Master Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

CheckIt: publication tool for gestors of Semantic vocabulary of terms - backend

Bc. Michal Švagr

Supervisor: Ing. Michal Med, Ph.D.
Supervisor–specialist: Ing. Martin Ledvinka, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2023

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Švagr** Jméno: **Michal** Osobní číslo: **483856**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

CheckIt: publikační nástroj pro správce Sémantického slovníku pojmů - backend

Název diplomové práce anglicky:

CheckIt: publication tool for gestors of Semantic vocabulary of terms - backend

Pokyny pro vypracování:

1) Become familiar with the tools TermIt, Ontographer, Mission Control and SGov Server for the creation, editing and publication of Semantic vocabulary of Terms (SSP). Tools are parts of the Assembly line designed and created in the project Quality Open Data and Infrastructure (KODI) on the Ministry of Interior of the Czech Republic.
2) Analyze processes related to the creation, editing and publication of semantic vocabularies within the Assembly line.
3) Design a tool for publication of the new versions of the semantic vocabularies created or edited by the tools of Assembly line. Focus on the backend part of the tool.
4) Implement design created in the previous point. Take into account involvement of the tool into the Assembly line environment and usage of GitHub as a storage of published vocabularies.
5) Evaluate functionality of a tool by the automated tests for the given scenarios - conflict resolution, publication, concurrent requirements etc.

Seznam doporučené literatury:

Křemen, P.; Nečaský, M., Improving discoverability of Open Government Data with rich metadata descriptions using Semantic Government Vocabulary, Journal of Web Semantics. 2019, 55 1-20. ISSN 1570-8268.
Křemen P., Pojmové znalostní grafy ve veřejné správě, available from:
https://data.gov.cz/%C4%8Dl%C3%A1nky/pojmov%C3%A9-znalostn%C3%AD-grafy-ve-ve%C5%99ejn%C3%A9-spr%C3%A1v%C4%9B
Křemen P.; Med M.;Nečaský M.;Domanská R., Metodika tvorby a údržby sémantického slovníku pojmů veřejné správy, 2022
Křemen P.; Med M.;Nečaský M.;Domanská R., Koncepce sémantického slovníku pojmů pro potřeby konceptuálního datového modelování agend, 2022
Fowler, M: Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Michal Med, Ph.D. katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.01.2023** Termín odevzdání diplomové práce: **26.05.2023**

Platnost zadání diplomové práce: **22.09.2024**

Ing. Michal Med, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

© VUT v Praze, Design: VUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank Ing. Michal Med, Ph.D. and Ing. Martin Ledvinka, Ph.D. for guidance, professional help, factual comments, and for providing the opportunity to work on this topic. I would also like to thank Bc. Filip Kopecký for the great cooperation and swift communication throughout the whole development. Finally, I would like to express my gratitude to my family for their support and encouragement.

Declaration

I declare that I prepared the submitted work independently and that I have listed all the literature used.

In prague, 25. May 2023

Abstract

This master thesis deals with the issue of revising created or modified vocabularies and their models in the Assembly line toolset. The work contains an introduction to RDF, analyses of processes of the Assembly line, designing of new processes, research on existing tools for comparing RDF datasets and designing and implementing of a solution in the form of an application. The result of the thesis is a back-end application implementing the new processes and a proposal of steps for future work.

Keywords: RDF, RDF Schema, RDF diff, SPARQL, SSP, Assembly line

Supervisor: Ing. Michal Med, Ph.D.

Abstrakt

Tato diplomová práce se zabývá problematikou revidování vytvořených nebo upravených slovníků a jejich modelů v sadě nástrojů zvané Výrobní linka. Práce obsahuje úvod do RDF, analýzu procesů Výrobní linky, návrh nových procesů, rešerši existujících nástrojů pro porovnávání RDF datových sad a návrh a implementaci řešení ve formě aplikace. Výsledkem práce je backendová aplikace realizující vzniklé procesy a návrh kroků pro budoucí práci.

Klíčová slova: RDF, RDF Schema, RDF diff, SPARQL, SSP, Výrobní linka

Překlad názvu: CheckIt: publikační nástroj pro správce Sémantického slovníku pojmů - backend

Contents

1 Introduction **1**

2 Introduction to RDF(S) **3**

2.1 RDF(S) 3

2.2 Triple 4

2.2.1 Literal 5

2.2.2 Blank node 6

2.2.3 Named graph 6

2.3 Concrete syntaxes for RDF 7

2.4 Example of RDF dataset 8

3 Assembly line processes **11**

3.1 Vocabulary modelling process . . 11

3.2 Current Publication process 13

3.3 New Publication process 15

3.4 Requesting to Gestor a vocabulary 17

4 Existing tools research **19**

4.1 Main requirements for the tool . 19

4.2 Researched tools 20

4.2.1 Python RDFLib 20

4.2.2 Protégé 21

4.2.3 OWLDiff 22

4.2.4 RDF Graph Normalization . . 23

4.2.5 Apache Jena 23

4.3 Conclusion 24

5 Ontology **27**

5.1 Main entities 27

5.2 Relationship diagram 29

5.2.1 Change entity 29

5.2.2 Publication context entity . . . 30

5.2.3 Comment entity 31

5.2.4 Notification entity 32

5.2.5 Gestoring request entity 32

6 Implementation **35**

6.1 Used technologies 35

6.2 Change resolving algorithm 36

6.2.1 Change resolving without blank nodes 37

6.2.2 Change resolving in blank nodes 38

6.3 Composing relationships 42

6.4 Updating Publication context . . 43

6.5 Maintenance improvements 43

7 Deployment 45

7.1 Existing deployment 45

7.2 New deployment 47

7.3 Deployment for development . . . 48

7.4 Deployment for user testing 49

8 Impediments and future work 51

8.1 Impediments along the way 51

8.2 Future work on CheckIt 52

8.2.1 Further integration with the Assembly line 52

8.2.2 Correcting typos while reviewing 53

8.2.3 Improving notifications 53

8.2.4 Gestoring requests 54

8.2.5 Testing 54

9 Conclusion 55

Bibliography 57

A Content of the electronic attachment 61

Figures

2.1 RDF triple example diagram 5

2.2 RDF literal example diagram 5

2.3 RDF blank node example diagram 6

2.4 RDF Named graph example diagram 7

2.5 RDF dataset example diagram . 10

3.1 Publication process (AS IS) 14

3.2 Publication process (TO BE) . . . 16

3.3 Gestoring request diagram 17

4.1 Python RDFLib output 20

4.2 Error classes in Protégé 21

4.3 Error entities in OWLDiff 22

4.4 Jena RDFDiff output 23

5.1 Ontology relationship diagram . . 28

5.2 Ontology relationship diagram - Publication context closeup 31

5.3 Ontology relationship diagram - Comment closeup 31

5.4 Ontology relationship diagram - Notification closeup 32

5.5 Ontology relationship diagram - Gestoring request closeup 33

6.1 Relationship `has manufacturer` (simple view) 38

6.2 Relationship `has manufacturer` (full view) 39

7.1 Deployment diagram (current) . . 46

7.2 Deployment diagram (new) 48

Tables

2.1 Building stones of RDF(S) 4

4.1 Tools capabilites 24

6.1 Used technologies 36

Chapter 1

Introduction

In the current day and age, we live in a world surrounded by data. The volume of data has grown exponentially since the second half of the last decade. In 2017, it was estimated that 2.5 exabytes of data are generated each day [1]. Brought to perspective, that is the capacity of 2.5 million one-terabyte disks ordinarily found in modern notebooks. One of the new contributors to the growth is the digitalization trend, which was recently accelerated with the pandemic and associated remote work from home.

Among the main pushes to digitize government documents in the public sector is the Open Data directive of European Union[1], which aims to make as much non-personal information available for re-use as possible. This directive gave rise to the KODI project[2] at the Czech Republic's Ministry of the Interior, which follows on from the previous two projects Open Data [2] and Open Data II [3]. This project aspires to increase transparency and availability of government data for other government entities and public use [4]. To achieve one of the goals of the KODI project, a set of tools called the Assembly line is being created, enabling the digitization of agendas, laws and decrees by creating and maintaining conceptual models.

Modelling of these legal documents in the Assembly line is done in several steps: defining new or selecting any existing conceptual model in the Mission control tool, creating or modifying key terms of the document in the

[1] You can see the directive on `eur-lex.europa.eu`

[2] The full name of the project is "Rozvoj datových politik v oblasti zlepšování kvality a interoperability dat veřejné správy" or translated to English "Developing data policies to improve the quality and interoperability of public administration data".

TermIt tool, modelling relationships between terms in the OntoGrapher tool, requesting publication in the Mission control tool, and subsequent reviewing and publication. The last step is the only one without a specialized tool for conceptual models, reducing the uniformity of the user experience. Reviews are done by manually comparing text differences of RDF files in GitHub Pull Requests, which requires extensive knowledge of RDF(S) and data architecture of the Assembly line. Also, discussion about discrepancies in the models is realized using email communication, therefore requiring a lot of additional information describing the location of the discrepancies. All this dramatically degrades the usability of the Assembly line as a whole.

This master thesis aims to solve these shortcomings by designing a tool that would allow reviewing in a human-readable way with built-in communication. So not only the expert members of the KODI team but also individuals from government entities responsible for the correctness of these conceptual models can review them and use the Assembly line to its full potential.

Chapter 2

Introduction to RDF(S)

First of all, let us familiarise ourselves with the technologies used in the Assembly line [5]. The central database of the toolset is GraphDB, which is a semantic graph database storing RDF data [6]. RDF is not a very well-known data model, so let us get familiar with it, as it will be essential throughout this thesis.

2.1 RDF(S)

Resource Description Framework (RDF) [7], specified by the World Wide Web Consortium (W3C), is a general language and vocabulary for representing data in the form of statements creating directed edges of a graph. The resource document is described in a way that both humans and machines can understand. Thanks to the graph format of RDF data, it is easy to visualize complicated datasets in the form of diagrams for better understanding by humans. RDF can be used to describe any resource that can be identified by an International Resource Identifier (IRI) [8]. The most well-known sub-type of IRI is Uniform Resource Locator (URL) [9], which is used for addressing web pages. As we will see later, these web pages can contain information in RDF format describing the identifier.

RDF Schema [10] (RDFS or RDF-S) is a vocabulary for describing RDF. It provides a set of terms that can be used to describe the structure and meaning of RDF data. When we talk about RDF data, the extension RDF Schema is

usually included as well, which is why in some literature, the abbreviation RDF(S) is used as both RDF and RDF Schema. In the table below, we can see some of the terms used as building stones, defined in RDF(S)[1]:

Construct	Syntax form	Description
Class (class)	C rdf:type rdfs:Class	C (source) is a RDFS class
Property (class)	P rdf:type rdf:Property	P (source) is a RDF property
type (property)	I rdf:type C	I (source) is an instance of C (class)
subClassOf (property)	C1 rdfs:subClassOf C2	C1 (class) is a subclass of C2 (class)
subPropertyOf (property)	P1 rdfs:subPropertyOf P2	P1 (property) is a subproperty of P2 (property)
domain (property)	P rdfs:domain C	The domain of (property) P is (class) C
range (property)	P rdfs:range C	The range of (property) P is (class) C

Table 2.1: Building stones of RDF(S)

2.2 Triple

As mentioned, RDF data is described using statements in the form of triples subject-predicate-object. Subject and object are two resources most commonly identified by IRI. The predicate in the middle of the triplet describes the relationship between subject and object in the direction of subject → object using IRI. Here we can see an example of a triple stating that Adam knows Lucy or, more precisely, that entity <http://example.com/Adam> has a relation <http://example.com/knows> to entity <http://example.com/Lucy> as we don't know how are entities or the relation defined.

```
<http://example.com/Adam> <http://example.com/knows> <http://example.com/Lucy> .
```

[1]Used prefix `rdf:` is of IRI <http://www.w3.org/1999/02/22-rdf-syntax-ns#> and prefix `rdfs:` is of IRI <http://www.w3.org/2000/01/rdf-schema#>. These and more constructs can be found on `https://www.w3.org/TR/rdf-schema/`.

Figure 2.1: RDF triple example diagram

2.2.1 Literal

In addition to the use of IRI, literals can be used in the object position of the triple, representing a value. A literal is bound to a data type for proper interpretation, such as a string, a number, a decimal number, or a date. Literals with data type string can be enriched furthermore with a language tag (e.g. "Building"@en). The example below shows three triples stating that Lucy's first name is "Lucy" in English and "Lucka" in Czech. The last triple tells us Lucy's birth date.

```
<http://example.com/Lucy> <http://xmlns.com/foaf/0.1/firstName> "Lucy"@en .
<http://example.com/Lucy> <http://xmlns.com/foaf/0.1/firstName> "Lucka"@cs .
<http://example.com/Lucy> <http://schema.org/birthDate>
                "1991-09-04"^^<http://www.w3.org/2001/XMLSchema#date> .
```

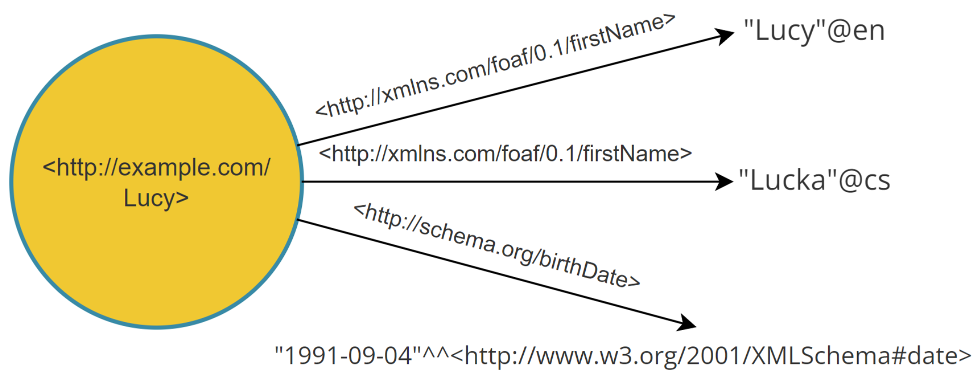

Figure 2.2: RDF literal example diagram

2.2.2 Blank node

There is one more defined concept, except IRI and Literal, in RDF called a blank node. It can only appear at the subject or object part of the triple. Blank nodes are placeholders for resources that do not have an IRI but still are part of the dataset. They represent intermediate nodes of a resource in a graph where the resource itself is not of direct interest. Blank nodes do not have identifiers, only temporary identifiers to show which blank node is which when printing out the data. In the example below, we can see statements telling us that Adam knows *someone* named Bob. The *someone* is represented by the blank node `_:bn1`.

```
<http://example.org/Adam> <http://xmlns.com/foaf/0.1/knows> _:bn1 .
_:bn1 <http://xmlns.com/foaf/0.1/firstName> "Bob" .
```

