KNOWLEDGE REPRESENTATION AND PROCESSING METHODS IN SEMANTIC WEB

MASTER OF SCIENCE THESIS

JAROSŁAW WALISZKO

KRAKOW 2010

DEGREE PROGRAMME: APPLIED COMPUTER SCIENCE
SUPERVISOR: ANTONI LIGĘZA, PROFESSOR
DECLARATION OF ORIGINALITY

I hereby declare, aware of criminal liability for false declaration, that this thesis represents my original work. I am the sole author and did not use sources other than those referenced.

Jarosław Waliszko
I gratefully acknowledge the help and support of my advisor, Professor Antoni Ligęza. Without his assistance the work embodied in this thesis would not have been possible. Thanks are also due to my Parents for support and understanding.
# Contents

1 Introduction
   1.1 Goals ................................................................. 8
   1.2 Content ............................................................. 9

2 Theoretical background
   2.1 Semantic Web ....................................................... 10
      2.1.1 Background .................................................. 10
      2.1.2 Evolution .................................................... 11
      2.1.3 Outline ....................................................... 12
         2.1.3.1 KR languages and tools ................................. 13
   2.2 Towards ontology development .................................. 14
      2.2.1 Introduction ................................................. 14
      2.2.2 Reasons and advantages .................................. 14
   2.3 XML ................................................................. 16
      2.3.1 Introduction ................................................. 16
      2.3.2 XML syntax .................................................. 17
         2.3.2.1 Elements ............................................... 17
         2.3.2.2 Attributes ............................................. 18
         2.3.2.3 Others ............................................... 19
      2.3.3 Structure and content validation ......................... 19
   2.4 RDF ................................................................. 21
      2.4.1 Introduction ................................................. 21
      2.4.2 RDF syntax .................................................. 22
   2.5 RDFS ............................................................... 23
      2.5.1 Introduction ................................................. 23
      2.5.2 RDFS constructs ............................................ 24
   2.6 DL ................................................................. 25
      2.6.1 Introduction ................................................. 25
      2.6.2 DL Knowledge Representations Systems .................. 25
## CONTENTS

2.6.3 Description languages ................................................................. 27
  2.6.3.1 Syntax and semantics of $\mathcal{ALC}$ .................................... 29

2.7 OWL ................................................................................................. 30
  2.7.1 Introduction .................................................................................. 30
  2.7.2 DLs and OWL ................................................................................ 30
    2.7.2.1 OWL and reasoners .............................................................. 34
  2.7.3 OWL sublanguages ....................................................................... 34
  2.7.4 OWL 2 versus OWL 1 ................................................................. 35

3 Traffic danger ontology ................................................................. 37
  3.1 Introduction ...................................................................................... 37
  3.2 Knowledge engineering methodology for development of traffic danger concept ........................................ 38
  3.3 Ontology parts description .............................................................. 40
    3.3.1 Individuals .................................................................................. 40
    3.3.2 Properties ................................................................................... 41
    3.3.3 Classes ....................................................................................... 42
  3.4 Existential and universal restrictions ............................................. 44
  3.5 Open world assumption and closure axiom .................................... 44
  3.6 Reasoning ......................................................................................... 47
    3.6.1 Introduction ................................................................................ 47
    3.6.2 First example ............................................................................. 49
    3.6.3 Second example ......................................................................... 51
  3.7 Summary .......................................................................................... 53

4 Ontology-driven web system .......................................................... 55
  4.1 Introduction ....................................................................................... 55
  4.2 Ontology-driven software development ......................................... 56
    4.2.1 Agile methodology ...................................................................... 56
    4.2.2 Model-driven architecture .......................................................... 57
  4.3 System use cases ............................................................................. 58
  4.4 Interaction with system for updating and inferring data .............. 59
  4.5 System flow ..................................................................................... 60
  4.6 Database structure .......................................................................... 62
  4.7 Synchronization ............................................................................... 64
  4.8 System overview ............................................................................. 64
    4.8.1 Traffic danger board ................................................................. 64
    4.8.2 Trusted area authentication form .............................................. 69
    4.8.3 Traffic danger control panel ....................................................... 70
| Section                                                                 | Page |
|------------------------------------------------------------------------|------|
| 4.9 Technologies inside                                                | 72   |
| 4.9.1 Introduction                                                     | 72   |
| 4.9.2 Brief overview                                                   | 72   |
| 4.9.2.1 PostgreSQL                                                     | 72   |
| 4.9.2.2 Hibernate                                                      | 72   |
| 4.9.2.3 Java Server Pages                                              | 72   |
| 4.9.2.4 Spring MVC                                                     | 73   |
| 4.9.2.5 jQuery                                                         | 74   |
| 4.9.2.6 The OWL API                                                    | 75   |
| 4.9.2.7 HermiT                                                         | 75   |
| 4.9.2.8 Log4j                                                          | 76   |
| 4.10 Possible directions of development                                | 76   |
| 4.11 Summary                                                           | 76   |
| 5 Conclusion                                                           | 77   |
| A Traffic danger web system deployment                                 | 78   |
| A.1 Environment setup                                                 | 78   |
| A.1.1 PostgreSQL installation                                          | 78   |
| A.1.2 Java Runtime Environment (JRE) installation                     | 79   |
| A.1.3 Apache Tomcat installation                                       | 79   |
| A.2 Loading and configuration                                          | 81   |
| B Additional tools installation instructions                            | 83   |
| B.1 Protégé installation                                              | 83   |
| B.1.1 Editor installation                                              | 83   |
| B.1.2 Plugins installation                                             | 85   |
| B.2 Maven installation                                                | 85   |
| C Working with sources                                                | 87   |
| List of figures                                                        | 89   |
| List of tables                                                         | 91   |
| Bibliography                                                           | 92   |
Chapter 1

Introduction

1.1 Goals

The goal of this thesis is to take a closer look at progress of knowledge engineering in the field of Semantic Web. Along with theory of Knowledge Representation (KR) [47] and knowledge processing methods such as Description Logic (DL) [27], reasoning mechanisms and ontology modeling languages (OWL [48], RDF [42], RDFS [34]), the thesis shows the practical usage of the mentioned approaches in building systems driven by ontologies.

A working prototype of ontology-driven application, written in Java, has been developed within the scope of this thesis. The system main assumption is an attempt to integrate database and ontology approach, for storing and inferring desired information about domain of traffic dangers. For the needs of the system, domain model of traffic danger concept has been also designed. The ontology has been built using Protégé [15] editor integrated with description logic reasoner.

In presented solution, ontology domain description is supported by data stored in database. Location details of traffic conditions are stored in database, while clean abstract of traffic danger domain is described by ontology. Such integration results in dynamic deduction possibility of specific facts, using DL reasoning services, instead of using only static relations defined in database. Ontology-based approach gives the meaning to the information.
1.2 Content

Chapter 1 briefly outlines the goals and contents of the thesis.

Chapter 2 introduces the concept of Semantic Web, Knowledge Representation languages, Description Logics and reasoning mechanisms. Future goals of Semantic Web applications is shown along with explanation why it is worth to develop ontologies.

Chapter 3 explains the traffic danger concept. Ontology development process based on Protégé tool from Stanford Center for Biomedical Informatics Research is shown and described. Protégé and knowledge engineering approach used for ontology development is introduced.

Chapter 4 shows ontology-based reasoning system built as a proposal of knowledge processing application. The system works as a web application and integrates database approach with ontology-based approach for combining knowledge. It can dynamically infer answers for user defined questions in real time. Particular parts of the application are explained. Model-driven architecture along with agile methodology is introduced. Technologies used in building process are also briefly outlined.

Chapter 5 shortly summarizes the work.
Chapter 2

Theoretical background

2.1 Semantic Web

2.1.1 Background

Today’s World Wide Web has achieved a great success. WWW is widely used all around the world. Simple foundations, on which it is based, has given its strength and global community. Almost every user is able to create and publish hypertext resources in a simple manner. Basically "web content consists mainly of distributed hypertext, and is accessed via a combination of keyword-based search and link navigation" [37].

Notably, upon explosion of information amount accessible through the web, current web has encountered drawbacks. The huge amount of content required to be processed in order to find desired facts, has "highlighted some serious shortcomings in the hypertext paradigm" [37]:

- difficulties in finding correct information through simple searching and browsing mechanisms,
- problems of finding facts, which has common correspondents,
- irrelevant results after providing of more complex queries in the browsing engines,
- lack of semantics,
- lack of deduction facilities.

People are facing inconveniences in web content browsing. It is obvious, that content processing tasks are much more complicated when it comes to implement them into software agents. It is because technologies used for hypertext documents creation are designed with the intention of describing and presenting information in the human way. The lack of formal, logic-based information structure is the big issue for machines, which require mathematically-based way for knowledge processing. Here are the challenges, current web is going to face.
2.1.2 Evolution

Today’s World Wide Web gives new opportunities for discipline called Knowledge Representation (KR). Web resources need to be better adopted for automatic processing by computer agents. That is possible by more systematic and formal description mechanisms used for defining contents of the pages. What is more, Uniform Resource Identifiers (URIs), which are the naming scheme for web resources, allows KR systems to improve linking between semantic documents by avoiding the ambiguities of natural language. That is why new possibilities and challenges are opened for knowledge representation assumptions [47].

One of the most powerful features of the World Wide Web is the possibility of interconnections between knowledge sources. Interconnections are provided by linking mechanisms on pages. In other words, instead of copying information from external sources created by other people to our page, we just provide links to such sources.

Semantic Web can be treated as an extension to current web and the successor of Web 2.0. In recent Web 2.0 architecture, there is a lot of unstructured data. Information is represented in various ways, incompatible with each other. What Semantic Web offers is defined meaning of information, machine-readable description of contents, simplification of information exchange, classification and inference mechanisms, data processing and integration in machine-automated manner.

Figure 2.1 Web evolution from old Web 1.0 up to Semantic Web [32]
2.1.3 Outline

"The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

Tim Berners-Lee [29]

"The goal of Semantic Web research is to transform the Web from a linked document repository into a distributed knowledge base and application platform, thus allowing the vast range of available information and services to be more effectively exploited."

Ian Horrocks [37]

The Semantic Web is not about links between web pages, but about relationships between things (e.g. A is a part of B and Y is a member of Z) and about properties of things (e.g. size, weight, age and price) [17].

"If HTML and the Web made all the online documents look like one huge book, RDF, schema, and inference languages will make all the data in the world look like one huge database."

Tim Berners-Lee [28]

As it was said, the Semantic Web is a web that is able to provide information about things in the way computers can understand. The human-understandable sentences which describe things, are going to be intelligently processed in the Semantic Web.

Statements appropriate for dedicated language are built using syntax rules, which are defined by this language. In Semantic Web field statements are built using rules provided by specific KR languages and such statements are semantic aware. Semantic analysis of such statements allows to convert them to knowledge base queries. To search and access the Semantic Web content, tools called Semantic Web Agents or Semantic Web Services are required. It is because searching process in Semantic Web will be much different than today’s searching in free text - complicated math algorithms are going to find answers for us.

"The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries."

World Wide Web Consortium [23]
The Semantic Web layered architecture (Figure 2.2) covers such technological aspects like application interfaces, security, rules and logic (SWRL, RIF), ontologies (RDFS, OWL), metadata (RDF), serialization (XML) and resource identification (URI).

![Figure 2.2 Semantic Web Stack [43]](image)

### 2.1.3.1 KR languages and tools

The KR languages: RDF, RDFS and OWL explained in the next sections, are "*without question the most widely used KR languages in history*" [47]. They were created to capture and describe knowledge gathered by the web resources, and help applications to use these resources in an intelligent way. These languages are shaping the new standards in web development, allowing for transformation to the new standards. The Web 3.0 is slowly coming to light.

Along with the mentioned languages, development of new tools cooperating with them has appeared. Tools are designed for creating and maintaining taxonomies, and are supported by DL reasoning services (see Section 2.7.2.1). It gives the wide group of Semantic Web community new possibilities and motivation for creation ontology-based applications and experimenting with that technology. New challenges have appeared such as large scale OWL-based applications development.


2.2 Towards ontology development

2.2.1 Introduction

"In recent years the development of ontologies has been moving from the realm of Artificial-Intelligence laboratories to the desktops of domain experts" [44]. Many organizations and disciplines has developed standardized ontologies that domain experts can exchange and reuse in building their own ones. An ontology, "explicit formal specifications of the terms in the domain and relations among them" [33], specifies set of common vocabulary, researchers can share within a domain of knowledge. Ontologies have become increasingly important with the global move towards the Semantic Web. Parallel with Semantic Web evolution, knowledge defined by ontologies can be available on the web to a multitude of machines.

2.2.2 Reasons and advantages

The key reasons for developing an ontology [44] are following:

1. To share common understanding of the structure of information among people or software agents.
2. To enable reuse of domain knowledge.
3. To make domain assumptions explicit.
4. To separate domain knowledge from the operational knowledge.
5. To analyze domain knowledge.

Ad 1. Sharing common understanding of knowledge domain is one of the most critical reason for developing ontologies. For example, we can imagine several web sites containing specific technical information. If that information is published under common comprehensive ontology, the software agents will be able to aggregate and process knowledge from them, and infer exhaustive answers for user queries to that domain.

Ad 2. Coherent domain knowledge sharing, along with reusing of such knowledge can provide the possibility of injecting new aspects in existing set of information. This possibility, will provide complex description of specific knowledge structures to researchers all around the world, where one team of experts can supplement existing structure with new elements, with standardized vocabulary set. Along with automatic processing of knowledge, answering to questions about domain related topics, is going to be coherent and comprehensive.

Ad 3. Explicit domain assumptions provide clear description of domain knowledge, and simplify knowledge extensibility for domain. New researchers have the possibility of view to comprehensive domain related information structure, and while learning about that structure, have easier task in understanding what specific terms mean. Besides, in the world of software applications, explicit assumptions are much better than hard coded assumptions. Hard coded information along with all relations, would be much harder to find, and any changes would be impossible to execute, without programming expertise.

Ad 4. When it comes to separation, between domain knowledge and operational knowledge, we can talk about some kind of loose coupling between them. We can develop abstract set of assumptions in the
domain, and next use that metadata for providing specific, context-based information about such domain. For example domain experts can research an ontology which describe traffic dangers aspect, and users can provide specific traffic danger information for that underlying ontology, by defining concrete locations where specific traffic conditions can appear.

