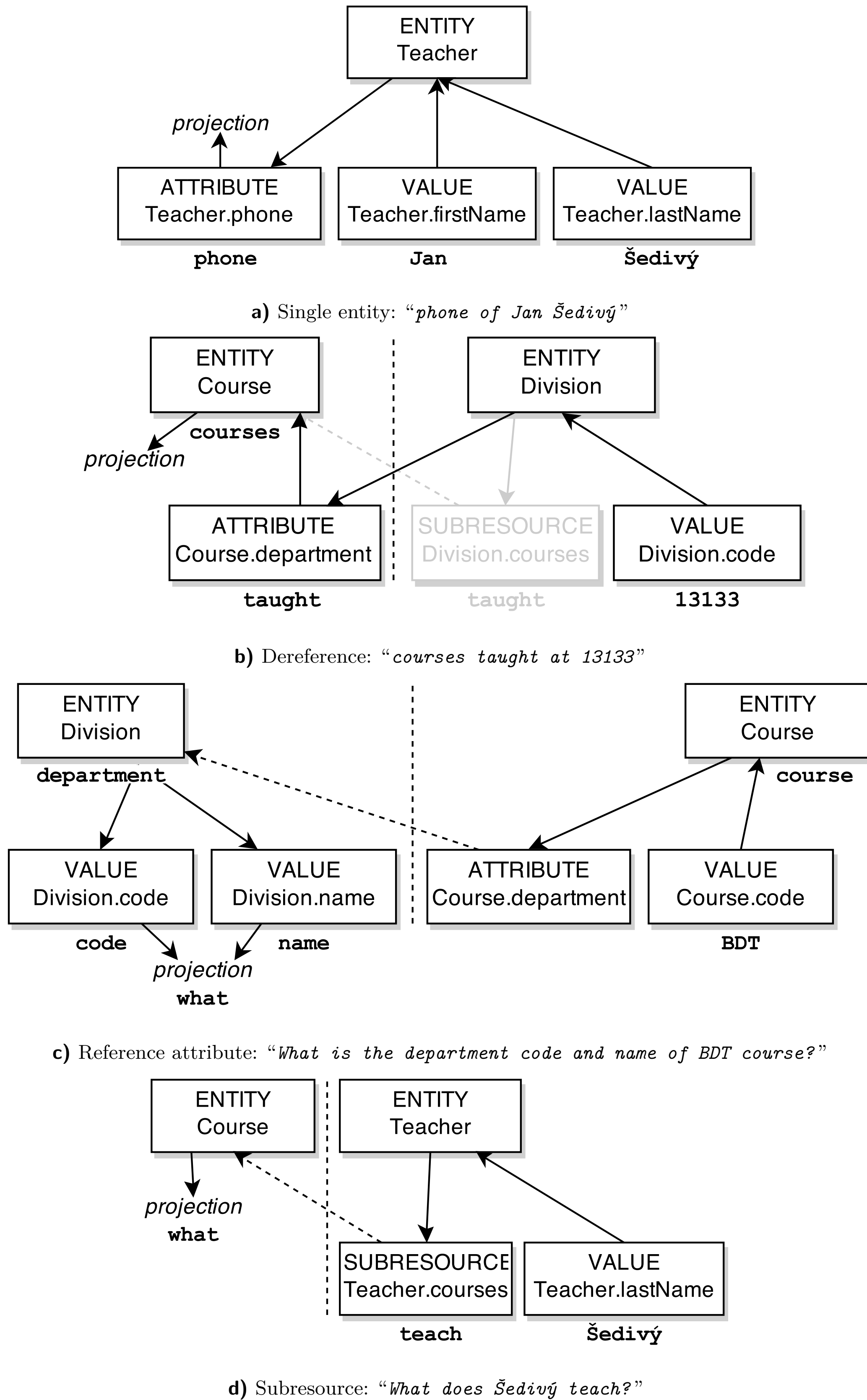
This section first demonstrates the translation of the different types of join paths segments to queries on concrete examples, then it describes the structure of the query objects and how they are derived from a triple $(proj_q, constr_q, path_q)$ and finally it explains how the actual REST requests are constructed from the query and how the query plan is executed.

Join path connection types

The Figure 4 depicts different types of connections on four input utterances and their interpretations. Each graph in the figure represents the triple $(proj_q, constr_q, path_q)$ of the respective utterance. The vertices of the graphs are the entity and attribute elements from $path_q$ and the value elements from $constr_q$. The vertices that are adjacent to special vertex *projection* are the elements from $proj_q$. The edges represent connections - solid lines for the direct connections and dashed lines for the indirect ones. The vertical dashed lines delimit the elements belonging to the different entities.

The first case is the simplest one, occurring when the constraints are on the projection entity. The utterance illustrated in 4a translates to single request looking essentially like this:
`/teachers?query=lastName==Šedivý;lastName==Šedivý&fields=content/phone`

The case in 4b shows an instance of a query that uses dereferencing to perform constraints on an entity different from the projection entity. This utterence is therefore also translated into a single query: `/courses?query=department.code=13133`. The figure also shows a grayed out alternative path. Even though this path would produce the same results, it contains indirect connection and thus needs two queries to be executed. That's why the first path is chosen by the shortest path algorithm instead.



a) Single entity: "*phone of Jan Šedivý*"

b) Dereference: "*courses taught at 13133*"

c) Reference attribute: "*What is the department code and name of BDT course?*"

d) Subresource: "*What does Šedivý teach?*"

Figure 4. Types of connections on the join path

The graph in 4c is an example of query plan containing an indirect reference attribute connection and also multiple projection attributes. Note that even though the connection comprises the very same elements as the 4b (Division, Course.department and Course) the connection is indirect in this case, because it has the opposite orientation $((e_a, ref(e_a), w_I)$ rather than $(ent(e_a), e_a, w_D))$.

All query plans containing indirect connections are realized by multiple queries, in this case two of them:

1. `/courses?query=code==BDT&fields=content/department`
2. `<id>?fields=content/code, content/name)`

The last graph in 4d is very similar to the previous one, but it is realized by different pair of queries, because it is a subresource connection. The first query extracts URIs of the Teacher records and the second one uses them to construct requests to subresource of the Teacher resource, Teacher.courses:

1. `/teacher?query=lastName==Šedivý;lastName==Šedivý`
2. `<id>/courses`

Query objects generation

A *query* is an object with four fields:

- *query.resource* is either an entity element or a subresource element and it identifies the requested resource.
- *query.projections* holds a set of attribute elements that defines what subset of entity representation should be projected to the query result.
- *query.constraints* holds a set of triples $(e_{ar}, e_c, t_c) \in E_{ar} \times E_p \times T$ that defines query constraint criteria.
- *query.collection* is a boolean value that denotes whether the resource represents collection or a single entity. It is `true` by default.

Each query is constructed from a subpath of $path_q$ delimited by edges representing an indirect connections (i.e. the edges of form $(e_a, ref(e_a), w_I)$).

The *query.resource* field is set according to the first vertex of the subpath. If the first vertex is a subresource element e_s, *query.resource* is set to e_s, if it is a reference attribute element, *query.resource* is set to empty value $\emptyset$, otherwise (that is for the first query of the plan) the entity element of the subpath is assigned to *query.resource*.

For the intermediate queries, the *query.projections* field points to the references that are to be passed to the subsequent query. In subresource queries the *query.projections* are not set, because the URIs of the subresource itself are the result. Similarly, the attribute elements are assigned to *query.projections* of a reference attribute queries. For the final query on the projection entity, the *query.projections* is set to $proj_e$.

The triples (e_{ar}, e_c, t_c) in *query.constraints* corresponds to all $(t_c, e_c) \in constr_q$ with e_c equal to the entities in the subpath. The first element of the triple, e_{ar}, is $\emptyset$ if e_c belongs to the query entity. If it does not belong to the query entity, it means that the entity of e_c is connected to the query entity via dereferencing of its reference attribute. The e_{ar} is the the reference attribute that connects query entity to e_c.