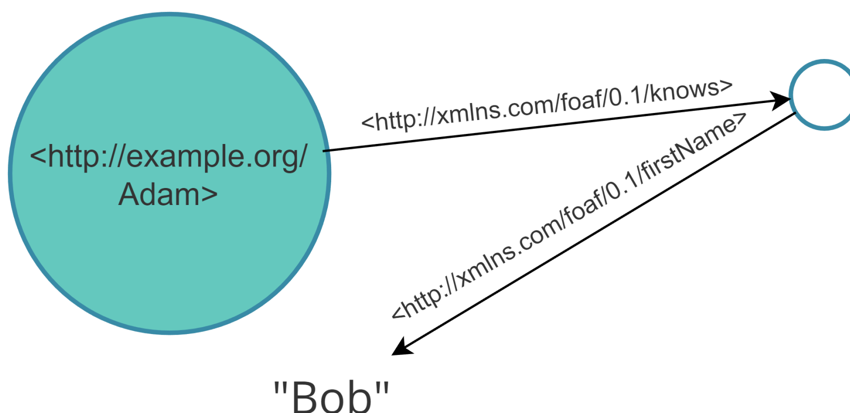

Figure 2.3: RDF blank node example diagram

2.2.3 Named graph

Every RDF triple of a dataset is located in the Default graph unless it is enclosed in a Named graph. Which serves as a mechanism for organizing and categorizing data in large RDF datasets by grouping statements of common context into a structure. Named graphs are identified (named) with an IRI, which must be unique inside a dataset, but other datasets can expand this graph by specifying triples in equally identified graphs. An example can be a dataset describing a library with members and books. All information about members can be stored in one Named graph, information about books in a

second one, and information about borrowing of books by members in the Default graph.

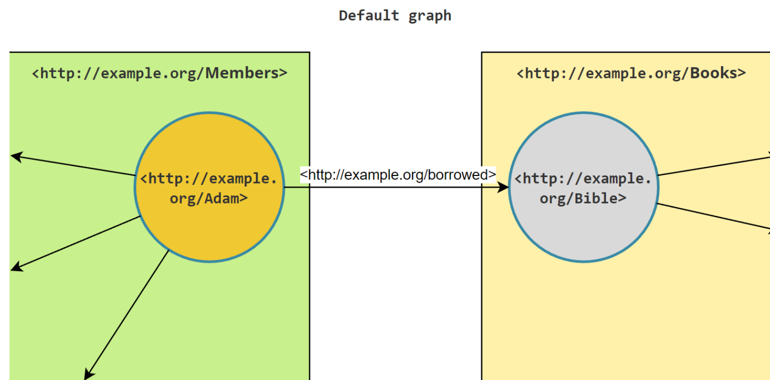

Figure 2.4: RDF Named graph example diagram

2.3 Concrete syntaxes for RDF

There is a number of serialization syntaxes for RDF [11]. These serialization algorithms are non-deterministic[2], or in other words, the output can change even though the data for serialization have not changed. In addition to serializing data, most of the syntaxes try to shorten the notation.

For example, Turtle [12] allows representation of the initial parts of IRI with short strings ending with ":" (colon) called Prefixes. The full form the Prefix is representing is specified once at the start of the serialized output. Turtle also shortens the RDF document by allowing it to write a semicolon-separated list of pairs of predicates and objects related to the same subject. Or a comma-separated list of objects instead of repeating the subject and predicate. Among the most used serialization syntaxes are the following:

- Family of RDF Turtle languages (N-Triples [13], N-Quads [14], Turtle [12], and TriG [15])
- RDF/XML [16] (XML-based syntax for RDF)
- JSON-LD [17] (JSON-based syntax for RDF)

[2]With the exception of specialized algorithms creating a normalized output, but they come with enormous performance hits. We will talk about one in chapter 4 - Existing tools research.

- RDF/JSON [18] (alternative JSON-based syntax for RDF)
- RDFa [19] (for expressing RDF in HTML documents)
- N3 [20] (non-standard serialization, similar to Turtle with the ability to define inference rules)

2.4 Example of RDF dataset

Here is an example of a very small RDF dataset containing everything discussed in this chapter except the use of the Named graph. Until this example, everything was in basic N-Triples syntax. Let us make it easier to read this example by showing it in Turtle syntax, which aims to be easily readable by humans.

```
1  @prefix e: <http://example.com/> .
2  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3  @prefix wd: <http://www.wikidata.org/entity/> .
4  @prefix schema: <http://schema.org/> .
5  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
7  @prefix skos: <http://www.w3.org/2004/02/skos/core#> .
8  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
9
10 e:Adam
11     a foaf:Person ;
12     e:knows e:Lucy ;
13     foaf:knows [ foaf:firstName "Bob" ] ;
14     foaf:interest wd:Q54872 .
15
16 e:Lucy
17     a foaf:Person ;
18     foaf:firstName "Lucy"@en , "Lucka"@cs ;
19     schema:birthDate  "1991-09-04"^^xsd:date .
20
21 e:knows
22     a rdf:Property ;
23     skos:definition "A person at least saw this person."@en ;
24     rdfs:domain foaf:Person ;
25     rdfs:range foaf:Person .
```

On lines 1 to 8, we can see the definition of Prefixes starting with one for our example namespace. On lines 10 to 14, is a definition of resource `e:Adam` or `<http://example.com/Adam>` in full IRI representation. Line

11 tells us that `e:Adam` is an instance of type `foaf:Person`, where 'a' in the data is a special abbreviation of `rdf:type`. Because the last triple ended with a semicolon, we know that on the next line, we still talk about `e:Adam`, and we learn that `e:Adam` knows `e:Lucy`, where relation (predicate) `knows` is defined in our example (on lines 21 to 25). On line 13, there is also a relation `knows`, but this one is from FOAF[3] Vocabulary Specification [21], which defines knows differently[4] than we do in this example. We can also notice a blank node defined in between '`[ ]`' (square brackets) on this line. All in all, line 13 tells us that `e:Adam` knows someone with first name Bob. Line 14 says that `e:Adam` have interest in `wd:Q54872`, a resource from WikiData [22], which is a free and open knowledge base project (sister to Wikipedia [23]). When we substitute its Prefix for its IRI, we get an URL `http://www.wikidata.org/entity/Q54872` linking to the definition of the resource, from which we can see that `e:Adam` is interested in RDF.

Because line 14 ended with a '`.`' (dot), we no longer read by pairs of predicate and object. We read a full triple split on lines 16 and 17, telling us that `e:Lucy` is also of type `foaf:Person`. Line 18 tells us first name of `e:Lucy` in two different languages using the comma-separated list of objects. From line 19, we learn that `e:Lucy` was born on the 4th of September 1991, by using the predicate `schema:birthDate` defined on `http://schema.org/birthDate` and `xsd:date` as the literal type.

The last entity in our example is on lines 21 to 25. From the first statement, we can see that `e:knows` is of type `rdf:Property` making it suitable to use at the predicate part of a triple. Next, we have a definition of property `e:knows` in the English language. The second to last line defines the `rdfs:domain` of this property, meaning that it can be used only in a triple where the subject is an instance of class `foaf:Person`. Similarly, the last line tells us the `rdfs:range` restricting the object part of a triple when using this property to be an instance of class `foaf:Person`.

[3] FOAF stands for "Friend of a friend".

[4] The definition of `foaf:knows` can be found on `http://xmlns.com/foaf/0.1/#term_knows`.

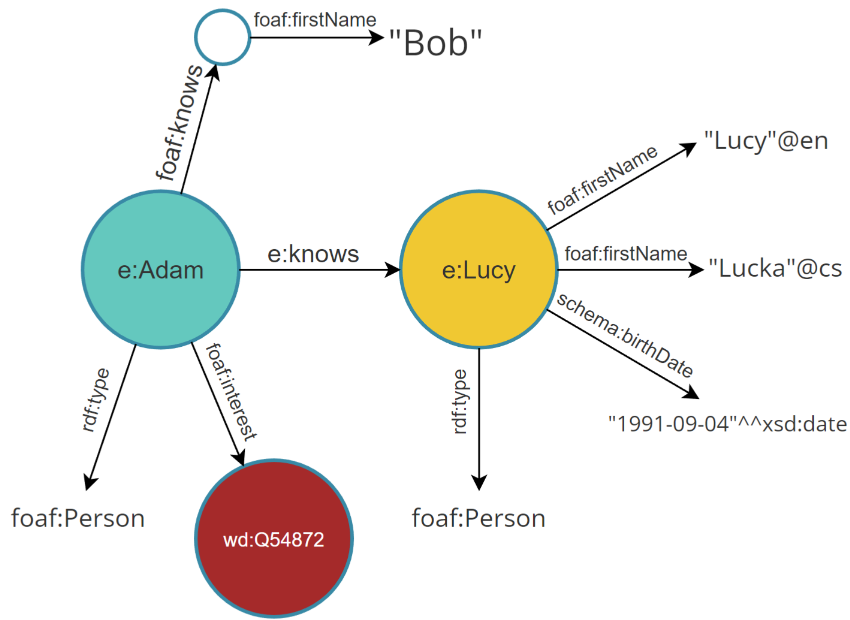

Figure 2.5: RDF dataset example diagram

The strength of RDF is in reusing resources someone else already created, as seen in this example, with resources mostly from `http://xmlns.com/foaf/0.1/` namespace. Making it perfect for use in the Assembly line, where models of decrees and laws are created with the use of terms from other decrees and laws.

Chapter 3

Assembly line processes

When we familiarized ourselves with the RDF technology used to store data in the Assembly line. We can proceed to analyze the existing processes in the Assembly line and design new ones to improve user experience with the new tool in mind.

3.1 Vocabulary modelling process

To better understand processes in the later part of the methodology of modelling physical law documents in the Assembly line, we first need to apprehend what is actually created before submitting it for review.

Consider that we, Editor users of the Assembly line, want to model the EU COUNCIL DIRECTIVE 1999/37/EC[1] on the registration documents for vehicles. As mentioned in Chapter 1 - Introduction, first, we create a new vocabulary (Assembly line terminology for a conceptual model) in the Mission control tool by specifying its name. This will create a project with a main entity called Vocabulary ("slovník" in Czech[2]), describing the vocabulary as a whole and linking to the other two created main entities that make the vocabulary dataset: Glossary and Model.

[1] More can be found about the directive on `www.eumonitor.eu`.

[2] Czech translations will help us understand example data because many ontologies that describe conceptual models in the Assembly line have IRIs based on the Czech language.

Next, we need to create and define terms ("pojem" in Czech) from the directive with the use of the TermIt tool. Entities of terms are collected under the Glossary entity ("glosář" in Czech). When we have our terms created, we need to add relationships between them in the OntoGrapher tool to give them the same hierarchy and structure as in the law document. These relationships are created under the third main entity called Model ("model" in Czech). When modelling relationships with OntoGrapher, we do so by creating a diagram, which is also saved as an attachment to the modelled vocabulary.

The following example shows parts[3] of the dataset of the COUNCIL DIRECTIVE 1999/37/EC modelled in the Assembly line:

```
1  # Vocabulary definition
2  <https://slovník.gov.cz/generický/eu-directive-1999-37-ec> a
3      a-popis-dat-pojem:slovník, owl:Ontology;
4    a-popis-dat-pojem:má-glosář g-sgov-eu-directive-1999-37-ec:glosář;
5    a-popis-dat-pojem:má-model g-sgov-eu-directive-1999-37-ec:model;
6    dcterms:created "2021-08-12T10:31:45.913Z"^^xsd:dateTime;
7    dcterms:rights <https://creativecommons.org/licenses/by-sa/4.0>;
8    dcterms:title "COUNCIL DIRECTIVE 1999/37/EC on the registration documents
   ↪  for vehicles"@en;
9    bibo:status "Specification"@en;
10   vann:preferredNamespacePrefix "g-sgov-eu-directive-1999-37-ec-pojem";
11   vann:preferredNamespaceUri
12     "https://slovník.gov.cz/generický/eu-directive-1999-37-ec/pojem/";
13   owl:imports g-sgov-eu-directive-1999-37-ec:glosář,
   ↪  g-sgov-eu-directive-1999-37-ec:model;
14   owl:versionIRI
15     "https://slovník.gov.cz/generický/eu-directive-1999-37-ec/verze/1.0.0" .
16
17 # Term definition
18 g-sgov-eu-directive-1999-37-ec-pojem:osvědčení-o-registraci a skos:Concept;
19   skos:definition "Doklad osvědčující, že vozidlo je registrováno v členském
   ↪  státě."@cs, "The document which certifies that the vehicle is
   ↪  registered in a Member State."@en;
20   skos:inScheme g-sgov-eu-directive-1999-37-ec:glosář;
21   skos:prefLabel "Osvědčení o registraci"@cs, "Registration certificate"@en.
22
23 # Definiton of relationship term
24 g-sgov-eu-directive-1999-37-ec-pojem:má-motor a skos:Concept,
 ↪  z-sgov-pojem:typ-vztahu;
25   skos:prefLabel "has engine"@en, "má motor"@cs;
26   rdfs:subClassOf [ a owl:Restriction;
27       owl:onProperty z-sgov-pojem:má-vztažený-prvek-1;
28       owl:someValuesFrom g-sgov-eu-directive-1999-37-ec-pojem:vozidlo
29     ], [ a owl:Restriction;
30       owl:allValuesFrom g-sgov-eu-directive-1999-37-ec-pojem:vozidlo;
31       owl:onProperty z-sgov-pojem:má-vztažený-prvek-1
32     ], [ a owl:Restriction;
33       owl:allValuesFrom g-sgov-eu-directive-1999-37-ec-pojem:motor;
34       owl:onProperty z-sgov-pojem:má-vztažený-prvek-2
35     ], [ a owl:Restriction;
36       owl:onProperty z-sgov-pojem:má-vztažený-prvek-2;
37       owl:someValuesFrom g-sgov-eu-directive-1999-37-ec-pojem:motor
38     ] .
```

[3] Full published dataset is available on GitHub in `opendata-mvcr/ssp` repository.

