Design and Implementation of a Linked Open Data Ontology Repository

with support for ontology comparision

Chen Cai
mail@chencai.info

August 12, 2013

Supervisor:
dr. G. H. L. Fletcher

Software Engineering tutor:
ir. M. R. Henrichs

Academic Research tutor:
dr. G. H. L. Fletcher

Assessment committee:
dr. G. H. L. Fletcher
ir. M. R. Henrichs
dr. M. Pechenizkiy
dr. A. Serebrenik
Abstract

Linked Data is currently a hot topic, both in industry and in academic research, because it provides a mechanism to facilitate the re-usability, cross-linking, integration and sharing of data on the web. Ontologies (i.e., data models for Linked Data) are encouraged to be reused to increase the value and usefulness of Linked Data. However, due to the absence of suitable ontology repositories (i.e., web-based systems that provide access to extensible collections of ontologies), publishing and finding ontologies remains difficult, which hampers the re-usability of ontologies. To cope with this problem, which was initially raised by the company Semmtech BV, in this thesis we present the design and implementation of a fully standards compliant ontology repository. Using the HTTP APIs provided by this ontology repository, ontologies can be easily published, retrieved and reused. Moreover, ontologies are often not static, but evolve over time. Domain changes, adaptations to different tasks, or changes in the conceptualization require modifications of the ontology. The evolution of ontologies causes operability problems not only on the data that conforms to the changed ontologies, but also on other ontologies that depend on the changed ontologies, which will hamper their effective reuse. It is important to provide mechanisms for storage of different versions of an ontology and for highlighting differences between versions. Detecting syntactic difference between versions of an ontology is not hard, but finding and understanding the logical impact (i.e., whether two versions of an ontology give the same answers to queries) of the difference remains very difficult, mainly due to the high complexity of reasoning over ontologies. In this thesis, we also present a novel approach for visualizing the logical impact of ontology difference. Explicitly, we demonstrate an algorithm for detection of existence of the logical impact and for finding succinct representations of the logical impact if it exists.
Preface

This thesis is the final product of my graduation project for my master Computer Science and Engineering at Eindhoven University of Technology. This graduation project was conducted in Web Engineering (former known as Database and Hypermedia) group of the department of Mathematics and Computer Science, with the company Semmtech BV involved in.

Firstly, I would like to thank my family members for their help and support during my studies. Secondly, I would like to express my greatest gratitude to my tutors, George Fletcher and Mike Henrichs, for their expertise and advice. Without their guidance this thesis would never came to be. Finally, I would also like to thank the assessment committee members, for reviewing my thesis, attending my presentations and giving me critical feedback.
# Contents

1 Introduction .................................................. 1
   1.1 Problem statement ...................................... 1
   1.2 Contributions .......................................... 2
   1.3 Thesis outline ........................................... 3

2 Preliminaries .................................................. 5
   2.1 RDF ....................................................... 5
   2.2 5-star Linked Open Data .................................. 6
   2.3 Linked Data Platform ..................................... 7
      2.3.1 Linked Data Platform Resources ...................... 8
      2.3.2 Linked Data Platform Containers ...................... 8
   2.4 OWL 2 Ontologies ....................................... 9
   2.5 Ontology Metadata Vocabulary ........................... 10
   2.6 SPARQL ................................................... 12
   2.7 REST ...................................................... 12
   2.8 Agile Development in Evolutionary Prototyping Technique .. 13
   2.9 Related works ............................................ 13

3 Software requirements .......................................... 15
   3.1 General Description ..................................... 15
      3.1.1 Product Functions .................................... 15
      3.1.2 User Objectives ...................................... 15
   3.2 Specific requirements .................................... 16
      3.2.1 Non-functional requirements .......................... 17
      3.2.2 Storage of ontologies ................................ 17
      3.2.3 Interfaces ............................................ 18
      3.2.4 Ontology publishing .................................. 18
      3.2.5 Modifying of ontologies ............................... 19
      3.2.6 Ontology versioning .................................. 20
      3.2.7 Accessing ontologies ................................ 20
      3.2.8 Administration ....................................... 21
3.3 Summary .................................................. 21

4 Software design and implementation ........................................ 23
   4.1 Storage of ontologies ..................................... 23
      4.1.1 Triplestores ....................................... 23
      4.1.2 AllegroGraph ...................................... 24
   4.2 Designed URIs ........................................... 26
   4.3 Security .................................................. 28
      4.3.1 User management ................................... 28
      4.3.2 Access control ..................................... 29
   4.4 Publishing and modifying of ontologies ......................... 30
      4.4.1 Publishing ontologies .............................. 30
      4.4.2 Modifying ontologies via HTTP requests ............ 32
      4.4.3 Modifying resources via HTTP requests ............ 35
      4.4.4 Collision detection ................................ 38
   4.5 Accessing ontologies .................................... 39
      4.5.1 Listing URIs of hosted ontologies ................. 39
      4.5.2 Retrieving ontologies via HTTP GET requests ..... 40
      4.5.3 Retrieving resources via HTTP GET requests ..... 40
      4.5.4 Support for keyword search ......................... 42
      4.5.5 Ontology Browser .................................. 44
      4.5.6 Support for SPARQL query execution ............... 45
   4.6 Ontology versioning ...................................... 46
      4.6.1 Definition .......................................... 46
      4.6.2 Detecting and representing ontology difference .... 46
   4.7 Implementation .......................................... 47
   4.8 Summary .................................................. 49

5 Performance testing ..................................................... 51
   5.1 Performance test on publishing ontologies .................. 51
   5.2 Performance test on retrieving ontologies .................. 52
   5.3 Summary .................................................. 54

6 Logic-based ontology comparison ...................................... 55
   6.1 OWL 2 QL and DL-LiteR ................................... 55
      6.1.1 OWL 2 QL .......................................... 55
      6.1.2 DL-LiteR ........................................... 56
   6.2 Logical difference between two ontologies .................. 58
   6.3 $\Sigma$-query entailment for DL-LiteR ontologies .......... 59
      6.3.1 Canonical models and $\Sigma$-Homomorphisms .......... 59
6.3.2 Incomplete algorithm for $\Sigma$-query entailment between DL-Lite$_R$ ontologies .................................................. 61
6.4 Summary ........................................................................ 62

7 Visualization of Extended-$\Sigma$-query difference 63
  7.1 Extended-$\Sigma$-query difference ........................................ 63
     7.1.1 Definition of Extended-$\Sigma$-query difference .......... 63
     7.1.2 Algorithm for detecting Extended-$\Sigma$-query difference .. 64
  7.2 Implementation .............................................................. 66
  7.3 Summary ....................................................................... 68

8 Conclusion 69
  8.1 Contributions ................................................................. 69
  8.2 Limitations .................................................................. 69
  8.3 Future work ................................................................. 70

Appendix 70

References 71
Chapter 1

Introduction

In this chapter, we describe the background of this graduation project and the problems addressed by this thesis. Also, the contributions of this project and the outline of this thesis are introduced.

1.1 Problem statement

Linked Data is currently a hot topic, both in industry and in academic research [14]. The Linked Data initiative\(^1\) encourages the interlinking of data on the web in order to increase its value and usefulness. One way of increasing the potential for interlinking when publishing data is to reuse terms from existing ontologies (i.e., data models for Linked Data) where possible, and only invent new ones if necessary. As a result, the current explosion in the amount of data published on the Semantic Web is accompanied by a rapid increase in the number of ontologies that are being developed and made available online [30].

In order for ontologies to be reused, users must be able to find relevant ontologies quickly and easily. They also must be able to share their ontologies in such a way that others can easily discover, access, and reuse them. A number of online ontology repositories have been created to address this need, which are defined as web-based systems that provide access to extensible collections of ontologies with the primary purpose of enabling users to find and use one or several ontologies from these collections [30]. However, there are still some issues that have rarely been addressed by existing ontology repositories, which hamper the re-usability of ontologies. An ontology is usually identified by a base URI (Uniform Resource Identifier), which allows other users to access it online through this URI. Most of existing ontology repositories are hosting ontologies in the same way of hosting other ordinary web documents (e.g.,

\(^1\)Linked Data initiative: http://linkeddata.org/
Users can download ontologies as web documents from these ontology repositories, but the base URIs of the downloaded ontologies may already be not accessible, in which case, the ontologies are still not directly reusable. Also, some ontology developers may not have their own infrastructure which can provide accessible base URIs for publishing ontologies. Without accessible base URIs, the ontologies made by these ontology developers would also not be reusable.

These issues described above were raised by the company *Semmtech BV*, when they were developing their own ontology modelling tool *SEMMweb*. With this tool, users can create ontologies. However, to our knowledge, in the open source community, there was no suitable ontology repositories available that can be associated with this tool to directly make created ontologies available online. Therefore, we decided to join our efforts to develop a 5-star compliant Linked Open Data ontology repository that can fulfill this demand.

Moreover, during the discussions on desired features of the ontology repository with *Semmtech BV*, we realized a challenging task which requires efforts from the academic research perspective. Ontologies are often not static, but evolve over time. Domain changes, adaptations to different tasks, or changes in the conceptualization require modifications of the ontology. The evolution of ontologies will hamper their effective reuse [17], because it causes operability problems not only on the data that conforms to the changed ontologies, but also on other ontologies that depend on the changed ontologies. Therefore, it is important to provide mechanisms for storage of different versions of the same ontology and for highlighting differences between versions. Detecting syntactic difference (i.e., the set of axioms added or removed) between different versions of an ontology is not hard. But finding and understanding the logical impact of the syntactic difference remains very difficult [9, 19, 20, 21]. Thus, apart from design and implementation of the ontology repository, we devote our efforts to exploring the possibility of visualization of logical impact of ontology difference, because no related research had been conducted in this approach.

### 1.2 Contributions

This project contributes two novel things. First, a design and implementation of the first 5-star compliant Linked Open Data ontology repository is made. This ontology repository provides means for ontology storage, sharing, retrieval and versioning. Second, a novel approach of visualization of logical impact of ontology difference is demonstrated.

---

25-star deployment scheme for Open Data: http://5stardata.info/
1.3 Thesis outline

The work demonstrated in this thesis consists of two parts. Part 1 is on software development of the ontology repository which is presented in Chapter 2 to Chapter 5:

- Chapter 2 introduces a number of basic concepts and definitions that are used throughout this thesis, such as RDF, Linked Data Platform, and SPARQL.
- Chapter 3 presents the software requirements for our 5-star compliant Linked Open Data ontology repository.
- Chapter 4 illustrates in detail how the software requirements are fulfilled by the design and implementation of the ontology repository.
- Chapter 5 demonstrates the results of performance testing of the implemented ontology repository.

Part 2 is on academic research of logic-based ontology comparison which is illustrated in Chapter 6 and Chapter 7:

- Chapter 6 introduces summarized research background materials for Chapter 7, such as OWL 2 QL, DL-Lite and notions of logical differences between ontologies.
- Chapter 7 demonstrates a novel approach for visualization of logical impact of ontology difference.

In addition, Chapter 8 describes concluding remarks, limitations and future work of this graduation project.
Chapter 2

Preliminaries

In this chapter we introduce basic concepts and definitions that are used throughout this thesis. Discussing these topics in-depth is beyond the scope of this thesis; only the required basic information will be described. Additionally, we introduce a number of existing ontology repositories and discuss their limitations.

2.1 RDF

RDF, stands for the Resource Description Framework, is a W3C recommenda-
dation for representation resources in the World Wide Web. The basic idea of RDF is identifying things by URIs\(^1\) and describing resources using unordered sets of RDF triples of the form \((\text{subject}, \text{predicate}, \text{object})\). Intuitively, we could express that the resource \((\text{subject})\) has a relation \((\text{predicate})\) to another resource \((\text{object})\). There are some commonly used RDF serialization formats, e.g., RDF/XML\(^3\), Turtle\(^4\) and RDFa\(^5\). For instance, the following triples describe that resource Chen is a student who takes a course named Database Technology:

\[
\begin{align*}
\text{(ex:Chen rdf:type ex:Student)} \\
\text{(ex:Chen ex:takesCourse ex:DatabaseTechnology)},
\end{align*}
\]

can be represented in RDF/XML and Turtle shown in Listing 2.1 and Listing 2.2, respectively.

\(^1\)World Wide Web Consortium: http://www.w3.org/  
\(^2\)Uniform Resource Identifiers  
\(^3\)RDF/XML: http://www.w3.org/TR/rdf-syntax-grammar/  
\(^4\)Turtle: http://www.w3.org/TeamSubmission/turtle/  
\(^5\)RDFa: http://www.w3.org/TR/rdfa-syntax/
Listing 2.1: RDF data represented in RDF/XML format

Listing 2.2: RDF data represented in Turtle format

A set of RDF triples intrinsically represents a labelled, directed graph. Each triple can be viewed as two nodes (subject and object) connected by an edge (predicate). For example, the RDF data presented in Listing 2.1 can be shown as the RDF graph in Figure 2.1.

Figure 2.1: RDF graph of the RDF data shown in Listing 2.1

In RDF, a blank node is a node in a RDF graph representing a resource without being given a URI reference. The resource represented by a blank node is called an anonymous resource. Blank nodes are defined in RDF as “existential variables” in the same way that has been used before in mathematical logic [24]. In some RDF serializations, blank nodes can be assigned with a node id of the form _:id which is a local identifier with a limited scope in a particular RDF graph.

2.2 5-star Linked Open Data

Linked Data, as the name indicates, is a data model that identifies, describes, links and relates structured data elements. The overall purpose of Linked Data
is facilitating the re-usability, cross-linking, integration and sharing of data on
the Semantic Web [34]. Linked Data principles start from the most fundamental
component of Linked Data, i.e., the use of globally unique URI as names for
distinctively denoting such things as information objects, people, places, and
events [2]. These URIs should be HTTP URIs so that people can look up
them. When someone looks up these HTTP URIs, useful information should be
provided using the standards (e.g., RDF, SPARQL). The provided information
shall include links to other URIs, so that people can discover more things.