Ad 5. Analyzing domain knowledge is an obvious value on its own. It is critical when it comes to extend existing, or providing new branches of specific domain. Analysis should be based on comprehensive, structured source of knowledge, which an ontology is, and should provide an easy way of global view of concept along with description of all related terms used to define it.

Similar classification provided by The Knowledge Systems, AI Laboratory (KSL) at Stanford University [24] is following:

- to enable a machine to use the knowledge in some application,
- to enable multiple machines to share their knowledge,
- to help yourself understand some area of knowledge better,
- to help other people understand some area of knowledge,
- to help people reach a consensus in their understanding of some area of knowledge.

These particular points, in a bit different form, were described earlier. Ontology development aims, described by different sources, are convergent. Ontologies development seems to be crucial in the near future, because the size of human knowledge is going to be too wide for processing in the meaning of standard way.
2.3 XML

2.3.1 Introduction

The starting point of introduction to world of KR languages, should be brief description of XML language. It is because XML lies on the bottom of Semantic Web layered stack and provides the serialization for mentioned KR languages.

XML is widely used technology so information stored in variety of papers, e.g. [25, 30], can be easily found to help understand, what exactly is XML.

XML facts:

- XML stands for EXtensible Markup Language,
- XML is a markup language much like HTML,
- XML allows to represent tree-like structures,
- XML was designed to carry data, not to display data,
- XML tags are not predefined - custom tags have to be defined,
- XML is designed to be self-descriptive,
- XML is platform independent,
- XML files are stored as plain text files,
- XML emphasizes simplicity, generality, and usability over internet,
- XML is a W3C Recommendation.

XML is used in many various aspects of coherent data transmission through the web. In the real world incompatible data formats coexist in computer systems, and are completely incomprehensible to each other. XML provides common, interchangeable way of storing and sharing data. It is a software and hardware independent tool for carrying information. It is also used as the serialization mechanism for various, newly created internet languages like: XHTML, WSDL (describing Web Services), RSS (used for news feeds), RDF and OWL (describing resources and ontologies), SMIL (describing multimedia for the web), etc.
Below, there is a sample XML document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<message status="critical" releaseDate="13th April, 1970">
    <to>Manned Spacecraft Center (building 30), Houston, Texas</to>
    <from>Commander James A. Lovell, Apollo 13 crew</from>
    <heading>Technical Fault</heading>
    <body>
        Houston, we've had a problem.
        We've had a main B bus undervolt.
    </body>
</message>
```

First line of the file indicates, that it is version 1.0 of XML standard, and information is encoded with UTF-8 character encoding. File content declares a message sent on the 13th of April, 1970, from Apollo 13 spacecraft to mission control center, about technical issue connected with oxygen tank explosion.

This document is quite self-descriptive. It contains sender and receiver information, heading and the message body. Notably, that piece of code does not provide any functionality, besides carrying and structuralizing the data. It is just information wrapped in tags. There is still a necessity for writing software which will be able to process and respond, or display that information in more friendly way.

### 2.3.2 XML syntax

#### 2.3.2.1 Elements

The tags used inside XML documents, are not defined in any XML standard. XML allows developers to invent custom tags, because as opposed to HTML where tags are the part of the standard, XML has no predefined tags.

XML documents have to contain one root element. Structure of XML documents is the structure of rooted tree. The tree starts at the root and branches to the lowest level of the tree. A rooted tree is a connected, acyclic, undirected graph, in which one of the vertices is distinguished from others. The distinguished vertex is called the root of the tree [46].
<root>
  <firstChild>
    <subChild>.....</subChild>
  </firstChild>
  <secondChild>
    <subChild>.....</subChild>
  </secondChild>
</root>

An XML element is everything from (including) the element start tag to (including) the element end tag. Elements can contain other elements, plain text or a mixture of both. Each XML tag have to close previously opened tag, in the correct order. It is illegal to have unclosed tag in XML document. All elements must be properly nested. What is more, XML is case sensitive.

<p>incorrect - no closing tag</p>
<p>correct</p>

<p><b>incorrect - improperly nested elements</b></p>
<p><b>correct</b></p>

<p>incorrect - tags are not matched in the manner of case sensitivity</p>
<p>correct</p>

2.3.2.2 Attributes

XML elements can have attributes. Attributes provide additional information about elements. Attribute is a simple key-value pair, in the format of key="value". Attribute values have to be quoted.

<message status="critical">.....</message>

There are some problems with attributes. In opposite to elements they cannot contain multiple values, as well as tree structures. They are not easily expandable for future changes.

- Attributes should be used for providing information, which is not relevant to the data itself. They are excellent for storing metadata (data about data) (such as ID of message, etc.).
- For storing data itself, elements are much better choice than attributes.
2.3.2.3 Others

XML supports comments. The syntax for writing comments in XML is similar to that of HTML:

`<!-- This is a comment -->`

Some characters in XML Standard have special purpose. It means, that XML parsers try to interpret such characters in a special way. To insert such a character in the XML document without its meaning, predefined entity references should be used:

| &lt;    | <  | less than |
| &gt;    | >  | greater than |
| &amp;   | &  | ampersand |
| &apos;   | '  | apostrophe |
| &quot;  | "  | quotation mark |

Table 2.1 Predefined entities in XML 1.0

As opposed to HTML, white-space characters in XML document are NOT truncated. When it comes to storing new lines, XML behaves in the same way as Unix/Mac applications, and uses line feed character (LF).

XML documents can follow a set of strict rules, to store information in a predefined format. If they follow such rules, they are valid. Such predefined data structure is critical during parsing of XML documents by various applications. It helps applications to properly interpret specific information, because when transporting data, it is essential that both sender and receiver have the same "expectations" about the content. If XML document doesn’t follow its schema, it can be rejected by application as invalid.

2.3.3 Structure and content validation

The standard defining syntax rules which should be followed by XML documents is called Document Type Definition (DTD). It describes the structure with a list of legal tags and their attributes. DTD can be embedded within XML file:
```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message [ 
<!ELEMENT message (to, from, heading, body)> 
<!ATTLIST message status (spam|normal|critical) "spam"> 
<!ATTLIST message releaseDate CDATA #REQUIRED> 
<!ELEMENT to (#PCDATA)> 
<!ELEMENT from (#PCDATA)> 
<!ELEMENT heading (#PCDATA)> 
<!ELEMENT body (#PCDATA)> ]>
<message status="critical" releaseDate="13th April, 1970">
  <to>Manned Spacecraft Center (building 30), Houston, Texas</to>
  <from>Commander James A. Lovell, Apollo 13 crew</from>
  <heading>Technical Fault</heading>
  <body>
    Houston, we’ve had a problem.
    We’ve had a main B bus undervolt.
  </body>
</message>
```

or can exist as a separate file, which can be referenced from the XML document, as marked out below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message SYSTEM "message.dtd">
<message status="critical" releaseDate="13th April, 1970">
  <to>Manned Spacecraft Center (building 30), Houston, Texas</to>
  <from>Commander James A. Lovell, Apollo 13 crew</from>
  <heading>Technical Fault</heading>
  <body>
    Houston, we’ve had a problem.
    We’ve had a main B bus undervolt.
  </body>
</message>
```

More powerful XML-based alternative to DTD is called XML Schema. The migration from older DTDs to XML Schema includes following benefits:

- Schemas support primitive (built-in) data types (e.g. `xsd:integer`, `xsd:string`, `xsd:date`, etc.) and data namespaces.
- Schemas are extensible. Schemas gives possibility of defining custom data types, using object-oriented data modeling principles: encapsulation, inheritance and substitution.
- It is possible to reuse them in other Schemas, create new data types derived from the standard types and reference multiple schemas within the same document.
• Schemas are compatible with other XML technologies (Web Services, XQuery, XSLT, etc.).