The first part of the example depicts the saved data structure of the Vocabulary, where we can notice the link to Glossary and Model via `owl:imports` predicate on line 13. A term is defined on lines 18 to 21 with `skos:definition` and `skos:prefLabel` in both the Czech and English languages. The last section on lines 24 to 39 represents the term "has engine" ("má motor" in Czech) created as a relationship. We can see the definition of the term followed by the data structure describing the relationship between the terms "vehicle" ("vozidlo" in Czech) and "engine" ("motor" in Czech).

3.2 Current Publication process

When we have our data created, we can click on Publish in the Mission control tool, which will begin the Publication process. After that, the back-end application SGoV will generate three files (in TriG[4] syntax) from the RDF dataset, each named by the vocabulary and suffixed with one of the main entities (Vocabulary, Glossary and Model). SGoV also generates a file with a list of attachments and the attachment files themselves, if there are any. These files are then committed to the GitHub[5] repository of Semantic government vocabulary (SSP[6]) [25], and a GitHub Pull Request[7] is created.

Experts on RDF data and modelling in the KODI team are notified about the creation of a new Pull Request by email. When they open the Pull Request, they are shown the textual differences between files currently present in SSP and newly committed changed files. The reviewer goes through these differences and evaluates them. If there is no problems with the data, the reviewer approves the Pull Request and merges it into the SSP repository. If there is any problem, the reviewer contacts the author (mainly via email, as this step is not standardized) about the issue. After the Editors resolve these problems, the Publication process starts over by clicking Publish in the Mission control tool.

[4] TriG is an extension of Turtle syntax, allowing to define Named graphs.

[5] GitHub is a web-based platform that enables collaboration, version control, and hosting of code repositories [24].

[6] Semantic government vocabulary is "Sémantický slovník pojmů" in Czech hence the abbreviation SSP.

[7] GitHub Pull Request is a mechanism allowing proposals of changes to data saved in a repository. It provides a way to merge changes to the repository [26].

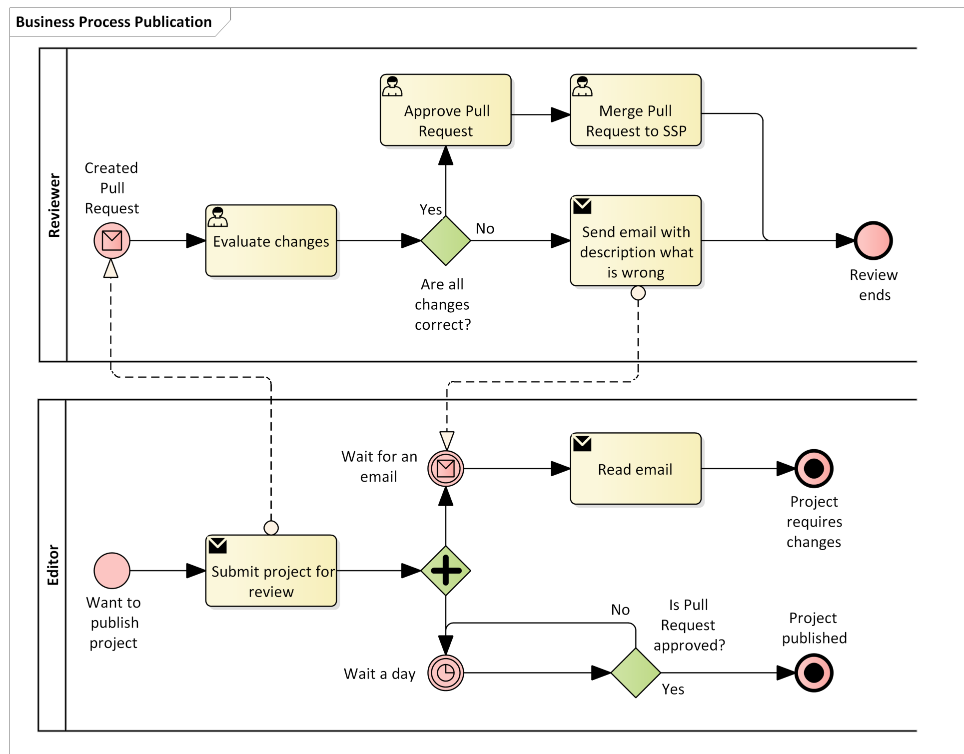

Figure 3.1: Publication process (AS IS)

The Publication process, as is, has a lot of issues. Mostly that reviewers are required to have an account in the external tool GitHub, hardly traceable communication, long descriptions of problem location (thanks to the fact that reviewers see RDF representation and Editors see the UI of Assembly line tools), and the fact that changes are shown based on text differences of the serialized dataset. Meaning that the reviewer needs to have extensive knowledge of RDF(S) and the data model of the Assembly line to understand what the data describes.

Also, as we learned in chapter 2 - Introduction to RDF(S), the serialized output of RDF is not deterministic, so even a change of one attribute in the data can and often does result in many differences shown by text comparison. Because even unchanged entities randomly change their order in the output file, it is sometimes next to impossible for a human to distinguish reordering changes from actual changes in hundreds of changed lines.

The communication between Editors and Reviewers is difficult and confusing because the user accounts of Editors are not propagated to GitHub. So no notifications about rejected or approved Pull Requests are sent to them. The fact that Pull Request was rejected Editors discover by receiving

an email from a Reviewer regarding problems in the changes, but if it was approved, no email is expected to be sent by Reviewers. Editors need to check the web page of the Pull Request periodically to find out if it was approved. This is aggravated by the reality that the link to this website is not persisted anywhere, so Editors need to save it on their own when it is shown in a notification after pressing Publish in Mission control, visible only for a few seconds.

3.3 New Publication process

So in collaboration with Bc. Filip Kopecký (assigned to create the front-end part of the new tool), we reworked this process to resolve these issues with the new tool we called CheckIt. As We can see in the diagram on the next page, a new role called Gestor is used. Each Gestor is responsible for a set of vocabularies that they can review, essentially replacing Reviewers.

An Editor starts the new process similarly by clicking on Submit button in Mission control. Which generates a list of actual changes (excluding reordering changes) in a Publication context, and the Gestors of modified vocabularies are notified. Gestors review each change and decide if it is correct. If the intention behind some change is unclear, Gestor can create a comment with a question or a simple task (e.g. correct a typo) for Editors. They are notified about a comment within their submitted project, to which they can reply and decide if more modifications are needed. If so, Editors modify the project and submit it again. This will notify Gestors about updated changes, and if they are currently making reviews, CheckIt will show them a message that new changes are available. If Gestor does not agree with any change, they can provide a reason after rejecting it.

When all changes are reviewed, all are approved, and Gestors are satisfied with the changes, one of them can approve the publication context with or without a final message (e.g. tips for work in the future). This will submit changes to SSP and send notifications to Editors about the changes' approval. After that, Editors can decide to delete the project or continue to work on it.

If some changes are rejected, or more changes are required to be made, Gestor rejects the publication context and provides a rejection message with a reason. This will notify Editors about the rejection, and they can decide if the reason is fundamental, so they need to delete the project a start from scratch or edit the project and submit it again.

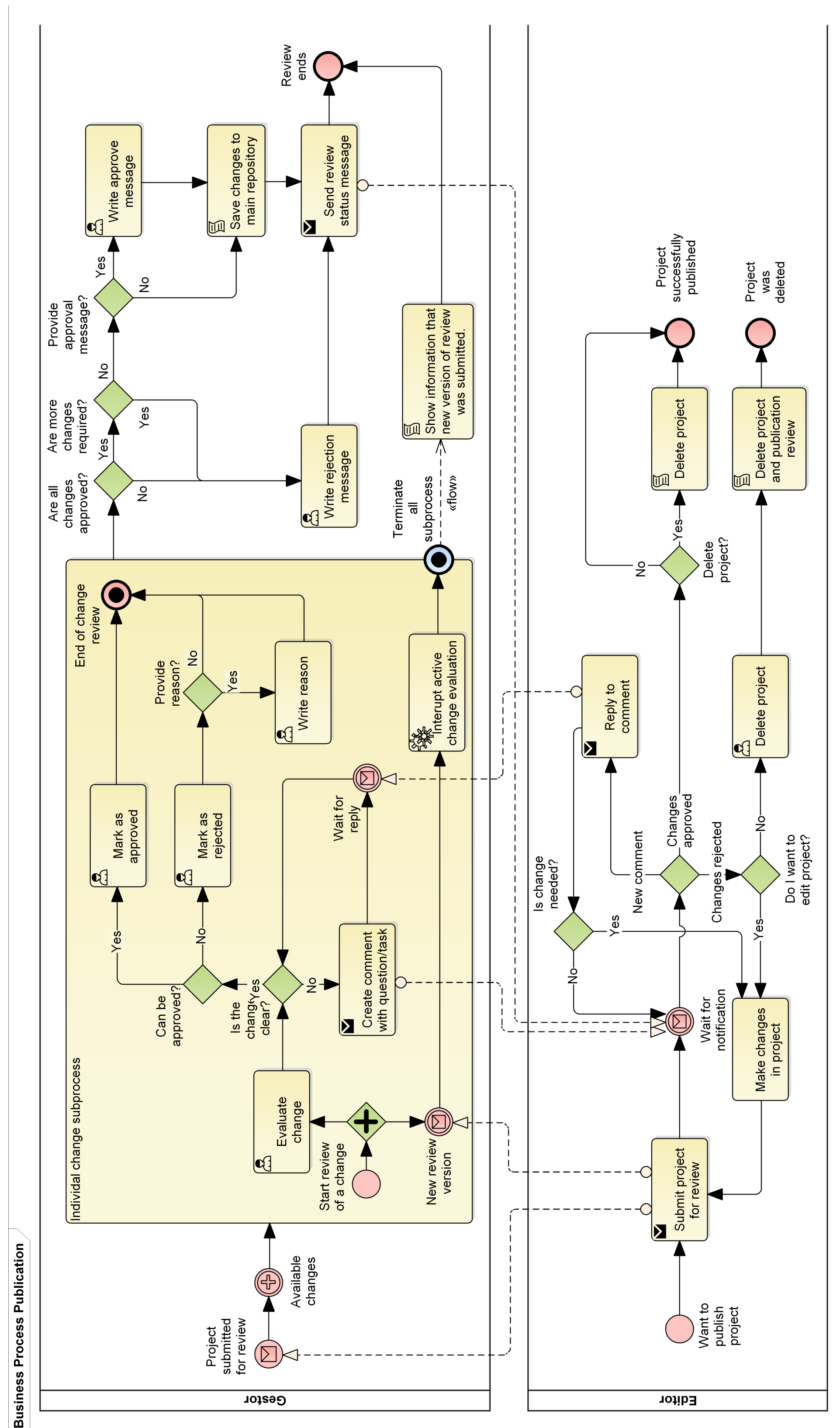

Figure 3.2: Publication process (TO BE)

3.4 Requesting to Gestor a vocabulary

To manage Gestors, an additional role of Admin is required. The system will need to be initiated with a single user that has an Admin role. This role gives the user privileges to assign Admin roles to other users and assign Gestors to vocabularies.

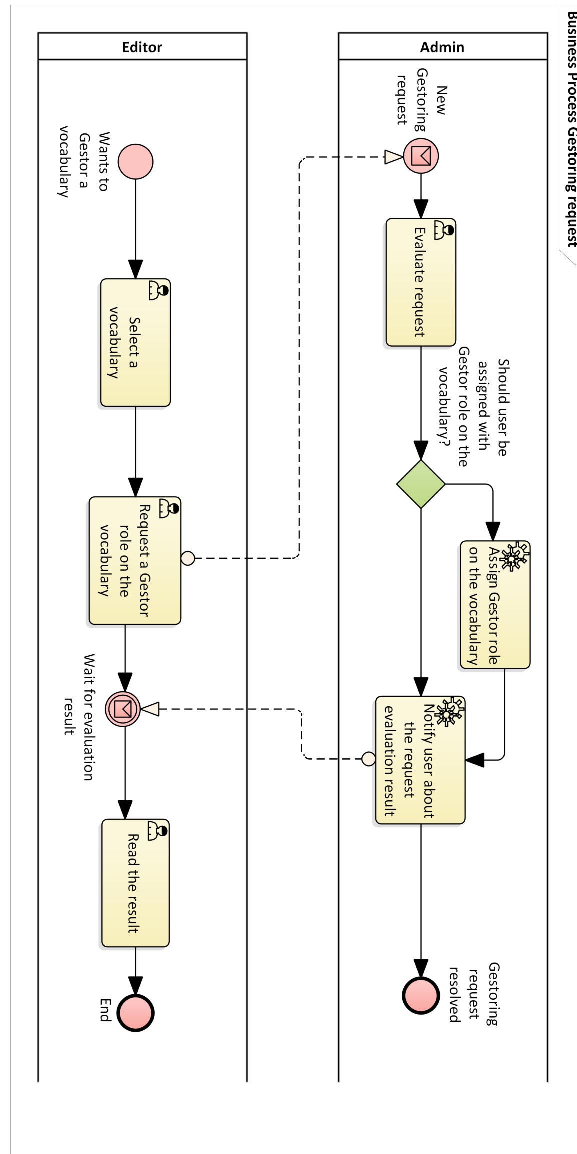

Figure 3.3: Gestoring request diagram

To further improve communication in the application, a new process for users was created to request a Gestor role on a vocabulary. In this process, an

Editor can view a list of vocabularies and select on which of them to request a Gestor role. Admins are notified about the request and can decide if the applicant should Gestor this vocabulary. If so, the system will assign a Gestor role to the applicant and send a notification that they can now review the vocabulary. Otherwise, the applicant is notified that the request was rejected by Admin.