Linked Open Data (LOD) is Linked Data which is released under an open
licence. Tim Berners-Lee, the inventor of the World Wide Web, suggested the
5-star scheme [2] for LOD services:

- Star 1: Available on the web with an open licence;
- Star 2: Use machine-readable structured data (e.g., excel);
- Star 3: Use non-proprietary format (e.g., CSV instead of excel);
- Star 4: Apply open standards (e.g., RDF) to describe things;
- Star 5: Link the data to other people’s data.

In additional, a web resource is said to be 5-star compliant if it satisfies the
5-star scheme.

2.3 Linked Data Platform

*Linked Data Platform* (LDP) is described by a W3C working draft [36] which
provides a set of best practices and simple approach for a read-write Linked
Data architecture. The basic idea of Linked Data Platform is the use of HTTP
requests for accessing, updating, creating and deleting resources from servers
that expose their resources as Linked Data. Linked Data Platform handles
two components: *Linked Data Platform Resource* (LDPR) and *Linked Data
Platform Container* (LDPC). A *LDP Server* is an application program that
processes HTTP requests for LDPRs or LDPCs and generates corresponding
HTTP responses. A *LDP Client* is an application program that generates
HTTP requests for LDPRs or LDPCs and processes corresponding HTTP re-
sponses.

To be noted that we applied the Linked Data Platform specification pub-
lished on March 7, 2013 in this project. Newer versions of the specification
would be published in the near future and might contain content that is differ-
ent with the content described in this section.
2.3.1 Linked Data Platform Resources

Linked Data Platform Resources (LDPRs) are HTTP resources that are provided by LDP and expressed in a RDF representation, and therefore 5-star compliant. The HTTP Request-URI of a LDPR is typically the subject of all triples in the response. Given the URI of a LDPR, LDP clients could send HTTP requests, e.g., HTTP GET, HTTP POST, HTTP DELETE, to the corresponding LDP server to access, modify and delete the LDPR. For example, after receiving the HTTP GET request shown in Listing 2.3, the LDP server would return the HTTP response (see Listing 2.4) includes content of the LDPR with the URI http://repo.chencai.info/ns/pizza/1373728064025/TomatoTopping.

```
GET /ns/pizza/1373728064025/TomatoTopping
Host: repo.chencai.info
Accept: text/turtle; charset=UTF-8
```

Listing 2.3: HTTP GET request sent to http://repo.chencai.info/ns/pizza/1373728064025/TomatoTopping

```
HTTP/1.1 200 OK
Content-Type: text/turtle; charset=UTF-8

@prefix : <http://repo.chencai.info/ns/pizza/1373728064025/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
:TomatoTopping a owl:Class ;
    rdfs:label "TomatoTopping"@en .
```

Listing 2.4: HTTP response contains content of a LDPR with the URI http://repo.chencai.info/ns/pizza/1373728064025/TomatoTopping

2.3.2 Linked Data Platform Containers

It is not uncommon that HTTP applications are organizing their resources into smaller containers. Blog posts are grouped into blogs, wiki pages are grouped into wikis, and products are grouped into catalogs. Each resource created in the application is created within an instance of one of these container-like entities, and users can list the existing artifacts within one [36]. Similarly, Linked Data Platform Containers (LDPCs) are collections of LDPRs represented in sets of triples. For instance, a LDP client could send a HTTP request to the LDP server to ask for the representation of a LDPC with the URI http://repo.chencai.info/ns/pizza. The responded representation of the LDPC is demonstrated in Listing 2.5.
2.4 OWL 2 Ontologies

In computer science and information science, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse [11]. Typically, the representational primitives consist of classes (or sets), attributes (or properties), and relationships (or relations among class members).

An ontology must be formulated in some representation language. The OWL 2 Web Ontology Language\(^6\) is an ontology language for the Semantic Web with formally defined meaning [26]. OWL 2 ontologies are the ontologies formulated in OWL 2. Based on RDF, OWL 2 adds more vocabulary for describing properties and classes. Entities (also known as resources) are the fundamental building blocks of OWL 2 ontologies, and they define the vocabulary, i.e., the named terms, of an ontology. The structure of entities and literals in OWL 2 is shown in Figure 2.2. In OWL 2, classes, datatypes, object properties, data properties, annotation properties, and named individuals are entities (also known as resources), and they are all uniquely identified by an IRI (a complement to URI), thus they can be compliant with Link Data Platform as well. The relations of entities and literals in OWL 2 is shown in Figure 2.2 [27].

Similar to RDF data, OWL 2 ontologies are also formed by triples which can be represented in some serialization formats, such as RDF/XML, OWL/XML\(^7\), Turtle\(^8\). If OWL 2 ontologies are published with an open licence, they could also be categorized as 5-star Linked Open Data. In this case, we could use Linked Data Platform to publish and manage them.

In this thesis, we focus on only OWL 2 ontologies. Thus all ontologies mentioned in this thesis are OWL 2 ontologies.

\(^6\)OWL 2 Web Ontology Language: http://www.w3.org/TR/owl2-overview/
\(^7\)OWL/XML: http://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/
\(^8\)Turtle: http://www.w3.org/TeamSubmission/turtle/
Ontology Metadata Vocabulary

Ontology Metadata Vocabulary\(^9\) (OMV) [13] is a proposed standard for ontology metadata. Metadata is meant as machine processable information for the Web, which is used for describing information resources and improving their accessibility.

OMV has been developed since 2005, which is based on the ontology development experience of a group of ontology developers. OMV is a vocabulary of terms and definitions describing ontologies. The main classes and properties of OMV is shown in Figure 2.3. An example of ontology metadata expressed using OMV is demonstrated in Listing 2.6.

Listing 2.6: Ontology metadata expressed using OMV

\(^9\)OMV: http://sourceforge.net/projects/omv2/
Figure 2.3: OMV overview [13]
2.6 SPARQL

SPARQL is a query language recommended by the W3C for querying RDF data. The latest version of SPARQL is SPARQL 1.1. Inspired by SQL, SPARQL uses a select-from-where syntax to query or update RDF graphs. There are four types of SPARQL queries: (1) SELECT queries which return variable bindings; (2) ASK queries, i.e., boolean “yes/no” queries; (3) CONSTRUCT queries, by which new RDF graphs can be constructed from the query results; (4) UPDATE queries, which support of updating, creating, and removing RDF graphs.

To find the student who takes the Database Technology course in the RDF data shown in Listing 2.1, the following SPARQL query could be applied, where $x$ is a variable indicated by a “?”.

```sparql
PREFIX ex: <http://example.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x
WHERE {
  ?x rdf:type ex:Student .
  ?x ex:takesCourse ex:DatabaseTechnology .
}
```

|Listing 2.7: A SPARQL query example|

2.7 REST

REST [8], stands for Representational state transfer, is an architectural style that defines a set of constraints that applies to the architecture of a distributed system. RESTful web services are the result of applying these constraints to services that utilize web standards such as URIs, HTTP and XML. REST is not protocol-specific, but when people talk about REST, they usually mean REST over HTTP [3]. The most noticeable feature of RESTful web services is the standardized format of URIs:

```
http://host:port/path?queryString#fragment
```

For example, in other web services, we may use the following URI to identify a document of a product:

```
http://example.com/product1.html
```

In RESTful web services, the URI would be:

```
http://example.com/product1/
```

In addition, REST has been widely applied by large web sites on the Internet, e.g., Facebook.com, Google.com, Amazon.com.

More details about REST can be found in Roy Thomas Fielding’s Doctoral dissertation [8].

---

10SPARQL 1.1: http://www.w3.org/TR/sparql11-query/
2.8 Agile Development in Evolutionary Prototyping Technique

Agile Software Development is a software development approach proposed by a group of software developers in February 2001 [25]. The following principles should be applied on Agile Software Development:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Evolutionary prototyping is an variant of Software prototyping, which refers to the activity of creating prototypes of software applications by continually refining and rebuilding the prototypes in a structured manner. Unlike Throwaway Prototyping, Evolutionary Prototyping would produce robust and working systems.

Based on principles of Agile Software Development and the approach of Evolutionary prototyping, Agile Development Development in Evolutionary Prototyping Technique (ADEPT) [6] is introduced by ABB Inc. to increase agility and maturity on their technology development projects.

ADEPT models has three phases:

1. Project Evaluation phase: team members meet with the customer to gather desired features and requirements.

2. Feature Development phase: developers employ principles of Agile Software Development to plan, design, develop, test, and integrate features for the software prototype. The customer would participate in initial evaluations. Based on feedback of the customer, features would be revised or added.

3. Project Completion phase: the development team validates the system, delivers the working prototype as the final system to the customer, and conducts a project evaluation.

2.9 Related works

A few ontology repositories have been developed. A survey of ontology repositories [30] was conducted in 2011, which provides an overview of eleven ontology
repositories. These ontology repositories were designed and implemented for different purpose and domain with a variety of features. Some of these ontology repositories, e.g., OBO Foundry [35], MMI [33], ONKI [37], are not domain-independent but explicitly focusing on a particular field and specific usage scenarios. The focus of these ontology repositories is on the content of the carefully curated collection of ontologies they provide, rather than on the features of the system. Only few ontology repositories, e.g., BioPortal [28] and OntoSearch2 [31], support publishing ontologies. However, they still remain as collections of ontologies, which do not provide accessible URIs for hosting ontologies. Ontology developers have to publish the ontologies online using their own accessible URIs, then submit the ontologies to these ontology repositories.
Chapter 3

Software requirements

This chapter describes the software requirements of our 5-star compliant Linked Open Data ontology repository. It includes functional requirements and non-functional requirements. The requirements documented in this chapter are elicited from the customer, i.e., Semmtech BV.

We applied the ADEPT model (see Section 2.8) to incrementally clarify requirements during the development of the software prototype. We recorded the requirements in different versions of software requirements document. The requirements shown in this chapter are from the latest version of the software requirements document.

3.1 General Description

3.1.1 Product Functions

This ontology repository is a prototyping software that provides users with a centralized platform to share and manage ontologies. It allows users to store (Section 3.2.2), publish (Section 3.2.4), access (Section 3.2.7), modify (Section 3.2.5) and version (Section 3.2.6) ontologies via RESTful APIs using HTTP commands.

3.1.2 User Objectives

This section addresses the different user roles and the main purpose the ontology repository will have for them.
Ontology users

Ontology users can view, browse and search accessible ontologies in the repository.

Ontology developers

Ontology developers are authorized users that can upload ontologies to the repository and view, modify, delete these ontologies. These users can also be the creators of ontologies.

System administrators

The system administrators oversee the entire repository and have the right to configure the system, to create and remove ontologies hosted in the repository. The system administrators are given fine-grain control over creation and management of users, passwords and roles.

3.2 Specific requirements

The specific requirements of the product discussed in this section are divided into logical subsections. The product will adhere to these requirements. Furthermore any requirements resulting from additional requests are included here.

For prioritizing the specific requirements for the software, the MoSCoW model [5] will be used. The capital letters in MoSCoW stand for:

M Must have; these requirements are essential for the software.

S Should have; these requirements are not critical for the product to work, but are nearly as important as the must haves, meaning they must be implemented if at all possible.

C Could have; requirements which are not critical to the product’s success. If they can be implemented with little development costs, they can increase the Clients satisfaction.

W Won’t have; these requirements will not be implemented in this Project. However, they would be nice to have in future versions of the product.
3.2.1 Non-functional requirements

UCR31 \(^1\)  
Must have  
The repository shall choose open source solutions which match up to the requirements. If no open source alternative exists which offers the functionality required, commercial packages will be compared.

UCR32  
Must have  
The repository shall respond to HTTP requests of publishing ontologies with 1000 triples represented in Turtle format, within 10 seconds after the HTTP requests are sent from an user who is located in the Netherlands using a dedicated Internet connection with more than 10 MB/s download speed and more than 2 MB/s upload speed.

UCR35  
Must have  
The repository shall respond to HTTP requests of retrieving ontologies with 1000 triples represented in Turtle format, within 10 seconds after the HTTP requests are sent from an user who is located in the Netherlands using a dedicated Internet connection with more than 10 MB/s download speed and more than 2 MB/s upload speed.

UCR33  
Must have  
The developed application should be accompanied by documentations on how it is structured.

3.2.2 Storage of ontologies

UCR1  
Must have  
Ontologies shall be stored in a RDF triplestore attached to the repository.

UCR2  
Must have  
Ontology metadata shall be specified and stored using a standard format based on the Ontology Metadata Vocabulary [13].

UCR3  
Must have  
Metadata of each ontology shall be stored as triples within the ontology itself.

\(^1\) To be noted that the labels of these software requirements shown in this chapter are not continuous, because some labels together with their corresponding software requirements have been removed from previous versions based on discussions with the customer.
3.2.3 Interfaces

**UCR26**  
Must have  
The repository shall provide RESTful web APIs according to [36].

**UCR27**  
Must have  
The repository shall provide users with a web application interface to browse ontologies. This web application will be created for Chrome, without guaranteeing other web browsers.

**UCR28**  
Should have  
The repository shall allow users to execute SPARQL queries on each stored ontology.

**UCR30**  
Should have  
The repository shall inform users with error messages if users try to access non-existing ontologies or resources in the repository. This error message is either a web page with a descriptive message, or a HTTP Error code if using the RESTful API.

3.2.4 Ontology publishing

**UCR4**  
Must have  
The repository shall allow ontology developers to publish ontologies via the HTTP POST request described in [36].