Sample schema shown below, can be used as an alternative to DTD introduced earlier, for defining the syntactic correctness of XML document also discussed earlier:

```xml
<xs:element name="message">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="status" type="xs:string" default="spam"/>
    <xs:attribute name="releaseDate" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

2.4 RDF

2.4.1 Introduction

If we take a look once again at Semantic Web Stack, we can see that just above XML lies Resource Description Framework (RDF). RDF is a simple assertional language used for describing resources in the World Wide Web. RDF is intended for situations where information is processed in automatic manner by applications, rather than being displayed to users. It is designed to represent information in the form of triples, such as subject, predicate and object. RDF predicates can be treated as attributes of resources. Sample subject-predicate-object (object-attribute-value) triple building block is shown on the directed graph below:

![Figure 2.3 Example of RDF statement](image)

J. Waliszko  Knowledge representation and processing methods in Semantic Web
Resources refers to things being described. Resources are identified by URIs. The binary relations between resources are called properties. RDF properties themselves also have URIs, to be more accurate in the identification of the relations between resources. Statements are created as the asserted properties of resources.

2.4.2 RDF syntax

RDF provides XML-based syntax, called RDF/XML. A piece of RDF/XML code written below corresponds to the graph in Figure 2.3:

1. `<?xml version="1.0"?>`
2. `<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:agh="http://agh.edu.pl#">`
3. `<rdf:Description rdf:about="http://agh.edu.pl#Student">`
4. `<agh:hasName>"Jarek"</agh:hasName>`
5. `</rdf:Description>`
6. `</rdf:RDF>`

Line numbers are added to the example. It is because they are referenced in the explanation part below:

- **Line 1** contains standard XML declaration `<?xml version="1.0"?>`, which indicates that the content of that file is XML, and provides XML version which is used inside.
- **Line 2** starts from tag `rdf:RDF`, which indicates that the following XML content syntax is RDF. The `xmlns:rdf` defines a namespace identified by the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, and tells that all tags prefixed with `rdf:` are parts of the namespace. That namespace is used for terms from RDF vocabulary.
- **Line 3** defines another prefix `agh:`, which represents namespace `http://www.agh.edu.pl#`. URI `http://agh.edu.pl#` is used for vocabulary terms defined by organization agh.edu.pl.
- **Lines 4-6** provide RDF/XML code, which describes the statement shown at graph 2.3. **Line 4** begins from tag `rdf:Description` which indicates the start of description of a resource. Next, using attribute `rdf:about`, identifies the resource the statement is about (the subject of the statement), by providing its URI `http://agh.edu.pl#Student`. **Line 5** declares property element `agh:hasName` for the subject resource, where both predicate and object of the statement are represented. The value of the property identified by namespace `http://agh.edu.pl#hasName` is a plain literal `Jarek`. **Line 6** closes `rdf:Description` element.
- **Line 7** indicates the end of `rdf:RDF` element.
RDF vocabulary is a defined set of predicates that can be used in an application. The vocabulary defined by the RDF specification is following: rdfs:type (predicate indicating that resource is an instance of class), rdfs:XMLLiteral (class of typed literals), rdfs:Property (class of properties), rdfs:Alt, rdfs:Bag, rdfs:Seq (containers), rdfs:List (class of lists), rdfs:nil (instance of rdfs:List, representing empty list), rdfs:Statement, rdfs:subject, rdfs:predicate, rdfs:object (used for reification, in which each statement (each triple) is assigned a URI and treated as a resource about which further statements can be made). Vocabulary shown here is used as a backbone for RDF Schema, where that limited vocabulary is extended.

RDF/XML is machine processable, just like HTML. By using URIs, it can link pieces of information across the web. Unlike conventional hypertext documents, URIs are used by RDF for any identifiable things, even those which are not directly retrievable on the web (such as the student Jarek). In addition to describing web resources (like pages, images, videos, etc.), RDF can also describes abstract things like people, robots, planets, events, etc. [42]

2.5 RDFS

2.5.1 Introduction

RDF vocabulary description language (RDF Schema), built on top of RDF, is a general-purpose language for representing information on the web [34].

RDF properties may be thought of as attributes of resources. In this sense they can be understand as traditional attribute-value pairs. In addition, RDF properties represent relationships between resources. RDF however, provides no mechanisms for describing these properties, nor does it provide any mechanisms for describing the relationships between these properties and other resources. That is the role of the RDF vocabulary description language (RDFS) [34].

RDF Schema extends RDF vocabulary to allow describing taxonomies of classes and properties. Classes (generalized categories or unary relations) and properties (predicates or binary relations) can be arranged into hierarchies. In addition it extends definitions for some of the elements of RDF, i.e. it sets the domain and range of properties [4]. RDFS also gives simple inferencing possibility of the following forms: inferring class membership and subclass relations through transitive inference in the subclass hierarchy, inferring class membership through occurrence in typed property-positions, and inferring property values and subproperty relations through transitive inference in the subproperty hierarchy.
2.5.2 RDFS constructs

Main constructs defined by RDFS are:

- **Classes**: `rdfs:Resource` (because all things described by RDF are resources, it is the class of everything), `rdfs:Class` (declares a resource as a class of another resource), `rdfs:Literal` (literal values, such as strings or integers), `rdfs:Datatype` (class of datatypes), `rdf:XMLLiteral` (the class of XML literal values) and `rdf:Property` (class of properties). Example:

  `<rdf:Description rdf:about="http://agh.edu.pl#Student">`  
  `<rdf:type rdf:resource=People/>`  
  `</rdf:Description>`

  `<rdfs:Class rdf:ID="Keyboard">`  
  `<rdfs:subClassOf rdf:resource="Notebook"/>`  
  `</rdfs:Class>`

- **Properties**: `rdfs:domain` (declares the class of the subject in a triple block), `rdfs:range` (declares the class or datatype of the object in a triple block), `rdf:type` (property used to state that a resource is an instance of a class), `rdfs:subClassOf` (allows to declare hierarchies of classes) and `rdfs:subPropertyOf` (allows to declare hierarchies of properties). Example:

  `<rdf:Property rdf:ID="hasUniqueSkills">`  
  `<rdfs:subPropertyOf rdf:resource="hasFeature"/>`  
  `<rdfs:domain rdf:resource="People"/>`  
  `<rdfs:ranage rdf:resource="AGHStudents"/>`  
  `</rdf:Property>`

- **Utility Properties**: `rdfs:seeAlso` (relates a resource to another with additional explanation), `rdfs:isDefinedBy` (relates a resource to its definition), `rdfs:label` (provides friendly name of resource) and `rdfs:comment` (provides a human-readable description of a resource).

The fact is, that there is lack of any notion of negation or disjunction in RDFS. It provides only a very limited notion of existential quantification. This makes for RDFS language "very limited expressive power" [47].
2.6 DL

2.6.1 Introduction

Description Logics (DLs) are a family of knowledge representation (KR) languages. They can be used for representing knowledge of application domains in a structured, formal way. The first part of the name description logics means that the knowledge is represented by concept descriptions. Elementary descriptions are the expressions built from atomics concepts (unary predicates) and atomic roles (binary predicates). Complex descriptions are created from elementary descriptions inductively by using concept constructors. The second part indicates that description logics are equipped with a formal logic-based semantics, unlike their predecessors: semantic networks and frames. DLs were introduced to KR systems as a reaction to overcome deficiencies derived from the lack of formal semantics. One of the key DLs features is that they provide reasoning algorithms for the composition of structured concept descriptions. Reasoning allows to infer implicitly represented knowledge from the explicitly provided information stored in the knowledge base [27, 38, 47].

Reasoning processes allow for classification of concepts and individuals. Classification of concepts determines the dependencies between them, thus creates hierarchies of concepts. Classification of individuals indicates whether given individual is an instance of certain concept. Such inferential mechanisms are used by many intelligent information processing systems. What is more, classifying and ordering attempts of every part of the world is the way humans try to understand reality.

2.6.2 DL Knowledge Representations Systems

Descriptions of relationships (concepts and roles) and their combinations can be used to build Description Logics Knowledge Representations Systems (DLKRSs). Reasoning services of such systems are divided into 2 parts: terminological component (called TBox) and an assertional component (called ABox). Such a hybrid architecture was pioneered by the KRYPTON system, and has been adopted by many DLKRSs. It is shown on the Figure 2.4.
TBox introduces the terminology which means the vocabulary of application domain. The vocabulary consist of concepts (set of individuals) and roles (binary relationships between individuals). TBox can also be used for creating names (abbreviations) for complex descriptions. ABox contains assertions about named individuals in terms of this vocabulary. When it comes to searching analogies for understanding what TBox and ABox component is, we can collate TBox part with database schema, and ABox with database data.

DL systems, besides storing terminologies and assertions, also provide reasoning mechanisms. Hence, TBox (term classifier) reasons about concepts and roles descriptions and their relationships, while ABox reasons about individuals and their relationships, e.g.

- TBox:

\[
\text{GoodCPU} \equiv \text{Thing} \sqcap (\exists \text{isMadeOf.Metalloid}) \\
\sqcap (\exists \text{isCreatedBy.ChipManufacturer}) \\
\sqcap (\forall \text{hasFeature.(} \geq 4 \text{ hasCore} \sqcup \neg \text{x86Arch})) \\
\sqcap \exists \text{hasFeature.}\top
\]

In one of its simplest form TBox introduces abbreviation for a complex description. The abbreviation here is GoodCPU, which indicates something described by following statement: "a thing made of semi metal, which is created by chip maker and all of its features are either more than 3 cores or not x86 architecture".

Figure 2.4 The architecture of Knowledge Representation System based on Description Logics [27]
• ABox:

\[
\begin{align*}
\text{GoodCPU}(\text{Itanium}) & \quad \text{Metalloid}(\text{Silicon}) \\
\text{ChipManufacturer}(\text{IntelCorp.}) & \quad \text{hasFeature}(\text{Itanium, IA64}) \\
\text{hasCore}(\text{Itanium, 12}) & \quad \neg x86\text{Arch}(\text{IA64})
\end{align*}
\]

We can see descriptions of specific situation, where properties are asserted for some individuals. \textit{GoodCPU} indicates the "Itanium chip made of Silicon by Intel Corporation, which has 12 cores based on IA64 architecture (not backward compatible with old x86)."

Reasoning about terminology (TBox) provides \textit{satisfiability} (i.e. non-contradictory) and \textit{subsumption} services. Satisfiability is used for finding an interpretation that makes the formula true and subsumption is used for organizing the concept hierarchy according to their generality (ordering of concepts based on subsumption relation).

Reasoning about assertions (ABox) provides \textit{consistency} and \textit{instantiation} mechanisms. A theory is consistent when it does not contain contradiction. In the semantic terms it means, that it has a model. Instantiation is used to perform \textit{realization} and \textit{retrieval}. Realization means computing the most specific concepts which an individual instantiates while retrieval means computing the individuals which are instances of a given concept.

Satisfiability checks of descriptions while consistency checks of sets of assertions to determine whether a knowledge base is meaningful or not.

On the start of typical DLKRS application, TBox reasoning services are invoked to ensure that all defined concepts are satisfiable, i.e. are not subsumed by the bottom concept, which is always interpreted as the empty set. Moreover, subsumption hierarchy is computed. This hierarchy would then be inspected to make sure that it coincides with the intention of the modeler. Next it comes to ABox, which first check for its consistency with the TBox and then, for example, compute the most specific concepts that each individual is an instance of (known as realizing the ABox).

### 2.6.3 Description languages

As it was mentioned at the beginning, elementary descriptions are the expressions built from \textit{atomic concepts} (unary predicates) and \textit{atomic roles} (binary predicates). Complex descriptions are created from elementary descriptions inductively by using of \textit{concept constructors}.

Description languages are distinguished by the constructors they provide. The language \(\mathcal{AL}\) (Attributive Language) is a minimal language of practical interest. The other languages of this family are extensions of \(\mathcal{AL}\), i.e. \(\mathcal{ALC}\) (Section 2.6.3.1) is obtained from \(\mathcal{AL}\) by adding the complement operator \(\neg\), \(\mathcal{ALE}\) is obtained from \(\mathcal{AL}\) by adding existential restrictions \(\exists R.C\), etc. [27, 47]. Before moving into details, we will introduce conventional notation in DL:
| Symbol | Description                  |
|--------|------------------------------|
| ⊤      | all concept names            |
| ⊥      | empty concept                |
| ⊓      | intersection or conjunction of concepts |
| ⊔      | union or disjunction of concepts |
| ¬      | negation or complement of concepts |
| ∀      | universal restriction        |
| ∃      | existential restriction      |
| ⊑      | concept inclusion            |
| ≡      | concept equivalence          |
| ≡      | concept definition           |
| :      | concept/role assertion       |

Table 2.2  DL notation

Let A and B be atomic concepts, R be atomic role, and C and D be concept descriptions. Based on that abstract notation, concept descriptions in \( \mathcal{AL} \) language are formed by following syntax rule [27]:

\[
C, D \rightarrow A | \text{ (atomic concept)} \\
⊥ | \text{ (universal concept)} \\
⊥ | \text{ (bottom concept)} \\
¬A | \text{ (atomic negation)} \\
C \sqcap D | \text{ (intersection)} \\
∀R.C | \text{ (value restriction)} \\
∃R.\top | \text{ (limited existential quantification)}
\]

In this section a basic example of what can be expressed using \( \mathcal{AL} \) is shown. Let’s assume Computer and Notebook are atomic concepts. Based on that Computer \( \sqcup \) Notebook is an \( \mathcal{AL} \) – concept describing computers which are notebooks. In analogical way Computer \( \sqcup \neg \text{Notebook} \) describes concept of computers which are not notebooks (they can be calculators, supercomputers, etc.). Suppose hasConnection is an atomic role - we can describe the concept Computer \( \sqcap \exists\text{hasConnection}.\top \) as indicating computers which have network connections. Computer \( \sqcap \forall\text{hasConnection}.\text{Notebook} \) describes such computers which are connected only to other notebooks. Concept of computers without any connections is Computer \( \sqcap \forall\text{hasConnection}.\bot \).

When it comes to define semantics of \( \mathcal{AL} – \) concepts we consider interpretation \( I \). The interpretation consists of a non-empty set \( \Delta^I \) (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept \( A \) a set \( A^I \subseteq \Delta^I \) and to every atomic role \( R \) a binary relation \( R^I \subseteq \Delta^I \times \Delta^I \). The interpretation function is extended to concept descriptions by the following inductive definitions [27]:

---

J. Waliszko  Knowledge representation and processing methods in Semantic Web
Two concepts $C$, $D$ are equivalent ($C \equiv D$), if $C^I = D^I$ for all interpretations $I$, e.g. $\forall \text{hasConnection}. \text{Notebook} \sqcap \forall \text{hasConnection}. \text{Ebook}$ is equivalent to $\forall \text{hasConnection}. (\text{Notebook} \sqcup \text{Ebook})$.

2.6.3.1 Syntax and semantics of ALC

The basic DL $ALC$ stands for Attributive Concept Language with Complements. It was introduced by Manfred Schmidt-Schauß and Gert Smolka in 1991. $ALC$ is obtained from $AL$ by adding the complement operator ($\neg$). Below, there are definitions of syntax and semantics of $ALC$ [47]:

Definition 1. ($ALC$ syntax). Let $N_C$ be a set of concept names and $N_R$ be a set of role names. The set of $ALC$-concept descriptions is the smallest set such that

1. $\top$, $\bot$, and every concept name $A \in N_C$ is an $ALC$-concept description,
2. if $C$ and $D$ are $ALC$-concept descriptions and $r \in N_R$, then $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall r.C$, and $\exists r.C$ are $ALC$-concept descriptions.

In the following, we will often use "$ALC$-concept" instead of "$ALC$-concept description". The semantics of $ALC$ (and of DLs in general) is given in terms of interpretations.

Definition 2. ($ALC$ semantics). An interpretation $I = (\Delta^I, \cdot^I)$ consists of a nonempty set $\Delta^I$, called the domain of $I$, and a function $\cdot^I$ that maps every $ALC$-concept to a subset of $\Delta^I$, and every role name to a subset of $\Delta^I \times \Delta^I$ such that, for all $ALC$-concepts $C, D$ and all role names $r$,

$$
\begin{align*}
\top^I &= \Delta^I \\
\bot^I &= \emptyset \\
(C \sqcap D)^I &= C^I \cap D^I \\
(C \sqcup D)^I &= C^I \cup D^I \\
(\neg C)^I &= \Delta^I \setminus C^I \\
(\forall r.C)^I &= \{ x \in \Delta^I \mid \text{For all } y \in \Delta^I \text{ if } < x, y > \in r^I, \text{ then } y \in C^I \} \\
(\exists r.C)^I &= \{ x \in \Delta^I \mid \text{There is some } y \in \Delta^I \text{ with } < x, y > \in r^I \text{ and } y \in C^I \}
\end{align*}
$$

We say that $C^I$ ($r^I$) is the extension of the concept $C$ (role name $r$) in the interpretation $I$. If $x \in C^I$, then we say that $x$ is an instance of $C$ in $I$. 

J. Waliszko  Knowledge representation and processing methods in Semantic Web
2.7 OWL

2.7.1 Introduction

The OWL Web Ontology Language (OWL), is an ontology language for the Semantic Web with formally defined meaning. It was released in February 2004 as a W3C recommendation. OWL lays on top of RDF and RDFS and comes with a larger vocabulary and stronger syntax. All of them have similar foundations, but OWL is a stronger language with greater machine interpretability than RDF. It can be used to define classes and properties like RDFS, but contains additional set of constructs, which gives to it more expressive power. OWL 2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents. OWL ontologies can be used along with information written in RDF, and OWL ontologies themselves are primarily exchanged as RDF documents. "OWL's expressivity is sufficient to cover most of the well-known Description Logic formalisms" [47, 48].

2.7.2 DLs and OWL

OWL is DL-based ontology language. Because the semantics of OWL (Lite and DL) "can be defined via a translation into an expressive DL" [47], implemented DL reasoners can be used for reasoning tasks in applications built on OWL. Analogies in the naming conventions between DL and OWL, are shown in Table 2.3.

| OWL  | DL   |
|------|------|
| class| concept |
| object| individual |
| property| role |

Table 2.3 OWL DL naming synonyms
An OWL ontology describes the domain in the terms of classes (concepts), objects (individuals) and properties (roles). OWL classes can be created from basic classes or properties, by variety of constructors. The constructors supported by OWL, are shown in Table 2.4. An ontology consists of groups of axioms. Axioms are used for making the assertions, e.g. assertions of subsumption relationships between classes or properties. They are presented in Table 2.5.

| Constructor          | DL Syntax       | Example                    |
|----------------------|-----------------|----------------------------|
| intersectionOf       | \( C_1 \sqcap \cdots \sqcap C_n \) | \( \text{Human} \sqcap \text{Male} \) |
| unionOf              | \( C_1 \sqcup \cdots \sqcup C_n \) | \( \text{Doctor} \sqcup \text{Lawyer} \) |
| complementOf         | \( \neg C \)    | \( \neg \text{Male} \)     |
| oneOf                | \( x_1 \cdots x_2 \) | \( \text{john, mary} \)    |
| allValuesFrom        | \( \forall P.C \) | \( \forall \text{hasChild}.\text{Doctor} \) |
| someValuesFrom       | \( \exists r.C \) | \( \exists \text{hasChild}.\text{Lawyer} \) |
| hasValue             | \( \exists r.\{x\} \) | \( \exists \text{citizensOf}.\{\text{USA}\} \) |
| minCardinality       | \( (\geq nr) \) | \( (\geq 2 \text{hasChild}) \) |
| maxCardinality       | \( (\leq nr) \) | \( (\leq 2 \text{hasChild}) \) |
| inverseOf            | \( r^- \)      | \( \text{hasChild}^- \)   |