Chapter 4

Existing tools research

From the new Publication process, a primary problem arose to solve, machine comparison of two RDF datasets and get their differences in a usable form. Unfortunately, that is an NP-hard problem, thanks to the existence of blank nodes [27]. Which, as we learned, are not identified uniquely across datasets, so even printing the same dataset twice generates different identifiers. With the size of RDF datasets planned to be compared, choosing the appropriate tool is critical for finding the changes in a reasonable amount of time.

4.1 Main requirements for the tool

Therefore we need to specify some requirements for the tools.

- Listing changes – output of the tool must contain the list of changed triples with distinguishment between removed and added ones,
- Actual changes – the tool needs to be able to distinguish between changed triples and triples that were only reordered,
- Identical blank nodes – if blank nodes differ only in generated blank node identifier, the tool needs to mark them as unchanged,
- Time constraint – finding these changes should be done within minutes so as not to interrupt the current workflow of the Assembly line.

Three RDF datasets were created in Turtle syntax to test these requirements with various copies with changed statements. The test datasets can be seen in doc/Test-datasets in the GitHub repository of the CheckIt server on `https://github.com/mighantos/checkit-server`.

- Simple dataset – containing eight triples without any blank node,
- Simple blank node dataset – containing six triples with one pointing at a blank node with a single triple in it,
- Real-world dataset – an average dataset from SSP containing almost five and a half thousand triples with blank nodes.

4.2 Researched tools

Let us look at some already existing tools that deal with comparing RDF datasets and find how they stack against each other.

4.2.1 Python RDFLib

Package RDFLib [28] for Python to work with RDF data contains a script able to compare two RDF datasets. The script's output has three text blocks: statements present in both, statements present only in the first file, and statements present only in the second file. That is precisely what we are looking for. And thanks to the open source 3-Clause BSD licence, the output can be modified to the needs of integration with the final application.

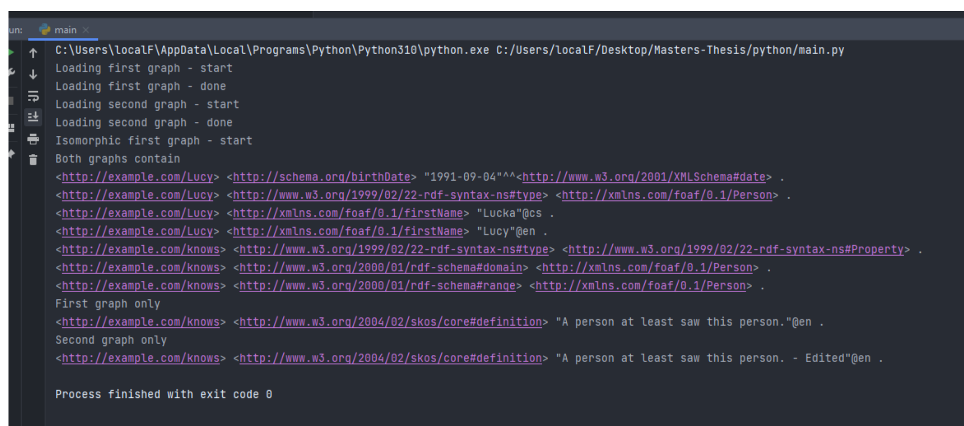

Figure 4.1: Python RDFLib output

Testing showed the script's ability to recognize identical blank nodes correctly. Unfortunately, it also revealed that the script ran on the Real-world dataset from SSP compared to its copy with one changed triple outside blank nodes for tens of minutes. And that is not compliant with the real-time workflow of the Assembly line.

4.2.2 Protégé

Protégé [29] is a free, open-source software platform for creating and managing ontologies, offering Protégé Desktop ontology editor with a plugin for comparing OWL [30] ontologies, both written in Java [31].

Testing with the Simple dataset with various changes worked well. The first problem arose when comparing datasets with blank nodes where the Protégé plugin failed to identify blank nodes differing only in the identifiers as identical. More problems occurred when trying to import the Real-world dataset to the Protégé Desktop. Where the application required to import all conceptual models of terms used in the Real-world dataset.

The class list displayed in the application then showed an alarming amount of classes (entities) called Error[number] (e.g. Error946), and these error classes were cyclicly connected to themselves. Furthermore, the result of comparing this dataset with its copy with one changed statement outside of blank nodes showed the actual change made, 50 changes on Error classes, 250 created statements and 249 removed statements.

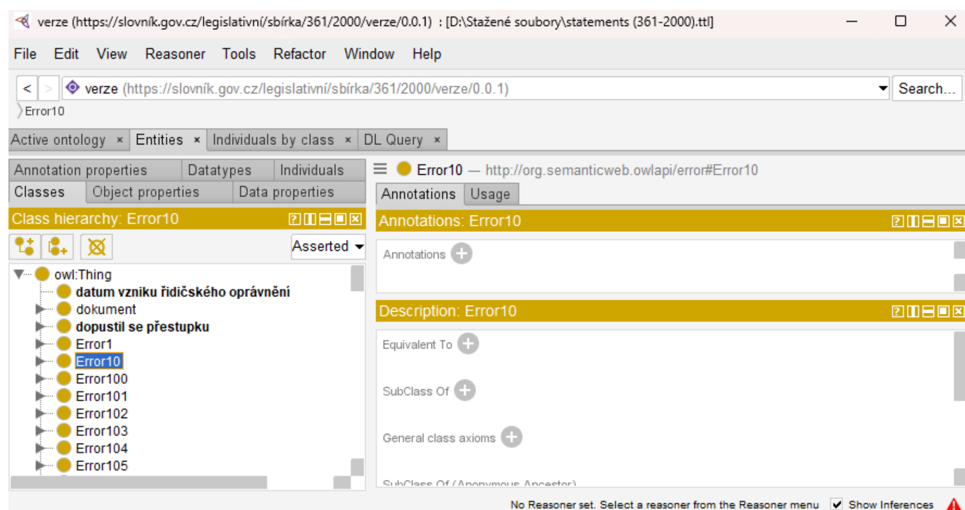

Figure 4.2: Error classes in Protégé

4.2.3 OWLDiff

OWLDiff [32] is an open-source tool to perform a two-way difference of OWL ontologies written in Java 14 under the GNU General Public License. It can be compiled as a stand-alone application as well as a plugin for Protégé or NeON [33]. A stand-alone version of the application was used for testing purposes to determine the capabilities.

The application prompts you to select the first and second files to compare. All test datasets were compared in a matter of seconds. Identical blank nodes were recognized correctly. The application also allows to select which statements, present only in the first file, to add, and which statements that are only present in the second file, to remove. The resulting dataset then can be saved by the application. But very similar Error classes were shown as in Protégé tools when testing Real-world SSP datasets.

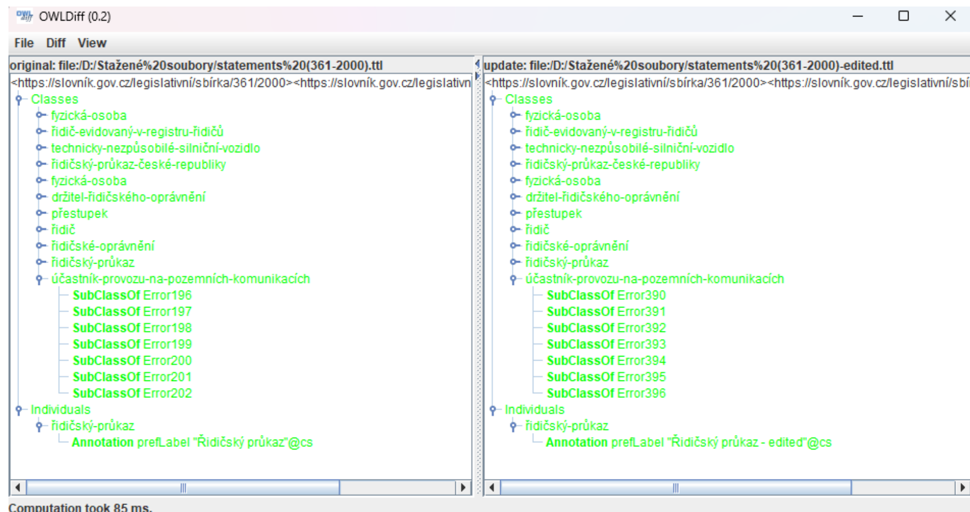

Figure 4.3: Error entities in OWLDiff

After contacting the author, it turned out that datasets in SSP are not fully following the rules of OWL, as was thought. This causes to show Error classes in OWLDiff and Protégé tool, meaning that any tool created to compare OWL ontologies would require correcting the data in SSP.

4.2.4 RDF Graph Normalization

One technique for comparing two RDF files is to normalize them using the RDF Graph Normalization Algorithm [34] and compare them with general text-based tools. JSONLD-JAVA [35] is a Java implementation of the JSON-LD 1.0 specification [17] and the JSON-LD-API 1.0 specification [36], allowing to convert JSON-LD file to normalized N-Quads file. Although RDF Graph Normalization Algorithm faces similar challenges with blank nodes, as mentioned at the beginning of this chapter, testing showed times below half of a minute, even on Real-world datasets from SSP.

But doing a simple text comparison comes with its shortcomings. The plain text output of this method needs to be backtracked in the file to corresponding lines. These lines then need to be mapped to other lines that refer to the same RDF entity. By doing so, we can extract the needed information about the entity and its surroundings to describe the change to the end user.

4.2.5 Apache Jena

Apache Jena [37] is a free and open-source Java framework, released under the Apache License 2.0, designed for writing Semantic Web and Linked Data applications. It provides an API to extract and write RDF data to graphs. Part of the Jena project is a program called RDFDiff, which is capable of comparing two RDF graphs. The output of this program is list of statements present only in the first graph followed by statements present only in the second graph.

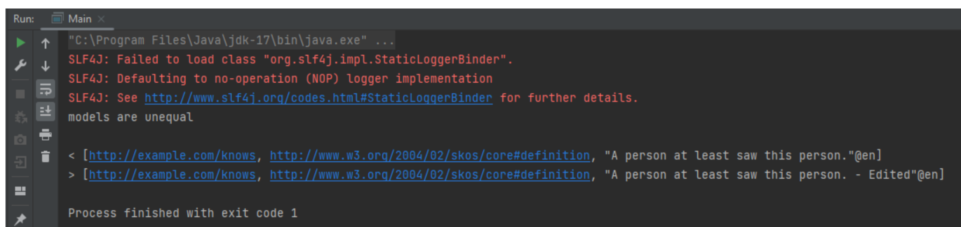

Figure 4.4: Jena RDFDiff output

While testing the program, it became clear that speed was a priority, as all comparisons were completed in fractions of a second. Blank nodes only differing in identifiers were correctly identified as identical. But when there was a change even in one blank node, all blank nodes were marked as removed

and created. Upon inspecting the code, it was clear that it is done by design to not deal with comparing matching blank nodes and be as fast as possible. But because the program is open-source, it can be modified to our needs.

4.3 Conclusion

	Listing changes	Actual changes	Identical blank nodes	Time constraint
Python RDFLib	✓	✓	✓	✗
Protégé	✓	✓	✗	✓
OWLDiff	✓	✓	✗	✓
RDF Graph Normalization	✓	✓	✓	✓
Apache Jena	✓	✓	✓	✓

Table 4.1: Tools capabilites

In conclusion, the Python script to compare RDF datasets from RDFLib was ruled out because of insufficient speed. Protégé tools problems found with recognizing identical blank nodes also ruled these tools out. The option to change SSP data and all tools of the Assembly line to create data compliant with OWL was deemed too costly, and consultation with the KODI team confirmed it. This ruled out OWLDiff or any other potential OWL comparing tools. So we are left with JSONLD-JAVA and Jena API, where neither of them is a complete tool for our needs, but both can recognize every required change in a reasonable time.

After more analyses of the Jena API RDFDiff program, it was found that it works by comparing lists of statements of both datasets. Meaning it is very similar to the RDF Graph Normalization method in the way of finding changes. But the internal representation of data in Jena API RDFDiff allows for easier access to information about changed entities and their surrounding.

For this reason, the Jena API RDFDiff was selected as the main inspiration for incorporating a similar algorithm with the use of Jena API in the back-end part of CheckIt for finding changes. It was furthermore supported by the

use of other features of Jena API in some Assembly line tools and prior experiences with it.

Chapter 5

Ontology

Before the implementation of the CheckIt back-end can start, we need to define the database structure. In traditional relational databases, this would be done by defining a schema with tables named by entities and columns representing properties of those entities. But in RDF(S), the schema is replaced by ontology, which is an RDF dataset defining entities and possible relationships between them. For this purpose, a new ontology called Change description ("popis-změn" in Czech) was created.

5.1 Main entities

The ontology is based on a few main entities allowing it to describe and save everything needed in the application, namely:

- Change – describing modifications in RDF triples of the reference dataset,
- Publication context – grouping Changes under one entity linking to the corresponding project they were created in,
- Comment – to save discussion comments on Changes and final messages on Publication contexts,
- Notifications – an entity containing in-application notification text and a link to the location in the application the notification is about,
- Gestoring request – representing the request of an Editor user asking to be assigned the Gestor role on a vocabulary.