**UCR6**  
Should have  
The repository shall check syntactic correctness of each published ontology.

**UCR7**  
Must have  
The repository shall automatically assign a resolvable URI to each published ontology.

**UCR36**  
Could have  
The repository shall check semantic consistency of each published ontology.

**UCR32**  
Must have  
The repository shall be able to assign and reserve a resolvable base URI for an ontology, after providing the mandatory metadata.
3.2.5 Modifying of ontologies

**UCR8**  
*Must have*  
The repository shall allow ontology developers to overwrite resources from ontologies via the HTTP PUT request described in [36].

**UCR34**  
*Should have*  
If HTTP PUT requests are received, the repository shall require the HTTP If-Match header and HTTP ETags to detect collisions as described in [36].

**UCR9**  
*Must have*  
The repository shall allow ontology developers to remove resources from ontologies via the HTTP DELETE request described in [36].

**UCR10**  
*Must have*  
The repository shall allow only modifications performed by authenticated users.

**UCR11**  
*Must have*  
The repository shall allow ontology developers to modify ontologies using the SPARQL UPDATE queries.

**UCR29**  
*Must have*  
The repository shall allow ontology developers to modify metadata of ontologies.
3.2.6 Ontology versioning

**UCR12**  
*Must have*  
The repository shall automatically assign a distinct version number\(^2\) to each ontology hosted in the repository.

**UCR13**  
*Must have*  
The repository shall contain a function which can input two ontology URIs and returns the structural difference between those ontologies in a structured format (e.g., XML, JSON).

**UCR14**  
*Must have*  
The repository shall be able to list URLs of all versions of an ontology hosted in the repository.

3.2.7 Accessing ontologies

**UCR15**  
*Must have*  
The repository shall allow users to perform keyword\(^3\) search over ontologies hosted in the repository.

**UCR16**  
*Must have*  
The repository shall allow users to retrieve triples in which the resource is the subject via the HTTP GET request described in [36].

**UCR17**  
*Should have*  
The repository shall allow users to download all the triples of an ontology hosted in the repository in either RDF/XML or Turtle format.

---

\(^2\) The repository is not responsible for the semantics of these identifiers and their possible relationships.

\(^3\) Terms appeared in literals and URIs.
3.2.8  Administration

UCR18  Must have
The repository shall allow users to create ontology developer accounts.

UCR19  Must have
The repository shall allow an ontology developer to change his/her account password.

UCR20  Must have
The repository shall allow an ontology developer to delete ontologies uploaded by him/her.

UCR21  Must have
The repository shall allow a system administrator to delete ontologies hosted in the repository.

UCR22  Must have
The repository shall allow a system administrator to delete an ontology developer account.

UCR23  Must have
The repository shall allow a system administrator to change the password of an ontology developer account.

3.3  Summary

In this chapter, we presented the software requirements of the ontology repository. We specified features of the ontology repository that were requested by the customer.
Chapter 4

Software design and implementation

In this chapter, we demonstrate the design and implementation of the ontology repository. We discuss in detail the realization of features described by the functional requirements documented in Chapter 3. These features include: ontology storage, ontology publishing, ontology accessing, ontology modification, ontology versioning and administration.

4.1 Storage of ontologies

4.1.1 Triplestores

According to the software requirement UCR1, the ontologies hosted in the repository have to be stored in a triplestore. Triplestore, as the name indicates, is a type of Database Management System (DBMS) specially designed for storage and retrieval of triples.

Triplestores can be broadly classified in two types categories: Native triplestores and Non-native triplestores [23]. Native triplestores are those that are implemented from scratch and exploit the RDF data model to efficiently store and access the RDF data, e.g., AllegroGraph\(^1\), Stardog\(^2\), OWLIM\(^3\). Non-native triplestores are built by adding a RDF specific layer to existing Relational Database Management Systems, such as Jena SDB\(^4\) and Virtuoso\(^5\).

\(^1\)AllegroGraph: http://www.franz.com/agraph/allegrograph/
\(^2\)Stardog: http://stardog.com/
\(^3\)OWLIM: http://www.ontotext.com/owlim
\(^4\)Jena SDB: http://jena.apache.org/documentation/sdb/
\(^5\)Virtuoso: http://virtuoso.openlinksw.com/
Since Native triplestores are usually more efficient than Non-native triplestores, we decided to integrate a Native triplestore into the repository application. Based on the requests and discussions with the customer, three popular Native triplestores were considered: AllegroGraph 4.11, OWLIM-Lite 5.3.5777 and Stardog 1.1.5. Several aspects of these triplestores, which are related to the software requirements, have been compared (see Table 4.1).

### 4.1.2 AllegroGraph

Based on the comparison shown in Table 4.1, we decided to select AllegroGraph 4.11 as the triplestore attached to the ontology repository. The reasons are listed below:

1. A user-friendly WEB GUI is provided by AllegroGraph 4.11 which is preferred by the customer.

2. A full-function free edition of AllegroGraph 4.11 is offered with the limitation on only the number of triples (5 million) can be stored. Since an ontology usually contains only thousands of triples, 5 million triples is sufficient for developing and testing the system. After the system is delivered to the customer, the customer could purchase the developer version that allows up to 500 million triples.

3. The function for management of user accounts supported by AllegroGraph 4.11 could meet the software requirements UCR18, UCR19, UCR21, UCR22 and UCR23, which is described in detail in Section 4.3.

4. The logical storage structure of AllegroGraph 4.11 could be fully utilized by the ontology repository, which is illustrated in Figure 4.1.

Moreover, a comprehensive documentation of AllegroGraph 4.11 is available in [http://www.franz.com/agraph/support/documentation/current/](http://www.franz.com/agraph/support/documentation/current/).
<table>
<thead>
<tr>
<th>Name</th>
<th>JAVA APIs</th>
<th>Security</th>
<th>WEB GUI</th>
<th>Scalability</th>
<th>SPARQL 1.1 Endpoint</th>
<th>License</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllegroGraph 4.11</td>
<td>Sesame, Jena</td>
<td>User role management, Triple Level access control</td>
<td>Good GUI with full management functions</td>
<td>Free edition: 5 million triples, Developer edition: 500 million triples</td>
<td>Fully supported</td>
<td>Proprietary</td>
<td>Linux</td>
</tr>
<tr>
<td>OWLIM-Lite 5.3.5777</td>
<td>Sesame</td>
<td>Not supported</td>
<td>Semi GUI with limited management functions</td>
<td>100 million triples</td>
<td>Fully supported</td>
<td>Licensed for use free of charge</td>
<td>Linux, Windows</td>
</tr>
<tr>
<td>Stardog 1.1.5</td>
<td>Stardog API with Sesame and Jena bindings</td>
<td>User role management, database level access control</td>
<td>Not supported</td>
<td>Free edition: 10 databases, 25 million triples per database, 4 users, 4 roles, Developer edition has no triple number limitations.</td>
<td>Fully supported</td>
<td>Proprietary</td>
<td>Linux</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of three Native triplestores
Figure 4.1: Mapping between the logical storage structure of AllegroGraph with the structure of our ontology repository. Catalogs are named by distinct ontology identifiers. Each catalog has a number of repositories which are named by distinct ontology version identifiers. Triples of different versions of ontologies are stored in corresponding repositories in corresponding catalogs.

4.2 Designed URIs

According to the software requirements UCR7 and UCR12, the ontology repository shall automatically assign a resolvable URI and a distinct version identifier to each hosted ontologies. The version identifier would only indicate the time at which the ontology is published to the ontology repository and makes sure the assigned URIs are distinct. The ontology repository would not be responsible for the semantics of this version identifier. The ontology repository would allow users to control and modify the content of the published ontologies, thus the users should be responsible for the content of ontologies and the semantics of the assigned version identifiers.

We design the URIs of the ontology repository using the well-known REST style that is also specified in the Software requirement UCR26. The following

6REST: Representational State Transfer
URI is exposed in the ontology repository:
http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}/

There are three variables in this URI: ontologyID, version and resource. The ontologyID indicates the identity of an ontology hosted by the repository. The version describes the version number of the ontology. As for the resource, it refers to a class, a datatype, an object property, a data property, an annotation property, or a named individual defined in the version of the ontology. To be noted that the resource identified by the designed URI is also a Linked Data Platform resource, thus the principles of the Linked Data Platform are supported by the ontology repository.

Based on the logical storage structure of AllegroGraph 4.11 depicted in Figure 4.1, we could directly use each ontologyID as a distinct catalog name and each version as a repository name in a catalog. For instance, an ontology with the URI http://repo.chencai.info/ns/pizza/1373728064025/ could be stored in the repository named “1373728064025” in the catalog named “pizza” in AllegroGraph 4.11.

An overview of supported HTTP requests are listed as follows (more details can be found in the rest of this chapter):

1. http://repo.chencai.info/ns/
   • HTTP GET: Get the list of URIs of hosted ontologies in the repository. If triples are requested, return the result as a LDP container.
   • HTTP POST: Send required metadata to the repository. A value included in the metadata would be used to determine the ontologyID. A resolvable URI will be automatically assigned and returned.

2. http://repo.chencai.info/ns/{ontologyID}/
   • HTTP GET: Get the list of URIs of different versions of the ontology annotated by the ontologyID.
   • HTTP POST: Send the metadata of a new version of the ontology to the repository. A resolvable URI will be automatically assigned and returned.

3. http://repo.chencai.info/ns/{ontologyID}/{version}/
   • HTTP GET: Get the content of the ontology version identified by the URI in Turtle or RDF/XML format.
   • HTTP POST: Append the ontology version identified by the URI according to the content of the HTTP request.
• HTTP PUT: Replace the content of the ontology version identified by this URI by the content of the HTTP request.

• HTTP DELETE: Remove the version of the ontology identified by the URI in the repository.


• HTTP GET: Get the content of the resource identified by the URI in Turtle or RDF/XML format.

• HTTP PUT: Replace the content of the resource identified by this URI by the content of the HTTP request.

• HTTP DELETE: Remove the resource identified by this URI in this version of the ontology.

4.3 Security

4.3.1 User management

We utilise the user management functions provided by AllegroGraph 4.11 to meet the software requirements.

In AllegroGraph 4.11, a user is given an username and a password and can only access the system using that username and password. An anonymous user may be allowed who would not require a password to access the system. Each user has a set of permissions that can be specified by the system administrators named super users. The permissions include read, write, and read/write permission for each catalog and repository.

Additionally, each user in AllegroGraph 4.11 can be assigned with several roles which are collections of permissions. One could achieve the same end result by simply giving each user the permissions associated with the user’s role(s).

According to the software requirements of the ontology repository, three types of users are defined: ontology users, ontology developers and system administrators. Based on the similarities of types of users in the ontology repository and in AllegroGraph 4.11, we specify the ontology users in the ontology repository as the anonymous user in AllegroGraph 4.11, the ontology developers in the ontology repository as the users with a role named “developer” in AllegroGraph 4.11, and the system administrators in the ontology repository as the super users in AllegroGraph 4.11. In this case, the ontology repository could use the access control functions provided by AllegroGraph 4.11 to authenticate and authorize users, which is discussed in detail in Section 4.3.2.
Additionally, AllegroGraph 4.11 provides a WEB GUI and Java APIs to manage users, roles and permissions, which could meet the software requirements UCR18, UCR19, UCR21, UCR22 and UCR23.

We have also considered to implement this feature from scratch, but due to the time constraint, we may not be able to design it as robust as the one supported by AllegroGraph 4.11. Furthermore, the customer prefers the WEB GUI for user management provided by AllegroGraph 4.11.

### 4.3.2 Access control

As mentioned above, each user in AllegroGraph 4.11 can be assigned with permissions for each catalog and repository. To ensure the ontologies hosted in the ontology repository is accessible, we grant read permissions of all catalogs to the anonymous user and to the role named “developer” in AllegroGraph 4.11. The write permission of a catalog is granted to the user who created the catalog.

In order to authenticate and authorize users, the ontology repository requires users to specify the HTTP basic authentication header in each HTTP request. If no HTTP basic authentication header is specified, the ontology repository would categorize it as an ontology user (corresponding to the anonymous user in AllegroGraph 4.11). For example, the HTTP DELETE request shown in Listing 4.1 includes an authentication header which specifies the authentication mechanism (in this case Basic) followed by base64 encoded username and password.

```
DELETE /ns/pizza/1373728064025/ HTTP/1.1
Host: repo.chencai.info
Authorization: Basic Y2hlbjpjYWk=
```

Listing 4.1: A HTTP request with the HTTP basic authentication header

After receiving a HTTP request, the ontology repository decodes the HTTP basic authentication header to get the username and the password. Then the ontology repository sends the username and the password to AllegroGraph 4.11 via Java APIs provided by AllegroGraph 4.11 to authenticate and authorize the user. If authentication or authorization fails, HTTP error code 401 (Access Denied) would be responded. For example, decoding the HTTP basic authentication header in Listing 4.1, the ontology repository could get the username “chen” and the password “cai”. According to the HTTP DELETE request, the ontology repository will send the username and the password to AllegroGraph 4.11 to check the correctness of the password and if the user has the write permission for catalog named “pizza”.

With the help of access control functions provided by AllegroGraph 4.11,
the ontology repository could allow only authorized users to publish and modify ontologies, which meets the software requirement UCR10.

In addition, if an user would like to prevent other users from accessing an ontology uploaded by her, the user could request the system administrators to remove the read permission of other users for the corresponding catalog or repository in AllegroGraph 4.11.

4.4 Publishing and modifying of ontologies

4.4.1 Publishing ontologies

Publishing an ontology to the ontology repository requires two steps: (1) Sending mandatory metadata of the ontology to the ontology repository to reserve a base URI; (2) Sending other content of the ontology to the ontology repository.