Rather then from the description of the assignments to each field, the process is probably more comprehensible from the point of traversing the query path and progressive construction of the individual queries. This process is illustrated in form of pseudocode in the Algorithm 1.

Algorithm 1 Generation of REST query plan

Require: $proj_q, constr_q, path_q$
Ensure: $plan$
  initialize empty $plan$, initialize new $query$
  for all (e_i, e_j, w) in $path_q$ **do**
    if e_i is entity **then**
      if e_j is reference attribute **then**
        if e_j belongs to e_i **then**
          set $query.resource$ to e_i if not already set
          set $query.projections$ to e_j
          add all $(\emptyset, e_c, t_c)$ where $(t_c, e_c) \in constr_q : ent(e_c) = e_i$ to $query.constraints$
          add $query$ to $plan$, initialize new $query$
        else
          add all (e_j, e_c, t_c) where $(t_c, e_c) \in constr_q : ent(e_c) = e_i$ to $query.constraints$
        end if
      else if e_j is subresource **then**
        set $query.resource$ to e_i if not already set
        add all $(\emptyset, e_c, t_c)$ where $(t_c, e_c) \in constr_q : ent(e_c) = e_i$ to $query.constraints$
        add $query$ to $plan$, initialize new $query$
        set $query.resource$ to e_j
      end if
    else if e_i is reference attribute **and** e_j is entity **and** e_i doesn't belong to e_j **then**
      set $query.resource$ to $\emptyset$ if not already set
      set $query.collection$ to `false`
    end if
  end for
  set $query.resource$ to e_{proj} if not already set
  add all $(\emptyset, e_c, t_c)$ where $(t_c, e_c) \in constr_q : ent(e_c) = e_{proj}$ to $query.constraints$
  set $query.projections$ to $proj_q$
  add $query$ to $plan$
  return $plan$

REST requests generation

Each query is realized by a set of requests to the underlying REST API. The fields of a query are compiled into a URI template with *id* parameter. In query plan execution time, the template parameter is supplied with a resource identifiers set from results of the precedent query to produce one concrete complete URI for each identifier. The URI templates have the following form:

$$\langle baseUri\rangle\langle id\rangle\langle resource\rangle\texttt{?fields=}\langle projections\rangle\texttt{\&query=}\langle constraints\rangle\texttt{\&}\langle defaults\rangle$$

The *baseUri* part is the URI of the REST API. It is static and same for all queries. For KOSAPI, *baseUri* is `https://kosapi.feld.cvut.cz/api/3/`.

The *resource* part corresponds to the elements in *query.resource* field. Each entity element keeps a resource name for this purpose, the element name is used for the subresource elements.

The *projections* part is a XPARTIAL expression corresponding to attributes in the field *query.projections*. Apart from these attributes, it also contains `title` (a short description of the records) and `link` (references to related resources). In the case of

queries on the collection resources (i.e. those with *query.collections* equal to `true`) the list of the projections is wrapped in `id,link,entry(...)`.

The *contraints* part contains a RSQL expression built from the content of the field *query.constraints*. It is semicolon separated list of criteria constructed from respective (e_{ar}, e_c, t_c) triples. If $e_{ar} \neq \emptyset$, a criterion starts with e_{ar} followed by a dot. The name of e_c separated by `==` from the value constructed from t_c then follows. For integer or enumerative attributes, the value is simply the t_c token as is. For string attributes, the words of t_c are wrapped and separated by wildcard `*`.

The $defaults$ part comprises other `&`-separated query string parameters ensuring that only relevant data are returned:

- `lang=en` and `multilang=false` - returns only English variant of multilingual data
- `limit=100` and `offset=0` - returns only first 100 results
- `sem=current,next` - returns only results relevant for the current and the next semesters
- `detail=1` - return the optional attributes (typically long texts)

(11) Example of REST query plan corresponding to interpretation in Example 9

```
https://kosapi.feld.cvut.cz/api/3/<id>courses/
?query=code==*bdt*
&fields=id,link,entry(title,link,content/lecturers)
&limit=100&offset=0&sem=current,next&lang=en&multilang=false&...

https://kosapi.feld.cvut.cz/api/3/<id>
?fields=content/lastName,title,content/phone,link
&limit=100&offset=0&sem=current,next&lang=en&multilang=false&...
```

REST query plan - URI templates

4.7. REST API

NaLIDa provides a RESTful application programming interface to the system functionality for client applications to use. The translation of natural language queries is provided in three variants, each having a separate resource: KOSapi response, SQL query and detailed intermediate outputs.

This section describes the HTTP requests that each resource accepts as well as the responses it produces. For details on response formats see section 5.7.2.

4.7.1. KOSapi response

The *KOSapi response* resource retrieves the result data corresponding to the submitted query by generating and executing a RSQL query plan.

resource URI: `GET /api/kos?q=<query>&t=<interpretationId>`

query parameters:

query (q) - required parameter, contains the natural language query to be translated.

interpretationId (t) - optional parameter, identifies the selected interpretation in the case that the query is ambiguous. Value -1 means that the ambiguity should be resolved automatically.

produces: application/xml, text/html

responses:

200 OK - query was successfully translated. The response contains either the result data or a list of possible interpretations in the case of ambiguity.

400 Bad request - no query was submitted or it was not translatable.
500 Internal server error - communication with KOSapi failed.

4.7.2. SQL query

The *SQL query* resource translates the submitted query into a SQL query plan.
resource URI: `GET /api/kos/sql?q=`*<query>*`&t=`*<interpretationId>*
query parameters: same as for KOSapi response
produces: application/xml, text/html
responses: same as for KOSapi response. The response contains a translation to SQL statement instead of the result data.

4.7.3. Detailed intermediate outputs

The *detailed intermediate outputs* resource provides a textual debugging log concerning RSQL and SQL query plan generation, along with intermediate results and the final XML result.
resource URI: `GET /api/kos/debug?q=`*<query>*`&t=`*<interpretationId>*
query parameters: same as for KOSapi response
produces: text/plain, text/html
responses: same as for KOSapi response. The response contains a text document with debug information including the submitted query, the selected interpretation, the translation to both RSQL and SQL query plans and the result data.

5. Implementation

The design introduced in the previous chapters has been implemented into a functional NLIDB system. This chapter covers the used technologies and its sections 5.1 to 5.7 describe the implementation details of each system component.

NaLIDa is divided into two modules, Core and Web. Both are written in Java programming language and use Apache Maven 2 to manage the project's build as well as its dependencies.

The project depends on a number of open source libraries to facilitate various subtasks. The Appendix B presents their detailed enumeration. Apache Common CLI validates and parses command line arguments, SnakeYAML parses database schema description, JGraphT builds a graph representation of the database schema and analyses it using graph algorithms, Stanford CoreNLP performs a syntactic analysis of the NL input, Google Guava is used throughout the application to ease the manipulation with collections and Jersey handles communication by means of RESTful Web services.

The Core module is a Java Standard Edition 7 project which can be used either directly as a standalone command line tool (taking as an argument a NL query or a path to file containing list of such queries) or as a library used by another Java application. This module consists of several components each handling a specific subtask from the schema configuration, NLP and interpretation to the query generation and result data retrieval.

The Web module is a Java Enterprise Edition 7 project providing an interface to Core module in form of RESTful Web service. Deployment of the application was tested on Apache Tomcat 7 web server, but it is expected to run seamlessly on most standard JEE application servers.

5.1. Schema

`Schema` consists of two data structures representing a layout of the underlying database, a *schema element tree* and an *entity-relationship graph*. Both are constructed upon the application initialization from the schema description, whose format is specified in section 5.1.3).

The classes comprising the schema are bundled in the `nalida.schema` package. This includes the `Schema` class and the `Element` class as well as all its subclasses.