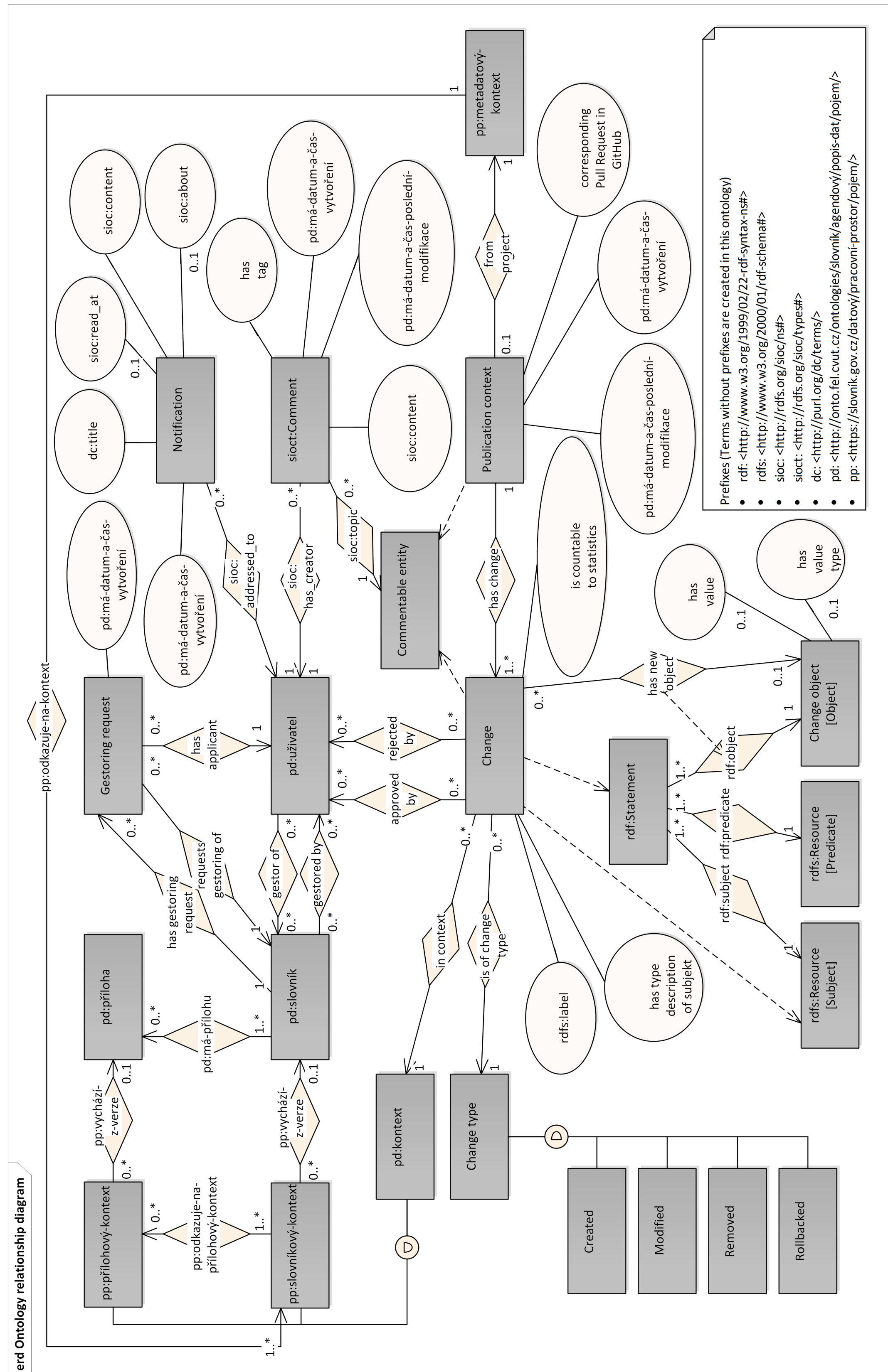

Figure 5.1: Ontology relationship diagram

5.2 Relationship diagram

Modelling of the ontology started with the use of the Assembly line, where new entities and relationships between them were created. But because the Assembly line can only use newly created terms or terms from vocabularies present in SSP, the rest of the relationships needed to be added manually afterwards by modifying the RDF output. A diagram, which we can see on the left, was also created to help grasp the relationships between the entities. The ontology[1] uses IRIs based on Czech words to maintain consistency with other ontologies created for the Assembly line, but English labels of the created entities will be used in the following text to simplify its description.

5.2.1 Change entity

As we can see from the diagram on the previous page, the `Change`[2] entity inherits from `rdf:Statement`[3] which points with `rdf:subject`, `rdf:predicate` and `rdf:object` to the RDF triple that was changed. To differentiate what happened with this triple a relationship `is of change type` pointing to abstract entity `Change type`, which can state if the triple was `Created`, `Modified`, `Removed` or `Rollbacked`. The `Modifed` type is a special case of `Created` and `Removed` on a statement that was only changed in object value. For this `Change type`, an additional relationship `has new object` is required to be able to store both the old and the new values of the object of the changed statement. The relationship `has new object` is inherited from `rdf:object` as it represents almost the identical bond. The `Rollbacked` type is used when a `Change` was already reviewed by a Gestor, but Editor submitted a new version of `Change`s that does not include the same `Change`.

The `Change` entity also points to a few `rdfs:Literals`[4] with relationships `rdfs:label` saving the preferred name of the `Change` entity (e.g. name of a term), `has type description of subject` describing the entity type of the subject in the changed statement (e.g. Term, Vocabulary or Blank node) and `is countable to statistics` saving if a `Change` should be counted to statistics on how many changes need reviewing in Publication context. This is incorporated because some changes will be grouped to show as a larger whole to help Gestors with reviewing.

[1]You can find the whole Ontology in GitHub repository kbss-cvut/popis-zmen-ontology.
[2]If an entity in the diagram does not have a Prefix, it is created and defined in the Change description ontology.
[3]List of used prefixes can be seen in the right bottom corner of the diagram.
[4]Bubbles represent relationships pointing to `rdfs:Literal`, to simplify the diagram.

Relationships `approved by` and `rejected by` pointing to entity `pd:uživatel` ("user" in English) are there to allow Gestors to review `Changes`. To ensure that only Gestors of a specific vocabulary can review it, a relationship `in context` points to abstract entity `pd:kontext`, that can be either `pp:slovníkový-kontext` ("vocabulary context" in English) which is a copy of the canonical vocabulary[5] with made changes or `pp:přílohový-kontext` ("attachment context" in English) a copy of the canonical attachment with made changes. These contexts point to the canonical version of vocabulary/attachment with `pp:vychází-z-verze` ("based on version" in English). A new relationship `gestored by` then completes the circle to `pd:uživatel`, allowing to check if a user can review the change. An inferred relationship `gestor of` in the opposite direction is specified to simplify queries to the DB, as it is going to be common to list the gestored vocabularies of a user.

In the later part of the implementation, a problem was encountered with saving object value to the database as it can represent literal, IRI or a blank node. So the generic `rdfs:Resource` was changed to `Change object`. This entity points to two optional `rdfs:Literals` with `has value` representing the value (e.g. IRI or string) and `has value type` describing the type of the value (e.g. xsd:integer[6]). The relationships are optional to allow representing Blank node, which does not have either, and also IRI, which has value but does not have a value type.

As we can also notice, the `Change` entity has an arrow to `rdfs:Resource` of the subject depicting that `Change` inherits from it. This relationship is there to highlight a possible use case where a `Change` in a blank node is using an identifier of another `Change` that has the blank node as an `Change object`. This is done to simplify the reconstruction of the changed RDF structure.

5.2.2 Publication context entity

As previously mentioned, `Publication context` groups changes made in a project. It does so by pointing at individual `Change` entities with relationship `has change` and the reference project entity `pp:metadatový-kontext`[7] pointed at by the relationship `from project`. Two `rdfs:Literal` entities saving the time and date of creation and last modification, are linked

[5]Canonical is used in the Assembly line to describe the original version or in other words the version currently saved in SSP.

[6]Prefix `xsd:` is representing `<http://www.w3.org/2001/XMLSchema#>` namespace.

[7]The `pp:metadatový-kontext` is "metadata context" in English, but it was renamed to "workspace" and then "project" as its purposed changed during the development of the data architecture of the Assembly line.

with pd:má-datum-a-čas-vytvoření and pd:má-datum-a-čas-poslední-modifikace relationships respectively. These rdfs:Literals are used to order loaded Publication context entities and check if a user is reviewing the latest version of changes. The third relationship corresponding Pull Request in GitHub also pointing at rdfs:Literal is an identification of GitHub Pull Request created by the SGoV server for the reference project.

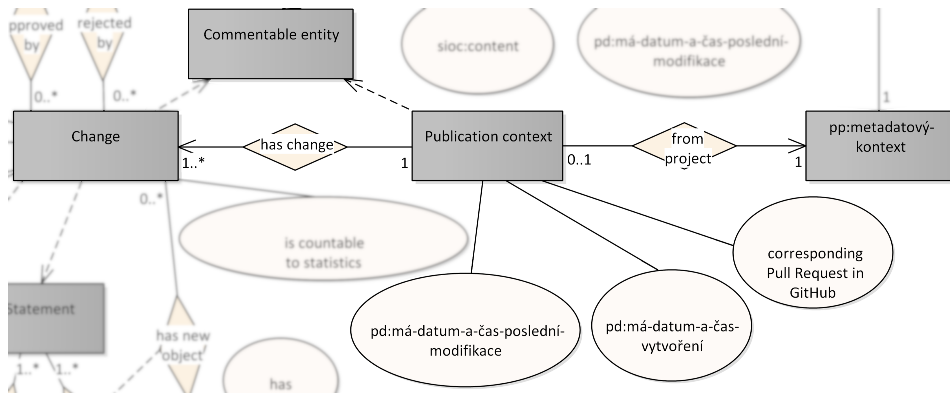

Figure 5.2: Ontology relationship diagram - Publication context closeup

5.2.3 Comment entity

The previous two entities we talked about can have some form of message attached to them. For that purpose, a sioct:Comment is reused from the http://rdfs.org/sioc/types# namespace with relationships sioc:has_creator to point at a user who created the comment and sioc:topic pointing at abstract Commentable entity, which both Change and Publication context inherit from. For very similar reasons as Publication context, sioct:Comment also has the time and date of creation and last modification attached to them.

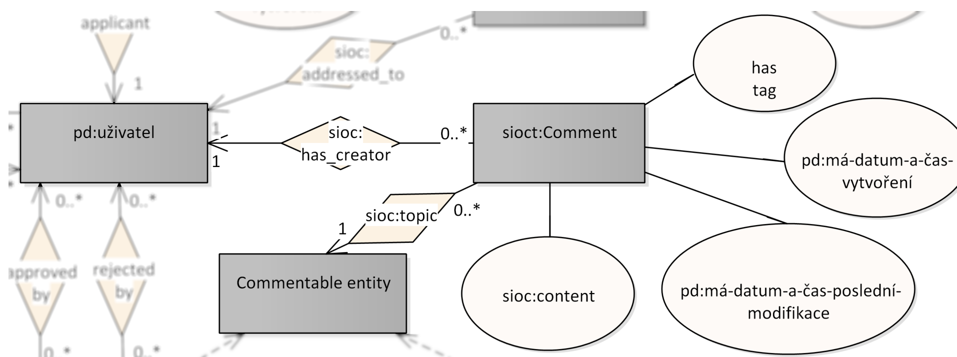

Figure 5.3: Ontology relationship diagram - Comment closeup

To save the text of the message, a sioc:content pointing at string rdfs:Literal is used. On Change entities, there can be two types of comments, a rejection or a discussion. To differentiate between them, a rdfs:Literal is attached with relationship has tag. The reason why comment tags are not done in the same way as Change type is that TermIt is using sioct:Comment in the same way, just without the tag, and it was seen as a solution, better compatible with future reuse in TermIt and other tools of Assembly line.

5.2.4 Notification entity

To inform users about new events in the application, a Notification entity is needed. It has the same relationships to save content and time and date of creation as sioc:Comment. Furthermore, it points to rdfs:Literals with relationships dc:title saving short title of the notification, sioc:about saving a path to a page where the user can find more about what the notification is talking about, and optional sioc:read_at marking the date and time when was the notification read. Each Notification is assigned to a single user with relationship sioc:addressed_to to allow personalization of notifications.

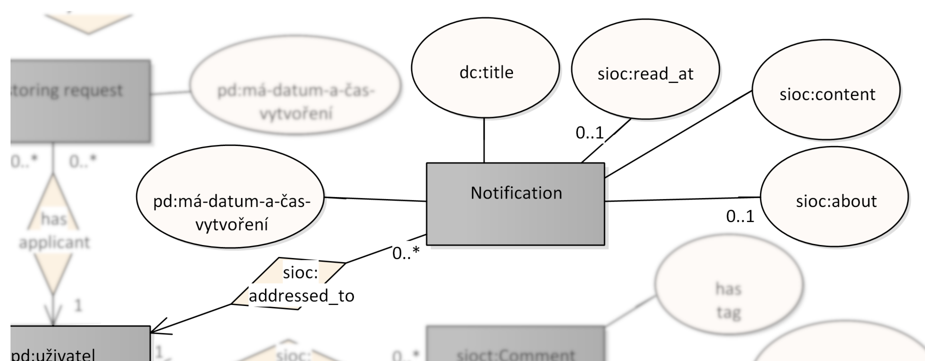

Figure 5.4: Ontology relationship diagram - Notification closeup

5.2.5 Gestoring request entity

As mentioned in Chapter 3 - Assembly line processes, Editors can request to be added as a Gestor to a vocabulary. To save these requests, a Gestoring request entity is created in the ontology. This entity is pointing at a canonical vocabulary (pd:slovník entity) with relationship requests gestoring of and to the user (pd:uživatel entity) who is requesting it with has applicant

relationship. A relationship `has gestoring request` going from canonical vocabulary to `Gestoring request` inferred as an opposite to `requests gestoring of` is there to once again simplify queries to database. To warn Admins about long pending requests, a creation time and date are again saved with `pd:má-datum-a-čas-vytvoření` relationship.

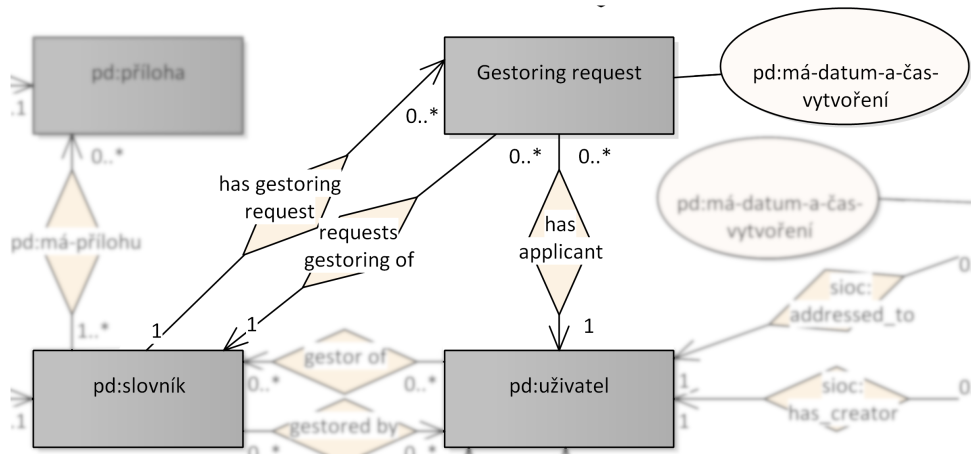

Figure 5.5: Ontology relationship diagram - Gestoring request closeup

Chapter 6

Implementation

With ontology created, the implementation of the solution could start. After a discussion with Bc. Filip Kopecký, we agreed on a simple client-server architecture with REST[1] endpoints on the server side and him creating a thin client. So in this chapter, we will learn about the technologies used and the main ideas and algorithms employed while implementing the back-end part of the CheckIt application[2].