Table 2.4 OWL constructors [47]

| Axiom                | DL Syntax       | Example                                      |
|----------------------|-----------------|----------------------------------------------|
| subClassOf           | \( C_1 \sqsubseteq C_2 \) | \( \text{Human} \sqsubseteq \text{Animal} \sqcap \text{Biped} \) |
| equivalentClass      | \( C_1 \equiv C_2 \) | \( \text{Man} \equiv \text{Human} \sqcap \text{Male} \) |
| subPropertyOf        | \( P_1 \sqsubseteq P_2 \) | \( \text{hasDaughter} \sqsubseteq \text{hasChild} \) |
| equivalentProperty   | \( P_1 \equiv P_2 \) | \( \text{cost} \equiv \text{price} \) |
| disjointWith         | \( C_1 \sqsubseteq \neg C_2 \) | \( \text{Male} \sqsubseteq \neg \text{Female} \) |
| sameAs               | \( x_1 \equiv x_2 \) | \( \text{President}_\text{Bush} \equiv \text{G}_\text{W}_\text{Bush} \) |
| differentFrom        | \( x_1 \sqsubseteq \neg x_2 \) | \( \text{john} \sqsubseteq \neg \text{peter} \) |
| TransitiveProperty   | \( P \triangleright \text{transitive role} \) | \( \text{hasAncestor} \text{is a transitive role} \) |
| FunctionalProperty   | \( T \sqsubseteq (\leq 1 P) \) | \( T \sqsubseteq (\leq 1 \text{hasMother}) \) |
| InverseFunctionalProperty | \( T \sqsubseteq (\leq 1 P^-) \) | \( T \sqsubseteq (\leq 1 \text{isMotherOf}^-) \) |
| SymmetricProperty    | \( P \equiv P^- \) | \( \text{isSiblingOf} \equiv \text{isSiblingOf}^- \) |

Table 2.5 OWL axioms [47]
Based on the DL syntax shown in tables above, we can construct OWL equivalents by serializing them to XML, e.g.

\[
\text{Human} \cap \text{Male}
\]

\[
\begin{align*}
&\text{Human} \cap \text{Male} \\
&<\text{owl:intersectionOf} \text{ rdf:parseType="Collection"}> \\
&\quad <\text{rdf:Description} \text{ rdf:about="#Human"}/>
\quad <\text{rdf:Description} \text{ rdf:about="#Male"}/>
\quad </\text{owl:intersectionOf}>
\end{align*}
\]

\[
\text{Doctor} \cup \text{Lawyer}
\]

\[
\begin{align*}
&\text{Doctor} \cup \text{Lawyer} \\
&<\text{owl:unionOf} \text{ rdf:parseType="Collection"}> \\
&\quad <\text{rdf:Description} \text{ rdf:about="#Doctor"}/>
\quad <\text{rdf:Description} \text{ rdf:about="#Lawyer"}/>
\quad </\text{owl:unionOf}>
\end{align*}
\]

\[
\exists \text{hasChild}\cdot \text{Lawyer}
\]

\[
\begin{align*}
&\exists \text{hasChild}\cdot \text{Lawyer} \\
&<\text{owl:Restriction}> \\
&\quad <\text{owl:onProperty} \text{ rdf:resource="#hasChild"}/>
\quad <\text{owl:someValuesFrom} \text{ rdf:resource="#Lawyer"}/>
\quad </\text{owl:Restriction}>
\end{align*}
\]

\[
\exists \text{citizensOf}\cdot \{\text{USA}\}
\]

\[
\begin{align*}
&\exists \text{citizensOf}\cdot \{\text{USA}\} \\
&<\text{owl:Restriction}> \\
&\quad <\text{owl:onProperty} \text{ rdf:resource="#isCitizenOf"}/>
\quad <\text{owl:hasValue} \text{ rdf:resource="#USA"}/>
\quad </\text{owl:Restriction}>
\end{align*}
\]

and last, slightly more complicated example:

\[
\text{GoodCPU} \equiv \text{Thing} \cap (\exists \text{isMadeOf}\cdot \text{Metalloid})
\quad \cap (\exists \text{isCreatedBy}\cdot \text{ChipManufacturer})
\quad \cap (\forall \text{hasFeature}.(\geq 4 \text{hasCore} \cup \neg \text{x86Arch}))
\]
<owl:Class rdf:about="#GoodCPU">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&owl;Thing"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#createdBy"/>
          <owl:someValuesFrom rdf:resource="#ChipManufacturer"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#madeOf"/>
          <owl:someValuesFrom rdf:resource="#Metalloid"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasFeature"/>
          <owl:allValuesFrom>
            <owl:Class>
              <owl:unionOf rdf:parseType="Collection">
                <owl:Class>
                  <owl:complementOf rdf:resource="#x86Arch"/>
                </owl:Class>
                <owl:Restriction>
                  <owl:onProperty rdf:resource="#hasCore"/>
                  <owl:someValuesFrom>
                    <rdf:Description>
                      <rdf:type rdf:resource="&rdfs;Datatype"/>
                      <owl:withRestrictions rdf:parseType="Collection">
                        <rdf:Description>
                          <xsd:minInclusive
                            rdf:datatype="&xsd;integer">4
                          </xsd:minInclusive>
                        </rdf:Description>
                      </owl:withRestrictions>
                    </rdf:Description>
                  </owl:someValuesFrom>
                </owl:Restriction>
              </owl:unionOf>
            </owl:Class>
          </owl:allValuesFrom>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
2.7.2.1 OWL and reasoners

Possibility of using DL reasoning services in OWL-based applications was one of the reasons why OWL is based on DL. "OWL DL has a formal model-theoretic semantics providing a rigorous and provably decidable semantics for the language" [47]. The decidability ensures, that consistency of OWL DL ontology can be checked by complete DL reasoners. Reasoners can be also used to infer information from asserted facts.

Popular reasoners in the OWL community are: FaCT, FaCT++, Pellet, RACER, KAON2 or HermiT (see Section 4.9.2.7). These systems provide reasoning support for a set of tools designed for ontology creation and maintenance which has already been created. There are: Protégé (used for creation of traffic danger ontology described in Chapter 3), Swoop, OilEd and TopBraid Composer.

Increasing number of tools designed for OWL is both a cause and a motivation for the community, to develop ontologies in many various fields, not only in the scope of the Semantic Web. There are areas like: biology, medicine, geography, geology, astronomy, agriculture or defense, in which ontologies become adopted [47].

The specialists from IBM departments in China and USA and Department of Biomedical Informatics in Columbia University, have prepared a report about their experiences with Semantic Web applications [39]. They were working through large terminology called MED (Medical Entities Dictionary) used at the Columbia Presbyterian Medical Center. MED which previously had been using frame-based logic, has been transformed into OWL ontology. After transformation DL subsumption reasoning service was used to classify that ontology, which has revealed many modeling errors. But, as it is said in the report, "the important result here is not that we identified these modeling errors due to the increased expressivity of DL. More important is our finding that the missed subsumptions could have cost the hospital many missing results in various decision support and infection control systems that routinely use MED to screen patients."

2.7.3 OWL sublanguages

W3C specification defines 3 variants of OWL. These sublanguages: OWL Lite, OWL DL, and OWL Full provides different level of expressiveness (increasingly in mentioned order). Each of them contains extensions to its predecessor. Based on scope and complexity of the application domain, appropriate version should be chosen for description of the domain. More complex version gives more freedom in domain modeling. The complexity however affects the cost of a learning, which results in higher learning curve. We have 3 sublanguages inside OWL:

- **OWL Lite**, the simplest version, useful enough to support creation of classification hierarchies with simple constraints. It is not widely used and acts as the entry point for Semantic Web application developers.
- **OWL DL**, the most widely used version, includes OWL Lite. It was designed to provide maximum expressiveness possible while retaining computational completeness, decidability, and automated rea-
soning. That is why OWL DL takes the advantage of reasoning services provided for description logics (see Section 2.7.2.1).

- **OWL Full**, the most powerful in expressiveness but also the heaviest in the meaning of a learning cost. Includes OWL DL. Provides different semantic than predecessors.

### 2.7.4 OWL 2 versus OWL 1

OWL 2 was announced by W3C working group on 27 October 2009, as the extension for previous versions (OWL and OWL 1.1). Like OWL 1, OWL 2 is designed to allow development of ontologies and to simplify this process. The second common design goal is to facilitate sharing ontologies via the web, with the aim of making web content more accessible to machines. Besides, OWL 2 has a very similar overall structure to OWL 1. Figure 2.5 shows main building blocks of OWL 2 and relations between them. Despite changes in naming, almost all the building blocks of OWL 2 were present in OWL 1. All OWL 1 ontologies are valid OWL 2 ontologies.

![Figure 2.5 The structure of OWL 2](image)

Figure 2.5 The structure of OWL 2 [48]
Looking at the structure diagram, we can see that abstract notion of the ontology (abstract structure or RDF graph) is placed in the middle of it. At the top there are various serialization syntaxes for ontologies. At the bottom there are the two semantic specifications. These specifications define the meaning of OWL 2 ontologies. For most of users from the community, only one syntax and one semantic is sufficient. OWL 2 adds some new functionality but still remains compatible with OWL 1. Some of the new features include:

- keys,
- property chains,
- richer datatypes, data ranges,
- qualified cardinality restrictions,
- asymmetric, reflexive, and disjoint properties,
- enhanced annotation capabilities.

OWL 2 also defines three new profiles (sublanguages) (OWL 2 EL, OWL 2 QL, OWL 2 RL) and a new syntax (Manchester Syntax). In addition, the set of RDF Graphs that can be handled by DL reasoners is slightly larger in OWL 2 than in its predecessor. It is connected with the fact that some of the restrictions applicable to OWL DL have been relaxed in the newer OWL version [48].
Chapter 3

Traffic danger ontology

3.1 Introduction

An ontology is described as formal representation of the knowledge about some domain, by a set of concepts within the domain, and the relationships between those concepts. It can be used either to define the domain, or to reason the properties of that domain. One of formal definitions of ontology is "formal, explicit specification of a shared conceptualization" [33]. An ontology provides a shared vocabulary, which can be used to model a domain - that is, the type of objects and/or concepts that exist, and their properties and relations [26].

Ontologies are used in artificial intelligence, Semantic Web, systems engineering, software engineering, biomedical informatics, etc. There are used as a form of knowledge representation about some part of world.

Ontologies may be represented by a lot of standards. Different ontology languages provide different facilities. The most recent development in standard ontology languages is OWL from the World Wide Web Consortium (W3C). One of the last tools for developing ontologies is Protégé [35]. Traffic danger ontology is developed using that tool. In Figure 3.1 there are annotations of that ontology, made in Protégé.
3.2 Knowledge engineering methodology for development of traffic danger concept

The real world provides a set of methodologies for developing an ontology, but there is no the fixed, best one. The domain experts can choose how to build ontologies in the way is best for them. While developing traffic danger ontology, the following fundamental rules described in [44] were taken into consideration:

- There is no one correct way to model a domain - there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.

- Ontology development is necessarily an iterative process.

- Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.

While developing an ontology it is important to have in mind that ontology is a model of real domain, so it must reflect reality. Details connected with that part of reality are going to be clearer and more crystallized down the road of building the ontology. After the initial version of the ontology, it is necessary to revise and rethink the design and repeat such iterative process to get the model that fulfill preferences and will be ready...
3.2 Knowledge engineering methodology for development of traffic danger concept

for giving answers for specific questions. Developing an ontology is not an aim on its own, usually it is a part of more complex software architecture which is driven by such formal domain model. Ontology should be applicable for the system it is going to cooperate.

Methodology outlined above derives from agile development practices based on short feedback loops, systematic tests and constant cooperation with domain experts.

Ontology designing process requires to determine its domain and scope. It should be the starting point of work. That part is definitely more crystallized after answering several helpful questions [44]:

- What is the domain that the ontology will cover?
- For what we are going to use the ontology?
- For what types of questions the information in the ontology should provide answers?
- Who will use and maintain the ontology?

The domain of traffic danger ontology consists of: traffic conditions, locations where such conditions can occur and the actual threats (related to specified conditions). The ontology will be used as the pillar of web system, which can provide real time information for traffic users about dangers connected with various areas. The ontology will be used by the public, it means, all end users who interact with the system. Besides, ontology will be widely accessible for anyone, who can use it for developing their own ideas based on that concept, or extend/improve existing formalization of the domain. Maintain of ontology will be narrowed for those trusted users, who can provide coherent, reliable data about current conditions on the roads. When it comes to competency questions [31] the ontology is going to answer, traffic danger ontology passed through the following sample ones:

- What are the traffic dangers on specific area?
- Are there any dangers connected with low friction on specific area?
- What are the subareas of specific location?
- What kind of dangers are connected with bad atmospheric conditions?
- Is there any danger connected with specific postal code on specific district?
- Are there any traffic conditions provided for specific location?

After that part, domain developer should have almost clear vision of what knowledge structure is going to be created. Searching for already defined ontologies, which could fit desired problem, or at least some of its parts, is a good starting point.

When it comes to the traffic danger ontology development process, the primary research has resulted in the finding of geographical locations ontology. Unfortunately that information structure was too complex to be included into the prototype. After all, it was not an easy task to find any helpful topics on the web - at least not formalized in a way simple enough that could be useful for the needs of this thesis. Because of that, creation of new ontology has started from scratch.

The top-down development process has been chosen for ontology creation. Starting from the definition of the most general concepts in the domain, ontology construction has been carried on, down to the most
detailed concepts. After one concept had been entirely built up, and has been sufficient for the current needs, development of the next one has been started in the similar top-down way.

After all parts of ontology had been built, analysis of the ontology was started. The analysis procedure could be compared to specific kind of debugging process. In that part, reasoner was used for verifying answers for competency questions. If some parts of ontology was not coherent with another one, or the ontology was not able to result in providing correct answers, the broken parts were rebuilt. It was an iterative process, which ended when all competency questions was covered. Some of the questions has changed during the time of development process, but the latest form of the ontology fulfills answers to all questions correctly.

### 3.3 Ontology parts description

In this section we will take a look at short outline of components that builds the OWL ontologies. We can talk about Individuals, Properties and Classes.

#### 3.3.1 Individuals

The individuals could be treated as instances of classes. They represent the concrete objects, in our domain. Figure 3.2 shows sample individuals, used in ontology we are talking about.

> These sample individuals shown below, are not present in the pure ontology which describes the prototype concept of traffic danger. Individuals, and all links connected to them, are added to ontology after synchronization with database, which I’ll explain in Chapter 4.

![Sample individuals](image)
3.3.2 Properties

Properties are binary relations between classes (or individuals). They account for linking individuals. In Figures 3.3 and 3.4 we can see respectively data and object properties used in traffic danger ontology.

![Figure 3.3 Only one data property in our ontology](image1)

Properties can have many features, e.g. they can have inverse properties, as discussed in the example below.

The property named `hasLocation` link the postal code individual `30-147` to the street individual called `ArmiiKrajowej` (Figure 3.5). The inverse property of `hasLocation` is `isLocationOf`. Because of the inversion, we can see, that the reasoner properly infer, that property `isLocationOf` link `ArmiiKrajowej` street to `30-147` postal code (Figure 3.6). Inferred features are marked by light yellow background. The reasoning is based on description logics (DL), which will be explained later.

![Figure 3.4 Object properties](image2)

![Figure 3.5 Property assertions (for the street named ArmiiKrajowej individual)](image3)
3.3.3 Classes

If we use analogy, the classes may be viewed as types in object oriented programming. They represent the concepts, in the other words, they define the abstracts. They also may be treated as sets of individuals, and contain all individuals in our domain of concept, that fill all requirements of membership. Classes are though described using formal, mathematical descriptions called description logics (OWL DL). These logics allow, to precisely certify, if certain subclasses or individuals are members of our superclass, or not. The process of deduction may be computed automatically using reasoners. That is one of the key features of OWL-DL. Figure 3.7 shows sample individuals assigned to their classes. Figure 3.8 shows all classes, traffic danger ontology is consisted of.

Figure 3.6 Property assertions (for the postal code 30-147 individual)

Figure 3.7 Individuals by class
Figure 3.8  Asserted class hierarchy
3.4 Existential and universal restrictions

This section considers the important issue of existential and universal restrictions in Protégé. As it was described in Sections 2.6 and 2.7, existential restrictions are denoted in formal Description Logics (DL) Syntax by existential quantifier (\(\exists\)), and in Ontology Web Language (OWL) by someValuesFrom. Universal restrictions on the other hand, are denoted in DL Syntax by universal quantifier (\(\forall\)), and in OWL by allValuesFrom.

- "Existential restrictions describe classes of individuals that participate in at least one relationship along a specified property to individuals that are members of a specified class" [35]. In Protégé the keyword some is used to denote existential restrictions.

- "Universal restrictions describe classes of individuals that for a given property only have relationships along this property to individuals that are members of a specified class" [35]. In Protégé the keyword only is used to denote universal restrictions.

3.5 Open world assumption and closure axiom

This section is directly connected with the reasoning domain (see Section 3.6). The reason of separation this field to its individual section is to emphasize its importance.

Open word assumption (OWA) is one of the foundations of reasoning mechanisms used in Description Logics and OWL. In other papers OWA can be referred as OWR which extends to open world reasoning.

The OWA denotes, that a fact which has not been stated to be true, cannot be assumed to be false. In the other words there is prohibited to assume that something does not exist until that fact is explicitly stated. Such an ambiguous situation simply means that the knowledge has not yet been added to the knowledge base.

In the case of described traffic ontology, that is stated that PoorVisibilityDanger has precipitation conditions that are kinds of FoggyCondition, RainyCondition and SnowyCondition. Because of the open world assumption, until we explicitly say that a PoorVisibilityDanger only has that kinds of conditions, it is assumed (by the reasoner) that a PoorVisibilityDanger could have other conditions such as SunnyCondition. To specify explicitly that a BlindingLightDanger has conditions that are kinds of FoggyCondition, RainyCondition and SnowyCondition only, a closure axiom has to be added on the hasPrecipitationCondition property.

A closure axiom on a property denotes that property can be filled by the specified set of fillers. The axiom consists of a universal restriction which has a filler that is the union of fillers that occur in the existential restrictions for the property.
3.5 Open world assumption and closure axiom

In traffic danger ontology for example, the closure axiom on the `hasPrecipitationCondition` for `PoorVisibilityDanger` defined by following existential restrictions:

$$\exists \text{hasPrecipitationCondition} . \text{FoggyCondition}$$
$$\exists \text{hasPrecipitationCondition} . \text{RainyCondition}$$
$$\exists \text{hasPrecipitationCondition} . \text{SnowyCondition}$$

