ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Binder** Jméno: **Alice** Osobní číslo: **469856**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Podpora tvorby ontologických konceptuálních modelů pomocí návrhových vzorů

Název diplomové práce anglicky:

Support for creating ontological conceptual models using design patterns

Pokyny pro vypracování:

OntoUML je formální jazyk pro tvorbu konceptuálních modelů založených na základní ontologii UFO. Jeho použití vyžaduje od modeláře značné znalosti z ontologického inženýrství. Práce si klade za cíl usnadnit tvorbu OntoUML modelů pomocí návrhových vzorů a formálních zkratek, které budou specifické pro vybranou aplikační doménu a budou využívat ontologii UFO.
Pokyny:
- seznamte se s problematikou tvorby doménových ontologií postavených na ontologii UFO, s důrazem na jazyk OntoUML
- seznamte se s existujícícmi ontologickými návrhovými vzory
- vytvořte formální jazyk pro reprezentaci a tvorbu návrhových vzorů nad OntoUML 2.0 modely
- navrhněte a implementujte algoritmus pro detekci návrhových vzorů na základě existujících ontologických modelů
- vyhodnoťte efektivitu a přesnost algoritmu na vybrané sadě ontologických modelů, případně syntetických datech

Seznam doporučené literatury:

[1] Martin G. Skjæveland, Daniel P. Lupp, Leif Harald Karlsen, and Henrik Forssell. Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates In: Vrandečić D. et al. (eds) The Semantic Web—ISWC 2018. ISWC 2018. LNCS vol 11136. Springer. 2018.
[2] Giancarlo Guizzardi, Gerd Wagner. Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages. In: Theory and Applications of Ontology: Computer Applications. 2010

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Petr Křemen, Ph.D. skupina znalostních softwarových systémů FEL

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

Datum zadání diplomové práce: **21.02.2021** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **19.02.2023**

Ing. Petr Křemen, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

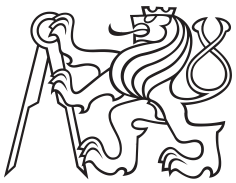
III. PŘEVZETÍ ZADÁNÍ

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

Diploma thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Support for creating ontological conceptual models using design patterns

Alice Binder

Supervisor: Ing. Petr Křemen, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2022

Acknowledgements

I would like to thank Dr. Křemen for his continued support throughout the many years of our collaboration on several projects, including this thesis. Another mention goes to my friends and family, whose assistance and kindness allowed me to continue despite many personal challenges.

Declaration

I declare that I have made the submitted work independently and that I have listed all the sources used in line with the Methodological Guideline on Compliance with Ethical Principles of preparation of academic final theses.

Prague, 20. May 2022

Abstract

Implementations of Semantic Web standards have been the focus of many enterprises. However, the process of data creation and manipulation can be quite time-consuming. It is often the case that a dataset has rigidly defined classes with a vast amount of instantiations of those classes. Ontology design patterns provide an efficiency boost to creating these instantiations - they formalize an often-repeatable set of data and allow the user to create instances of those patterns easily instead of manually copying the data each time. In this thesis, the potential applications of ontology design patterns is discussed, research of current formal ontology design pattern description languages is performed, and the resulting language is then tested on OntoUML model examples. Finally, a service functioning as an intermediary between datasets and patterns is designed and implemented. Interactions with the endpoints are implemented in the conceptual modeling tool Ontographer and evaluated on a repository of vocabularies of legal terms from pieces of Czech legislation. The evaluation showed that while shortcomings are present, the overall project has enough potential to warrant further study.

Keywords: ontology design patterns, semantic web, thesaurus, skos, owl, ontouml, conceptual modeling, web application, ontology, knowledge management system, software engineering, ontology engineering

Supervisor: Ing. Petr Křemen, Ph.D.
Karlovo náměstí 13, Praha 2

Abstrakt

Implementace standardů Semantic Web je středem zájmu mnoha organizací. Vytváření dat a manipulace s nimi však může být značně časově náročné. Často se stává, že datová sada má pevně definované třídy s velkým množstvím instancí těchto tříd. Návrhové vzory ontologií poskytují zvýšení efektivity při vytváření těchto instancí – formalizují často opakovatelnou sadu dat a umožňují uživateli snadno vytvářet instance těchto vzorů namísto ručního kopírování dat. V této práci jsou diskutovány možné aplikace návrhových vzorů ontologií a je proveden výzkum současných jazyků pro popis formálních návrhových vzorů ontologií. Výsledný jazyk je následně testován na příkladech OntoUML modelů. Poté je navržena a implementována služba fungující jako prostředník mezi datovými sadami a návrhovými vzory. Interakce s funkcemi služby jsou implementovány v nástroji pro koncepční modelování Ontographer a vyhodnocovány na úložišti slovníků právních pojmů z české legislativy. Hodnocení ukázalo, že ačkoliv existují nedostatky, tak celkový projekt má dostatečný potenciál k tomu, aby si zasloužil další výzkum.