6.1 Used technologies

The CheckIt application will be handed over to the working group responsible for the Assembly line tools consisting of just five people. To simplify this transfer and help with maintaining the toolset as a whole, technologies similar to those already used in the Assembly line were preferred.

Java was chosen as the programming language because all server applications in the Assembly line are written in Java, and the Jena API is written for it as well.

When deciding what to use for building the application, a choice had to

[1]REST, or Representational State Transfer, is a set of principles for designing networked applications [38].

[2]You can find the implemented code in the GitHub repository mighantos/checkit-server.

be made between the two most popular ones, Maven and Gradle [39]. Both are widely used automated building tools, and both are used in the Assembly line. However, Gradle was chosen for its superior performance [40], simple and readable configuration, and compatibility with a wider range of IDEs.

The Spring Boot framework used in the Assembly line was also selected, as it helps simplify the code, configuration, database transactions, security and overall experience of writing Java applications [41]. Spring Boot framework is designed for creating RESTful API applications, which is ideal for use in the client-server architecture.

To communicate with the RDF database, a Java OWL Persistence API (JOPA) framework aimed at efficient programmatic access to OWL2 ontologies and RDF graphs in Java has been chosen [42]. JOPA is also used in all back-end applications of the Assembly line toolset. The API aims to resemble Java Persistence API (JPA), an object-relational mapping (ORM) specification in Jakarta EE [43], commonly used in Java applications. To integrate JOPA transactions into Spring, the JOPA-Spring-transaction[3] dependency was added to take advantage of the transaction notation.

Technology	Version	Description
Java	17	Programing language
Spring boot	3.0.6	Main Framework
JOPA	0.22.0	Database persistence framework
JOPA OntoDriver	0.22.0	Database connector driver
Apache Jena API	4.8.0	Library for manipulating with RDF data
Keycloak API	21.1.1	Library for communication with Keycloak Admin API
Kohsuke GitHub API	1.314	Library for communication with GitHub

Table 6.1: Used technologies

6.2 Change resolving algorithm

One of the most essential algorithms of the CheckIt server is the resolving of changes made in a project. To do so, a list of vocabulary contexts present in the specified project is found. On each of these vocabulary contexts, a query to the database is called that loads the entire content of the vocabulary

[3] Link to repository: `https://github.com/ledsoft/jopa-spring-transaction`

graph in vocabulary context to a `Model` object of the Jena API. The same thing is done for the canonical vocabulary graph. These two `Models` are then compared using the method `isIsomorphicWith` of the Jena API. This method returns if the two graphs are identical while correctly recognizing identical blank nodes. If the two `Models` are isomorphic, the algorithm will return an empty list of changes.

6.2.1 Change resolving without blank nodes

If there is at least one change in the two `Models`, the algorithm[4] continues by iterating over all statements of vocabulary context that do not include a blank node identifier and checks on each, if the canonical vocabulary statements include it. If the statement is not in the canonical vocabulary, it is added to a Java `Map` with subjects of changed statements as keys mapping to a list of changed statements of this subject. When all statements have been iterated over, the `Map` will contain all statements that are new, separated into lists with a common subject. If we do the same thing, but with vocabulary context and canonical vocabulary interchanged, we will get a `Map` of removed statements.

With these two `Maps`, a list of `Changes` can be created by iterating over all subjects (keys) of removed statements `Map` and checking if they are present in the keyset of new statements `Map`. If not, the list of statements corresponding to the subject in the removed statements `Map` is converted to `Change` objects with `Removed` change type. But if the subject is in new statements `Map` keyset, each removed statement needs to be compared to the list of new statements of the subject (key). In this comparison, if there is a new statement that has the same subject and predicate and the object is of the same type[5] as the removed statement, the new statement is removed from its list and converted to `Change`, which is given the `Modified` change type. An object part of the new statement is set in the `Change` as `newObject` property. If the statement is only present in removed statements, it is created as a `Change` with `Removed` change type. The rest of the new statements can then be converted into `Changes` with `Created` change type. Using `Maps` instead of lists significantly narrows the comparison part to speed up the algorithm.

All `Changes` also set a `label` property, which, thanks to the Jena `Model` representation of the graphs, can be easily found without calling the database.

[4]This algorithm can be found in the `ChangeResolver` class in GitHub repository `mighantos/checkit-server` of CheckIt server.

[5]Meaning that they are both IRIs or both literals of the same type and if they are of type string, the languages must also match.

Labels are saved in all available languages by finding statements with the wanted subject and predicates `skos:prefLabel`[6] or alternatively `dc:title`[7]. Maps also help here because the labels can be searched only once for all changes on the same subject.

6.2.2 Change resolving in blank nodes

The next step is to resolve changes in blank nodes. As mentioned in Chapter 4 - Existing tools research, finding changes in generic blank nodes is an NP-hard problem. But in the case of the Assembly line, created blank nodes have structure and occurrence rules that reduce the complexity of matching them. They are created almost exclusively for modelled relationships by the OntoGrapher tool. To better understand the implemented algorithm, let us first look at an example of a relationship and its representation in data.

Relationships in the Assembly line

A vocabulary *REGULATION (EU) 2018/858 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL on the approval and market surveillance of motor vehicles and their trailers, and of systems, components and separate technical units intended for such vehicles* modelled in the Assembly line after identically named EU regulation[8] has a relationship `has manufacturer` between terms `Vehicle` and `Manufacturer`.

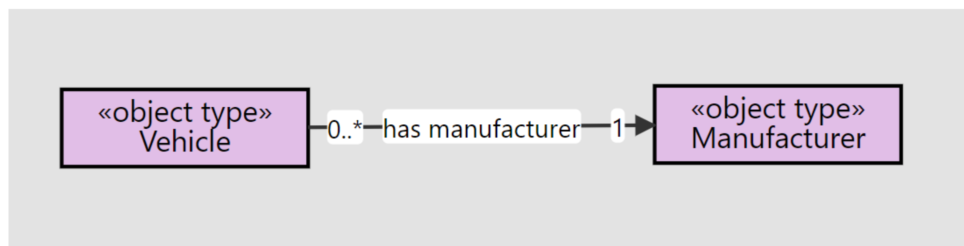

Figure 6.1: Relationship `has manufacturer` (simple view)

The diagram above, created in the OntoGrapher tool, shows the relationship in a simplified view. Because the relationship `has manufacturer` itself is

[6] Prefix `skos:` is `<http://www.w3.org/2004/02/skos/core#>` namespace.
[7] Prefix `dc:` is `<http://purl.org/dc/terms/>` namespace.
[8] You can find its wording on EUR-Lex.

also a term, just of a `relator`[9] type. A better representation would actually be two relationships going from `relator` term to the two terms `Vehicle` and `Manufacturer`. The relationship going from `relator` to `Vehicle` is `mediates 1`[10] denoting that the relationship `has manufacturer` is starting on term `Vehicle`. To define where the relationship `has manufacturer` is pointing to, a relationship `mediates 2`[11] going from it to `Manufacturer` is used.

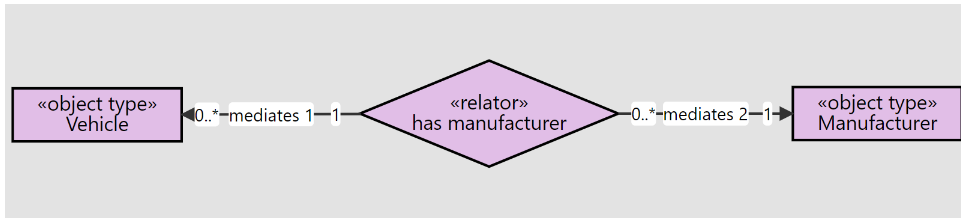

Figure 6.2: Relationship `has manufacturer` (full view)

This is described in RDF with the use of restriction from the Web Ontology Language (OWL)[12]. Restrictions are created in blank nodes on the relationship and on both terms at the ends of the relationship by the predicate `rdfs:subClassOf`[13]. These blank nodes are of type `owl:Restriction` and include the direction of the restriction with predicate `owl:onProperty`. Then there are essentially two types: one defining cardinality and the other restricting presence of a relationship. The data representation can be seen in the two following examples. With the description of the used terms in the first one and the description of the modelled relationship in the second one.

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix z-sgov: <https://slovník.gov.cz/základní/pojem/> .
@prefix g-sgov-eu-regulation-2018-858:
    <https://slovník.gov.cz/generický/eu-regulation-2018-858/> .
@prefix eu-reg-858:
    <https://slovník.gov.cz/generický/eu-regulation-2018-858/pojem/> .

eu-reg-858:has-manufacturer a skos:Concept, z-sgov:typ-vztahu;
  skos:inScheme g-sgov-eu-regulation-2018-858:glosář;
  skos:prefLabel "has manufacturer"@en, "má výrobce"@cs;
  skos:scopeNote ""@cs .

eu-reg-858:vehicle a skos:Concept;
  skos:inScheme g-sgov-eu-regulation-2018-858:glosář;
  skos:prefLabel "Vehicle"@en, "Vozidlo"@cs;
  skos:scopeNote ""@cs .

eu-reg-858:manufacturer a skos:Concept;
  skos:inScheme g-sgov-eu-regulation-2018-858:glosář;
  skos:prefLabel "Manufacturer"@en, "Výrobce"@cs;
  skos:scopeNote ""@cs .
```

[9] `relator` is a label of entity `<https://slovník.gov.cz/základní/pojem/typ-vztahu>`..

[10] IRI: <https://slovník.gov.cz/základní/pojem/má-vztažený-prvek-1>

[11] IRI: <https://slovník.gov.cz/základní/pojem/má-vztažený-prvek-2>

[12] You can find more on W3C OWL.

[13] You can find the prefixes in the example on the next page.

```
@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 z-sgov: <https://slovník.gov.cz/základní/pojem/> .
@prefix eu-reg-858:
    <https://slovník.gov.cz/generický/eu-regulation-2018-858/pojem/> .

eu-reg-858:has-manufacturer rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-1 ;
        owl:someValuesFrom eu-reg-858:vehicle
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-1 ;
        owl:allValuesFrom eu-reg-858:vehicle
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-1 ;
        owl:onClass eu-reg-858:vehicle ;
        owl:minQualifiedCardinality "0"^^xsd:nonNegativeInteger
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-2 ;
        owl:someValuesFrom eu-reg-858:manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-2 ;
        owl:allValuesFrom eu-reg-858:manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-2 ;
        owl:onClass eu-reg-858:manufacturer ;
        owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger
    ], [
        a owl:Restriction ;
        owl:onProperty z-sgov:má-vztažený-prvek-2 ;
        owl:onClass eu-reg-858:manufacturer ;
        owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
    ] .

eu-reg-858:vehicle rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-1 ] ;
        owl:someValuesFrom eu-reg-858:has-manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-1 ] ;
        owl:allValuesFrom eu-reg-858:has-manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-1 ] ;
        owl:onClass eu-reg-858:has-manufacturer ;
        owl:minQualifiedCardinality "0"^^xsd:nonNegativeInteger
    ] .