5.1.1. Schema element tree

The *element tree* is a three-level tree structure with vertices of `Element` class. Abstract `Element` class has 5 subclasses: `Entity`, `Attribute`, `Subresource`, `Value` and `WhWord`. Their type and composition hierarchy is depicted in Figure 5. Apart from the references to its parent and a collection of its children, each element object has a list of tokens, that represent it in NL queries, and a unique name.

Each level of the tree consists of specific types of elements as follows:

1. Root of the tree is the `Schema` itself with `Entity` objects as children.

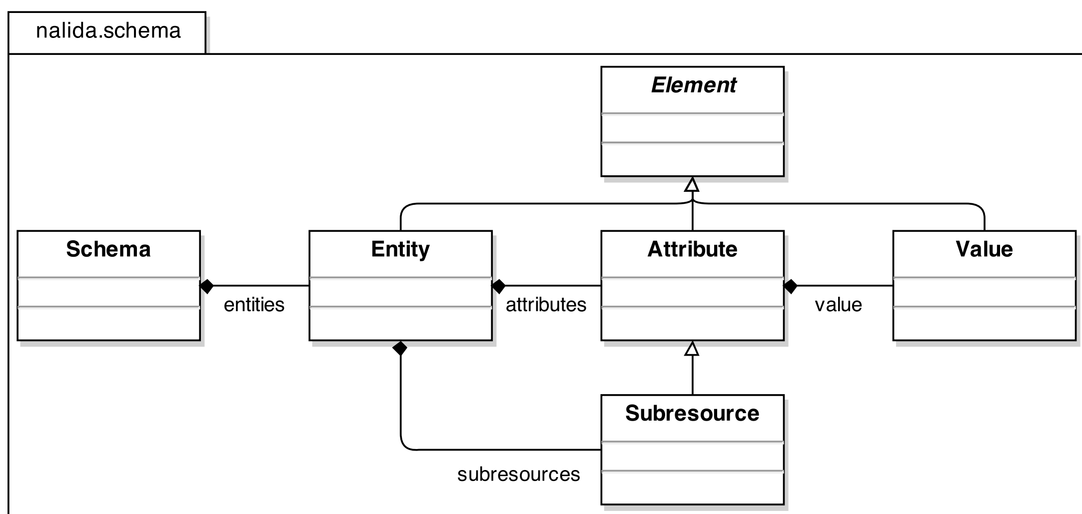


Figure 5. Class diagram of `nalida.schema` package

2. Each `Entity` has children of two classes: `Attribute` and `Subresource`. An object of `Attribute` class is either primitive (with string, integer or enumeration type) or a reference to an `Entity`, a `Subresource` is always a reference.
3. Each primitive `Attribute` has a single `Value` child.

Singleton object of `WhWord` type represents an interrogative word, not a database element, and is therefore not present in the *element tree*.

5.1.2. Entity-relationship graph

The entity-relationship graph is an object stored in a field of `Schema` object and it has type `DirectedWeightedMultigraph<Element, DefaultWeightedEdge>`, where `DirectedWeightedMultigraph` and `DefaultWeightedEdge` are classes provided by the JGraphT library. It is created from the element tree by a procedure described in section 4.6.2. The values of weights w_D and w_I are set to 1 and 1.1, respectively.

5.1.3. Schema description format

The *schema description* is a file in YAML format that completely defines the structure of the schema. YAML is a human-readable data serialization format. It relies on outline indentation for structure which makes it clutterless and comfortably readable and editable by a human.

The definition of the schema description is the first step in the system configuration, prior to the execution or deployment of the system. For REST APIs that are not defined in terms of machine-readable specification, such as KOSapi, the schema description must be defined manually by the system administrator, which makes the YAML format particularly suitable for this task.

There are many machine-readable formats for definition of REST APIs, such as WADL, RAML, or RSDL. Should the underlying REST API be defined in terms of such definition format, the schema description could be defined semi-automatically. However, because the target REST API of this thesis, KOSapi, does not have such

definition, the task of the automatic schema description generation is not addressed in this thesis.

The YAML schema description is a nested structure of associative arrays and lists. In the root of the hierarchy, there is an array of the of the records, `baseUri` and `entities`. Under `baseUri` key, there is the address of underlying REST API. In `entities`, there is a list of arrays defining the individual entities.

Each entity array contains these keys: `name`, `resource`, `tokens`, `attributes` and `subresources`. The values of `name` and `resource` are strings that specify the name and the resource URI of the entity. The list in the `tokens` enumerates token strings that can represent the entity in a natural language query. In `attributes` and `subresources`, there are lists of arrays specifying individual attributes of the respective type.

Each attribute is defined in terms of three keys: `name` and `tokens`, which are similar to the same records of entity, and `type`. Type of the attribute is denoted by a string in `type`. For primitive attributes, it contains either `string`, `integer` or `enum`. For reference attributes and subresources, it contains the name of the references entity, with `*` appended in the case of collection attributes.

Listing 5.1 Snippet of the schema description file

```
baseUri: https://kosapi.feld.cvut.cz/api/3/
entities:
  - name: Teacher
    resource: teachers/
    tokens: [teacher]
    attributes:
      - name:   email
        type:   string
        tokens: [email]
      - name:   division
        type:   Division
        tokens: [division, department, work, employ]
      - name:   supervisionPhDStudents
        type:   enum
        tokens: [supervision]
    subresources:
      - name:   courses
        type:   Course
        tokens: [teach]
  - name: Course
    resource: courses/
    tokens: [course]
    attributes:
      - name:   credits
        type:   integer
        tokens: [credit]
      - name:   instance/lecturers/teacher
        type:   Teacher*
        tokens: [lecturer, lecture]
  - name: Division
    ...
```

A snippets of the schema description file is shown in Listing 5.1. Note that the schema does not describe the KOSapi in its entirety, because some parts that does not contain information realistically queryable by natural language queries, such as entities Pathway or CoursesGroup, were omitted. Full entity-relationship model of KOSapi is depicted in Appendix C.

5.1.4. Value tokens

The schema description specifies tokens that can represent the given entity element or attribute element in the natural language queries. However, in order to successfully interpret an input, the tokens of value elements must be correctly interpreted as well.

Because there is often a great number of possible values for a given attribute, the value tokens are not stored directly in the schema description, but in separate files, one file listing all possible values for each primitive attribute.

For the very same reason, it is infeasible to create the value tokens files by hand. The system therefore includes `ValueExtractor`, a simple utility for automatic extraction of the value tokens from the underlying REST API. It takes the `Schema` object as an input, it constructs and executes a series of queries for each entity and then it saves the values of the results attributes to the respective files.

The resources exposed by KOSapi usually contain hundreds or thousands of records. The maximal number of results for one request is limited to 1000 records, but the requests with great number of results sometimes take very long or even cause time out. The records are therefore retrieved in batches of 100 using the `limit` and `offset` query parameters.

5.2. Lexicon

The single purpose of the Lexicon component is to provide mapping from a lemma to all the tokens it appears in. This is implemented by a simple lookup in a structure of type `Map<String, Set<Token»`.

During its initialization, the Lexicon traverses the schema element tree and creates a `Token` object for each token of each entity, attribute and value. Each lemma of the `Token` objects created in this way are then inserted into the map as a key pointing to the set of all corresponding tokens.

While the components described in following sections perform their tasks on per-request basis, the schema and the lexicon are loaded only once in the execution lifecycle of the application.

5.3. Syntactic analysis

Because the implementation of syntactic analysis tools such as tokenizer, POS tagger or parser is a complex task on its own and it typically requires a large training corpus of annotated data, an of-the-shelf general-purpose solution is used instead of creating a new one specifically for purposes of this project.