Described in the requirements UCR2 and UCR3, the metadata of each published ontology shall be specified based on the Ontology Metadata Vocabulary (OMV) [13] and stored as triples within the ontology itself. According to the request from the customer, the following classes and properties of the OMV are selected as mandatory metadata:

Class:
- omv:Ontology
- omv:OWL

Property:
- omv:URI
- omv:creationDate
- omv:resourceLocator
- omv:acronym
- omv:name
- omv:description
- omv:hasOntologyLanguage

In addition, at least one of the following two collections of classes and properties should be included in the mandatory metadata:

1. omv:hasCreator, omv:Person, omv:firstName, omv:lastName, omv:eMail

2. omv:hasCreator, omv:Organisation, omv:name, omv:acronym

where omv:Person and omv:Organisation are classes, others are properties.

According to the requirement UCR32, after providing the mandatory metadata, the ontology repository shall assign and reserve a resolvable base URI
for an ontology. An example of such mandatory metadata is shown in Listing 4.2, in which there is an instance of the omv:Ontology class that is provided as an anonymous resource. The ontology repository requires only such anonymous resources because the URIs of the ontologies could not be provided by users in advance. The same situation holds for the value of omv:URI and omv:resourceLocator. The ontology repository should automatically complete them after other mandatory metadata is provided. Additionally, the ontology repository would take the value of the omv:acronym property as the ontologyID requested by the user.

```turtle
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix omv: <http://omv.ontoware.org/2005/05/ontology#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
[] a owl:Ontology , omv:Ontology ;
  omv:uri ""^^xsd:string ;
  omv:resourceLocator ""^^xsd:string ;
  omv:version ""^^xsd:string ;
  omv:acronym "pizza"^^xsd:string ;
  omv:name "The Pizza Ontology"^^xsd:string ;
  omv:description "This ontology is used for modeling pizza."^^xsd:string ;
  omv:creationDate "2013-05-10 14:57:00"^^xsd:dateTime ;
  omv:hasOntologyLanguage omv:OWL ;
  omv:hasCreator
    [ a omv:Person ;
      omv:firstName "Chen"^^xsd:string ;
      omv:lastName "Cai"^^xsd:string ;
      omv:eMail "mail@chencai.info"^^xsd:string
    ] ;
```

Listing 4.2: Mandatory metadata required for reserving a URI

We design the procedure of publishing an ontology to the ontology repository as follows:

1. According to the username and the password obtained from the system administrator, the user specifies the HTTP basic authentication header of a HTTP POST request. Then the user adds the mandatory metadata (see Listing 4.2 for example) in Turtle format in the message body of the HTTP POST request.

2. The user sends the HTTP POST request to URI `http://repo.chencai.info/ns/`.

3. The ontology repository authenticates the user and verifies the correctness of the mandatory metadata.

4. Take the value of omv:acronym as ontologyID (e.g., pizza). The ontology repository would check if this ontologyID is already occupied. If yes, HTTP error code 400 (Bad Request) would be returned. Otherwise
the ontology repository would create a catalog named `ontologyID` in AllegroGraph 4.11 and a repository named `version` (e.g., 1373728064025) in the catalog. The `version` is generated based on the Unix epoch time when receives the HTTP request. Complete the triples of the metadata using the reserved URI (e.g., `http://repo.chencai.info/ns/pizza/1373728064025`) and store the triple into the repository named `version` in AllegroGraph 4.11. In the meantime, update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details).

5. The ontology repository responds HTTP code 201 (Created) with the reserved URI.

6. The user could append triples to the ontology by sending HTTP POST requests to the reserved URI.

### 4.4.2 Modifying ontologies via HTTP requests

According to the software requirements, the ontology repository shall support three types of HTTP requests for modifying ontologies: HTTP POST requests, HTTP PUT requests and HTTP DELETE requests.

**HTTP POST requests for modifying ontologies**

Based on the specification of Linked Data Platform, the ontology repository shall allow users to append triples to hosted ontologies by sending HTTP POST requests to the URI of the ontology, i.e., `http://repo.chencai.info/ns/{ontologyID}/{version}`. In order to accurately process the HTTP POST requests, the following HTTP headers are required:

- **Authorization**: contains base64 encoded username and password.
- **Content-Type**: specifies the format (application/rdf+xml or text/turtle) of triples enclosed in the message body of the request.
- **If-Match**: refers to the ETag value provided by the users. More details are described in Section 4.4.4.

After receiving the HTTP POST request with required HTTP Headers, the ontology repository would process the request as follows:

1. Check the existence of the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.
2. Verify the correctness of the username and password and the write permission of the user for the repository named \(version\) in the catalog named \(ontologyID\) in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no write permission.

3. Compare the ETag value included in the If-Match header with the ETag value stored in the ontology repository. If they do not match, respond HTTP code 412 (Condition Failed).

4. Examine the syntactic correctness of the triples included in the request message body. If the triples are syntactic incorrect, return HTTP error code 400 (Bad request).

5. Using Java APIs provided by AllegroGraph 4.11, append the triples into the repository named \(version\) in the catalog named \(ontologyID\) in AllegroGraph 4.11. In the meantime, update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details). Return HTTP code 200 (OK) if no error occurs.

To be noted that the ontology repository would not check the semantic consistency of uploaded triples, because this process is very time consuming and resource consuming.

**HTTP DELETE requests for modifying ontologies**

The ontology repository supports using HTTP DELETE requests to remove hosted ontologies. By sending HTTP DELETE requests with required HTTP headers to URIs of the form `http://repo.chencai.info/ns/{ontologyID}/{version}`, the user would be able to remove the corresponding hosted ontologies from the ontology repository.

For the HTTP DELETE requests, the following HTTP headers are required

- **Authorization**: contains base64 encoded username and password.
- **If-Match**: refers to the ETag value provided by the users. More details are described in Section 4.4.4.

The ontology repository handles the HTTP DELETE requests in the following steps:

1. Check the existence of the repository named \(version\) in the catalog named \(ontologyID\) in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.
2. Verify the correctness of the username and password and the write permission of the user for the repository named version in the catalog named ontologyID in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no write permission.

3. Compare the ETag value included in the If-Match header with the ETag value stored in the ontology repository. If they do not match, respond HTTP code 412 (Condition Failed).

4. Using Java APIs provided by AllegroGraph 4.11, delete the repository named version in the catalog named ontologyID in AllegroGraph 4.11. Update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details). Return HTTP code 200 (OK) if no error occurs.

**HTTP PUT requests for modifying ontologies**

HTTP PUT requests can be used for replacing the content of the hosted ontologies according to the triples specified in the message body of the requests. The following HTTP headers are required for the HTTP PUT requests:

- Authorization: contains base64 encoded username and password.

- Content-Type: specifies the format (application/rdf+xml or text/turtle) of triples enclosed in the message body of the request.

- If-Match: refers to the ETag value provided by the users. More details are described in Section 4.4.4.

After receiving the HTTP PUT request with required HTTP Headers, the ontology repository conducts the procedure as follows:

1. Check the existence of the repository named version in the catalog named ontologyID in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.

2. Verify the correctness of the username and password and the write permission of the user for the repository named version in the catalog named ontologyID in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no write permission.

3. Compare the ETag value included in the If-Match header with the ETag value stored in the ontology repository. If they do not match, respond HTTP code 412 (Condition Failed).
4. Examine the syntactic correctness of the triples included in the request message body. If the triples are syntactic incorrect, return HTTP error code 400 (Bad request).

5. Using Java APIs provided by AllegroGraph 4.11, remove all triples stored in the repository named version in the catalog named ontologyID in AllegroGraph 4.11. Then append the triples included in the request message body into the same repository in AllegroGraph 4.11. In the meantime, update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details). Return HTTP code 200 (OK) if no error occurs.

To be noted that the function of this feature does not overlap with the function of modifying ontologies via HTTP POST requests described above. Using the HTTP PUT requests, the triples that are stored previously would firstly be removed, then new triples would be added. But using the HTTP POST requests, users are only allowed to add new triples. The triples that have been stored previously would not be altered.

4.4.3 Modifying resources via HTTP requests

The ontology repository allows users to modify resources via HTTP PUT requests and HTTP DELETE requests as described in the software requirements UCR8 and UCR9. In the ontology repository, all resources are also Linked Data Platform resources (LDPRs), which are suggested to be identified by URIs of the form http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}. If resources included in a hosted ontology are identified by URIs of other forms, these resources would not be able to be modified via the HTTP requests described in this section.

HTTP DELETE requests for modifying resources

The users could send a HTTP DELETE request to the URI of a resource to remove the triples describe the resource. Recall the description of LDPRs in Section 2.3.1, the subject of all triples of a LDPR is the URI of the LDPR. To accurately and efficiently locate the triples of a LDPR, SPARQL queries could be used. For example, to remove the triples describe a LDPR identified by the URI http://repo.chencai.info/ns/pizza/1374089830603/Italy, the SPARQL query shown in Listing 4.3 could be used.
It is worth to mention that the Java APIs provided by AllegroGraph 4.11 also supports removing triples. However, these Java APIs achieve this function as follows: (1) retrieve all triples in a repository in AllegroGraph 4.11 and store these triples in a data structure (e.g., Jena model); (2) remove some triples from the data structure; (3) dump all the triples of the modified data structure back to the repository in AllegroGraph 4.11. Considering the repository contains millions of triples, this process would be very inefficient. Applying SPARQL queries, the ontology repository can command the query engine to directly work on a small number of triples, which is more efficient.

For the HTTP DELETE requests, the following HTTP headers are required:

- Authorization: contains base64 encoded username and password.
- If-Match: refers to the ETag value provided by the users. More details are described in Section 4.4.4.

The ontology repository would process a HTTP DELETE request sent to URI \texttt{http://repo.chencai.info/ns/\{ontologyID\}/\{version\}/\{resource\}} in the following steps:

1. Check the existence of the resource in the repository named \texttt{version} in the catalog named \texttt{ontologyID} in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.

2. Verify the correctness of the username and password and the write permission of the user for the repository named \texttt{version} in the catalog named \texttt{ontologyID} in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no write permission.

3. Compare the ETag value included in the If-Match header with the ETag value stored in the ontology repository. If they do not match, respond HTTP code 412 (Condition Failed).

4. Using JAVA APIs, execute the SPARQL query composed based on the URI of the resource in the repository named \texttt{version} in the catalog named \texttt{ontologyID} in AllegroGraph 4.11. Update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details). Return HTTP code 200 (OK) if no error occurs.
HTTP PUT requests for modifying resources

Using a HTTP PUT request, users could replace all triples of a resource by the triples provided in the request message body. This modification could be divided into two steps: (1) removing all triples of the resource; (2) adding triples provided via the HTTP request. For step (1), the SPARQL query described above (e.g., the query shown in Listing 4.3) could be applied. As for step (2), the following SPARQL UPDATE query could be used.

```sparql
INSERT DATA
{
    ......
    (Triples enclosed in the request message body)
    ......
}
```

Listing 4.4: SPARQL query to add triples

Similarly, in order to accurately process the HTTP PUT requests, the following HTTP headers are required:

- Authorization: contains base64 encoded username and password.
- Content-Type: specifies the format (application/rdf+xml or text/turtle) of triples enclosed in the message body of the request.
- If-Match: refers to the ETag value provided by the users. More details are described in Section 4.4.4.

The following steps would be performed after the ontology repository receiving the HTTP PUT request with required HTTP headers sent to the URI `http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}`:

1. Check the existence of the resource in the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.

2. Verify the correctness of the username and password and the `write` permission of the user for the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no `write` permission.

3. Compare the ETag value included in the If-Match header with the ETag value stored in the ontology repository. If they do not match, respond HTTP code 412 (Condition Failed).
4. Examine the syntactic correctness of the triples included in the request message body. Also, verify if all triples enclosed in the request message body all have the same subject http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}. If the triples are syntactic incorrect or some subjects of the triples are different with the URI, return HTTP error code 400 (Bad request).

5. Using Java APIs provided by AllegroGraph 4.11, execute the SPARQL query to remove all triples and the other SPARQL query to add new triples in the repository named version in the catalog named ontologyID in AllegroGraph 4.11. In the meantime, update the stored ETag value (see Section 4.4.4 for details) and the index of SIREn (see Section 4.5.4 for details). Return HTTP code 200 (OK) if no error occurs.

### 4.4.4 Collision detection

By the software requirement UCR34, the ontology repository is required to support collision detection after receiving HTTP PUT requests. The purpose of collision detection is to ensure the user is not modifying an ontology that has changed since the user last retrieved its representation. Therefore, even though it is not required, we implemented this feature for other HTTP requests that could be used to modify ontologies.

In the ontology repository, collision detection is achieved by comparing the value included in the If-Match header with the corresponding ETag value stored as an object of a triple stored in the repository named “etags” in the system catalog in AllegroGraph 4.11:

- After an ontology is published, a resolvable URI would be reserved for the ontology. Store the triple of the form

  (URI of an ontology, hasETag, ETag value),

  in the repository named “etags” in the system catalog in AllegroGraph 4.11, where the ETag value is equal to the Unix epoch time at which the ontology is published.

- An ETag value would be included in the response of a HTTP GET request as described in Section 4.5.2. This ETag value is obtained from the triple stored in “etags” repository, where the triple’s subject is the URI of the retrieved ontology. The user should remember this ETag value and provide it when If-Match headers are required.
• When a modification to an ontology is performed by sending a HTTP request to the URI of the ontology or to the URI of a resource in the ontology, the ontology repository would change the object of the corresponding triple to the Unix epoch time at which the modification is performed. In such way the stored ETag values would be updated.

• The If-Match header is required to be specified in each HTTP request that could be used to modify an ontology. The value included in the If-Match header would be compared with the ETag value stored in corresponding triple in “etags” repository. If they are not equal, a collision is detected.