eu-reg-858:manufacturer rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-2 ] ;
        owl:someValuesFrom eu-reg-858:has-manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-2 ] ;
        owl:allValuesFrom eu-reg-858:has-manufacturer
    ], [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-2 ] ;
        owl:onClass eu-reg-858:has-manufacturer ;
        owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger
    ], [
        a owl:Restriction ;
        owl:onProperty [ owl:inverseOf z-sgov:má-vztažený-prvek-2 ] ;
        owl:onClass eu-reg-858:has-manufacturer ;
        owl:maxQualifiedCardinality "1"^^xsd:nonNegativeInteger
    ] .
```

Change resolving in relationships

As we can see from the example, created blank nodes are predictable with very similar structures. For this reason, an algorithm[14] was implemented that starts by creating a `Map` containing blank node identifiers of all statements, that contain blank node identifier in the subject position, as keys mapping to a `Model` containing all statements of this blank node. This makes it a map for finding blank node subgraphs by identifiers. A second `Map` is created containing all non-blank node subjects, that point at blank node identifiers, as keys mapping to lists of these statements with common subject. In our example, this would be a `Map` with `eu-reg-858:has-manufacturer`, `eu-reg-858:vehicle` and `eu-reg-858:manufacturer` as keyset and lists containing the statements with predict `rdfs:subClassOf` and objects being the generated identifiers of the blank nodes.

These two `Maps` are created for both the content of vocabulary context and the content of canonical vocabulary. The `Maps` can be then used together to find subgraphs a subject is pointing to.

The data architecture of the Assembly line does not allow blank nodes that can not be found without traversing all statements, where the subject is not a blank node identifier. This means comparing subgraphs of a subject from canonical vocabulary with subgraphs of the same subject in vocabulary context, by utilizing the same method `isIsomorphicWith` of Jena API, will find isomorphic subgraphs that can be removed, leaving only the changed subgraphs. The only case where there is a blank node inside a blank node[15] is in restriction on one of the terms at the ends of a relationship. And there can be only one nested statement, so they can just be compared with each other. As OWL restrictions are the only blank nodes and their content can be modified[16] only in cardinality number or removed/added as a whole, a decision was made to store all statements of the changed subgraph as `Changes` with change type `Created` or `Removed` depending on if they are in vocabulary context or canonical vocabulary.

For blank node `Changes`, a label is also set, but it is inherited from the subject pointing at the blank node. To maintain re-buildability of stored `Changes` back to subgraphs, the identifier of the "parent" `Change` is saved as a subject of the blank node `Change`[17].

[14] This algorithm can be found in the `ChangeResolver` class in GitHub repository `mighantos/checkit-server` of CheckIt server.

[15] Shown in the example on lines 43, 47, 51 and more.

[16] By the Assembly line tools.

[17] The "parent" `Change` is the one that stored statement pointing to the blank node.

6.3 Composing relationships

With the blank node changes stored in the database, the main goal of CheckIt to simplify reviewing for Gestors is still not achieved. Because showing them a list of changed triples in blank nodes is not making their work that much simpler. For that, the `Changes` are composed back[18] into an entity `RelationshipDto` before the list of `Changes` is sent to the client. This entity contains names, IRIs, and cardinalities of the terms at the start and end of the relationship, the name and IRI of the relationship, and a "compose" list of stored `Changes` from which the relationship is composed of.

This list is created by fetching all `Changes` that have a blank node as an `object` property and creating a `Map` of subjects mapping to lists of `Changes` (very similarly as in relationships resolving). Then a subject in the `Map` that maps to blank nodes with `owl:onProperty` not pointing at a blank node (meaning the subject is a relationship) is found. By extracting the list of changes corresponding to the subject from the `Map` and finding the changes in blank nodes (they are pointing at), we get a list of all changes in blank nodes of the relationship term. This list can be transferred to the "compose" list of changes in the newly created `RelationshipDto` with the name and IRI of the relationship set from information within these changes.

As can be seen in the example, each restriction on relationship points to a term IRI on one of the ends with `owl:someValuesFrom`, `owl:allValuesFrom`, or `owl:onClass`[19]. Which end is described by the `owl:onProperty`[20]. These term IRIs can then be found in the `Map` mapping to their restriction `Changes`. The restrictions pointing back at the relationship[21] are transferred to the "compose" list in the `RelationshipDto` and the names, IRIs and cardinalities of ends filled. When all relationships were found, the algorithm returns the list of `RelationshipDto`.

This encapsulation of change into a larger whole also affected the calculation of statistics sent to the front end about the amount of `Changes` that need to be reviewed. To send correct values, this algorithm is run before `Changes` are saved to the database and from each `RelationshipDto`, one `Change` is selected from the "compose" list, which is marked as countable. This `Change` is also used for attaching comments to the relationship change.

[18] This algorithm can be found in the `ChangeDtoComposer` class in the GitHub repository of CheckIt server.

[19] For example on lines 11 and 24.

[20] As can be seen for example on lines 10 and 23

[21] For example on lines 44 and 59.

6.4 Updating Publication context

As shown in the figure 3.2 Publication process (TO BE), Editors can edit the project with existing Publication context a submit it for review again. This will result in stopping all reviews of Gestors and providing them with the updated list of changes. This list is created by comparing existing `Change`s in the Publication context with `Change`s resolved from the current form of the vocabulary context. If an exact match of an existing `Change` (meaning the subject, predicate and object of these `Change`s are identical), the existing `Change` is placed into the list of newly formed changes for the updated Publication context. The label of the existing `Change` must be replaced with the label of the current `Change`, as newly created terms could have been renamed. If no existing `Change` matches the current `Change`, it is placed in the list of newly formed changes, with all its reviews still present.

After matching all current `Change`s to existing ones, the rest of the unmatched existing `Change`s are checked if they were reviewed by any Gestor. If not, they are simply removed from the database, but if they were reviewed, their change type is set to `Rollbacked`, and they are added to the list of newly formed changes. This type of `Change` is then sent only to the Gestors who approved or rejected it before to inform them that this `Change` was scraped by Editors. It can not be reviewed but only acknowledged because the review state of the `Change` is used for resolving the affected Gestors that this `Change` should be sent to.

6.5 Maintenance improvements

Because the back-end implementation contains over a hundred Java classes with more than six and a half thousand lines of code, JavaDoc was written for every public method[22] to help understand the code better. CheckStyle plugin [44], widely used in the Assembly line tools, is also present in the application. This tool statically analysis the code and checks its formatting, naming conventions, correct use of JavaDoc and code complexity. Unit tests covering the service layer were also implemented to ensure the main functionality of the application. Both the static analysis and tests are run with GitHub actions on Pull Requests merging to the main branch and before creating a Docker image to prevent publishing an implementation with critical

[22]The generated JavaDoc can be found in doc/JavaDoc the GitHub repository of ChekcIt server.

bugs.

Chapter 7

Deployment

Because the development of the back end is intertwined with the development of the front end, and the back end is dependent on tools in the Assembly line, the need for deployment of this toolset, with the whole CheckIt application integrated into it, quickly arose. To satisfy this need, the deployment configuration of the Assembly line needed to be modified to include CheckIt.

7.1 Existing deployment

The current deployment is done with the use of Docker, where every tool runs as a service container. The working group behind the Assembly line created a GitHub repository with instructions for deployment and a Docker compose YAML file with a few simple helper scripts[1]. The compose file consists of the ten following services:

- GraphDB database – Java server with the central database for all tools, containing a copy of canonical vocabularies from SSP and their modified version in vocabulary contexts,
- SGoV server – Java back-end application managing projects and adding vocabularies into them,

[1]The instructions (only in the Czech language) and deployment files can be found on GitHub in datagov-cz/sgov-assembly-line repository .

- Mission control – front end for SGoV server, served by Nginx,
- TermIt server – Java back end creating, modifying and removing terms from vocabulary contexts,
- TermIt UI – front end for TermIt server, served by Nginx,
- OntoGrapher – thick front-end client for modelling relationship between terms, served by Nginx,
- Keycloak – a modified Java bundle of Keycloak, with a Java broker that copies user info into the central database,
- Postgress database – for Keycloak configuration and user data,
- Annotator – for analysing texts provided by TermIt and finding occurrence of existing terms in them,
- Nginx – the main reverse proxy to hide all services behind a path in URL requests.

In the deployment diagram below, we can see how the services communicate with each other. Connections of the Keycloak server are not included to make the diagram more transparent because it connects to every service with HTTPS communication, as it is the OAuth2 authorization server of the system. Another simplification is the absence of the main Nginx reverse proxy that is used to resolve URL requests coming from client browsers to servers.

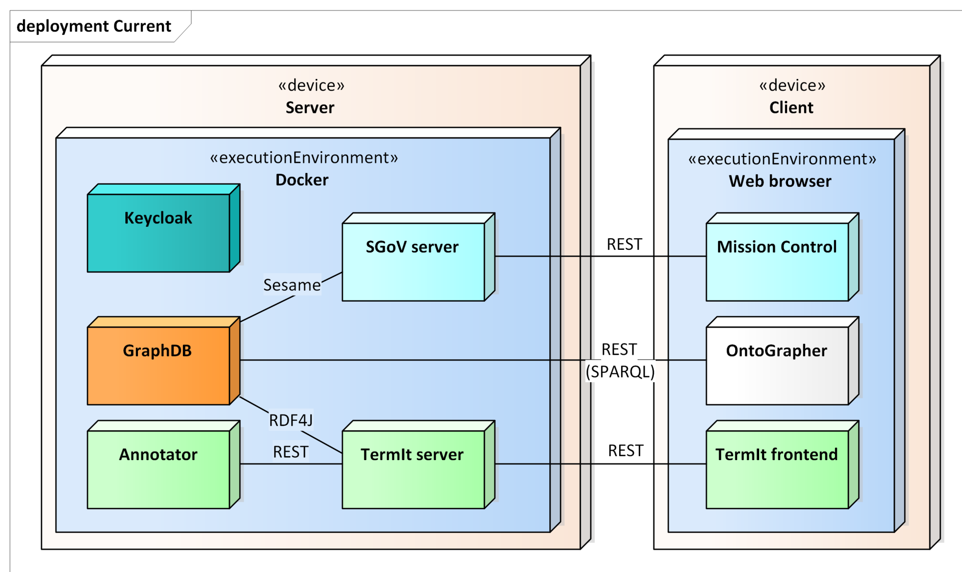

Figure 7.1: Deployment diagram (current)

7.2 New deployment

Integration of CheckIt into this ecosystem needs two services to be added, one for the CheckIt server and one for CheckIt UI. To create the server service, a Docker image of the server application is required.

First, the Gradle build configuration of the CheckIt server was enriched with a packaging task. This task bundles all compiled Java classes into a single JAR file. Java Runtime Environment (JRE) can execute this file, so a simple Dockerfile was created that builds an environment from OpenJDK 17 and copies the JAR file into it. This Dockerfile was then used to create the Docker image for the server. To add the Docker image to the Docker compose file of the Assembly line, a new GitHub branch called "checkit-deploy" was created. And the server service was added with the necessary environment variables into this branch. A ReadMe file was created in the GitHub repository of the CheckIt server with a startup guide and a description of possible environmental variables for the Assembly line maintainers to modify the deployment for their needs.

The second service was straightforward as Bc. Filip Kopecký created a Docker image of his front-end part and published it in GitHub Packages. So only a few environmental variables needed to be set after specifying the image location in the second service.

To allow access to the two new services by URL paths, they were added to the configuration of the main Nginx with respect to the existing path naming conventions. The ReadMe file, with instructions on deploying the Assembly line, also needed to be modified to reflect the addition of CheckIt by mentioning it in the list of tools present in the Assembly line and expanding the list of necessary variables to set. Some changes were also suggested to the deployment instructions in parts that were found unclear in an attempt to deploy it beforehand. The Keycloak's main configuration file for initialization also required editing to allow the new clients (services) to be able to authorize users and use their needed roles.

The CheckIt server requires a Keycloak user that will represent it and have permission to modify the roles of other users via Keycloak Admin API. This user must be created in the Keycloak after its initialization. This could not have been done by modifying the already existing main configuration file. So a launch script for the Keycloak Docker image was written that executes a CLI script for adding users (bundled with the used JBOSS Keycloak Docker

image). With this, the startup script[2] creates a user configuration with credentials from the environment variables of Docker compose, replaces its default roles with roles that allow API access and user role management, and then starts up Keycloak itself, which consumes both the user and main configuration files.

By doing all these modifications, the maintainers of the Assembly line can redeploy it following the slightly modified, yet to them well-known, deployment instructions. The redeployment will create the same stack with two new services added, as depicted by the diagram below.

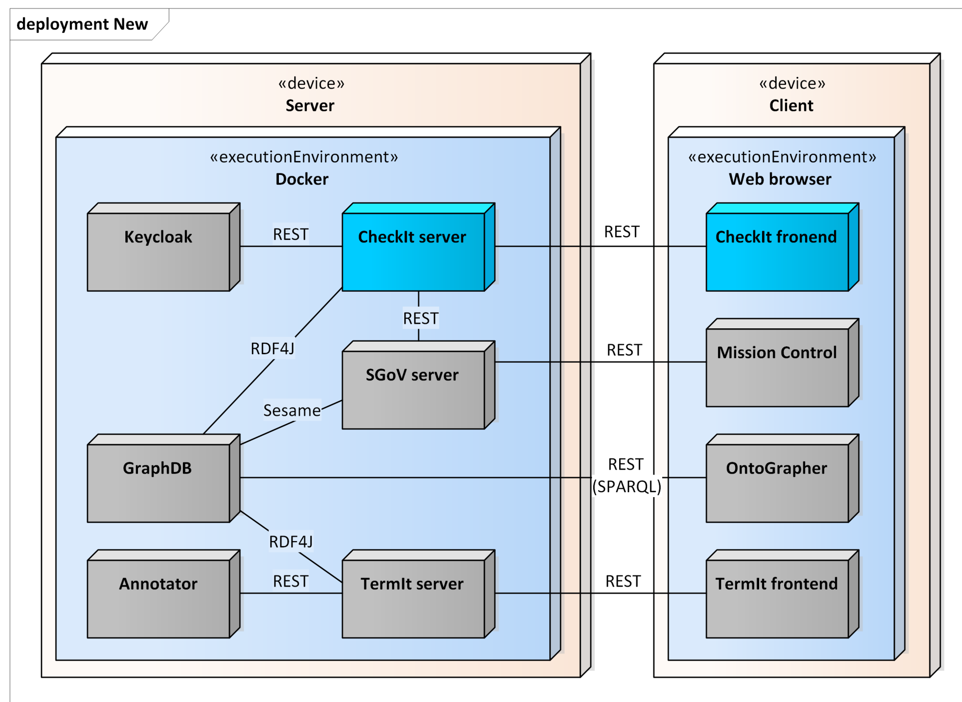

Figure 7.2: Deployment diagram (new)

7.3 Deployment for development

This new deployment configuration was first deployed on a home server only a few weeks into development. This deployment instance was updated frequently during the entire time of implementing the application. It was used for testing that both the developing front-end part and the developing back-end part could communicate with each other correctly and test both

[2]The script can also be found in al-auth-server/startup.sh on GitHub in the created branch in `datagov-cz/sgov-assembly-line` repository.

parts with real-world scenarios more easily. This cooperation with Bc. Filip Kopecký helped us both align our applications and find bugs in each other's work and fix them quickly during development.

7.4 Deployment for user testing

Besides deploying the new Assembly line stack to the home server, it was also deployed to the Demo instance (`https://onto.fel.cvut.cz/modelujeme`)[3] of the Assembly line on servers of Czech technical university. This deployment instance was redeployed with the new application for the KODI team to experience the new tool and for user testing. Although all testing scenarios were created by Bc. Filip Kopecký for testing of the UI, some of the answers provided relevant feedback even for the back end.

Bugs in the back end were also caught by the KODI team on this deployment. Thanks to that, they could have been fixed before letting users of the Assembly line access the new tool as well.

This deployment also tested the modified instructions and configuration files for deploying the Assembly line in a real environment. Although the instructions were understandable, the modified Docker compose file included a line with a boolean not surrounded by quotation marks. This was not an issue while deploying to the home server, but the older version of Docker compose used on the server of the Demo instance threw an error. But it was swiftly corrected and deployed successfully after that.

[3] Back-end API on `https://onto.fel.cvut.cz/modelujeme/sluzby/checkit-server/` and front-end part on `https://onto.fel.cvut.cz/modelujeme/v-nástroji/checkit`.

Chapter 8

Impediments and future work