After considering several available NLP toolkits, Stanford CoreNLP was selected as it best meets the requirements of the system. Firstly, it performs all required syntactic analysis tasks out of the box and provides great accuracy and performance without a need of any configuration and customization. Secondly, it is well documented, designed to be usable with minimal effort and also implemented in Java, which means that

its incorporation into NaLIDa was a matter of five lines of code. CoreNLP is also highly extensible, which means that the NaLIDa-specific part of the syntactic analysis, namely matching the word lemmata to the tokens, could be easily integrated into its workflow so that all the results of the syntactic analysis can be accessed by Interpreter component in a uniform way.

The class `nalida.syntax.stanford.SyntacticAnalysis` encapsulates a pipeline of five annotators from CoreNLP toolkit: tokenizer, sentence splitter, POS tagger, lemmatizer and parser [24]. The pipeline also includes `SemanticAnnotator`, a custom annotator (called tokenizer in section 4.4), that uses Lexicon to assign a set of corresponding tokens to each lemma produces by CoreNLP lemmatizer.

5.4. Interpreter

The `Interpreter`, along with other related classes bundled together in the package `nalida.interpretation` (shown in Figure 6), implements the semantic analysis. `Interpreter` is an interface declaring a single method `Set<Interpretation> interpret(T annotatedLine)`, which takes an annotated line created by a syntactic analysis component and produces a set of valid interpretations. The interface is implemented by CoreNLP-specific class `StanfordInterpreter`, possibly the most involved class of the system, that uses the Stanford parse tree and annotations created by `SemanticAnnotator` to produce the interpretations, as described in section 4.5.

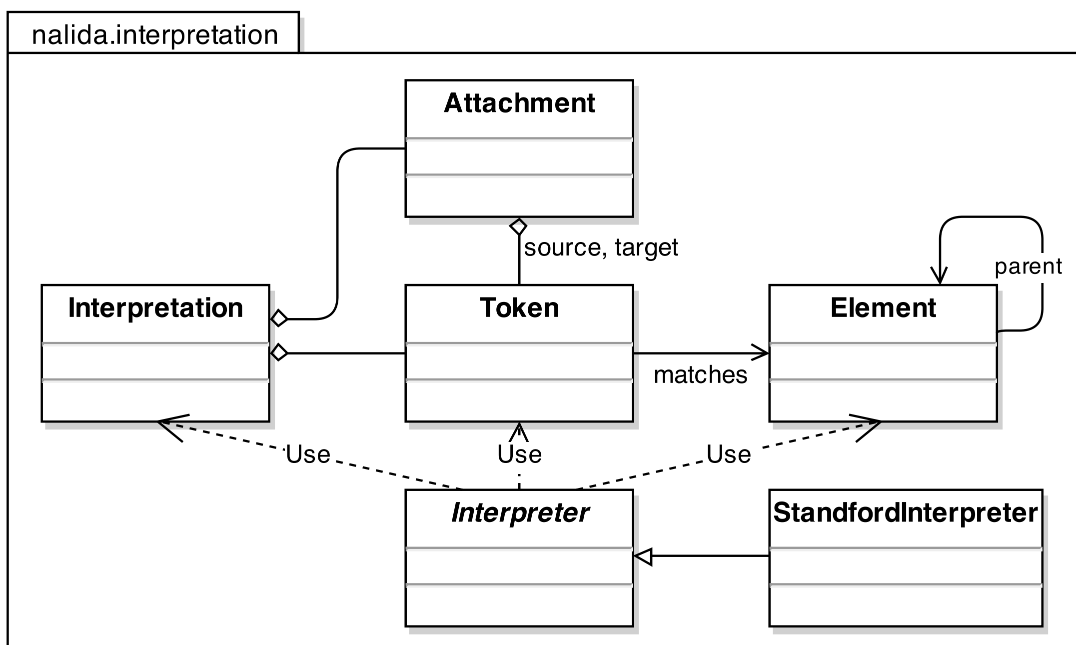


Figure 6. Class diagram of `nalida.interpretation` package

5.5. Query generator

Because there is practically no standardization in the field of REST API query languages, a NLIDB system that aims to be portable to various REST API must allow

for customization of the query generation module to every new REST API. NaLIDa furthermore considers the relational databases as a its potential DBMS platform.

These issues are addressed by concentrating as much of the query generator functionality (bundled in the `nalida.query` package) as possible into general purpose classes `QueryGenerator`, `QueryPlan` and `Query`, and extending them by concrete classes in packages `nalida.query.rest` and `nalida.query.sql` that implement the DBMS-specific parts.

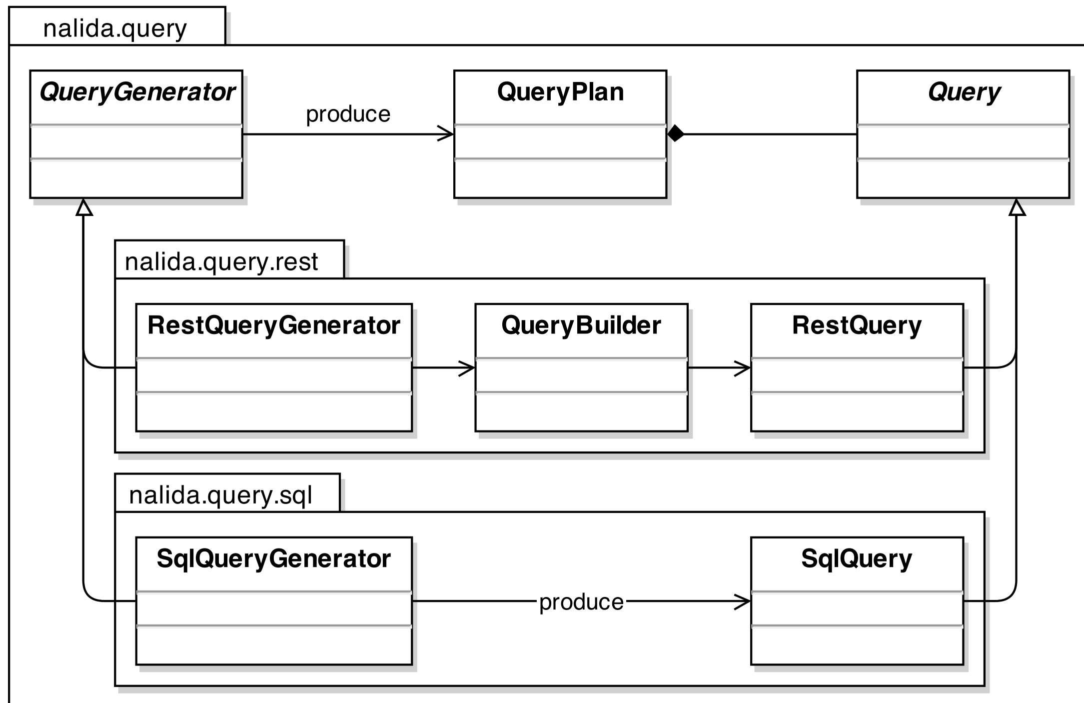


Figure 7. Class diagram of `nalida.query` package

As stated in section 4.6.5, the algorithm of `SqlQueryGenerator` is quite simple. It creates a `QueryPlan` with a single `SqlQuery` whose execution produces a SQL SELECT statement.

On the other hand, `RestQueryGenerator` implements the Algorithm 1 to build a `QueryPlan` consisting of a sequence of chained `RestQuery` objects. Because the `Query` objects are immutable, the generator uses `RestQueryBuilder` to progressively construct the queries. When a query plan is executed, the queries use Jersey Client API library to perform the HTTP requests to the underlying REST API. Successful HTTP requests return a set of XML documents, which are combined into a single result with the help of utility class for XML processing called `nalida.util.XmlParser`. In order to retrieve URIs that are to be passed to the subsequent query, the same utility class constructs an XPath expression from *query.projections* field and evaluates it on the XML result document.

5.6. Command line interface

Even though the main objective of this thesis is to provide interface to the system in the form of RESTful Web service, the ability to use the system via a command line is very useful for a number of use cases, such as testing and experimentation.