4.5 Accessing ontologies

4.5.1 Listing URIs of hosted ontologies

As mentioned above, users could send HTTP GET requests to the ontology repository to list URIs of hosted ontologies.

Firstly, an user could send a HTTP GET request to the URI http://repo.chencai.info/ns/ to retrieve a list of URIs of the hosted ontologies. These URIs are generated based on the names of catalogs in AllgroGraph 4.11. If a representation in Turtle is requested, the response shown in Listing 4.5 would be returned, which is a Linked Data Platform container.

```prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
prefix owl: <http://www.w3.org/2002/07/owl#> .
prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
prefix ldp: <http://www.w3.org/ns/ldp#>.
<http://repo.chencai.info/ns/> a ldp:Container ;
  rdfs:member <http://repo.chencai.info/ns/pizza> ,
  <http://repo.chencai.info/ns/country> ,
  <http://repo.chencai.info/ns/school> ,
  <http://repo.chencai.info/ns/people> .
```

Listing 4.5: List of URIs of hosted ontologies

Similarly, if the HTTP GET request is sent to http://repo.chencai.info/ns/{ontologyID}, the ontology repository would list URIs of all versions of the ontology named ontologyID. This list is generated based on the names of repositories in the catalog named ontologyID in AllegroGraph 4.11. An example of such list is demonstrated in Listing 2.5.

39
4.5.2 Retrieving ontologies via HTTP GET requests

Users could retrieve all triples of an ontology by sending a HTTP GET request to the URI of the form \texttt{http://repo.chencai.info/ns/{ontologyID}/\{version\}}. Two HTTP headers can be given for this HTTP GET request:

- **Accept** (required): indicates the desired format (application/rdf+xml or text/turtle) of triples
- **Authorization** (optional): contains base64 encoded username and password

Once the ontology repository receives this HTTP GET request, it processes the request as follows:

1. Check the existence of the repository named \texttt{version} in the catalog named \texttt{ontologyID} in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.

2. If the Authorization header is specified, verify the correctness of the username and password. If no Authorization header is given, classify this user as the anonymous user in AllegroGraph 4.11. Check the \texttt{read} permission of the (anonymous) user for the repository named \texttt{version} in the catalog named \texttt{ontologyID} in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no \texttt{read} permission.

3. Using Java APIs provided by AllegroGraph 4.11, retrieve all triples in the desired format as specified in the HTTP Accept header and include the triples in the message body of a HTTP response. As described in Section 4.4.4, get corresponding ETag value from the “etags” repository and contain this value in the ETag header of the HTTP response. If any error occurs, return HTTP error code 500 (Internal server error).

4.5.3 Retrieving resources via HTTP GET requests

Users could also retrieve all triples that describe a resource by sending a HTTP GET request to the URI of the resources. The resource should be identified using the URI of the form \texttt{http://repo.chencai.info/ns/{ontologyID}/\{version\}/{resource}}, otherwise the ontology repository cannot retrieve it. To efficiently and accurately retrieve all triples of the resource, the SPARQL CONSTRUCT query described in Listing 4.6 could be used.

We have also considered to use the SPARQL DESCRIBE query which is designed to retrieve the relevant triples that describe a resource. However,
there is no standardized mechanism on determining which triples are relevant to a resource. In this case, different SPARQL query engines may give different answers to the same SPARQL DESCRIBE query. Also, AllegroGraph 4.11 does not give detailed description on how SPARQL DESCRIBE queries would be processed in its SPARQL query engine. Therefore, we decided to use the SPARQL CONSTRUCT queries instead.

```
CONSTRUCT
{
  <http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}> ?p ?o
}
WHERE
{
  <http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}> ?p ?o
}
```

Listing 4.6: SPARQL query to retrieve all triples of a resource

Two HTTP headers are used in the HTTP GET requests for retrieving resources:

- Accept (required): indicates the desired format (application/rdf+xml or text/turtle) of triples
- Authorization (optional): contains base64 encoded username and password

After receive a HTTP GET request that is sent to a URL http://repo.chencai.info/ns/{ontologyID}/{version}/{resource}, the ontology repository would process the following steps to respond the request:

1. Check the existence of the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. Return HTTP error code 404 (Not Found) if it does not exist.

2. If the Authorization header is specified, verify the correctness of the username and password. If no Authorization header is given, classify this user as the anonymoususer in AllegroGraph 4.11. Check the read permission of the (anonymous) user for the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. Return HTTP error code 403 (Forbidden) in case of incorrect username and password or no read permission.

3. Based on the URI of the resource, generate a SPARQL CONSTRUCT query as described above. Execute the SPARQL query in the repository named `version` in the catalog named `ontologyID` in AllegroGraph 4.11. If the answer to the SPARQL query is empty, respond HTTP error code 41...
4.5.4 Support for keyword search

According to the software requirement UCR15, the ontology repository shall allow users to perform keyword search over the hosted ontologies. By keyword search, we understand: given a set of terms, find the URIs of the ontologies that contain these terms. Simple keyword-based search has the advantages of (1) being easy to use by users since it hides from the user any structural information of the underlying data collection, and (2) of being applicable on any scenarios [7].

Ontologies need to be indexed before keyword search can be performed over them. Unlike traditional web documents (e.g., HTML, PDF) which are seen as a set or sequence of words, an ontology is semi-structured [1] which is composed of a set of attribute and value pairs and possibly a set of relations to other entities. Also, ontologies, as Linked Data, frequently use URIs to identify resources. Each URI can be split into two parts: a prefix and a local name. For ontologies, terms appeared in the local names of URIs are more frequently used as search keywords by users. For example, in Listing 4.7, usually only the coloured terms are applied as search keywords. Therefore, instead of applying traditional index structures that indistinguishably handle URIs and literals, new index structures are needed to incorporate ontologies.

Listing 4.7: Two triples with coloured terms

```xml
<http://repo.chencai.info/ns/pizza/20050418/GreenPepperTopping>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<http://repo.chencai.info/ns/pizza/20050418/PepperTopping> .

<http://repo.chencai.info/ns/pizza/20051018/GreenPepperTopping>
<http://www.w3.org/2000/01/rdf-schema#label>
"CoberturaDePimentaoVerde"@pt .
```

Several applications have applied such index structures that are specially designed for Linked Data, e.g., uSeekM\(^7\), LuceneSail\(^8\), SARQ\(^9\). However, most of them could not meet our requirements, since they integrate keyword search in the SPARQL query language (see Listing 4.8 for example) which cannot be simply learned and utilized by ordinary users.

---

\(^7\)uSeekM: https://dev.opensahara.com/projects/usekm

\(^8\)LuceneSail: http://dev.nepomuk.semanticdesktop.org/wiki/LuceneSail

\(^9\)SARQ: https://github.com/castagna/SARQ
Listing 4.8: Search for any resource whose RDFS-label value contains the string “person” using LuceneSail

Fortunately, SIREn [7], an open source semi-structured information retrieval engine based on Apache Lucene\textsuperscript{10}, can meet the requirements. According to the test results shown in [7], SIREn applies a node-based indexing model to handle semi-structured data, which is scale to billions of triples on a single commodity server. SIREn can also sustain a query rate of 17 queries per second to 292 queries per second.

To integrate SIREn in the ontology repository, we need to synchronize the data stored in AllegroGraph 4.11 with the index of SIREn. The synchronization would be conducted using Java APIs provided by SIREn, when (1) a new ontology is published; (2) an existing ontology is modified or deleted.

One can perform keyword search in SIREn by specifying parameters in the HTTP APIs provided by SIREn. SIREn would return a ranked list of URIs of ontologies that contain the search terms given by users. For example, send a HTTP GET request to SIREn to search ontologies contain a term “country” using the URL http://repo.chencai.info/siren/select?indent=on&q=country&fl=*%2Cscore&qt=siren, SIREn would respond the result in XML as it is shown in Listing 4.9.

Since all URIs of ontologies hosted in the ontology repository are resolvable, to further discovery the content of ontologies included in the search results, users could utilize other interfaces provided by the ontology repository (e.g., HTTP APIs, Ontology Browser).

\textsuperscript{10}Apache Lucene: http://lucene.apache.org/core/
4.5.5 Ontology Browser

To meet the software requirement UCR27, a web application that supports browsing ontologies shall be provided. In this case, the users would be provided with a user-friendly presentation interface to view published data.

There are few off-the-shelf LOD browsers available. However, to our knowledge, only two of them support browsing OWL 2 ontologies: OntologyBrowser\(^{11}\) and Graphity\(^{12}\), which were considered by us. Compare to Graphity, OntologyBrowser has the following advantages:

- OntologyBrowser can load content of an ontology from a resolvable URI, while Graphity must load content of an ontology from SPARQL endpoints;
- OntologyBrowser is developed for only the purpose of browsing OWL 2 ontologies, which is more lightweight than Graphity;
- OntologyBrowser requires no configuration.

Therefore, we have deployed OntologyBrowser as a component of the ontology repository, which is accessible on http://repo.chencai.info/browser/ontologies. OntologyBrowser supports loading ontologies from resolvable

\(^{11}\text{OntologyBrowser: https://code.google.com/p/ontology-browser/}\)

\(^{12}\text{Graphity: https://github.com/Graphity}\)
Since all ontologies hosted in the ontology repository would be assigned with resolvable URIs, we could fully utilize this feature. Moreover, OntologyBrowser provides expandable class and property hierarchies and auto-completed term search which are shown in Figure 4.2.

![Class hierarchy and auto-completed term search provided by OntologyBrowser](image)

**Figure 4.2**: Class hierarchy and auto-completed term search provided by OntologyBrowser

### 4.5.6 Support for SPARQL query execution

To support SPARQL query execution in an ontology, we design to use a parameter appended to the designed URI to specify SPARQL queries: `http://repo.chencai.info/ns/{ontologyID}/{version}?query={QueryString}`. Users could send HTTP GET requests to the URI to execute queries. The query string would be passed to AllegroGraph 4.11 and executed in corresponding repository in AllegroGraph 4.11. The query results would be returned in the HTTP responses.
Four types of SPARQL queries are supported:

- **SPARQL SELECT query**: Results are given in the SPARQL query results XML format designed by W3C.
- **SPARQL CONSTRUCT query**: Results are given in Turtle format.
- **SPARQL ASK query**: Results (true or false) are returned as strings.
- **SPARQL UPDATE query**: Only authorized users are allowed to execute SPARQL UPDATE queries. String “true” would be returned if the query is successfully executed, otherwise string “false” would be given.

### 4.6 Ontology versioning

#### 4.6.1 Definition

In the ontology community, ontology versioning is defined as the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology [17].

The ontology repository already supports using version number in the designed URIs to identify different variants of an ontology. As the ontology repository already provides users with mechanisms to create new variants and to modify existing variants, the users should take full responsibility of the semantics of the version identifier.

As for managing ontology changes and their effects, our ontology repository provides a diff function to compare two different versions of an ontology in a computationally efficient manner. We will continue discussing this topic from an academic research perspective in Chapter 6.

#### 4.6.2 Detecting and representing ontology difference

Determining the significant syntactic differences between two documents (so-called “diff”) is a standard problem across a wide range of activities, notably software development. Regular textual diffs rely on the assumption that order matters which no longer holds in ontologies, as the order of the triples in an ontology is irrelevant to conceptual meaning of the ontology. Therefore, a notion of “structural difference” has been proposed based on the notion of structural equivalence [27] defined by OWL 2. In turn, using this notion, certain

---

13SPARQL Query Results XML Format: http://www.w3.org/TR/rdf-sparql-XMLres/
types of negligible changes could be ignored, such as the order of axioms or concrete format (e.g., Turtle, RDF/XML).

A variety of tools have been developed to support detecting the structural difference, e.g., OWLDiff [22], SemVersion [38], ContentCVS [16] and Prompt-Diff [29]. However, none of them could return differences in a structured format (e.g., XML, JSON) which is specified in the software requirement UCR13.

Fortunately, Ecco [10], an open source hybrid diff tool for OWL 2 ontologies, could meet this requirement. Ecco has the following features:

- structurally compare two ontologies to obtain difference, specifically, additions (added axioms) and removals (removed axioms)
- distinguish which additions and removals are semantic effectual or inef-fectual
- output the difference in XML format

To allow users to utilize Ecco via HTTP requests, we design the following URI: http://repo.chencal.info/diff/?ont1={URI1}&ont2={URI2}. Users could simply specify two URIs of ontologies as parameters and send a HTTP GET request to the URI designed for running Ecco diff. The differences would be included in the message body of the HTTP response in XML format. To be noted that two provided ontologies must be semantic consistent, otherwise errors may occur.

4.7 Implementation

We applied the Agile Development in Evolutionary Prototyping Technique (see Section 2.8 for details) on implementation of the ontology repository. During the feature development phase, a regular meeting between the developer and the customer was conducted every week. During the meetings, the developer demonstrated implemented features to the customer and asked for feedback. Based on the feedback of the customer, features of the ontology repository were added or revised.

The ontology repository is implemented in Java programming language (Standard Edition 7), mainly based on the following libraries:

- Jersey 1.17: an open source framework for developing RESTful Web Services in Java. Website: https://jersey.java.net/.

- AllegroGraph Java APIs: a number of interfaces offering convenient and efficient access to an AllegroGraph server from a Java-based application. Website: https://github.com/franzinc/agraph-java-client
- Apache Jena 2.6.2: a Java framework for building Semantic Web applications. We use it for reading, processing and writing triples. Website: http://jena.apache.org/.

- Sesame 2.6.2: an open-source framework which is connected to AllegroGraph Java APIs to process RDF data. It is also used for sending SPARQL queries and processing query results. Website: http://www.openrdf.org/index.jsp.