Although the main features of the CheckIt application were implemented, some features planned in the design phase of the tool were not implemented because of unexpected delays with unforeseen obstacles.

8.1 Impediments along the way

The biggest impediments during the development of CheckIt were caused by the rapid evolution of the Assembly line combined with two recent consecutive personal changes in the leadership of the working group responsible for the Assembly line. This created gaps in the knowledge of the exact details of the data architecture. The provided incomplete or outdated information led to implementing some code in CheckIt back end and consequently debugging it to find why the code was not behaving as intended. Only to discover that the issue was caused by mistakes in supplied testing data. These mistakes were created by some data architecture changes that were not correctly updated in existing data or even data being created differently from the specification at the time of implementing CheckIt.

An example can be an inconsistent creation of restrictions on the terms at the ends of a relationship. Originally found and reported by Bc. Filip Kopecký as a bug in server implementation of updates of changes in an existing Publication context. Discovering this issue's true cause has cost over ten hours of debugging. Because all information received from the KODI

team (until then) was accurate, the issue was searched for in the algorithms of the implemented back end. And debugging changes in blank nodes of two versions of a vocabulary is no easy task, as mentioned in Chapter 3 - Assembly line processes.

It turned out that creating a relationship between two terms replaces any existing restrictions placed on these terms by other relationships beforehand. The issue was reported to the Assembly line team, and they managed to fix it in time to verify the algorithm in the back end.

8.2 Future work on CheckIt

These slowdowns meant that there was no time to implement change resolution in diagrams. To visualise these changes, an old and new diagram was planned to be shown to the user with highlighted changes and a list of concrete data changes like coordinates of elements and newly shown or hidden terms. Gestors then could approve or reject these changes as a whole. For this, the back end would need to compare the two diagrams (which are saved in RDF) and create the list of concrete data changes. This would then be sent to the front end for highlighting.

This also means that the current changes created by CheckIt back end can not be published to the SSP repository as they do not contain changes in diagrams. This is now mitigated by creating a Pull Request in parallel to the Publication context in CheckIt back end. When the Publication context is approved, CheckIt back end automatically approves and merges the Pull Request or closes it upon rejection. But everything is prepared for creating the new RDF dataset with the use of the canonical vocabularies and attachments and saved `Change` objects of a Publication context.

Besides this, a lot of improvements planned in development or found during user testing can be still implemented in future work. So let us look at them.

8.2.1 Further integration with the Assembly line

Currently, there is a new button in Mission control that creates a Publication context on a project. But more robust integration will be needed to prevent

the deletion of a project without deleting its corresponding Publication context, as CheckIt relies on the reference project's existence. Although deleting projects before successful publication is not a common practice, it is not an edge-case scenario either.

Integration with the rest of the tools in the Assembly line would also improve the user experience. For example, links from changes to the OntoGrapher or TermIt tool would allow Gestors to effortlessly explore other relationships in the vicinity of the changed one or examine existing properties of a changed term.

Or to update all Publication contexts automatically when a new version of SSP is propagated to the Assembly line's database. Instead of now, Editors needing to submit their projects again would also enhance user experience.

8.2.2 Correcting typos while reviewing

To speed up the workflow, CheckIt could allow Gestors to make some small changes to the data, such as correcting typos, during a review. This would eliminate the waiting time for Editors to make the slight changes, submit the project again, and then wait for the Gestor to approve these changes.

8.2.3 Improving notifications

While the CheckIt application was still in the design phase, the Assembly line team was planning to make a change that would record all users that contributed to a vocabulary in a project. This would allow notifications in CheckIt to be addressed to more relevant users, but sadly these changes were not realized as they were not a priority.

Notifications are currently present only in the application requiring users to log in to see them. Connection to Keycloak allows CheckIt to access the email addresses of users, which could be used to send emails about new unread notifications in the application.

8.2.4 Gestoring requests

Gestoring requests can be approved or rejected, but no message can be added to explain this decision. This was an oversight in the design phase discovered when evaluating the deployed application by users. By implementing such a feature, Notifications about these requests could have more meaningful content, and the communication between Admins and Editors would be easier and clearer.

A way for Editors to cancel a Gestoring request would also be nice, as they currently can not do so or even contact Admins in the application that it was not their intention to send that request.

8.2.5 Testing

More thorough testing is also needed, as the current automated tests cover only the main functionality of the back-end application. And as described at the start of this chapter, not all information about the data architecture is known, which can lead to unforeseen exceptions.

Chapter 9

Conclusion

The goal of this thesis was to familiarize ourselves with the Assembly line toolset, redesign the way how modified vocabularies are reviewed and published, and realize this new process.

We acquainted ourselves with RDF and RDF Schema, which are used for storing data in the Assembly line. The Vocabulary modelling process with an example of created data was presented, followed by the process of current Publication. In this process, a very difficult comparison of changes and complicated communication between Editors and Reviewers were identified as the main issues hampering the user-friendliness of the Assembly line.

A new Publication process was invented to make the Assembly line more comprehensive and usable for a wider range of institutions. For the new process to be realized, research on existing tools capable of comparing RDF datasets was conducted. Resulting in the usage of a modified Apache Jena API RDFDiff comparator as a helper library to implement in a new custom application. To define the data structure for information about changes needing to be saved, a Change description ontology was created. A back-end Spring boot Java application was implemented, utilizing this ontology to save changes found with the help of Jena API, to realize the new Publication process.

The created back-end application called CheckIt was deployed with the front-end part created by Bc. Filip Kopecký on a server provided by Czech technical university, where the Assembly line was already available as a Demo instance. This was done by modifying the existing deployment configurations

of the Assembly line to incorporate both the front-end part and the back-end part of CheckIt.

Capabilities of the created back-end of CheckIt include: finding changes in modified vocabularies, serving these changes to the front end, allowing users to communicate with in-application messages, requesting Gestoring roles on vocabularies by Editors, serving notifications about actions of other users, allowing Gestors to review the changes, and propagating approved projects to SSP.

This makes this master's thesis successful in reaching its goals, but a lot of features could still be added in future work to improve the final software.

Bibliography

[1] Domo, Inc. Data Never Sleeps 5.0. Visited on 17.05.2023 https://www.domo.com/learn/infographic/data-never-sleeps-5.

[2] Ministerstvo vnitra České republiky. Implementace strategií v oblasti otevřených dat veřejné správy ČR. Visited on 02.11.2022 https://esf2014.esfcr.cz/PublicPortal/Views/Projekty/Public/ProjektDetailPublicPage.aspx?action=get&datovySkladId=2D15BC0F-D539-4225-A7BC-DADF1860AA30.

[3] Ministerstvo vnitra České republiky. Implementace strategií v oblasti otevřených dat II. Visited on 02.11.2022 https://www.mvcr.cz/clanek/otevrena-data-ii.aspx.

[4] Ministerstvo vnitra České republiky. KODI - Rozvoj datových politik v oblasti zlepšování kvality a interoperability dat veřejné správy. Visited on 02.11.2022 https://www.mvcr.cz/clanek/kodi.aspx.

[5] Project KODI. Návrh a prototypování výrobní linky pro tvorbu a údržbu konceptuálních modelů agend, 2022. Visited on 15.01.2023 https://data.gov.cz/kodi/v%C3%BDstupy/C5V3.pdf.

[6] Ontotext. Graphdb - introduction. https://graphdb.ontotext.com/. Visited on 15.01.2023 https://graphdb.ontotext.com/.

[7] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 concepts and abstract syntax. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[8] M. Suignard and M. Duerst. IRI. Technical report, W3C, January 2005. Visited on 15.01.2023 https://datatracker.ietf.org/doc/html/rfc3987.

[9] Anne van Kesteren and Sam Ruby. URL. Technical report, W3C, December 2016. Visited on 15.01.2023 https://www.w3.org/TR/2016/NOTE-url-1-20161206/.

[10] Dan Brickley and Ramanathan Guha. RDF schema 1.1. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[11] Ora Lassila. Resource description framework (RDF) model and syntax specification. W3C recommendation, W3C, February 1999. Visited on 15.01.2023 https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[12] Eric Prud'hommeaux and Gavin Carothers. RDF 1.1 turtle. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-turtle-20140225/.

[13] Andy Seaborne and Gavin Carothers. RDF 1.1 n-triples. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[14] Gavin Carothers. RDF 1.1 n-quads. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-n-quads-20140225/.

[15] Gavin Carothers and Andy Seaborne. RDF 1.1 trig. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-trig-20140225/.

[16] Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. Technical report, W3C, February 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[17] Gregg Kellogg, Markus Lanthaler, and Manu Sporny. JSON-LD 1.0. Technical report, W3C, January 2014. Visited on 15.01.2023 https://www.w3.org/TR/2014/REC-json-ld-20140116/.

[18] Thomas Steiner, Arnaud Le Hors, and Ian Davis. RDF 1.1 JSON alternate serialization (RDF/json). Technical report, W3C, November 2013. Visited on 15.01.2023 https://www.w3.org/TR/2013/NOTE-rdf-json-20131107/.

[19] Mark Birbeck, Ben Adida, Ivan Herman, and Manu Sporny. RDFa 1.1 primer - third edition. Technical report, W3C, March 2015. Visited on 15.01.2023 https://www.w3.org/TR/2015/NOTE-rdfa-primer-20150317/.

[20] Tim Berners-Lee and D. Connolly. Notation3 (N3): A readable RDF syntax. Technical report, W3C, March 2011. Visited on 15.01.2023 http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/.

[21] Dan Brickley and Libby Miller. FOAF Vocabulary Specification. Namespace Document 2 Sept 2004, FOAF Project, 2004. Visited on 15.01.2023 http://xmlns.com/foaf/0.1/.

[22] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, September 2014. Visited on 15.01.2023 http://doi.acm.org/10.1145/2629489.

[23] Wikipedia. Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Wikipedia&oldid=1133819130, 2023. Visited on 15.01.2023 https://en.wikipedia.org/wiki/Wikipedia.

[24] GitHub, Inc. GitHub - Introduction, 2022. Visited on 15.01.2023 https://github.com/.

[25] Projekt KODI. Koncepce sémantického slovníku pojmů. Visited on 17.02.2023 https://opendata.gov.cz/_media/dokumenty:s%C3%A9mantick%C3%BD-slovn%C3%ADk-pojm%C5%AF:c1v2d1_n%C3%A1vrh_koncepce_s%C3%A9mantick%C3%A9ho_slovn%C3%ADku_pojm%C5%AF.pdf.

[26] GitHub, Inc. GitHub, code review, 2022. Visited on 15.01.2023 https://github.com/features/code-review.

[27] Yannis Tzitzikas, Christina Lantzaki, and Dimitris Zeginis. Blank node matching and rdf/s comparison functions. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2012*, pages 591–607, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[28] RDFLib Team. RDFLib 6.2.0, 2022. Visited on 15.01.2023 https://rdflib.readthedocs.io/en/stable/.

[29] Stanford Center for Biomedical Informatics Research. Protégé, 2020. Visited on 15.01.2023 https://protege.stanford.edu/.

[30] Deborah McGuinness and Frank van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004. Visited on 15.01.2023 https://www.w3.org/TR/2004/REC-owl-features-20040210/.

[31] Oracle Corporation. Java. https://www.java.com/, 1995. Visited on 15.01.2023 https://www.java.com/.

[32] Petr Křemen, Marek Šmid, and Zdenek Kouba. OWLDiff: A Practical Tool for Comparison and Merge of OWL Ontologies. In *2011 22nd International Workshop on Database and Expert Systems Applications*, pages 229–233, 2011.

[33] NeOn Technologies Foundation. Neon toolkit, 2014. Visited on 15.01.2023 `http://neon-toolkit.org/`.

[34] Dave Longley. Rdf dataset canonicalization. W3C recommendation, W3C, October 2022. Visited on 15.01.2023 `https://www.w3.org/community/reports/credentials/CG-FINAL-rdf-dataset-canonicalization-20221009/`.

[35] Peter Ansell, Stian Soiland-Reyes, and Tristan King. JSONLD-JAVA, 2021. Visited on 15.01.2023 `https://github.com/jsonld-java/jsonld-java`.

[36] Gregg Kellogg, Markus Lanthaler, and Manu Sporny. JSON-LD 1.0 processing algorithms and API. W3C recommendation, W3C, January 2014. `https://www.w3.org/TR/2014/REC-json-ld-api-20140116/`.

[37] The Apache Software Foundation. Apache Jena, 2023. Visited on 15.01.2023 `https://jena.apache.org/`.

[38] Red Hat, Inc. What is a REST API? Visited on 13.02.2023 `https://www.redhat.com/en/topics/api/what-is-a-rest-api`.

[39] JRebel. 2021 Java Developer Productivity Report. Visited on 13.02.2023 `https://www.jrebel.com/blog/java-build-tools-comparison`.

[40] Gradle Inc. Gradle vs Maven: Performance Comparison. Visited on 13.02.2023 `https://gradle.org/gradle-vs-maven-performance/#`.

[41] Pivotal Software. Spring boot. Visited on 15.01.2023 `https://spring.io/why-spring`.

[42] Ing. Martin Ledvinka, Ph.D. JOPA - Java OWL Persistence API. Visited on 15.01.2023 `https://github.com/kbss-cvut/jopa/blob/master/README.md`.

[43] Eclipse Foundation. Jakarte EE. Visited on 14.02.2023 `https://jakarta.ee/`.

[44] Oliver Burn. CheckStyle. Visited on 19.05.2023 `https://checkstyle.org/`.

Appendix A

Content of the electronic attachment

```
application.zip
├── checkit-server/ .................... Created back-end application
│   ├── .github/...................... Configurations of GitHub actions
│   ├── .idea/
│   ├── config/............... Configs for CheckStyle and IntelliJ IDEA
│   ├── doc/
│   │   ├── JavaDoc/............................... Generated JavaDoc
│   │   ├── Test-datasets/ ........... Testing data used in tool research
│   │   └── Assembly-line.postman-collection.json .. Postman export
│   ├── gradle/
│   ├── src/ ................................. Code of the application
│   ├── .gitignore
│   ├── build.gradle ................................. Gradle config
│   ├── Dockerfile
│   ├── gradle.properties
│   ├── gradlew
│   ├── gradlew.bat
│   ├── README.md ................................. Installation guide
│   └── settings.gradle
├── img/ .................................. Images used in this thesis
└── popis-zmen-ontology/ ......................... Created ontology
    ├── d-sgov-popis-zmen-glosář.ttl
    ├── d-sgov-popis-zmen-model.ttl
    ├── d-sgov-popis-zmen-slovník.ttl
    ├── Ontology_Relationship_Diagram_CZ.png
    ├── Ontology_Relationship_Diagram_EN.png
    └── README.md
```