The NaLIDa CLI supports several options to be passed as input arguments as well as multiple ways to submit the input data. All the CLI capabilities are implemented in the `nalida.Main` class, which handles everything from the input parsing, NLP components orchestration, dialog management and the results retrieval and presentation.

For the arguments parsing, it uses the Apache Common Cli library, which provides easy, flexible and robust way to specify command line options. If the application is executed without any parameters, a default output shown in Listins 5.2 is printed that describes the purpose of the application and its supported options.

Listing 5.2 Default output of NaLIDa command line interface

```
Missing required option: [-f process and answer each query from file,
-e show example queries and their processing,
-q process and answer the query from argument]

usage: -example | -file <queriesListFileName> | -query <query>
 -d,--dry-run       translate a query without executing it
 -e,--example       show example queries and their processing
 -f,--file <arg>    process and answer each query from file
 -g,--graph         visualize entity-relationship graph
 -q,--query <arg>   process and answer the query from argument
 -s,--sql           translate a query into SQL instead of REST request
 -v,--verbose       prints out detailed information about what is being
                    done
```

A user can input a single natural language query into the `query` option or multiple queries by providing a path into the `file` option. The path must point to a plain text file with one query per line. The lines starting with `#` character are considered to be comments and they are not evaluated. To showcase its functionality without the need of providing a custom input, the application contains a list of sample queries, that can be executed by `example` option.

The remaining options can be set to modify the output of the application. By default, a RSQL query plan is executed and only its results are printed to the standard output stream. The `dry-run` option only prints out the query plan instead of executing it, the `sql` option translates the input into SQL command, the `graph` option visualizes the ERM graph (works only in graphical environments) and the `verbose` option prints the progress of the translation and several intermediate results instead of just the final result.

5.7. Web service

While all the components described in the previous sections are parts of NaLIDa Core module, the Web service that provides RESTful API to the formerly described functionality resides in a separate module called NaLIDa Web. Besides the resources described in section 4.7 the module also provides a HTML home page that serves as a user-friendly introduction to the NaLIDa project and API.

5.7.1. Home page

The home page resides on the root URI of the application (in the development environment, for example `http://localhost:8080/nalida-web/`). It is a single HTML page build with Twitter Bootstrap front-end framework that allows fast and effortless development of sleek and responsive web design. It also uses JQuery JavaScript library for AJAX calls and DOM manipulation

The appearance of the home page is shown in Figure 8. From top to bottom, it consists of navigation bar, introduction text, query forms and API description.

NaLIDa Home About GitHub

Welcome to NaLIDa a Natural Language Interface for Database

Ask a question from the domain of KOS, information system of CTU in Prague. NaLIDa knows stuff about students, teachers, divisions, courses, parallels and exams.

Check the enumeration of supported entities, attributes and their natural language tokens in XML format for inspiration. Click on the examples or submit your own natural language question in English. Use either a wh-question or a noun phrase.

KOSapi response	SQL query	Detailed log
Submit	Submit	Submit
auto-disambiguate	auto-disambiguate	auto-disambiguate
Examples	Examples	Examples
What is phone of Jan Šedivý?	What is phone of Jan Šedivý?	What is phone of Jan Šedivý?
Which teachers are from division 13133?	Which teachers are from division 13133?	Which teachers are from division 13133?

Figure 8. NaLIDa RESTful API Home page

The navigation bar provides anchors to individual page components as well as the link to the project repository. The introduction text briefly describes the purpose of the system, the available actions and supported inputs. The API description lists the provided resources and describes their properties similarly to how they are described in section 4.7.

The most important part of the page are the query forms. For each of the three resources, there is a form consisting of an text input field, auto-disambiguation checkbox, submit button and list of examples.

A user can write his natural language query into the textbox and send it for translation by pressing the submit button. The auto-disambiguation checkbox alters the behavior of the system in the case that the input is ambiguous and multiple valid interpretations are available. By checking the auto-disambiguation, the user sets the parameter `t=-1` and leaves it up to the system to decide which interpretation to evaluate. If the checkbox is unchecked, the XML representations of all valid interpretations are returned instead and the user can select the correct one to obtain the results. The

list of the examples consists of predefined input sentences that can be submitted by clicking on them.

The submitted requests are sent asynchronously to the REST API resources and their results are displayed in a modal window without the need of reloading the page.

5.7.2. REST resources

The RESTful API of NaLIDa is exposed at URI `/api` relative to the root of the application, for instance `http://localhost:8080/nalida-web/api`.

The implementation of the three resources is bundled in `nalida.webapp` package. Apart from the classes implementing each resource, `ResponseResource`, `SqlResource` and `DebugResource`, and their common abstract superclass `AbstractResource`, the package also includes classes for error handling and serialization. The resources are implemented in terms of Java API for RESTful Web Services (JAX-RS), a standard for defining REST resources by annotating plain Java classes, implemented by Jersey library.

Each resource provides response in its default format (XML or plain text) and also in HTML by wrapping the default format in HTML code. Examples of the XML response from `ResponseResource` and `SqlResource` are provided in Listings 5.3 and 5.4. The HTML representation is used by the asynchronous calls from home page.

Listing 5.3 XML serialization of KOSapi response

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
  <atom:feed xml:base="https://kosapi.feld.cvut.cz/api/3" ... >
    ...
  </atom:feed>
</results>
```

Listing 5.4 XML serialization of SQL response

```
<?xml version="1.0" encoding="UTF-8"?>
<sql>
  SELECT ... FROM ... WHERE ...;
</sql>
```

Listing 5.5 XML serialization of interpretations

```
<?xml version="1.0" encoding="UTF-8"?>
<query>What is the username of Jakub Stejskal?</query>
<interpretations>
<interpretation>
  <link href="/kos?q=What+is+the+username+of+Jakub+Stejskal%3F&t=0" />
  <tokens>
    [WH_WORD, ATTRIBUTE/Student.username,
     VALUE/Student.firstName, VALUE/Student.lastName]
   </tokens>
</interpretation>
<interpretation>
  <link href="/kos?q=What+is+the+username+of+Jakub+Stejskal%3F&t=1" />
  <tokens>
```

```
    [WH_WORD, ATTRIBUTE/Teacher.username,
    VALUE/Student.firstName, VALUE/Student.lastName]
  </tokens>
</interpretation>
</interpretations>
```

The class `AbstractResource` contains methods for the HTML wrapping as well as for the disambiguation of interpretations and their serialization. An example of a disambiguation response with interpretations serialized to XML is presented in Listing 5.5 Each interpretation in the XML is accompanied by the URI that identifies the resource representing the evaluation of the respective interpretation. The URIs differ from each other only in the values of the parameter `t`.

Several error states may occur in the system, such as invalid input or unavailable KOSApi. In order to propagate information about their origin to the client of the API, the system must be able to serialize the resulting exceptions into the format requested by the client. This is solved by wrapping the exception into custom `NalidaException` which encapsulates its information into a serializable `ErrorBean` object.

Listing 5.6 XML serialization of error message

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<error>
  <errorCode>400</errorCode>
  <errorMsg>Failed to translate query 'example of invalid input'.
    Try to reformulate it.</errorMsg>
</error>
```

6. Evaluation

Based on the research of the evaluation methods used by other NLIDB systems presented in section 3.1.1, a set of experiments was performed to measure the quality and performance of NaLIDa as well as to identify its shortcomings.

As this thesis focuses on development of an interface to REST APIs and KOSapi in particular, the system was evaluated on the configuration for the KOSapi knowledge domain. The testing corpus was collected from the university students and it concerns the data in the study information system.

Section 6.1 describes the testing data and how they were acquired, section 6.2 covers how the data were classified and evaluated as well as what was measured during the experiments and how, section 6.3 presents the measured results and discusses the findings.

6.1. Data collection

Because a part of the assignment of this thesis is to perform a testing with users, the corpus for evaluation of NaLIDa was collected from members of the potential user base, rather than being created by the author of the system.