- Solrj 3.50: a Java interface to add, update, and query the index of SIREn. It is applied for synchronization of the data stored in AllegroGraph 4.11 and the data in the index of SIREn. Website: http://wiki.apache.org/solr/Solrj.

![The ontology repository](image)

Figure 4.3: Framework of the ontology repository

The framework of the ontology repository is demonstrated in Figure 4.3. As we can see from the framework, communication between the user and the ontology repository is purely through the HTTP protocol which is easier to be understood compared to the SPARQL protocol\(^\text{14}\) provided by some other repositories.

\(^{14}\)SPARQL protocol: http://www.w3.org/TR/sparql11-protocol/
The ontology repository has been deployed on Apache Tomcat\(^{15}\) 7.0.26 running on a Virtual Private Server located in Milan, Italy, with shared hardware resources (a 6-core CPU, 4 GB RAM, 40 GB hard disk, Ubuntu server 12.04) and shared Gigabit Internet connection. Users could access the ontology repository via the following URIs:

- **Linked Data Platform**: http://repo.chencai.info/ns/
- **SIREn**: http://repo.chencai.info/siren/
- **Ontology Browser**: http://repo.chencai.info/browser/
- **Ecco diff**: http://repo.chencai.info/diff/

### 4.8 Summary

In this chapter, we demonstrated in detail the design and implementation of our ontology repository. Firstly, we discussed the selection of triplestore that is used for storage of ontologies. We described the utilization of security features of the selected triplestore. We also presented the designed URIs of the ontology repository. Secondly, we demonstrated the design of HTTP APIs that conforms to the specification of Linked Data Platform. Thirdly, we illustrated the integration of OntologyBrowser and SIREn which provide features for accessing ontologies. Fourthly, we described the definition of ontology versioning and our solution of supporting ontology versioning. Lastly, we discussed the implementation of the ontology repository.

We could conclude that all the functional requirements (except UCR36 which is an optional requirement) listed in Chapter 3 have been fulfilled by our design and implementation.

---

\(^{15}\)Apache Tomcat: http://tomcat.apache.org/
Chapter 5

Performance testing

According to the software requirements UCR32 and UCR35, the ontology repository shall respond to HTTP requests of publishing an ontology or retrieving an ontology within 10 second in the specified situation. We have conducted a series of tests to verify if the implementation can meet these requirements. The tests were carried out in Eindhoven, the Netherlands, using a dedicated Internet connection with 25 MB/s download speed and 2.5 MB/s upload speed. We wrote a Java program that can repeatedly send HTTP requests and record the corresponding HTTP response time.

Software testing on other aspects, e.g., scalability, security, compatibility, have also been considered. However, due to time constraints, we conducted only tests on the aspects explicitly specified in the software requirements.

5.1 Performance test on publishing ontologies

Publishing an ontology to the ontology repository requires two steps: (1) reserving a URI by sending required metadata; (2) appending content of the ontology by sending another HTTP POST request. Thus we test the response time of the two steps separately.

Firstly, we repeated 50 times of sending a HTTP POST request contains required metadata (see Listing 4.2 for example) to reserve a URI from the ontology repository. The response time ranges from 2.316 seconds to 8.105 seconds with an average of 4.472 seconds, as it is demonstrated in Figure 5.1. The reason why the response time fluctuated might be that reserving URIs requires hard disk I/O operations to create indexes in AllegroGraph 4.11 on the server, but the server is using shared hardware resources whose I/O speed could not be guaranteed.

Secondly, we generated 50 different ontologies in Turtle format using the
well-known LUBM [12] benchmark that has been frequently applied on testing performance of triplestores. Each of the generated ontologies contained 1000 triples. For each generated ontology, we sent a HTTP POST request that included its triples to a freshly reserved URI. The reserved URI identified an ontology that contained only required metadata (see Listing 4.2 for example). The response time is shown in Figure 5.2, which varies from 0.804 seconds to 2.831 seconds with an average of 1.133 seconds.

Based on the test results shown above, on average, the ontology repository is able to publish an ontology with 1000 triples in around 6 seconds, which can meet the software requirement UCR32.

5.2 Performance test on retrieving ontologies

To test the performance of retrieving ontologies, we published the 50 generated ontologies used in Section 5.1 to the ontology repository. Each of these ontologies was identified by a distinct URI provided by the ontology repository. We sent HTTP GET requests to the URIs to retrieve triples in Turtle format and recorded the response time. The test results (shown in Figure 5.3) are very stable. All responses were returned in less than 1 second, which could meet the requirement UCR35.
Figure 5.2: Response time of appending 1000 triples

Figure 5.3: Response time of retrieve 1000 triples
5.3 Summary

In this chapter, we presented the results of performance testing on two aspects specified in the software requirements. We could conclude that the design and implementation of our ontology repository fulfilled the non-functional software requirements UCR32 and UCR35.
Chapter 6

Logic-based ontology comparison

We have discussed in Section 4.6.2 the detection of structural difference between two ontologies. Though very helpful, the notion of structural difference still has the deficiencies of having no unambiguous semantic foundation and being syntax dependent [19]. Therefore, in this chapter, we introduce another notion of ontology difference, logical difference, together with other background knowledge related to this notion, namely, the OWL 2 QL profile and the description logic DL-Lite$_R$. In addition, we illustrate the work conducted by Konev et al. [18] that provides an algorithm to check existence of logical difference between two ontologies expressed in DL-Lite$_R$.

To be noted that the materials shown in this chapter are not new. They are summarized research background materials directly from Konev et al. [18] necessary for Chapter 7. To our knowledge, the approach presented by Konev et al. [18] is the state of the art of detection of logical difference between two ontologies expressed in DL-Lite$_R$.

6.1 OWL 2 QL and DL-Lite$_R$

6.1.1 OWL 2 QL

The OWL 2 QL profile is designed with the aim of supporting ontology-based data access, which admits sound and complete reasoning in LOGSPACE with respect to the size of instance data stored in a data repository. The full profile specification of OWL 2 QL can be found in [26].

Moreover, OWL 2 QL includes many of the main features of conceptual models such as UML class diagrams and ER diagrams. For example, the UML class diagram shown in Figure 6.1 can be expressed in OWL 2 QL (see Listing 6.1).
Figure 6.1: A UML class diagram describes class Employee, class Manager, class Project, and the relationships between them.

Listing 6.1: OWL 2 QL expression of the UML class diagram depicted in Figure 6.1

6.1.2 DL-Lite\textsubscript{R}

DL-Lite\textsubscript{R} [4] is the description logic underlying OWL 2 QL. The alphabet of DL-Lite\textsubscript{R} contains sets of individual names \( a_i \), concept names \( A_i \), and role names \( P_i \). Concepts \( B \) and roles \( R \) are defined inductively as follows:

\[
R ::= P_i \mid P_i^- \quad B ::= \bot \mid \top \mid A_i \mid \exists R.
\]

where \( P_i^- \) is the reversed role of \( P_i \), \( \bot \) indicates the bottom concept (i.e., nothing), \( \top \) refers to the top concept (i.e., everything). Note that \( \exists R \) is the standard DL construct of unqualified existential quantification on basic roles, which can be seen as the abbreviation of \( \exists R.\top \).
A DL-Lite\(\text{R}\) knowledge base (KB) \(K = (\mathcal{T}, \mathcal{A})\) consists of two parts a TBox \(\mathcal{T}\) and an ABox \(\mathcal{A}\). A DL-Lite\(\text{R}\) TBox \(\mathcal{T}\) is a finite set of inclusions

\[
B_1 \sqsubseteq B_2, \quad R_1 \sqsubseteq R_2, \quad B_1 \sqcap B_2 \sqsubseteq \bot, \quad R_1 \sqcap R_2 \sqsubseteq \bot
\]

where \(B_1, B_2\) are concepts and \(R_1, R_2\) roles. A DL-Lite\(\text{R}\) ABox \(\mathcal{A}\) is formed by a set of assertions

\[
B(a_i), \quad R(a_i, a_j)
\]

where \(a_i \neq a_j\), \(a_i\) and \(a_j\) are individual names, \(B\) a concept and \(R\) a role.

**Example 1.** Considering the ontology shown in Figure 6.1 and Listing 6.1, we could express it in DL-Lite\(\text{R}\) as follows:

\[
\text{Manager} \sqsubseteq \text{Employee} \\
\exists \text{worksFor} \sqsubseteq \text{Employee} \\
\exists \text{worksFor}^{-} \sqsubseteq \text{Project}
\]

In this thesis, we consider the TBox \(\mathcal{T}\) as the ontology itself, the ABox \(\mathcal{A}\) as the associated instance data. For example, consider two KBs \(K_1 = (\mathcal{T}_1, \mathcal{A})\) and \(K_2 = (\mathcal{T}_2, \mathcal{A})\), \(\mathcal{T}_1\) and \(\mathcal{T}_2\) can be seen as different versions of an ontology, \(\mathcal{A}\) is the instance data associated with them. Additionally, we denote the set of individual names occurring in \(\mathcal{A}\) by \(\text{Ind}(\mathcal{A})\).

The semantic of DL-Lite\(\text{R}\) is defined in terms of interpretations, where an interpretation \(\mathcal{I} = (\Delta^\mathcal{I}, \mathcal{I})\) is formed by a non-empty interpretation domain \(\Delta^\mathcal{I}\) and an interpretation function \(\mathcal{I}\) that assigns a subset \(B^\mathcal{I}\) of \(\Delta^\mathcal{I}\) to concept \(B\), and a binary relation \(R^\mathcal{I}\) over \(\Delta^\mathcal{I}\) to each role \(R\).

For DL-Lite\(\text{R}\), we have

\[
B^\mathcal{I} \subseteq \Delta^\mathcal{I} \\
P^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I} \\
(P^{-})^\mathcal{I} = \{(o_2, o_1) \mid (o_1, o_2) \in P^\mathcal{I}\} \\
(\exists R)^\mathcal{I} = \{o \mid \exists o'.(o.o') \in R^\mathcal{I}\}
\]

An interpretation \(\mathcal{I}\) is a model of a concept inclusion \(B_1 \sqsubseteq B_2\) if \(B_1^\mathcal{I} \subseteq B_2^\mathcal{I}\). It is a model of a role inclusion \(R_1 \sqsubseteq R_2\) if \(R_1^\mathcal{I} \subseteq R_2^\mathcal{I}\). An interpretation \(\mathcal{I}\) is a model of an inclusion or assertion \(\alpha\) is denoted by \(\mathcal{I} \models \alpha\). A model of a KB \(\mathcal{K} = (\mathcal{T}, \mathcal{A})\) is an interpretation \(\mathcal{I}\) if \(\mathcal{I} \models \alpha\) for all \(\alpha \in \mathcal{T} \cup \mathcal{A}\). A KB is consistent if it has at least one model. A concept \(B\) is \(\mathcal{T}\)-consistent if \((\mathcal{T}, \{B(a)\})\) has a model. We write \(\mathcal{K} \models \alpha\) if \(\mathcal{I} \models \alpha\) for all models \(\mathcal{I}\) of \(\mathcal{K}\).
6.2 Logical difference between two ontologies

The notion of logical difference has been recently introduced [19, 20, 21] and completely abstracts from the representation of the ontology. From the view of logical difference, ontology is regard as a set of axioms expressed in a description logic with a formal and unambiguous semantics. In such case, ontologies give answers to queries of interest. Typical queries include subsumption queries between concepts or roles, instance assertion queries and conjunctive queries.

For a large-scale ontology, it is possible that users are interested in only small subset of the vocabulary of the ontology. Therefore the notion of signature Σ is introduced, which is a finite set of concept and role names selected by users. Given a concept, role, concept inclusion, TBox, ABox, or query E, sig(E) denotes the signature of E, that is, the set of concept and role names occurring in it. A concept (role, concept inclusion, TBox, ABox, etc.) E is called a Σ-concept (role, concept inclusion, TBox, ABox, etc., respectively) if sig(E) ⊆ Σ.

**Definition 1 ([18]).** The Σ-concept difference between ontologies T₁ and T₂ formulated in a DL L is the set cDiff_L^[Σ](T₁, T₂) of all Σ-concept inclusions B₁ ⊆ B₂ in L such that T₁ ⊄ B₁ ⊆ B₂ and T₂ ⊃ B₁ ⊆ B₂. T₁ Σ-concept entails T₂ if cDiff_L^[Σ](T₁, T₂) = 0.

For OWL 2 QL ontologies expressed in DL-LiteR, answers to queries over ABoxes are of greater importance than concept inclusion, since the OWL 2 QL profile is designed for ontology-based data access. In this thesis, we focus on a specific type of query: conjunctive query, which is defined as follows.

**Definition 2 ([4]).** The conjunctive query is a first order formula

\[ q(x_1, \ldots, x_n) = \exists y_1 \ldots \exists y_m \varphi(x_1, \ldots, x_n, y_1, \ldots, y_m) \]

where \( \varphi \) is constructed, using only \( \wedge \), from atoms of the form \( B(t) \) or \( R(t_1, t_2) \), with \( B \) being concept name, \( R \) a role and \( t_i \) being a variable from the list \( x_1 \ldots x_n, y_1 \ldots y_m \).

The variables in \( \bar{x} = x_1, \ldots, x_n \) are named answer variables of \( q \). We say some tuple \( \bar{a} \) of object names from an ABox \( A \) is an answer to \( q \) in an interpretation \( I \) if \( I \models q(\bar{a}) \). For a KB \( K = (T, A) \), we write \( K \models q(\bar{a}) \) if \( I \models q(\bar{a}) \) for all models \( I \) of \( K \). Then we formally define the notion of Σ-query difference.