```xml
<owl:Class rdf:about="#PoorVisibilityDanger">
  <rdfs:label xml:lang="en">PoorVisibilityDanger</rdfs:label>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPrecipitationCondition"/>
      <owl:someValuesFrom rdf:resource="#FoggyCondition"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPrecipitationCondition"/>
      <owl:someValuesFrom rdf:resource="#RainyCondition"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPrecipitationCondition"/>
      <owl:someValuesFrom rdf:resource="#SnowyCondition"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 3.9 Existential restrictions on `hasPrecipitationCondition` property
is stated as a universal restriction, which acts along the `hasPrecipitationCondition` property, with a filler that is the union of `FoggyCondition`, `RainyCondition` and also `SnowyCondition`:

\[
\forall \text{hasPrecipitationCondition}. (\text{FoggyCondition} \sqcup \text{RainyCondition} \sqcup \text{SnowyCondition})
\]

```xml
<owl:Class rdf:about="#PoorVisibilityDanger">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasPrecipitationCondition"/>
      <owl:allValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <rdf:Description rdf:about="#FoggyCondition"/>
            <rdf:Description rdf:about="#RainyCondition"/>
            <rdf:Description rdf:about="#SnowyCondition"/>
          </owl:unionOf>
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 3.10 Closure axiom on `hasPrecipitationCondition` property
3.6 Reasoning

3.6.1 Introduction

Such classes that do not have any sets of necessary and sufficient conditions, but have only necessary conditions, are known as primitive classes. In Protégé they are marked by plain round yellow icon. Yet the classes marked by 3 horizontal white lines, are called defined classes. It means that such classes have at least one set of the necessary and sufficient conditions for the reasoner, to make assumptions based on their descriptions (to infer relationships). The reasoner is able to infer dependencies only for defined classes [35]. When we turn on the reasoner, the result of inferred classes will be like that presented in Figure 3.11.

![Diagram of defined classes](image)

Figure 3.11 Defined classes with inferred memberships from *NamedTrafficDanger* subclasses
Another view of this taxonomy can be generated using Graphviz tool. It is open source graph (network) visualization project from AT&T Research. Graphviz is integrated with Protégé as a plugin called OWLViz. We can take a look of TrafficDanger class subclasses structure in Figure 3.12.

That orange ellipses shows the defined classes, which we were talking about earlier. The light yellow ellipses show primitive types. At the above picture we can see all subclasses of NamedTrafficDanger class. All of them have conditions based on which reasoner can make assumptions, and infer them to according defined classes. After inferring we can use OWLViz to show us the inferred relationships. Figure 3.13 shows the inferred memberships of WeatherDanger class.
The reasoner infer memberships of \texttt{WeatherDanger} type. We can see that \texttt{LowFrictionDanger} defined class is also subclass of \texttt{WeatherDanger}. Additionally that class has its own deducted members such as: \texttt{WetSurfaceDanger}, \texttt{FreezingSurfaceDanger} and \texttt{MeltingAsphaltDanger}.

How the classes were inferred, to be finally matched under appropriate supertypes? It is all because of description logics, the reasoning process of the reasoner engine is based on.

\subsection*{3.6.2 First example}

In this example \texttt{LowFrictionDanger} deduction case is exercised. Below, in Figure 3.14, we can see DL description, which tell us something about that class.

Based on DL description, the subclasses tree is build. All we want to do is to classify specific dangers from \texttt{NamedTrafficDanger} type, to wider danger type called \texttt{LowFrictionDanger}. For that danger to occur, the following complex criteria must be fulfilled:
1. there has to occur precipitation like rain or snow, and simultaneously the road has to have a damaged system of draining water out from surface, or
2. the air temperature has to be high while we are driving on asphalt road, or finally
3. the temperature has to be below freezing, (because the surface is frozen and hence slippery)

Because we know now (from Figure 3.13), the inferred members of LowFrictionDanger (WetSurfaceDanger, FreezingSurfaceDanger and MeltingAsphaltDanger), we should take a look of theirs DL descriptions. Figures 3.15, 3.16 and 3.17 show the dangers in mentioned order.

![Superclasses for WetSurfaceDanger](image)

**Figure 3.15**  **WetSurfaceDanger** concept superclasses set

![Superclasses for FreezingSurfaceDanger](image)

**Figure 3.16**  **FreezingSurfaceDanger** concept superclasses set

![Superclasses for MeltingSurfaceAsphalt](image)

**Figure 3.17**  **MeltingSurfaceAsphalt** concept superclasses set
If we spent a while on analyzing the descriptions, the deduction which reasoner makes should by obvious and easy. Take a closer look at FreezingSurfaceDanger (Figure 3.16). We can see that such a cold situation may occur only when the temperature is lower than 0. That is necessary condition that has to occur for the class, to be matched. BelowFreezingTemperatureCondition is defined as a subset of AirTemperatureCondition, and that one in turn is a subset of WeatherCondition (Figure 3.18).

![Superclasses path of BelowFreezingTemperatureCondition class](image)

The description of WeatherDanger class (Figure 3.19) tells, that such kinds of dangers can be caused by inappropriate weather conditions.

![WeatherDanger concept superclasses set](image)

Because BelowFreezingTemperatureCondition is weather condition (subset of WeatherCondition type), FreezingSurfaceDanger is a member of WeatherDanger type.

### 3.6.3 Second example

We can take a closer look at another example of reasoning. This time we will ask the ontology to fetch dangers, which may occur on district named StareMiasto. There is no class prepared for answering such a question on class hierarchy shown in Figure 3.8. We have to write the query directly, using DL Query tab in Protégé. Below, in Figure 3.20, the complete query is prepared.

![Query for dangers possibilities on district StareMiasto](image)

More precisely, the query fetches all classes of dangers, raised as a results of road conditions appeared on district StareMiasto.

We have to inform ontology about some details, for associated reasoner to give the correct answer. I would like to show how it can be done using Protégé editor. It is why I have prepared simple tutorial. In the next few steps we will add required information. The information we are going to add is appropriate just for this example, and can be very different in reality.
The analogical process, of filling ontology with additional knowledge, is called synchronization in this paper. It is done automatically, as a part of functionality provided by a system described in the Chapter 4. The detailed information will be provided later.

We are going to execute following steps using Protégé:

1. Create individuals: postal code 30-020, street Szpitalna, and district StareMiasto. The postal code should belong to street, and the street should belong to district (through hasLocation relation). Filling ontology with new individuals data is done using Protégé Individuals tab:
   - add hasLocation relation to appropriate individual, by repeating following process for postal code and street:
     - select appropriate individual on Individuals tree,
     - click on Object property assertions button in the Property assertions window,
     - in the opened window write "hasLocation Szpitalna" for postal code 30-020 and "hasLocation StareMiasto" for Szpitalna street,
   - define types for provided individuals, by repeating following process for postal code, street and district:
     - select appropriate individual on Individuals tree,
     - click on Types button on Description tab,
     - in the opened window write appropriate type for each individual: "PostalCodeLocation" for 30-020, "StreetLocation" for Szpitalna, and "DistrictLocation" for StareMiasto.

2. Associate the location defined by postal code 30-020 with HighCongestionCondition condition, to assert that such a condition occurs in this area. That process is done using Classes tab:
   - select HighCongestionCondition node on Asserted class hierarchy tree,
   - click Superclasses button,
   - write "hasLocation value 30-020" in the newly opened window.

Now ontology is prepared. We can execute query shown in Figure 3.20 using DL Query tab. After executing, there will be one subclass in the result set. It is TrafficCongestionDanger. Take a closer look on description of that class, in order to understand why it has been fetched. In Figure 3.21 we can see, that TrafficCongestionDanger class is connected to HighCongestionDanger condition.
Taking under consideration connections between postal code, street, and district individuals (Figure 3.22), reasoner can assume, that postal code area 30-020 belongs to StareMiasto district.

The reasoner deduction for StareMiasto is shown in Figure 3.23.

In the above steps, we have synchronized the core ontology with sample knowledge. The way we have done that (information shown in Figure 3.21 and Figure 3.23) allow to deduct, that TrafficCongestionDanger occurs on StareMiasto district.

3.7 Summary

Modern ontology development tools such as Protégé allow users to exploit ontologies conveniently, and provide intelligent guidance to find mistakes similar to a debugger in a programming environment. The only thing users have to do, for classes classification and inconsistencies detection process to be executed, is to choose a Classify option shown in Figure 3.24.
In addition, Protégé is ideal as a rapid prototyping environment in which ontology designers can instantly create individuals of specific classes in their ontology and experiment with semantic restrictions [40].

Furthermore, as a support for ontology researchers, Protégé has an open architecture and is easily extensible. That allows programmers to integrate arbitrary components with the tool. As a result ontology designers have a wide range of third-party plugins, which they can use. Protégé plugin library can be found under wiki page of the tool [16].
Chapter 4

Ontology-driven web system

4.1 Introduction

As it was mentioned at the beginning of the thesis, the system creation aim is an attempt to integrate database and ontology approach, for storing and inferring desired information about some domain in real time. In this chapter, such a composite approach is presented, in which location details of traffic conditions are stored in database, while clean abstract of traffic danger concept is described by core ontology. Such an integration results in dynamic deduction possibility of desired knowledge, using the ontology-based approach, instead of using static relations defined in database only.

The program cooperates with traffic danger ontology, and synchronizes that ontology with information stored in database. Next part of the process is traffic dangers inference. In this part reasoner searches for dangers, which occur on selected locations. Traffic dangers fetch the criteria only, when potentially dangerous traffic conditions are localized on area, we are looking for dangers. Locations of traffic conditions occurrences are defined by postal codes. In database, postal codes are connected with streets. Streets in turn are connected with districts. Ontology describes the way of connections between mentioned locations. Because of that fact, ontology gives the possibility of answering, which kind of traffic danger can occur on the desired location (postal code, street or district). As it was said before, the deduction is based on specific traffic conditions connected to specific postal codes.

Users are allowed to create dynamic questions, and getting results of inferred traffic dangers. That functionality is provided by the front dashboard page. Additionally, system allows trusted users to make changes to locations of traffic conditions occurrence. For working with latest data, provided by trusted users, synchronization mechanism is implemented. Synchronization integrates core ontology, describing the abstract of traffic dangers, with specific real time data. Synchronization process is executed first time on application startup. The startup can be understand as first request to the server while accessing main page. That functionality is also available on demand, after pressing synchronization button. After synchronization, we are cooperating with ontology cached in memory.
The project is coded in Java. Dependencies management and versioning is the task of Apache Maven tool. Main technologies used inside are: PostgreSQL, Hibernate, JSP, Spring MVC, jQuery, The OWL API, HermiT and Log4j.

4.2 Ontology-driven software development

An important aspect in developing any Semantic Web application, including application described in that chapter, is the ontology-driven architecture. In a simplification ontology-driven concept is extracted below:

We can see that Semantic Web application in general is divided into 2 separate but linked layers [40]:

- internal layer - contains all software logic: storage, control and deduction processes,
- Semantic Web layer - makes ontologies and interfaces available to the public.

Semantic Web layer is the external part. Interfaces of that layer are used to control the internal behavior, in particular the outcome of reasoning algorithms. Because of reusability of the ontologies, it is required to spend relatively lot of effort to provide good design and tests of that part. Domain experts should be involved in developing these tasks in parallel with programmers, to provide short iterations in developing as a results of quick responses for every inconsistencies in ontology like irrelevant hidden relationships, domain descriptions mistakes or usability problems.

4.2.1 Agile methodology

All of that suggests, that software development based on agile methodologies is going to be the very dedicated while working on Semantic Web systems. Agile software development is based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. The term was introduced in the Agile Manifesto in 2001, which says:
"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **individuals and interactions** over processes and tools,
- **working software** over comprehensive documentation,
- **customer collaboration** over contract negotiation,
- **responding to change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more" [10].

### 4.2.2 Model-driven architecture

Ontology-driven software design approach has a lot in common with model-driven architecture (MDA). MDA movement was launched by the Object Management Group (OMG) in 2001. That approach shows "how to better integrate high-level domain models into the development cycles of conventional software" [40].

A central idea of MDA is to defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language (DSL). The PIM is next translated to one or more platform-specific models (PSMs) that computers can run. In the other words the goal of MDA is to employ languages like UML and generate from them a code appropriate for specific platform. Ontology-driven software development follows similar idea, but the way it does it is much more aggressive. It is because domain models can be used not only for code generation, but what is more remarkable, they can be involved in run time executable tasks.

Nevertheless, any progress in MDA technology and tools may be useful for the Semantic Web community. In addition, MDA has aggregated wide range of modeling language standards under one metamodel writing standard called Meta-Object Facility (MOF). The OMG has created an initiative for rapid development ontology-related technologies. One of the OMG’s efforts are directed towards mapping definition between OWL and MOF/UML. That integration can become a key for bringing that technologies much closer together. Because OWL does not need to be the best modeling language for everyone who is involved in domain design tasks, domain-specific languages can be used, depending on the task and the expertise of the domain modelers. Platform dependent languages can be then translated into the details of OWL or any other desired language.
4.3 System use cases

On the diagram below, actors and use cases of the system are shown:

![Diagram of system use cases](image)

Figure 4.2 Use cases of the system
4.4 Interaction with system for updating and inferring data

On the diagram below, sequence diagram for updating and inferring data is shown:

Figure 4.3 Sequence diagram for updating and inferring data
4.5 System flow

On the diagram below, data flow in the system is shown:

![Diagram of system flow]

Figure 4.4 Data flow in the system
The system is divided into 3 functionally different layers: web dashboard layer cooperating with users (through browser clients), platform layer which is the core of system and storage layer, where data is stored, including ontology which drives the system.

On the left part of the diagram we can see session established for trusted users (after authentication pass), who can modify system database data. Database data can be modified through assigning specified information to dynamically built tree of current data state.

On the top, diagram shows, that all users can interact with dashboard for querying system, to get desired information.

In the center of the figure there are 3 processes: downloading of the core ontology, synchronization core ontology with current data uploaded by trusted users and inferring ontology dependencies. Core ontology can be stored on local or remote server and is accessed by URI of location. Synchronization is based on OWL API library, and provides fresh information (cached in memory) for semantic reasoner to infer, as a response to end users queries. Deduction of classes is provided by HermiT reasoner.

Cooperation with database is provided through Hibernate ORM technology. User interface is built with Java Server Pages and jQuery JavaScript library, while requests from users and appropriate responses, are controlled by Spring MVC. For logging the results of particular operations, Log4j library is used. Ontology can be provided in different formats like OWL2 XML, RDF/XML or Manchester Syntax. PostgreSQL is chosen as SQL database server. All of the mentioned technologies are free and open source.

It is just a brief preview of the system main tasks and particular technologies. The layers cooperation will be explained in detail in the next chapters. There will be also very short preview of mentioned technologies, which are used to build the project.
4.6 Database structure

Entity relationship diagram is shown in Figure 4.5. The diagram was made using Toad Data Modeler [21].

![ER diagram of traffic database]

Database tables structure consist of tables describing locations: street, district and postal_code, tables handling many-to-many relationships: postal_code_2_street, street_2_district, traffic_condition_2_postal_code, table with traffic conditions structure traffic_condition and table providing authentication for users: access.
4.6 Database structure

Table contains streets names, used for defining locations of specific traffic conditions.

| street   |
|----------|
| id       | street unique identifier |
| name     | street name              |

Table contains district names, used for defining locations of specific traffic conditions.

| district |
|----------|
| id       | district unique identifier |
| name     | district name              |

Table contains postal codes values, used for defining locations of specific traffic conditions.

| postal_code |
|-------------|
| id          | postal code unique identifier |
| value       | postal code value             |

Table contains traffic conditions names and descriptions. Besides that table defines relations between traffic conditions structure.

| traffic_condition |
|-------------------|
| id                | traffic condition unique identifier |
| parent_id         | parent traffic condition unique identifier |
| name              | traffic condition unique name |
| description       | traffic condition description |

Table maps streets to districts, because one street can belong to many districts as well as one district can gather many streets.

| street_2_district |
|-------------------|
| street_id         | street unique identifier |
| district_id       | district unique identifier |

Table maps streets to postal codes, because one street can gather many postal codes as well as one postal code can belong to many streets.

| street_2_postal_code |
|----------------------|
| street_id            | street unique identifier |
| postal_code_id       | postal code unique identifier |

Table maps traffic conditions to postal codes, because one traffic condition can occur on location defined by many postal codes as well as one postal code can define location of many traffic conditions occurred in parallel.

| traffic_condition_2_postal_code |
|---------------------------------|
| traffic_condition_id            | traffic condition unique identifier |
| postal_code_id                  | postal code unique identifier |

Table contains credentials of trusted users. The password is encoded in database with SHA-1 function.

| access |
|--------|
| id     | access id |
| username | user login |
| password | user password |
4.7 Synchronization

One of the most important aspects of the system is possibility of integration data from database with ontology data. I have called that process as synchronization. It allows to complete traffic danger ontology with additional information from database. These additional knowledge consist of locations structure and traffic conditions occurrence in that locations.