Six student of the Czech Technical University were asked to provide a list of natural language queries. They were briefly introduced to the purpose of the evaluated system and they were provided with a document with instructions regarding the limitations described in 4.1. The document included following information:

- the supported entities with a link to the KOSapi documentation
- the supported utterance types
- the limitation of the knowledge base to the data in KOSapi

The respondents were also asked to try to cover the knowledge domain as much as possible. They submitted their queries “offline”, i.e. without directly interacting with the system.

Some of the students did not read the whole instructions thoroughly and they submitted many questions that do not comply to the defined constraints. While these utterances are inherently untranslatable by the current systems and thus provide no information in terms of translation accuracy, they are still utilized in this evaluation for a different purpose.

Invalid inputs are useful in the sense of modeling the behavior of a typical user. The experience says that many users simple do not read any documentation or instructions. By examining the input utterances provided by such users, we can learn important lessons about what the users expect from the system and how to deal with this discrepancy of the expected and the actually supported functionality.

The total number of natural language queries collected from the students was 100. This number is sufficiently large to be comparable with results of other NLIDBs and sufficiently small to allow manual evaluation.

6.2. Evaluation methodology

Before running the actual experiments, the collected corpus was divided into two groups based on whether the utterance represented a valid input to the system or not.

The invalid inputs were further classified based on what constraints, from those defined in 4.1, they violated:

- knowledge domain – queried information is not available in KOSAPI
- time criteria – utterance contains date attribute constraint
- comparison criteria – utterance contains comparison operator criteria such as "`less than`" or "`greater than`"
- aggregate functions – utterance requires aggregate functions keywords such as "`smallest`", "`biggest`" or "`average`"
- utterance type – utterance was neither wh-question nor noun phrase
- spelling mistakes – utterances contains misspelled or incorrectly written keywords

The grammatical errors described in section 4.1.5, such as missing articles or incorrect verb forms were not considered as invalidating.

Both the valid and the invalid inputs were evaluated using the command line interface of NALIDA. The program was set to produce and execute the REST query plans so that the well-formedness of the REST requests and the relevance of their answers could be validated. For the very same reason, the SQL query generator was not evaluated, as there was no simple way to empirically verify its results. If the interpreter returned multiple interpretations for disambiguation, the best one was selected manually. The correctness of the generated query plan and its results was also checked manually. In the case of the successful queries, various additional features were observed for each utterance:

- how many interpretations were returned for disambiguation
- how complex the selected interpretations were, i.e. how many entities were involved in each interpretation
- length of the query plan, i.e. number of its queries
- computation time of the translation and the execution

In the field of information retrieval with binary classification, the relevance of the results is expressed in terms of *precision* (the fraction of answers that are correct) and *recall* (the fraction of correct answers from all asked questions). In the case of ambiguity, a query is considered as correctly answered if it can be answered by one of the offered interpretations. However, a NLIDB system has three possibilities: to answer correctly, to answer incorrectly, or not to answer at all. An additional metric, called *willingness*, has been introduced to represent tendency of the system to actually provide answers to queries [5]. The definitions of all three terms follow.

$$precision = \frac{correct}{correct + incorrect}$$

$$recall = \frac{correct}{correct + incorrect + unanswered}$$

$$willingness = \frac{correct + incorrect}{correct + incorrect + unanswered}$$

6.3. Results

The results of the evaluation are presented in Table 1. Most importantly, the experiments have shown that the translation succeeds in 85.9 % of cases for the valid inputs and in 61.0 % for the whole test set including the invalid inputs. The invalid inputs constituted 29.0 % of the test set.

	absolute number			percentage [%]		
	valid	invalid	total	valid	invalid	total
unanswered	8	12	20	11.3	41.4	20.0
incorrectly answered	2	9	11	2.8	31.0	11.0
partially answered	0	8	8	0.0	27.6	8.0
correctly answered	61	0	61	85.9	0.0	61.0
Σ	71	29	100			

Table 1. Number and percentage of inputs by validity and correctness of results

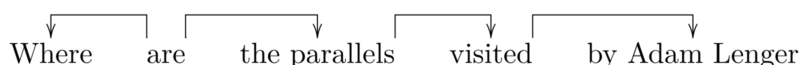
The translation of a valid input did not lead to the correct query plan and its answer in 10 cases. In principle, there are 3 points of the process where the translation may fail: the syntactic analysis, the interpretation and the join path finding in query generator.

The examination of the translation process of the concerned inputs revealed that only a single input failed in the query generation phase. The input "*`teachers of parallels in room T2:A4-202a`*" was correctly parsed and interpreted, but it failed to find a link between the entities Room and Parallel, because there simply is none. KOSapi does include reference attribute link from Parallel to Room, but it is part of its inner entity TimetableSlot and therefore does not allow dereferencing. In another words, the said query cannot be answered with current capabilities of KOSapi (this could however change soon, see section 7.1).

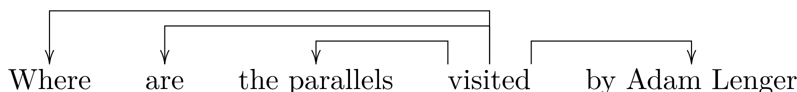
The other 9 inputs failed in the syntactic analysis phase. For instance, the input "*`What is the number of occupied seats of YO1ALG course?`*" was not correctly translated, because the phrase "*`the number of occupied seats`*" was not recognized as a token matching element Parallel.occupied. Every such concrete case can be easily solved by extending token set of corresponding element, but that is not a viable approach in principle, because it would require a continuous monitoring and manual editing of the schema description by the system administrator. This issue should be therefore addressed in the future research.

Another category of failures can be represented by the query "*`Where is the test for KO?`*". While it is evident to a person that the word "*`Where`*" refers to the room and that the question implicitly asks for Exam.room attribute, NaLIDa does not assigns this level of semantics to interrogative words. Another NLIDB system, Precise, addresses this issue by storing a list of interrogative words compatible with each attribute. Wh-words are then regarded as special kind of value elements belonging to all compatible attributes.

The last and the most prevalent cause of valid input translation failure is the incorrect parse tree. Consider the sentence "*`Where are the parallels visited by Adam Lenger?`*". The correct parse tree of this sentence is following:



However, the parser misinterprets the sentence and it provides following tree:



To explain the difference, the second interpretation can be paraphrased as "Where does Adam Lenger visit the parallels?" rather than "Where are the parallels that are visited by Adam Lenger?" Technically speaking, this is not a mistake on the parser side, because the second interpretation is also grammatically correct and even makes perfect sense. However, this prevents the correct projection token "`parallels`" to be identified and it makes the translation fail. To parse this type of questions correctly, the parser must be reconfigured, retrained or even replaced with a different parser, that prefers the correct interpretation.

The incorrectly answered inputs were divided into those whose answers were plainly wrong and those that produced answers that were close enough to the correct interpretation to be practically useful to the questioner. While these cases did not occur for the valid inputs, almost half of the incorrectly answered invalid inputs fell to this category.

One of such queries was "`Who's the reviewer of thesis with type B and with student rychtluk?`". In KOS, "`B`" stands for bachelor study programme type. However, in KOSAPI this flag is mapped to enumeration value `BACHELOR`, and because the enumeration attributes criteria do not support wildcards, the question would have to contain the exact value "`bachelor`" or "`BACHELOR`" to be recognized. That being said, the rest of the input query was interpreted correctly and its answer consisted of two records: the Bachelor's thesis and the Master's thesis of the student in question.

	valid inputs [%]	all inputs [%]
precision	96.8	77.2
recall	85.9	61.0
willingness	88.7	79.0

Table 2. Precision, recall and willingness for all inputs and the valid ones

From Table 2 that shows precision, recall and willingness, it stands out that the system is quite conservative in terms of providing a wrong answer to valid input. Indeed, its answer is incorrect in only 2.8 % of cases. The same is not true for the whole test set, because the invalid inputs produce an incorrect answer more often than no answer at all.