**Definition 3 ([18]).** The Σ-query difference between ontologies T₁ and T₂ formulated in a DL L is the set qDiff_L^[Σ](T₁, T₂) of pairs of the form (\( \mathcal{A}, q(\bar{x}) \)), where \( \mathcal{A} \) is a Σ-ABox and \( q(\bar{x}) \) a Σ-query, such that (\( T_1, \mathcal{A} \) \( \not\models q(\bar{a}) \)) and (\( T_2, \mathcal{A} \) \( \models q(\bar{a}) \)), for some tuple \( \bar{a} \) of object names from \( \mathcal{A} \). T₁ Σ-query entails T₂ if qDiff_L^[Σ](T₁, T₂) = 0.

58
To be noted that in the definition of \( \Sigma \)-query difference, any arbitrary \( \Sigma \)-ABoxes rather than a fix one are considered, since the ABox is typically unknown at the ontology design phase.

6.3 \( \Sigma \)-query entailment for DL-Lite\(_R\) ontologies

6.3.1 Canonical models and \( \Sigma \)-Homomorphisms

We start by characterising \( \Sigma \)-query entailment between DL-Lite\(_R\) ontologies in terms of \( \Sigma \)-homomorphisms between certain canonical models.

The canonical model \( \mathcal{M}_K \) of a consistent KB \( K = (\mathcal{T}, \mathcal{A}) \) gives correct answers to all conjunctive queries. Although \( \mathcal{M}_K \) is generally infinite, it can be folded up into a small and finite generating model \( \mathcal{G}_K = (\mathcal{I}_K, \leadsto_K) \), where \( \mathcal{I}_K \) is a finite interpretation and \( \leadsto_K \) a generation relation defines the unfolding. We denote the set \( \{S \mid R \sqsubseteq_T S \text{ and } S \sqsubseteq_T R\} \) by \([R]\), where \( \mathcal{T} \) is a DL-Lite\(_R\) TBox, \( R \) and \( S \) are roles, \( \sqsubseteq_T \) is the reflexive and transitive closure of the role inclusion relation given by \( \mathcal{T} \). For example, given \( \mathcal{T} = \{R \sqsubseteq S, S \sqsubseteq T, T \sqsubseteq R\} \), where \( R, S, T \) are roles, we can get \([R] = [S] = [T] = \{R, S, T\}\). Let \( \leq_T \) be a partial order on the set \( \{R \mid R \text{ a role in } \mathcal{T}\} \), such that \([R] \leq_T [S] \) if \( R \sqsubseteq S \).

For each \([R]\), we introduce a witness \( w_{[R]} \), and define a generating relation \( \leadsto_K \) as follows:

\[ a \leadsto_K w_{[R]} \text{ if } a \in \text{Ind}(\mathcal{A}) \text{ and } [R] \text{ is } \leq_T\text{-minimal such that } K \models \exists R(a) \text{ and } K \not\models \exists R(a) \text{ for all } b \in \text{Ind}(\mathcal{A}); \]

\[ w_{[S]} \leadsto_K w_{[R]} \text{ if } [R] \text{ is } \leq_T\text{-minimal, } [S^-] \neq [R] \text{ and } \mathcal{T} \models \exists S^- \sqsubseteq \exists R. \]

We say a role \( R \) is generating in \( K \) if in the generating model \( \mathcal{G}_K \) there is a path \( a \leadsto_K w_{[S_1]} \leadsto_K \cdots \leadsto_K w_{[S_n]} \leadsto_K w_{[R]} \). Then we define the interpretation \( \mathcal{I}_K \) as follows:

\[
\begin{align*}
\Delta^\mathcal{I}_K &= \text{Ind}(\mathcal{A}) \cup \{w_{[R]} \mid R \text{ is generating in } K\}, \\
a^\mathcal{I}_K &= a, \text{ for all } a \in \text{Ind}(\mathcal{A}), \\
A^\mathcal{I}_K &= \{a \in \text{Ind}(\mathcal{A}) \mid K \models A(a)\} \cup \{w_{[R]} \mid \mathcal{T} \models \exists R^- \sqsubseteq A\}, \\
P^\mathcal{I}_K &= \{(a, b) \in \text{Ind}(\mathcal{A}) \times \text{Ind}(\mathcal{A}) \mid \text{there is } R(a, b) \in \mathcal{A} \text{ s.t. } [R] \leq_T [P] \}
\cup \{(x, w_{[R]}) \mid x \leadsto_K w_{[R]} \text{ and } [R] \leq_T [P]\}
\cup \{(w_{[R]}, x) \mid x \leadsto_K w_{[R]} \text{ and } [R] \leq_T [P^-]\}.
\end{align*}
\]

For an interpretation \( \mathcal{I} \) and a signature \( \Sigma \), the \( \Sigma \)-types \( t^\mathcal{I}_\Sigma(x) \) and \( r^\mathcal{I}_\Sigma(x, y) \),
for $x, y \in \Delta^I$, are given by:

$$t^I_\Sigma(x) = \{\Sigma\text{-concept } B \mid x \in B^I\},$$
$$r^I_\Sigma(x, y) = \{\Sigma\text{-role } R \mid (x, y) \in R^I\}.$$

By unfolding the generating model $G_K = (I_K, \leadsto_K)$ along $\leadsto_K$, we can construct the canonical model $M_K$.

**Example 2.** For DL-Lite $T$Box $T$ and KB $K = (T, \{\text{Student}(a)\})$ where

$$T = \{\text{Student} \sqsubseteq \exists \text{takesCourse}, \exists \text{takesCourse}^\bot \sqsubseteq \exists \text{taughtBy},$$
$$\exists \text{taughtBy}^\bot \sqsubseteq \exists \text{worksFor}, \exists \text{worksFor}^\bot \sqsubseteq \exists \text{worksFor}\},$$

the models $G_K$ and $M_K$ are shown as follows ($\leadsto_K$ is displayed as $\rightarrow$):

$G_K$:

\[
\begin{array}{cccc}
\text{Student} & \text{takesCourse} & \text{taughtBy} & \text{worksFor} \\
\rightarrow & a & \rightarrow & \rightarrow \\
\end{array}
\]

$M_K$:

\[
\begin{array}{cccc}
\text{takesCourse} & \text{taughtBy} & \text{worksFor} & \text{worksFor} \\
\rightarrow & a & \rightarrow & \rightarrow \\
\end{array}
\]

Since the canonical model gives correct answers to all conjunctive queries, in order to check $\Sigma$-query entailment between KBs $K_1$ and $K_2$, it is sufficient to check whether $M_{K_2} \models q(\bar{a})$ implies $M_{K_1} \models q(\bar{a})$ for all $\Sigma$-queries $q(\bar{x})$ and tuple $\bar{a}$. This implication can be characterised in terms of finite $\Sigma$-Homomorphisms that is formally defined as follows.

**Definition 4** ([18]). A $\Sigma$-homomorphism from an $I_1$ to $I_2$ is a function $h : \Delta^I_1 \rightarrow \Delta^I_2$ such that $h(a^{I_1}) = a^{I_2}$ for all individual names $a$ in $I_1$, $t^I_\Sigma(x) \subseteq t^I_\Sigma(h(x))$ and $t^I_\Sigma(x, y) \subseteq t^I_\Sigma(h(x), h(y))$, for all $x, y \in \Delta^I_1$, where $\Sigma$-types $t^I_\Sigma(x)$ and $t^I_\Sigma(x, y)$ are given by $t^I_\Sigma(x) = \{\Sigma\text{-concept } B \mid x \in B^I\}$ and $t^I_\Sigma(x, y) = \{\Sigma\text{-role } R \mid (x, y) \in R^I\}$.

It has been proved that answers to conjunctive $\Sigma$-queries are preserved under $\Sigma$-homomorphism [32]. Therefore, $K_1 \Sigma$-query entails $K_2$ if there is a $\Sigma$-homomorphism from $M_{K_2}$ to $M_{K_1}$. $I$ is finitely $\Sigma$-homomorphically embeddable.
into $\mathcal{T}'$ if there is a $\Sigma$-homomorphism from $\mathcal{I}_1$ to $\mathcal{T}'$ for every finite sub-interpretation $\mathcal{I}_1$ of $\mathcal{I}$. We also denote the models $G_{(T_i, \{B(a)\})}$ and $M_{(T_i, \{B(a)\})}$ by $G^B_i$ and $M^B_i$, respectively. Then we formally characterise $\Sigma$-query entailment between DL-Lite$_R$ ontologies as follows.

**Theorem 1** ([18]). Ontology $\mathcal{T}_1$ $\Sigma$-query entails ontology $\mathcal{T}_2$ iff

- (p) $\mathcal{T}_2 \models \alpha$ implies $\mathcal{T}_1 \models \alpha$ for all $\Sigma$-inclusion $\alpha$, and
- (h) $M^B_{\mathcal{T}_2}$ is finitely $\Sigma$-homomorphically embeddable into $M^B_{\mathcal{T}_1}$, for all $\mathcal{T}_1$-consistent $\Sigma$-concepts $B$.

However, based on Theorem 1, checking $\Sigma$-query entailment between DL-Lite$_R$ ontologies is fairly complex. We know from Konev et al [18] that it is PSPACE-hard and in EXPTIME.

### 6.3.2 Incomplete algorithm for $\Sigma$-query entailment between DL-Lite$_R$ ontologies

The high complexity of checking $\Sigma$-query entailment between DL-Lite$_R$ ontologies is mainly caused by role inclusions. However, in “real-world” ontologies, the number of inclusions is very small. Thus Konev et al. [18] implemented a polynomial-time incomplete algorithm for $\Sigma$-query entailment between DL-Lite$_R$ ontologies, which is based on testing simulations between generating models.

**Definition 5** ([18]). Given two DL-Lite$_R$ ontologies $\mathcal{T}_1$ and $\mathcal{T}_2$, a signature $\Sigma$, a $\Sigma$-concept $B$, KBs $K_i = (T_i, \{B(a)\})$ and interpretations $\mathcal{I}_i$, where $i = 1, 2$ and $\mathcal{I}_i = I_{K_i}$, a relation $\rho \subseteq \Delta_{\mathcal{I}_2} \times \Delta_{\mathcal{I}_1}$ is called a $\Sigma$-simulation of $G_{K_2}$ in $G_{K_1}$ if the following conditions hold:

- (s1) the domain of $\rho$ is $\Delta_{\mathcal{I}_2}$ and $(a^{\mathcal{I}_2}, a^{\mathcal{I}_1}) \in \rho$;
- (s2) $r_{\Sigma}^{\mathcal{I}_2}(x) \subseteq r_{\Sigma}^{\mathcal{I}_1}(x')$, for all $(x, x') \in \rho$;
- (s3) if $x \sim_{K_2} w_{[R]}$ and $(x, x') \in \rho$, then there is $y' \in \Delta_{\mathcal{I}_1}$ such that $(w_{[R]}, y') \in \rho$ and $S \in r_{\Sigma}^{\mathcal{I}_2}(x', y')$ for every $\Sigma$-role $S$ with $[R] \leq_{\mathcal{T}_2} [S]$.

Moreover, $\rho$ is a forward $\Sigma$-simulation, if the condition (s1), (s2) and (s3') (strengthened (s3)) holds.

- (s3') if $x \sim_{K_2} w_{[R]}$ and $(x, x') \in \rho$, then there is a role $T$, with $x' \sim_{K_1} w_{[T]}$ and $[T] \leq_{\mathcal{T}_1} [S]$ for every $\Sigma$-role $S$ with $[R] \leq_{\mathcal{T}_2} [S]$. 

61
Two new conditions that can be checked in polynomial time are proposed in:

(y) condition (p) holds and there is a forward $\Sigma$-simulation of $G^B_{T_2}$ in $G^B_{T_1}$ for every $T_1$-consistent $\Sigma$-concept $B$;

(n) condition (p) does not hold or there is no $\Sigma$-simulation of $G^B_{T_2}$ in $G^B_{T_1}$ for any $T_1$-consistent $\Sigma$-concept $B$.

Theorem 2 ([18]). Given two DL-Lite$\mathcal{R}$ ontologies $T_1$ and $T_2$, a signature $\Sigma$, if (y) holds, then $T_1$ $\Sigma$-entails $T_2$; if (n) holds, then $T_1$ does not $\Sigma$-entail $T_2$.

Thus, an algorithm checking conditions (y) and (n) can be applied as an incomplete $\Sigma$-query entailment checker. This algorithm is incomplete because it cannot determine the situation that condition (p) holds but there is no $\Sigma$-simulation of $G^B_{T_2}$ in $G^B_{T_1}$ for some $T_1$-consistent $\Sigma$-concept $B$.

6.4 Summary

In this chapter, we introduced OWL 2 QL profile, description logic DL-Lite$\mathcal{R}$ and the notion of logical difference. We directly summarized the approach presented by Konev et al. [18] on checking existence of logical difference between two DL-Lite$\mathcal{R}$ ontologies. This approach introduced two important concepts, namely, generating model and forward $\Sigma$-simulation, which are utilized in our novel approach on visualization of logical difference described in Chapter 7.
Chapter 7

Visualization of Extended-Σ-query difference

Applying the algorithm presented in Section 6.3.2, one could verify if Σ-query entailment holds between two DL-LiteR ontologies. But simply telling users the result (yes/no) has limited value in real life cases, especially in ontology engineering, where ontology developers may would like to know supporting evidences of the result as well. If Σ-query entailment between two DL-LiteR ontologies does not hold, there must exist Σ-query difference that is a set of pairs of the form \((A, q(\bar{x}))\), which is a concrete evidence of the result. Therefore, in this chapter, we present a novel approach of visualization of a new type of Σ-query difference which we propose here, namely, Extended-Σ-query difference.