This approach provides loose coupling between core ontology, describing the abstract of traffic dangers, and synchronized ontology, which is filled with specific data connected with real time conditions on specific area. Generally, in computing, coupling refers to the degree of direct knowledge that one class has of another. Loose coupling was introduced by Karl Weick [49]. The term can also refer to our case. The only difference is that instead of classes we are now talking about ontologies. Core ontology describes clear concept of traffic danger, while synchronized one is related to specified environment. In the other words, synchronized ontology can differ on various environments where it is deployed. For example traffic conditions information for Krakow are very different than those for Warsaw. Of course we can synchronize our ontology at once with all global data, but it can result in system overloading and decreasing performance while inferring dependencies.

4.8 System overview

4.8.1 Traffic danger board

When user types URL of the system board page into the browser, e.g. http://localhost:8080/traffic_web-1.0.0/board.html, he will see the front page of the system. Overview of this page, called Traffic Danger Board, is shown in Figure 4.6.

When this first request to the board page is called, the internal synchronization process is also executed. After the request is completed, we have the possibility of interaction with ontology which is integrated with current traffic conditions data. Such a synchronized ontology is cached in memory. It is why the execution time of first request is longer than further requests. Further requests, connected with refreshing page or invoking reasoner related operations, are very fast, because synchronization process is not being recalled. All system operations are processed on cached ontology.
4.8 System overview

Figure 4.6  Overview of the system front page
If we want to synchronize our ontology with the latest data from database, we have to explicitly invoke synchronization task. It is provided after pushing button called *Synchronize*. That button is located in top right corner of main page, in the toolbar. Top toolbar of the dashboard is shown in Figure 4.7.

In the bottom toolbar (Figure 4.8), we have the possibility of downloading 2 kinds of ontologies: original (core) and updated (synchronized). It is just a simple feature, which can be helpful for developers or other interested people who want to preview ontology in a raw state - from file, or by loading such a file into an ontology editor such as Protégé.

Locations data in database are well known. That kind of information is fixed and depends on area we are working with. Based on provided postal codes, we can assign traffic conditions to selected ones (which are further connected to streets and districts). Traffic conditions structure in database is identical as traffic condition subtree structure defined in ontology, which was previously shown in Figure 3.8. In Figure 4.9 we can see core part of the main page.

The main page is divided into 3 tabs. First one, named *Dangers by location*, allows for previewing dangers on desired locations. We can choose dangers connected directly to desired postal code, or indirectly to street or even wider - to district. After choosing desired location, system queries ontology for occurred traffic dangers. Ontology querying means here that system invokes reasoner on cached ontology, and demands answer for question similar to that shown in Section 3.6.3. This time, in the place of *StareMiasto*, there can be inserted any selected location value. It is shown in Figure 4.10.
4.8 System overview

The second tab (called Any questions?), shown in Figure 4.11, displays responses for example predefined questions, our ontology is able to answer.

These questions are not build dynamically on runtime, just like it was always after changing drop down lists values on the previous tab. They are defined statically in the ontology, as defined classes. Defined classes topic was explained earlier in Chapter 3. In the practical shortcut, the reasoner can only automatically classify classes under classes which are defined, because they have fulfilled all necessary and sufficient conditions for reasoning to happen. From the technical point of view, to get answers for such questions, the system has to invoke reasoner on ontology. After that operation, dependencies of all defined classes will be automatically deducted. Developer has to get all that inferred dependent subclasses and provide them to the user. In OWLAPI, the OWL parsing library, that is much easier task to do, than building complex questions dynamically.

Working with OWLAPI while doing more complex operations, can be a bit confusing for developers, who have never been working with ontologies before. Nevertheless, after review of source code examples provided by the author, library usage will be straightforward. These examples can be found inside trunk branch of the library source code.

The last tab on the page, called About, provides a brief explanation of the system aim.

Dashboard provides also the possibility of changing language in which ontology displays data. Traffic danger ontology defines 2 languages: English and Polish. That is the reason why we have only that 2 options provided in drop down list shown on top toolbar (Figure 4.7). Various languages support is provided by the ontology itself. System will data in user preferred language, while fetching data from ontology. Database does not interfere with translations.

When the user choose Give me some knowledge hyperlink, located in the right top corner of the dashboard (Figure 4.7), he will be redirected to login page, described in the next section.

Figure 4.10  Dangers by location - inferred values
Figure 4.11 Ontology responses for predefined questions
4.8 System overview

4.8.2 Trusted area authentication form

This page provides logging panel (Figure 4.12) for authenticate trusted users, who can modify the database state.

![Trusted Area Authentication Form](image)

Figure 4.12 Logging panel for trusted area

Database data modification is directly connected with results, users will get while working with ontology, just after synchronization with that updated data. That is the reason of restricted access to control panel. That is be strongly unexpected situation for users, to get inappropriate or even adulterated information about traffic dangers on dedicated areas. This can even provide to potentially unsafe situations on roads, as a result of faulty information about real traffic dangers possibilities. The access is widely restricted but provided for those, who declared inclination for filling database with correct and coherent data.

The database created from scripts, contains example trusted user - username: sa, password: traffic. The user is created only for presentation purposes and should be deleted in production environment. When user provides invalid credentials, appropriate message will be shown on the page.

After login, user will be persisted in the server session. The next entrance to Traffic Danger Control Panel will be transparent for user, without necessity of redundant authentication.
4.8.3 Traffic danger control panel

This page allows trusted users for filling current information about specific traffic conditions locations. In the top left corner system displays user, who is actually logged in. The artificially provided user for presentations purposes is called sa:

Hello sa

Figure 4.13 Logged user name

Below we can see the base view of the page content:

Traffic Danger Control Panel

Location: postal code

Figure 4.14 Control panel raw view

After selection of desired postal code value from drop down list, system is building a dynamic tree of traffic conditions. The tree construction is based on database data. Traffic conditions structure in database (provided through table called traffic_condition shown in Figure 4.5 diagram) is identical, as traffic conditions tree structure in ontology, shown in Chapter 3 in Figure 3.8. ERD diagram of database schema shows, that traffic_conditions table is connected with postal_code through intermediate many-to-many relationship table traffic_condition_2_postal_code. Drop down list is filled with postal codes from database. After selection of desired postal code from the list, system is building current data structure. It is a traffic conditions tree, shown in Figure 4.15.
Users can assign selected location to desired condition by checking radio button next to condition name. When the information is correct, it is ready to be persisted inside database. That is done by pushing Update button, next to the drop down list. Knowledge is now updated and can be used for dangers deduction process on front dashboard page, just after the synchronization is done.
4.9 Technologies inside

4.9.1 Introduction

This chapter provides a short preview of main technologies used to build described system. The project is written in Java language in using of Eclipse Java EE IDE. Dependencies management and versioning is the task of Apache Maven tool [11].

4.9.2 Brief overview

4.9.2.1 PostgreSQL

Database was designed using latest version of PostgreSQL, which can be downloaded from its home page [14]. According to documentation, "PostgreSQL is an object-relational database management system (OR-DBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later" [13].

PostgreSQL is released under the PostgreSQL License, a liberal Open Source license, similar to the BSD or MIT licenses.

4.9.2.2 Hibernate

For interaction with database, system uses object-relational mapping (ORM) library called Hibernate. Hibernate is a library designed for Java language, and provides a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate home page [3] provides all necessary information about that technology. There is also a great tutorial [2], just excellent for learning how to make not only the first steps, but also advanced operations.

One of the most primary features of Hibernate technology, is the possibility of mapping from Java classes to database tables (and from Java data types to SQL data types). Hibernate also provides data query and retrieval facilities. Hibernate automatically generates the SQL calls and relieves the developer from manual result set handling and object conversion, keeping the application portable to all supported SQL databases, with database portability delivered at very little performance overhead.

Hibernate is free software that is distributed under the GNU Lesser General Public License.

4.9.2.3 Java Server Pages

According to home page of JavaServer Pages, "JSP technology enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of web-based applications that are platform independent. JSP technology separates the user interface from content gener-
JSP technology allows developers to build pages using XML-like tags. Such tags encapsulates the logic that generates the content for the page. The application logic can reside in server-based resources (such as JavaBeans component architecture) that the page accesses with these tags. By separating the page logic from its design and supporting a reusable component-based design, JSP technology to fast and easy build web-based applications. Besides standard markup tags (HTML and XML), JSP uses also scriplet tags for building pages. Scriptlet elements are delimited blocks of Java code which may be intermixed with the markup tags.

JavaServer Pages technology is an extension of the Java Servlet technology. A servlet is a Java class which conforms to the Java Servlet API, a protocol by which a Java class may respond to HTTP requests. Therefore developers may use a servlet to add dynamic content to a web server using the Java platform. JSPs are compiled into servlets by a JSP compiler. The compiler can generate a servlet in Java code that is then compiled by the Java compiler. It can also compile the servlet to byte code which is directly executable. The bytecode must be executed within a Java virtual machine (JVM). It provides an abstract platform-neutral runtime environment. To sum up, servlets are platform-independent, server-side modules extending capabilities of a web server.

### 4.9.2.4 Spring MVC

Model View Controller (MVC) is a software architecture which has its roots in Smalltalk [45]. Currently is considered an architectural pattern used in software engineering.

Applications can contain a mixture of data access code, business logic code, and presentation code. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. That situation is called as high coupling. Classes depends on so many other classes, that reusing is impossible. Adding new data access code or new data views often requires reimplementing or coping blocks of business logic code, which then requires maintenance in multiple places. The Model View Controller design pattern solves these problems by decoupling data access, business logic, and data presentation and user interaction [19]. The following diagram represents the MVC pattern:
SpringMVC in one of the implementations of model view controller pattern. It is a part of Spring Framework, an open source application framework for the Java platform. A key design principle in Spring Web MVC and in Spring in general is the open for extension, closed for modification principle (OCP principle). Open for extension means, that the behavior of the module can be extended. The module behavior can be changed in new and various ways as the requirements of the application change, or to meet the needs of new applications. Closed for modification means that source code of module designed in this way is inviolate. There is not allowed to make source code changes to it [41].

I have used the latest version of SpringMVC, which has a lot of improvements to the previous versions. Simplification in development, part of dynamically evolving Spring Framework application platform and integration with JSP made me choose that technology to build the traffic danger web system. Good source of information about that technology can be found in the documentation on the home page of Spring Framework [18].

4.9.2.5 jQuery

According to jQuery home page, "jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is designed to change the way that you write JavaScript" [7].
jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting. Cross-browsing refers to the ability for a websites or client-side scripts to support all the web browsers. jQuery’s syntax simplifies navigation through the HTML documents, selection of DOM elements, handling events, creating animations and even building Ajax oriented applications in a gentle way. jQuery was released in January 2006 at BarCamp NYC by John Resig. Nowadays the library is used by over 27% of the 10,000 most visited websites, jQuery is the most popular JavaScript library in use today [8, 22].

jQuery is free, open source software, under the terms of either the MIT License or the GNU General Public License (GPL) Version 2.

4.9.2.6 The OWL API

According to OWL API home page, "the OWL API is a Java API and reference implementation for creating, manipulating and serializing OWL Ontologies. The latest version of the API is focused towards OWL 2" [12].

The OWL API includes the following components:

- an API for OWL 2 and an efficient in-memory reference implementation,
- RDF/XML parser and writer,
- OWL/XML parser and writer,
- OWL Functional Syntax parser and writer,
- Turtle parser and writer,
- KRSS parser,
- OBO Flat file format parser,
- reasoner interfaces for working with reasoners such as FaCT++, HermiT, Pellet and RACER.

The API is closely aligned with the OWL 2 structural specification. It supports parsing and rendering in the syntaxes defined in the W3C specification, namely, the Functional Syntax, RDF/XML, OWL/XML and the Manchester OWL Syntax. Library is written in Java. "The latest version of the OWL API has been designed to meet the needs of people developing OWL-based applications, OWL editors and OWL reasoners. It is a high level API that is closely aligned with the OWL 2 specification. It includes first class change support, general purpose reasoner interfaces, validators for the various OWL 2 profiles, and support for parsing and serializing ontologies in a variety of syntaxes. The API also has a very flexible design that allows third parties to provide alternative implementations for all major components" [36].

The OWL API is open source and is available under the LGPL License.

4.9.2.7 HermiT

According to HermiT home page, "HermiT is reasoner for ontologies written using the Web Ontology Language (OWL). Given an OWL file, HermiT can determine whether or not the ontology is consistent, identify subsumption relationships between classes, and much more."

J. Waliszko  Knowledge representation and processing methods in Semantic Web
"HermiT is the first publicly-available OWL reasoner based on a novel "hypertableau" calculus which provides much more efficient reasoning than any previously-known algorithm. Ontologies which previously required minutes or hours to classify can often by classified in seconds by HermiT, and HermiT is the first reasoner able to classify a number of ontologies which had previously proven too complex for any available system to handle. HermiT uses direct semantics and passes all OWL 2 conformance tests for direct semantics reasoners" [1].

The version (v1.2.3) used in the Traffic Danger Web System is fully compatible with OWLAPI 3.0.0. It is the reason I have chosen that reasoner for working with, because for the time of writing that thesis the Fact++ and Pellet reasoners was not compatible with the newest version of the OWLAPI.

HermiT is open-source and released under LGPL.

4.9.2.8 Log4j

Apache Log4j is a Java-based logging utility. It was originally written by Ceki Gülcü and is now a project of the Apache Software Foundation.

4.10 Possible directions of development

Future directions of project development can be focused on extensions such as Web Services. Web Services will give the possibility of communication with external systems. That external systems can be perceived as software agents. Their tasks could be focused on periodic connections to our primary system, getting latest information set (serialized into ontology), and creating statistics about traffic dangers. Statistics can visualize frequencies of desired dangers on specific area, classification of safety in desired district at the turn of fixed time, etc.

4.11 Summary

To summarize, the development of Semantic Web applications with the support of great tool like Protégé, supported by DL reasoner, is quite inspiring development task. Such development process is called as driven by ontology. With the support of agile methodologies (like extreme programming) where frequent feedback is crucial for fast and stable system implementation, such development process comes to be powerful. Ontologies can be developed directly by domain experts. Because of direct access to executable systems, charged by recently developed domain models, feedbacks can be made frequently. Best practices from agile methodologies, effective at delivering a particular outcome, can be used in development high-quality domain models. Domain experts may work together with programmers, which can guarantee coherent testing and fast implementation processes.
Chapter 5

Conclusion

“The goal of Semantic Web research is to transform the Web from a linked document repository into a distributed knowledge base and application platform, thus allowing the vast range of available information and services to be more effectively exploited.”

Ian Horrocks [37]

According to Handbook of Knowledge Representation [47], "OWL has become the most used KR language in the history of the field, not because of its particular representational power, but rather because it was designed to be a common syntax usable by many KR systems, to be webized for easier sharing of ontologies and concepts, and to be expressive enough for many problems without totally sacrificing scalability".

Increasingly number of tools designed for OWL is both cause and motivation for the community, to develop ontologies in many various fields, not only in the scope of Semantic Web. There are areas like: biology, medicine, geography, geology, astronomy, agriculture or defense, in which ontologies are becoming adopted [47].

Modern ontology development tools such as Protégé allow users to maintenance and built ontologies in quick, efficient way. They provide intelligent guidance to find mistakes similar to a debugger in a programming environment because of description logics reasoners support. That is also the reason, why Protégé is ideal as a rapid prototyping environment in which ontology designers can instantly create individuals of specific classes in their ontology and experiment with semantic restrictions [40].

Development of Semantic Web applications with the support of recent powerful tools is quite inspiring development task. Ontology-driven development with the support of agile methodologies is very efficient implementation process. Ontologies can be developed directly by domain experts. Because of direct access to executable systems, charged by recently developed domain models, feedbacks can be made frequently. Best practices from agile methodologies, effective at delivering a particular outcome, can be used in development high-quality domain models. Domain experts may work together with programmers, which can guarantee coherent testing and fast implementation processes.
Appendix A

Traffic danger web system deployment

A.1 Environment setup

Java applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Because of Java platform independence, the system can be deployed on various environments.

For the project to be deployed, following applications should be pre-installed:

- PostgreSQL database engine,
- Java Runtime Environment (JRE),
- Apache Tomcat servlet container.

A.1.1 PostgreSQL installation

WINDOWS

Get the latest version of PostgreSQL executable package from http://www.postgresql.org/download/windows/. After downloading simply double click on that package. The installer will do the job.

LINUX

Get the latest version of PostgreSQL sources. For the date of writing that file latest version of PostgreSQL was v9.0beta2. Sources can be obtained by anonymous FTP from ftp://ftp.postgresql.org/pub/source/v9.0beta2/postgresql-9.0beta2.tar.gz. After downloading, unpack the file:

```
gzip -dc postgresql-9.0beta2.tar.gz | tar xf -
```

This will create a directory postgresql-9.0beta2 under the current directory with the PostgreSQL sources. Enter into that directory and execute installation procedure:
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data/
chown postgres /usr/local/pgsql/data/
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data/
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data/ >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test

A.1.2 Java Runtime Environment (JRE) installation

Download the Java 2 Standard Edition Runtime (JRE) release version 5.0 or later, from http://www.java.com/en/download/manual.jsp and install the JRE according to the instructions included with the release.

Set the environment variable named JRE_HOME to the pathname of the directory into which you installed the JRE, e.g.

LINUX

# for Bourne, bash, and related shells
export JRE_HOME=/usr/local/java/jre5.0

# for csh and related shells
setenv JRE_HOME=/usr/local/java/jre5.0

WINDOWS

set JRE_HOME=C:\jre5.0

You can also use the full JDK rather than just the JRE. In this case set the JAVA_HOME environment variable to the pathname of the directory into which you installed the JDK.

A.1.3 Apache Tomcat installation

Download the latest version of Tomcat binary distribution from http://tomcat.apache.org/, appropriate for your system and unpack the file into convenient location so that the distribution resides in its own directory:
A.1 Environment setup

**LINUX**

```
cp apache-tomcat-6.0.26.tar.gz /usr/local/apache/
cd /usr/local/apache/
gzip -dc apache-tomcat-6.0.26.tar.gz | tar xf -
```

Set the `CATALINA_HOME` environmental variable (it is used to refer to the full pathname of the release directory):

**LINUX**

```
# for Bourne, bash, and related shells
export CATALINA_HOME=/usr/local/apache/apache-tomcat-6.0.26
# for csh and related shells
setenv CATALINA_HOME=/usr/local/apache/apache-tomcat-6.0.26
```

**WINDOWS**

```
set CATALINA_HOME=C:\apache\apache-tomcat-6.0.26
```

**Startup Tomcat:**

**LINUX**

```
$CATALINA_HOME/bin/startup.sh
```

**WINDOWS**

```
%CATALINA_HOME%\bin\startup.bat
```

After starting the default web applications included with Tomcat will be available at address: [http://localhost:8080/](http://localhost:8080/)

---

If Tomcat is not responding, the reason can be another web server (or process) running and using provided port 8080. The solution is to edit `$CATALINA_HOME/conf/server.xml` configuration file and change the default port.

**Shutdown Tomcat:**

**LINUX**

```
$CATALINA_HOME/bin/shutdown.sh
```

**WINDOWS**

```
%CATALINA_HOME%\bin\shutdown.bat
```
A.2 Loading and configuration

Open the Tomcat Manager available under the Tomcat Administration panel at address http://localhost:8080/manager/html/:

If you are not authorized to view this page, you should probably examine \$CATALINA_HOME/conf/tomcat_users.xml file and, if necessary, define a new user with appropriate rights, e.g. `<user username="root" password="toor" roles="manager-gui" />

Figure A.1 Tomcat administration panel

Under the WAR file to deploy section shown in Figure A.2, deploy traffic_web-1.0.0.war archive, containing prebuild web application for traffic danger web system.

Figure A.2 WAR archive deployment section

After deployment is complete, open for edition \$CATALINA_HOME/webapps/traffic_web/WEB-INF/web.xml file and change value of OntologyURI parameter for appropriate path indicating ontology file for traffic danger TrafficDanger.owl.

The URI can identify local or remote resource, for example:

- file:///C:/Users/jwa/masters/TrafficDanger.owl (local Windows location),
- file:///home/jwa/masters/TrafficDanger.owl (local Linux location),
- http://host/TrafficDanger.owl (remote location).

Final step is about SQL scripts execution, required for database creation. Run the following scripts in the given order: postgres_traffic_user.sql, postgres_traffic_database.sql, postgres_traffic_schema.sql and finally postgres_traffic_data.sql. The last one actually contains sample data, so its execution is optional.

---

J. Waliszko  Knowledge representation and processing methods in Semantic Web
Change user to `postgres`, and change catalog to directory containing database scripts and invoke following commands:

```
psql -f postgres_traffic_user.sql
psql -f postgres_traffic_database.sql
psql -f postgres_traffic_schema.sql traffic
psql -f postgres_traffic_data.sql traffic
```

After that, system should be ready to cooperate under address `http://localhost:8080/traffic_web-1.0.0/board.html`. 
Appendix B

Additional tools installation instructions

B.1 Protégé installation

B.1.1 Editor installation

The installation instructions can be found on Protégé home page [15]. First, download the latest version of Protégé tool from the Protégé home page. Choose installation variant based on your system type.

**WINDOWS**

After downloading double-click `install_protege_4.0.2.exe`.

If you do not have a Java virtual machine installed, be sure to download the package above which includes one.

**LINUX**

After downloading open a shell and go to the directory of downloaded installer. At the prompt type:

```
sh ./install_protege_4.0.2.bin
```

If you do not have a Java virtual machine installed, be sure to download the package above which includes one. Otherwise you may need to download one from Sun’s Java website [20] or contact your OS manufacturer.
ALL OTHER PLATFORMS

1. Instructions for Unix or Unix-like operating systems
   - For Java 2, after downloading, type
     \texttt{java -jar install_protege_4.0.2.jar}
   - For Java 1.1, after downloading, type
     \texttt{jre -cp install_protege_4.0.2.jar install}
   - If that does not work, try
     \texttt{java -classpath \{path to\} classes.zip:install_protege_4.0.2.jar install}
   - If that does not work either, on sh-like shells, try
     \texttt{cd \{to directory where install_protege_4.0.2.jar is located\}
     \texttt{CLASSPATH=install_protege_4.0.2.jar export CLASSPATH java install}
   - Or for csh-like shells, try
     \texttt{cd \{to directory where install_protege_4.0.2.jar is located\}
     \texttt{setenv CLASSPATH install_protege_4.0.2.jar java install}

2. Instructions for other platforms
   - Be sure you have Java installed. You can download Java from Sun’s website \cite{20}.
   - In a console window, change to the directory where you downloaded \texttt{install_protege_4.0.2.jar} to before running the installer.

Your operating system may invoke Java in a different way. To start the installer, add \texttt{install_protege_4.0.2.jar} to your CLASSPATH, then start the main class of the installer named \texttt{install}. 
B.1.2 Plugins installation

After installation of Protégé there can be custom need of installation additional plugins. For plugins installation go to File->Preferences->Plugins tab and click Check for downloads now button (Figure B.1).

![Plugins tab](image)

Figure B.1 Plugins tab

For all additional instructions check the wiki page for Protégé [16]. That is very competitive and reliable source of all information that may be needed. It includes documentation, tutorials, sample ontologies, plugin libraries, etc.

B.2 Maven installation

Maven will be very helpful while working with sources of the project, because it can automatically download all dependencies tree from the network.

Maven is a Java tool, so you must have Java installed in order to proceed. Java Development Kit (JDK) is required (Java Runtime Environment (JRE) is not sufficient). Installation instructions provided below can be found on Maven home page [11].
WINDOWS

1. Download the latest Maven archive from Maven home page and unzip the distribution archive, i.e. apache-maven-2.2.1-bin.zip to the directory you wish to install Maven 2.2.1. These instructions assume you chose C:\Program Files\Apache Software Foundation\.
2. Add the M2_HOME environment variable by opening up the system properties (WinKey+Pause), selecting the Advanced tab, and the Environment Variables button, then adding the M2_HOME variable in the user variables with the value C:\Program Files\Apache Software Foundation\apache-maven-2.2.1.
3. In the same dialog, add the M2 environment variable in the user variables with the value %M2_HOME%\bin.
4. In the same dialog, update/create the PATH environment variable in the user variables and prepend the value %M2% to it (in order to make Maven available in the command line).
5. In the same dialog, make sure that JAVA_HOME exists in your user variables or in the system variables and it is set to the location of your JDK, e.g. C:\Program Files\Java\jdk1.5.0_02 and that %JAVA_HOME%\bin is in your PATH environment variable.

UNIX-BASED OPERATING SYSTEMS

1. Download the latest Maven archive from Maven home page and unzip the distribution archive, i.e. apache-maven-2.2.1-bin.zip to the directory you wish to install Maven 2.2.1. These instructions assume you chose /usr/local/apache-maven/. The subdirectory apache-maven-2.2.1 will be created from the archive.
2. In a command terminal, add the M2_HOME environment variable, e.g.

   export M2_HOME=/usr/local/apache-maven/apache-maven-2.2.1

3. Add the M2 environment variable, e.g.

   export M2=$M2_HOME/bin

4. Add M2 environment variable to your path, e.g.

   export PATH=$M2:$PATH

5. Make sure that JAVA_HOME is set to the location of your JDK, e.g.

   export JAVA_HOME=/usr/java/jdk1.5.0_02

   and that $JAVA_HOME/bin is in your PATH environment variable.
Appendix C

Working with sources

When it comes to editing sources (for reasons like extending current functionality, fixing unexpected bug, etc.), we should prepare convenient developing environment. Project is written using Eclipse IDE, and is supported by Maven (installation steps in Section B.2). For Eclipse to support Maven, we should install M2Eclipse plugin. Installation is straightforward, because it is automatic, and can be found on M2Eclipse home page [9]. For me, that 3 mentioned tools are synonym of convenient developing environment in the Java world.

Traffic danger application consists of 3 projects:

- traffic_ont - contains logic responsible for interacting with ontologies,
- traffic_domain - contains logic responsible for interacting with database,
- traffic_web - integration part, contains web logic, and UI.

For opening the sources, we should import that 3 projects into Eclipse workspace. When edition is complete, we can compile and install project to our local repository using Maven. What is more, complete web archive will be created, with all dependent resources and libraries. Such prepared package can be then deployed through Tomcat manager (as shown in Section A.2).

Because of comprehensive configuration files and appropriate projects structure (compatible with Maven standard layout [5]), installation can be accomplished in 3 steps shown below (suppose, that all projects are located in C:\workspace directory):

```bash
cd C:\workspace\traffic_ont\nmvn clean install
cd C:\workspace\traffic_domain\nmvn clean install
cd C:\workspace\traffic_web\nmvn clean install
```
After execution of that trivial commands, in our local Maven repository, suppose it is C:\.m2\repository, a subdirectory containing 3 projects will be created. Directory structure looks like that:

|--repository
  |--...
  |--agh
      |--traffic
          |--traffic_ont
              |--1.0.0
                  |--traffic_ont-1.0.0.jar
                  |--...
          |--traffic_domain
              |--1.0.0
                  |--traffic_domain-1.0.0.jar
                  |--...
          |--traffic_web
              |--1.0.0
                  |--traffic_web-1.0.0.war
                  |--...

The complete web archive traffic_web-1.0.0.war is now created, and ready to be deployed on server (see Appendix A).

Alternative way of quick loading Maven webapp project into Tomcat servlet container is available by invoking following command:

mvn tomcat:run
# List of figures

| Figure | Description                                                                 | Page |
|--------|------------------------------------------------------------------------------|------|
| 2.1    | Web evolution from old Web 1.0 up to Semantic Web [32]                      | 11   |
| 2.2    | Semantic Web Stack [43]                                                      | 13   |
| 2.3    | Example of RDF statement                                                     | 21   |
| 2.4    | The architecture of Knowledge Representation System based on Description Logics [27] | 26   |
| 2.5    | The structure of OWL 2 [48]                                                  | 35   |
| 3.1    | Traffic danger ontology annotations in Protégé                               | 38   |
| 3.2    | Sample individuals                                                           | 40   |
| 3.3    | Only one data property in our ontology                                       | 41   |
| 3.4    | Object properties                                                            | 41   |
| 3.5    | Property assertions (for the street named ArmiiKrajowej individual)          | 41   |
| 3.6    | Property assertions (for the postal code 30-147 individual)                  | 42   |
| 3.7    | Individuals by class                                                         | 42   |
| 3.8    | Asserted class hierarchy                                                     | 43   |
| 3.9    | Existential restrictions on hasPrecipitationCondition property               | 45   |
| 3.10   | Closure axiom on hasPrecipitationCondition property                          | 46   |
| 3.11   | Defined classes with inferred memberships from NamedTrafficDanger subclasses | 47   |
| 3.12   | Graphviz made view showing subclasses structure of TrafficDanger class       | 48   |
| 3.13   | Inferred members of WeatherDanger type                                        | 49   |
| 3.14   | OWL DL description of LowFrictionDanger                                       | 49   |
| 3.15   | WetSurfaceDanger concept superclasses set                                    | 50   |
| 3.16   | FreezingSurfaceDanger concept superclasses set                               | 50   |
| 3.17   | MeltingSurfaceAsphalt concept superclasses set                               | 50   |
| 3.18   | Superclasses path of BelowFreezingTemperatureCondition class                 | 51   |
| 3.19   | WeatherDanger concept superclasses set                                        | 51   |
| 3.20   | Query for dangers possibilities on district StareMiasto                      | 51   |
| 3.21   | TrafficCongestionDanger concept superclasses set                             | 53   |
| 3.22   | Connections chain from postal code to district                               | 53   |
| 3.23   | Inferred property assertions for StareMiasto district                         | 53   |
| 3.24   | Reasoner invoking in Protégé                                                  | 54   |
LIST OF FIGURES

| Figure | Description                                      | Page |
|--------|--------------------------------------------------|------|
| 4.1    | Ontology-driven approach                         | 56   |
| 4.2    | Use cases of the system                          | 58   |
| 4.3    | Sequence diagram for updating and inferring data | 59   |
| 4.4    | Data flow in the system                          | 60   |
| 4.5    | ER diagram of traffic database                   | 62   |
| 4.6    | Overview of the system front page                | 65   |
| 4.7    | Top toolbar of main page                         | 66   |
| 4.8    | Bottom toolbar of main page                      | 66   |
| 4.9    | Dangers by location                              | 66   |
| 4.10   | Dangers by location - inferred values            | 67   |
| 4.11   | Ontology responses for predefined questions      | 68   |
| 4.12   | Logging panel for trusted area                   | 69   |
| 4.13   | Logged user name                                 | 70   |
| 4.14   | Control panel raw view                           | 70   |
| 4.15   | Part of dynamically created view                 | 71   |
| 4.16   | Model View Controller [19]                       | 74   |
| A.1    | Tomcat administration panel                      | 81   |
| A.2    | WAR archive deployment section                   | 81   |
| B.1    | Plugins tab                                      | 85   |
## List of tables

| Table | Title                                                                 | Page |
|-------|----------------------------------------------------------------------|------|
| 2.1   | Predefined entities in XML 1.0                                       | 19   |
| 2.2   | DL notation                                                          | 28   |
| 2.3   | OWL DL naming synonyms                                               | 30   |
| 2.4   | OWL constructors [47]                                                | 31   |
| 2.5   | OWL axioms [47]                                                      | 31   |
Bibliography

[1] HermiT reasoner home page. http://hermit-reasoner.com/. Accessed September 2010.

[2] Hibernate documentation. http://docs.jboss.org/hibernate/stable/core/reference/en/html/. Accessed September 2010.

[3] Hibernate home page. http://www.hibernate.org/. Accessed September 2010.

[4] Introduction to Ontologies and Semantic Web. http://obitko.com/tutorials/ontologies-semantic-web/. Accessed September 2010.

[5] Introduction to the Standard Directory Layout for Maven. http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html. Accessed September 2010.

[6] JavaServer Pages home page. http://java.sun.com/products/jsp/. Accessed September 2010.

[7] jQuery home page. http://jquery.com/. Accessed September 2010.

[8] jQuery usage statistics. http://trends.builtwith.com/javascript/jquery/. Accessed September 2010.

[9] M2Eclipse home page. http://m2eclipse.sonatype.org/. Accessed September 2010.

[10] Manifesto for Agile Software Development. http://agilemanifesto.org/. Accessed September 2010.

[11] Maven home page. http://maven.apache.org/. Accessed September 2010.

[12] OWL API home page. http://owlapi.sourceforge.net/. Accessed September 2010.

[13] PostgreSQL documentation. http://www.postgresql.org/docs/8.4/static/index.html. Accessed September 2010.

[14] PostgreSQL home page. http://www.postgresql.org/. Accessed September 2010.
[15] Protégé home page. http://protege.stanford.edu/. Accessed September 2010.

[16] Protégé wiki page. http://protegewiki.stanford.edu/. Accessed September 2010.

[17] Semantic Web tutorial. http://www.w3schools.com/semweb/default.asp/. Accessed September 2010.

[18] Spring home page. http://www.springsource.org/. Accessed September 2010.

[19] SpringMVC Java BluePrints. http://java.sun.com/blueprints/patterns/MVC.html. Accessed September 2010.

[20] Sun home page. http://java.sun.com/. Accessed September 2010.

[21] Toad Data Modeler. http://www.casestudio.com/enu/default.aspx. Accessed September 2010.

[22] Usage of JavaScript libraries for websites. http://w3techs.com/technologies/overview/javascript_library/all/. Accessed September 2010.

[23] W3C Semantic Web Activity. http://www.w3.org/2001/sw/. Accessed September 2010.

[24] Why Develop an Ontology? http://www-ksl-svc.stanford.edu:5915/doc/frame-editor/guided-tour/why-develop-an-ontology.html. Accessed September 2010.

[25] XML tutorial. http://www.w3schools.com/xml/. Accessed September 2010.

[26] F. Arvidsson and A. Flycht-Eriksson. Ontologies I, November 2008.

[27] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. The Description Logic Handbook: Theory, implementation, and applications. Cambridge University Press, 2003.

[28] Tim Berners-Lee and Mark Fischetti. Weaving the Web: The Past, Present and Future of the World Wide Web by its Inventor. Britain: Orion Business, 1999.

[29] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. Scientific American, 2001.

[30] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[31] Mark S. Fox and Michael Gruninger. Enterprise Modeling. The American Association for Artificial Intelligence, 445 Burgess Drive Menlo Park, California 94025.
[32] Pieter W. Groenendijk. Wordpress blog. http://pietergroenendijk.wordpress.com/. Accessed September 2010.

[33] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2):199–220, June 1993.

[34] Ramanathan V. Guha and Dan Brickley. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[35] Matthew Horridge. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools. The University Of Manchester, 1.2 edition, March 2009.

[36] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. Technical report, The University of Manchester, UK, October 2009. 6th OWL Experienced and Directions Workshop, Chantilly, Virginia.

[37] Ian R. Horrocks. Semantic Web: The Story So Far. University of Manchester, Manchester, UK.

[38] Ian R. Horrocks. Optimising Tableaux Decision Procedures For Description Logics. PhD thesis, University of Manchester, 1997.

[39] Aaron Kershenbaum, Achille Fokoue, Chintan Patel, Christopher Welty, Edith Schonberg, James Cimino, Li Ma, Kavitha Srinivas, Robert Schloss, and J William Murdock. A View of OWL from the Field: Use cases and Experiences. Technical report, IBM T. J. Watson Research Lab, Hawthorne, USA and IBM China Research Lab, Beijing, China and Department of Biomedical Informatics, Columbia University, New York, USA, 2006.

[40] Holger Knublauch. Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protégé/OWL. Stanford Medical Informatics, Stanford University, Stanford, CA, 94305.

[41] Robert C. Martin and Micah Martin. Agile Principles, Patterns, and Practices in C#. Helion, 2008.

[42] Eric Miller and Frank Manola. RDF Primer. W3C Recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[43] Grzegorz J. Nalepa. Semantic Web. Presentation at Institute of Automatics, AGH University of Science and Technology, Poland.

[44] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Stanford University, Stanford, CA, 94305.

[45] Trygve Reenskaug. MVC XEROX PARC 1978-79.
[46] Ronald L. Rivest, Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*, volume 1. The MIT Press and McGraw-Hill, 1990.

[47] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*. Elsevier B.V., 1 edition, 2008.

[48] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Technical report, W3C, October 2009. http://www.w3.org/TR/2009/REC-owl2-overview-20091027/.

[49] Karl Weick. Educational organizations as loosely coupled systems. *Administrative Science Quarterly*, 21:1–9, 1976.