Figure 9 summarizes the other features of the translation measured for the valid inputs. Graph 9a shows that in more than 75 % of cases the disambiguation did not take place, because a single interpretation was selected automatically.

As for the rest, the only two cases where the experimenter had to choose between more than two options were really short utterances containing the word "`teach`", e.g. "`Who teaches Programming 1?`". Given the concerned domain, the word can describe various relations between many different entities. Furthermore, the small number of other tokens caused that the interpreter lacked the context to decide what entities were relevant.

Graphs 9b and 9c confirm that although, in theory, NALIDA is able to construct and answer complex queries traversing many entities and chaining several requests to KOSAPI, realistic inputs do not concern more than 3 entities and they are realizable in two consecutive queries or less.

The distribution of time that it takes to perform the syntactic analysis, interpretation and query generation is depicted on graph 9d. It shows that the vast majority of the valid inputs is translated in less than 150 ms, while taking 92 ms in average, which is fast enough to be perceived as an instant reaction by a user [25]. However, graph 9e shows that translation time is not that much of a concern, because the response time of a query plan execution against KOSApi ranges in units of seconds. This is partly caused by the fact that some queries comprise several HTTP requests, but even for a single request the response time varies wildly between different KOSApi resources. Still, the average time of the whole process, including the query processing and the answer retrieval, is 1624 ms for the valid input test set. Such performance is considered acceptable for the web-based applications.

Let us now focus on the input utterances that somehow violated the conditions for validity. Table 3 shows the distribution of invalid inputs to groups based on the violated constraints. Note that the numbers do not add up to the total number of invalid inputs, because the groups overlap and some inputs are thus in multiple groups.

violated constraint	number	percentage [%]
knowledge domain	10	31.0
aggregate functions	10	34.5
utterance type	6	20.7
spelling mistakes	6	20.7
time criteria	2	6.9
comparison criteria	1	3.5
Σ	29	

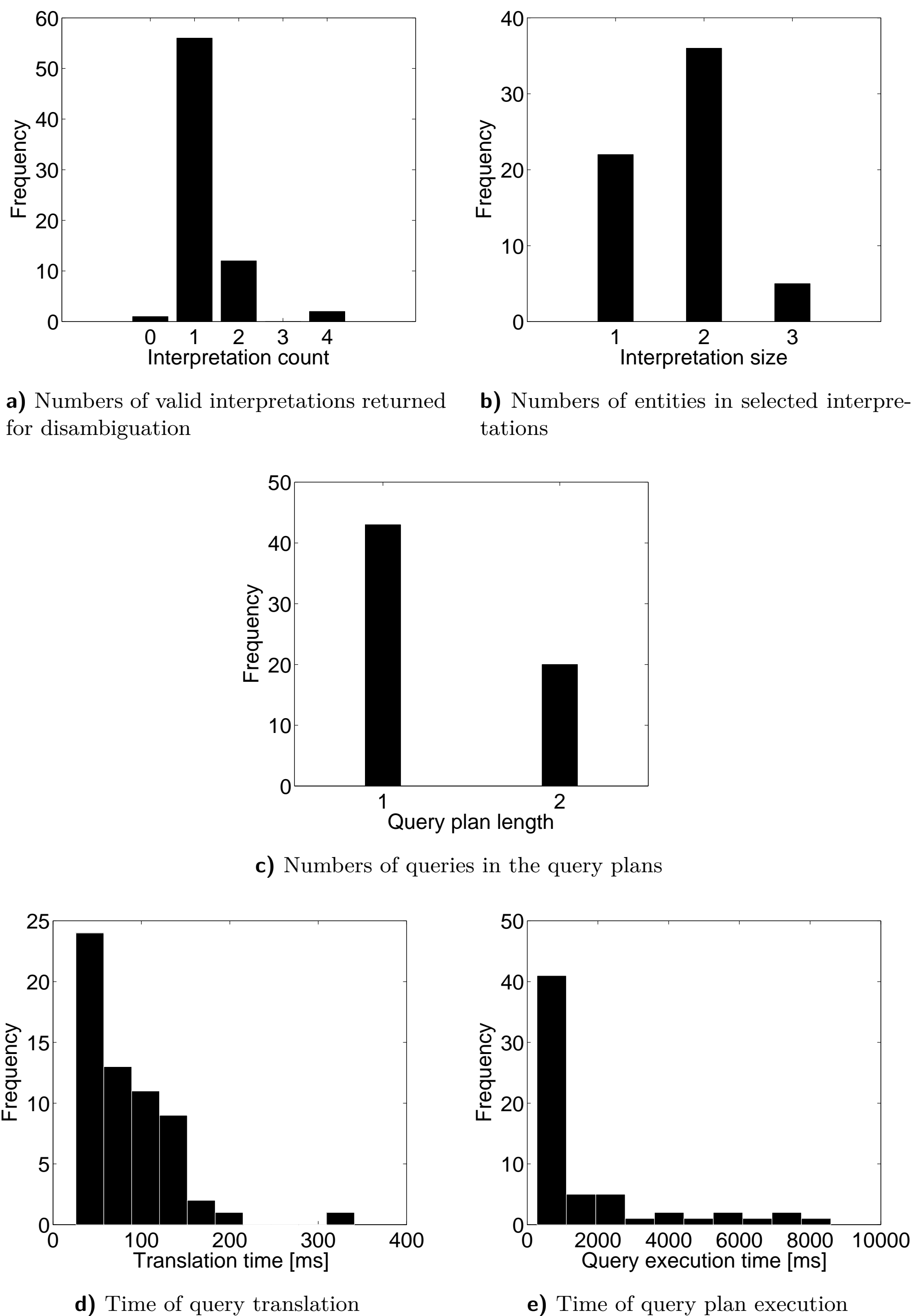
Table 3. Number and percentage of invalid inputs by violated constraint

The prevalent groups are the queries regarding information that is out of the knowledge domain and the queries using aggregate functions. The first group includes questions about domain information that is not available in KOSApi such as student's age or phone number, teacher's consultation hours and study department opening hours and personnel. While some the requested information is either irrelevant or confidential, the rest is just not included in the KOSApi interface. The knowledge domain would have to be extended to resolve these queries. This could be done on the NaLIDa level by using multiple databases in parallel, but given the nature of the missing information, a solution on the KOSApi level seems to be more reasonable, because such information may be useful for its other consumers as well.

The second group comprise the queries about the largest capacity, smallest department, total number of students with given name etc. Similarly to the first group, this issue can be resolved on both NaLIDa and KOSApi side, but again, the solution on the other side is more reasonable, in this case for the performance reasons. Executing a single SELECT count statement against the KOS database can be performed much faster than retrieving all relevant records to NaLIDa and post-processing them to compute the same result.

The nature of remaining groups was already discussed in previous chapters, namely sections 4.1.3, 4.1.4 and 4.1.5.

To conclude, we must stress that it is very tricky to compare the results of our experiments with others that were performed not only under different assumptions, on different knowledge domains and on different test sets, but also on a different DBMS architecture. That being said, we dare to say that the success rate of NaLIDa is well



a) Numbers of valid interpretations returned for disambiguation

b) Numbers of entities in selected interpretations

c) Numbers of queries in the query plans

d) Time of query translation

e) Time of query plan execution

Figure 9. Evaluation measurements for valid inputs

within the interval that is common for NLIDB systems. Roughly speaking, the success rate of other systems, usually reported in terms of recall, ranges between 80 % to 100 % [12, 11, 9].

More importantly, the experiments have provided a valuable insight to the nature of both the successful and unsuccessful queries. This knowledge may serve to determine the directions of the future research and development, but it may also serve to the developers of client applications to help them communicate the limitations of the system to the end users.

7. Conclusion

In this work, a natural language interface to database of the university information system KOS was proposed, implemented and evaluated. The resulting system is able to translate a natural language query into a set of requests to a RESTful Web service that encapsulates the KOS database.