7.1 Extended-Σ-query difference

7.1.1 Definition of Extended-Σ-query difference

The purpose of applying the signature Σ is to allow users to choose concepts and roles of their interest. However, due to limited understanding on the semantics of the ontologies, the users may not be able to properly select the signature Σ. For example, given two university ontologies \(T_2 = \{PostDoc \sqsubseteq \exists hasDocDegreeFrom, Chair \sqsubseteq \exists headOf, \ldots\}\), \(T_1 = \{\ldots\}\) (omitted parts are the same), the user may select the signature \(\Sigma = \{Chair, PostDoc\}\), which excludes some necessary elements that should be included, namely, role \(hasDocDegreeFrom\) and \(headOf\). As the inclusion \(Chair \sqsubseteq \exists headOf\) states each named individual of the type Chair has to be at least head of something, concept Chair and role headOf become closely related in this logical context. It is indispensable to add \(headOf\) into \(\Sigma\) as well if the user has in-
cluded Chair in Σ. The same situation occurs for the inclusion PostDoc \subseteq \exists\text{hasDocDegreeFrom}.

We observe that the construction of the generating model \( G_K \) described in Section 6.3.1 could properly determine closely related roles for each concept. Thus we extend the original \( \Sigma \) selected by users based on \( G_K \) to cope with the situations described above. We formally define this extended \( \Sigma \) as follows:

\[
\text{Extended-}\Sigma = \\
\Sigma \cup \{ R | R \text{ is generating in } (T_2, B(a)), R \in \text{sig}(T_1), B \in \text{\Sigma-concept} \}
\]

Intuitively, for each \( \Sigma \)-concept \( B \), all roles that are closely related to \( B \) are included in the Extended-\( \Sigma \). Additionally, we define the Extended-\( \Sigma \)-query difference as follows:

**Definition 6.** The Extended-\( \Sigma \)-query difference between ontologies \( T_1 \) and \( T_2 \) formulated in DL-LiteR is the set \( \text{eqDiff}(T_1, T_2) \) of pairs of the form \((A, q(\bar{x}))\), where \( A \) is an Extended-\( \Sigma \)-ABox and \( q(\bar{x}) \) an Extended-\( \Sigma \)-query, such that \((T_1, A) \not\models q(\bar{a})\) and \((T_2, A) \models q(\bar{a})\), for some tuple \( \bar{a} \) of object names from \( A \).

### 7.1.2 Algorithm for detecting Extended-\( \Sigma \)-query difference

As the Extended-\( \Sigma \)-ABox \( A \) and the Extended-\( \Sigma \)-query \( q(\bar{x}) \) are arbitrary, \( \text{eqDiff}(T_1, T_2) \) can be either infinite or empty. Instead of attempting to compute the infinite set, we focus on detecting some representative pairs of the form \((A, q(\bar{x}))\) in \( \text{eqDiff}(T_1, T_2) \). For each representative pair, the following condition holds: (1) \( A \) should contain as few elements as possible; (2) \( q(\bar{x}) \) should contain as few conjunctions as possible. The purpose of providing representative pairs to users is to avoid confusing users with complex ABoxes and queries.

Moreover, we show that \( \text{eqDiff}(T_1, T_2) \neq \emptyset \) if one of the following conditions holds:

- (ma) \( T_2 \models \alpha \) does not imply \( T_1 \models \alpha \) for some Extended-\( \Sigma \)-inclusion \( \alpha \);
- (mb) there is no forward \( \Sigma \)-simulation of \( G_B^{T_2} \) in \( G_B^{T_1} \) for some \( T_1 \)-consistent \( \Sigma \)-concept \( B \).

For condition (ma), the following types of Extended-\( \Sigma \)-inclusions should be considered:

1. \( A_1 \subseteq A_2 \). We can construct the representative pair \((A, q(x))\) where Extended-\( \Sigma \)-query \( q(x) = A_2(x) \) and Extended-\( \Sigma \)-ABox \( A = \{ A_1(a_1) \} \);
2. \( \exists R \subseteq A \). We can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = A(x)\) and Extended-\(\Sigma\)-ABox \(A = \{R(a_1, a_2)\}\);

3. \( A \subseteq \exists R \). We can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = \exists y_1 R(x, y_1)\) and Extended-\(\Sigma\)-ABox \(A = \{A(a_1)\}\);

4. \( \exists R_1 \subseteq \exists R_2 \). We can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = \exists y_1 R_2(x, y_1)\) and Extended-\(\Sigma\)-ABox \(A = \{R_1(a_1, a_2)\}\);

5. \( R_1 \subseteq R_2 \). We can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x_1, x_2) = R_2(x_1, x_2)\) and Extended-\(\Sigma\)-ABox \(A = \{R_1(a_1, a_2)\}\);

As for condition \((mb)\), we consider the generating models as directed labelled graphs. For each \(T_i\)-consistent \(\Sigma\)-concept \(B\) and corresponding \(G^{B \upharpoonright T_2}_{T_2}\) and \(G^{B \upharpoonright T_1}_{T_1}\), we can apply the well-known breadth-first search strategy to find the nearest node where the conditions of forward \(\Sigma\)-simulation do not hold. There are following cases:

1. \( I^2_{T_2}(a) \not\subseteq I^2_{T_1}(a) \). Then there exists some \(\Sigma\)-concept \(A\), s.t. \((T_1, \{B(a)\}) \not\vDash A(a)\) and \((T_2, \{B(a)\}) \models A(a)\). We can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = B(x) \land A(x)\) and Extended-\(\Sigma\)-ABox \(A = \{B(a)\}\);

2. \( x \sim^{\Sigma K_2} w_{[R]} \) and \((x, x') \in \rho\), there does not exist a role \(T\) with \(x' \sim^{\Sigma K_1} w_{[T]}\). If \(x = a\), we can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = \exists y_1 B(x) \land R(x, y_1)\) and Extended-\(\Sigma\)-ABox \(A = \{B(a)\}\); If \(x = w[S]\), we find the shortest path from \(a\) to \(w[S]\) in \(G^{B \upharpoonright T_2}_{T_2}\), i.e., \(a \sim^{\Sigma K_2} w_{[U]} \sim^{\Sigma K_2} \cdots \sim^{\Sigma K_2} w_{[U]} \sim^{\Sigma K_2} w_{[S]}\). Based on this shortest path, we can construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = \exists y_1, \ldots, y_{i+1} B(x) \land U_1(x, y_1) \land \cdots \land U_i(y_{i-1}, y_i) \land S(y_i, y_{i+1}) \land R(y_{i+1}, y_{i+2})\) and Extended-\(\Sigma\)-ABox \(A = \{B(a)\}\);

3. \( x \sim^{\Sigma K_2} w_{[R]} \) and \((x, x') \in \rho\), for every role \(T\) with \(x' \sim^{\Sigma K_1} w_{[T]}\), \(I^2_{T_2}(w_{[R]}) \not\subseteq I^2_{T_1}(w_{[T]})\). In this case, there exists some \(\Sigma\)-concept \(A\), s.t. \((T_2, \{B(a)\}) \models \exists R^\uparrow \subseteq A\) and \((T_1, \{B(a)\}) \not\models \exists T^\uparrow \subseteq A\). If \(x = a\), we construct the representative pair \((A, q(x))\) where Extended-\(\Sigma\)-query \(q(x) = \exists y_1 B(x) \land R(x, y_1) \land A(y_1)\) and Extended-\(\Sigma\)-ABox \(A = \{B(a)\}\); If \(x = w[S]\), we find a shortest path from \(a\) to \(w[S]\) in \(G^{B \upharpoonright T_2}_{T_2}\), i.e., \(a \sim^{\Sigma K_2} w_{[U]} \sim^{\Sigma K_2} \cdots \sim^{\Sigma K_2} w_{[U]} \sim^{\Sigma K_2} w_{[S]}\). Based on this shortest path, we can construct the representative pair \((A, q(x))\) where \(q(x) = \exists y_1, \ldots, y_{i+1} B(x) \land U_1(x, y_1) \land \cdots \land U_i(y_{i-1}, y_i) \land S(y_i, y_{i+1}) \land R(y_{i+1}, y_{i+2}) \land A(y_{i+2})\) and Extended-\(\Sigma\)-ABox \(A = \{B(a)\}\).

65
4. \( x \sim_{K_2} w_{\|R} \) and \((x, x') \in \rho\), for every role \( T \) with \( x' \sim_{K_1} w_{\|T} \) and \([T] \leq_{\mathcal{T}_1} [S]\), there exists some Extended-\( \Sigma \)-role \( S \) with \([R] \not\leq_{\mathcal{T}_2} [S]\). This case would never occur, since \( w_{\|R} \) must be \( \leq_{\mathcal{T}_2} \)-minimal according to the definition of generating model.

Therefore, an algorithm checking conditions \((ma)\) and \((mb)\) according to the cases listed above can be used to detect the representative pairs of Extended-\( \Sigma \)-query difference between two DL-Lite\(_R\) ontologies.

### 7.2 Implementation

We have implemented the algorithm presented in Section 7.1.2. This implementation demonstrates our novel approach of visualization of Extended-\( \Sigma \)-query difference, which is written in Java programming language (Standard Edition 7) and based on OWLAPI 3.43 [15]. To be noted that this implementation has not been interoperated with the ontology repository, because detecting Extended-\( \Sigma \)-query difference is very resource consuming and time consuming.

First of all, the user is requested to provide two DL-Lite\(_R\) ontologies \( \mathcal{T}_1 \) and \( \mathcal{T}_2 \) and to select a set of concepts and roles as the signature \( \Sigma \), which is shown in Figure 7.1.

Secondly, after the user clicks the Run Diff button, the program starts to execute the algorithm for detecting Extended-\( \Sigma \)-query difference. If a representative pair \((A, q(\bar{x}))\) could be detected, display \(q(\bar{x})\) as a SPARQL query and visualize it as a labelled graph. In additional, to help users understand the difference, we extract two modules \(Mod_1\) and \(Mod_2\) based on the concepts and roles appeared in \(q(\bar{x})\) from \(\mathcal{T}_1\) and \(\mathcal{T}_2\), respectively. By module, we understand a subset of the ontology that contains concepts and roles of interest. In principle, \(Mod_1\) and \(Mod_2\) would contain a limited number of axioms that trigger the Extended-\( \Sigma \)-query difference. The module extraction method is provided by OWLAPI 3.43. Additionally, corresponding ABox \(A\) would be included in the modules as well. We display \(Mod_1\) and \(Mod_2\) in Turtle format in a twin-view in the GUI, as it is shown in Figure 7.2.
Figure 7.1: GUI for importing ontologies and selecting signature.

Figure 7.2: Visualization of extracted modules and the generated query.
7.3 Summary

In this chapter, we demonstrated our novel approach for visualization of a new type of \( \Sigma \)-query difference, Extended-\( \Sigma \)-query difference. We first formally defined Extended-\( \Sigma \)-query difference. To avoid confusing users with complex queries and ABoxes, we proposed the notion of representative pair. We presented an algorithm for detecting the representative pairs of Extended-\( \Sigma \)-query difference. Also, we demonstrated our implementation for visualization of Extended-\( \Sigma \)-query difference, which applied the presented algorithm.
Chapter 8

Conclusion

In this chapter, we summarize the contributions of this graduation project. We describe the limitations of our ontology repository and work that could be conducted in the future.

8.1 Contributions

In this thesis, we have presented two novel things. Firstly, we demonstrated our work on the design and implementation, to our knowledge, of the first 5-star compliant Liked Open Data ontology repository. The APIs provided by this ontology repository are based on the HTTP protocol, thus it is convenient for other applications to utilize the ontology repository. The ontology modelling tool developed by the company Semmtech BV would be the first application to be associated with our ontology repository. Secondly, we presented our academic research on logic-based ontology comparison. We summarized previous work in this field and discussed their limitations. Based on the previous work, we proposed two new notions for logic-based ontology comparison: Extended-$\Sigma$-query difference and representative pairs. We presented an algorithm for detection of representative pairs of Extended-$\Sigma$-query difference between two DL-Lite$^R$ ontologies. To further demonstrate this algorithm, we also implemented a Java application based on this algorithm, which supports visualization of Extended-$\Sigma$-query difference.

8.2 Limitations

There are three major limitations of our ontology repository. First of all, the scalability of the ontology repository is unknown. Due to time constraints, we conducted software testing on only those aspects explicitly specified in the
software requirements. Thus the scalability of the ontology repository has not been tested. Second, the ontology repository is not able to properly display blank nodes nested in resources of ontologies. As blank nodes do not have URI reference, it is very difficult for the ontology repository to retrieve the triples of blank nodes using SPARQL queries. Third, security is a weakness of our ontology repository. We implemented only HTTP basic authentication, which is not considered to be a secure method of user authentication, as the username and password are passed as clear-text.

8.3 Future work

Conforming to newer versions of the Linked Data Platform specification

A newer version of the Linked Data Platform specification has been published on July 30, 2013, which is a last-call working draft and is intended to become a W3C recommendation. Compared to the version that has been applied in this project, a few changes have been made in the newer version. Future work is required to conform the ontology repository to newer versions of the Linked Data Platform specification.

Improving performance of the ontology repository

Since the design and implementation of the ontology repository presented in this thesis mainly focus on feature development, some performance related aspects have not been tested. For instance, the stability of this system is still unknown if a large group of users are using it simultaneously. It is possible the current implementation may not perform as well as expected in some situations. Thus it requires further tests and improvements.

Advanced security support for the ontology repository

The design and implementation of the ontology repository demonstrated in this thesis support only HTTP basic authentication which provides no confidentiality protection for the transmitted credentials. More advanced security features, e.g., support for HTTPS protocol and Digest access authentication, could be implemented.
Appendix A

Figure 8.1: Class diagram of the major components of the ontology repository
Bibliography