The design of the system builds on the previous research in the field, but because the existing solutions predominantly assume direct access to the underlying relational database, their methods had to be adapted to cope with the granularity and limitations of the intermediate layer.

The exploration of the NLIDB methods and approaches for use in the context of RESTful application programming interfaces is thus the main contribution of this thesis.

The quality of the solution was assessed by experiments with a test set collected from the potential users. We may conclude that the system exhibits a good performance and accuracy for the queries from the targeted subset of the natural language. The unsuccessful translations of both the valid and invalid inputs were analyzed in order to determine the importance of the individual possible future research directions.

7.1. Future work

Several constraints were defined for the supported input utterances. While some of them were introduced simply to provide a realistic scope for this thesis, others were based on the technical limitations of KOSApi. During the writing of this text, these limitations were discussed with the author of KOSApi and he went so far as to initiate work on the removal of some of them. For instance, the dereferencing of collection attributes should be supported in near future. These changes will have to be addressed accordingly in NaLIDa.

The experimental evaluation revealed the most pressing shortcomings of the current solution as well as the most promising opportunities for the future improvements. Again, some of them are a matter of coordinated effort on both NaLIDa and KOSApi side, but others, such as spelling correction or comparison criteria, are solely in the scope of NaLIDa.

While this thesis focused primarily on the university domain and communication with KOSApi, the system was designed with the portability to different domains and database managements systems in mind. The success of this objective could be assessed by using the NaLIDa framework to create a NLIDB for a different domain.

Finally, in order for the system to be actually useful to end users, it must be integrated into an existing client application or a dedicated client must be developed.

Appendix A.

Contents of the enclosed CD

The CD-ROM enclosed to this thesis contains various data including this document in PDF format, source files of this document and of the implemented system, a guide describing the compilation, installation and execution of the system, and the data necessary to run the application.

The `schema` directory includes the schema description and the primitive attribute value files. The `example.txt` contains input utterances used when CLI application is executed with `-e` option. The `evaluation.txt` contains input utterances used for the experimental evaluation. Note that only the attribute values necessary to run the `example.txt` are included. For full-fledged NaLIDa experience, the value data must be extracted as described in `installation.txt`.

```
/
├─ stejskal-msc.pdf ........................ this document
├─ installation.txt ........ installation and usage guide of the system
├─ readme.txt ........... description of the structure of the distribution
├─ src
│  ├─ nalida-core ............ source files of the NaLIDa Core module
│  │  └─ data ........................ data files of the NaLIDa Core
│  │     ├─ schema ............ directory containing schema description
│  │     ├─ examples.txt .....queries used when CLI is executed with -e option
│  │     └─ evaluation.txt ................ queries used for evaluation
│  └─ nalida-web ............. source files of the NaLIDa Web module
└─ tex ........................ LaTeX source files of this document
```

Appendix B.

Dependencies

B.1. NaLIDa Core dependencies

groupId	artifactId	version
commons-cli	commons-cli	1.3-SNAPSHOT
org.yaml	snakeyaml	1.13
edu.stanford.nlp	stanford-corenlp	3.3.1
com.google.guava	guava	16.0.1
org.jgrapht	jgrapht-core	0.9.0
org.jgrapht	jgrapht-ext	0.9.0
com.sun.jersey	jersey-client	1.18

B.2. NaLIDa Web dependencies

groupId	artifactId	version
cz.cvut.fel.nalida	nalida-core	0.0.1-SNAPSHOT
com.sun.jersey	jersey-server	1.18
com.sun.jersey	jersey-servlet	1.18
com.sun.jersey	jersey-json	1.18
org.webjars	bootstrap	3.1.1
org.webjars	jquery	1.11.0

Appendix C.

Entity-relationship model of KOSapi

Figure 10. KOSAPI entity-relationship model

Taken form `https://kosapi.fit.cvut.cz/projects/kosapi/wiki/Er_model`.

Bibliography

[1] Jakub Jirůtka. “KOSapi – third version”. czech. Master’s Thesis. FEE CTU in Prague, 2013.

[2] L. Androutsopoulos. “Natural Language Interfaces to Databases - An Introduction”. In: *Journal of Natural Language Engineering* 1 (1995), pp. 29–81.

[3] *ProgrammableWeb*. Apr. 2014. URL: `http://www.programmableweb.com/`.

[4] William A. Woods, Ronald M. Kaplan, and Bonnie Nash-Webber. *The Lunar Sciences Natural Language Information System: Final report*. Tech. rep. 2378. Cambridge, MA: Bolt, Beranek, and Newman, Inc., 1972.

[5] Michael Minock. “C-Phrase: A system for building robust natural language interfaces to databases.” In: *Data Knowl. Eng.* 69.3 (Mar. 9, 2010), pp. 290–302.

[6] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2000.

[7] *A Novel Approach Towards Incorporating Context Processing Capabilities in NLIDB System*. Proceedings of the Sixth International Joint Conference on Natural Language Processing. Nagoya, Japan: Asian Federation of Natural Language Processing, 2013, pp. 1216–1222.

[8] Safwan Shatnawi and Rajeh Khamis. “An Approach for Developing Natural Language Interface to Databases Using Data Synonyms Tree and Syntax State Table.” In: *SCSS*. Springer, 2009, pp. 509–514. ISBN: 978-90-481-9111-6.

[9] Ana-Maria Popescu et al. “Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability”. In: *In Proceedings of the Twentieth International Conference on Computational Linguistics (COLING-04)*. 2004.

[10] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. *Towards a Theory of Natural Language Interfaces to Databases*. 2003.

[11] Neelu Nihalani et al. “An Intelligent Interface for relational databases”. In: *IJSSST* 11 (2000).

[12] Rodolfo A. Pazos Rangel et al. “A Domain Independent Natural Language Interface to Databases Capable of Processing Complex Queries.” In: *MICAI*. Vol. 3789. Lecture Notes in Computer Science. Springer, May 2, 2006, pp. 833–842. ISBN: 3-540-29896-7.

[13] *Web Services Glossary*. Tech. rep. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211. W3C, Feb. 2004.

[14] Jong Hyun Lim and Kyong-Ho Lee. “Constructing Composite Web Services from Natural Language Requests”. In: *Web Semantics* 8.1 (2010), pp. 1–13. ISSN: 1570-8268. DOI: `10.1016/j.websem.2009.09.007`.

[15] Alessio Bosca et al. “Composing Web Services on the Basis of Natural Language Requests.” In: *ICWS*. IEEE Computer Society, Jan. 12, 2006, pp. 817–818. ISBN: 0-7695-2409-5.

[16] Roy Thomas Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.

[17] T. Berners-Lee, R. Fielding, and L. Masinter. *RFC 3986, Uniform Resource Identifier (URI): Generic Syntax*. Ed. by Internet Engineering Task Force (IETF). Request For Comments (RFC). 2005. URL: `http://www.ietf.org/rfc/rfc3986.txt`.

[18] Jakub Jirůtka. "KOS API as a web service". czech. Bachelor's Thesis. FEE CTU in Prague, 2010.

[19] Mark Nottingham. *FIQL: The Feed Item Query Language*. Internet Draft draft-nottingham-atompub-fiql-00. Dec. 2007.

[20] Google. *Google Data APIs*. 2012. URL: `https://developers.google.com/gdata/docs/2.0/reference`.

[21] Marie-Catherine de Marneffe and Christopher D. Manning. *Stanford typed dependencies manual*. Sept. 2008. URL: `http://nlp.stanford.edu/software/dependencies_manual.pdf`.

[22] Richard M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[23] Robert W. Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

[24] Dan Klein and Christopher D. Manning. "Accurate Unlexicalized Parsing". In: *Proceedings of the 41st Meeting of the Association for Computational Linguistics*. 2003.

[25] Jakob Nielsen. *Usability engineering.* Academic Press, 1993, pp. I–XIV, 1–358. ISBN: 978-0-12-518405-2.