Klíčová slova: ontologické návrhové vzory, semantický web, tezaurus, skos, owl, ontouml, konceptuální modelování, webová aplikace, ontologie, knowledge management system, softwarové inženýrství, inženýrství ontologií

Překlad názvu: Podpora tvorby ontologických konceptuálních modelů pomocí návrhových vzorů

Contents

Figures

Chapter 1

Introduction

The concept of ontologies, linked open data and the technologies within the Semantic Web paradigm have been the focus of many projects spearheaded by private firms[22], municipalities[21], countries[11] and even supranational entities[1] all over the world. The goals of many of these projects are to increase transparency and legibility of public data primarily for governmental organizations, and also to provide context, interoperability and formal definitions of data models, which can be useful for private enterprises as well.

Within the Semantic Web, data is usually described in the RDF[19] format, using certain kinds of publicly available or private ontologies or thesauri which offer standardized shorthands for descriptions of several "things" (terms, persons, products), relationships between those things (employments, ownerships, responsibilities), and description of those things (names, editorial notes, provenance data). However, these shorthands are oftentimes (in the cases of SKOS[23] and OWL[16], for example) very simple individual relations or classifications; in order to model a single class of "thing" and its relations formally and completely, one probably needs tens if not hundreds of lines of RDF data. A portion of these shorthands is then replicated for each member of the class, as a member usually has e.g. a different name, address, ID number, responsible persons, and so on.

Ontology Design Patterns (ODPs) offer a hypothetical efficiency improvement in this process. Instead of typing out each description of a class member by individual lines, a pattern would allow a data entry algorithm or user to "call" it with simple parameters and the instantiation of the pattern would take care of the rest. Chapter 3 discusses in detail the benefits of this approach

on a real example from the Czech government's linked open data project. This project is discussed in greater detail in the same Chapter.

This enhancement, however, implies the existence of a language able to describe such patterns and their instantiations. There are many options currently available with varying degrees of generalizability, legibility, and ease of use. Chapter 3 provides an overview of the current state of affairs and details research determining which language is the most applicable to the aforementioned example.

This research picks the OTTR (Reasonable Ontology Templates[1]) language, which is described in Chapter 4. The language and its supporting structure is analyzed and tested on OntoUML, an UFO-based extension of UML, which works as a base which is then expanded into the Czech project's base vocabulary.

There is a significant barrier to overcome in order to take advantage of OTTR on real-life RDF datasets, which is that there is the need for an interpreter between OTTR and the actual datasets. In the examples researched in Chapter 3, the patterns do not generally interact with the actual data directly, but through an intermediary *expander*[2], which takes a pattern and instantiates it with given parameters. This interactions is roughly analogous to creating objects of a certain class in an object-oriented programming paradigm; data describing an object (e.g. a Ford car with a specific VIN) is created from a class' constructor (e.g. a Car pattern instantiated with parameters including the VIN number and make).

OTTR already has an implementation of an expander, but this is not the complete package for datasets, as the maintainers also need to provide a way to create the patterns themselves (ideally through a frontend or an abstraction, not manually) and then similarly to call the expander with a pattern and filled out parameters. This paper, specifically in Chapter 5 attempts to implement this package as a server interacting with a database solely for patterns and instantiations and providing endpoints providing useful functionality.

The endpoints are tested with Ontographer, a web-based tool for conceptual

[1] In this paper, *templates* and *patterns* are oftentimes interchangeable concepts, but note that *templates* are generally meant as a formal description of a pattern that can then be easily recreated. *Pattern* is a much more general term describing recognizable and predictable phenomena.

[2] To *expand* a pattern is to instantiate it with given parameters and return the actual RDF data.

modeling used within the project to see if the solution is also useful for visual modeling tools concerning multiple terms and relationships between them.

The entire package (OTTR, the server, the Ontographer enhancement) is evaluated based on its performance on a repository semantic vocabularies of (mostly) legal terms in Chapter 7, which prompts a discussion of the results of the thesis' effort in Chapter 8, as well as potential avenues for future development.

Chapter 2

The Semantic Web

The traditional way of recording data in enterprise settings has been the Relational Database Management System. In it, data are primarily defined through basic types (i.e. strings, various number types, etc.) and are collected through variously connected tables containing the individual entries according to the design specifications.

This approach is very practical for conventional use-cases, where the semantic inferences from those tables are easily legible (e.g. tables of employees, customer orders or products), the connections between tables are evident and unchanging, or the database is meant for mostly private applications - for example simply serving as a backend to a website or an enterprise solution.

Needless to say, there are limitations whenever there is a requirement for representing deeper semantic relations between individual "things". For example, how would one ask an RDBMS "What are the most common types of road accidents during thunderstorms in Prague"? Of course, if the system has all the information about weather patterns, locations, and road accidents, the task is easy. However, the majority of relational databases are context and architecture specific, so interoperability of databases within the same organization, let alone different ones, is mostly out of the question. In addition, a relational database does not really know what an "accident" or a "thunderstorm" is, as it simply stores data rows.

This interferes also with the scenario where the data is supposed to be public; as relational databases require either vast software abstraction or detailed documentation to be legible to those not in the know of the given

database design. There are organizations which forgo any type of conventional database whatsoever - rather preferring to store information in spreadsheets, word processors, or PDF files. While technically "open" data if published onto the web for everyone, the de facto usability of the data when attempting to correlate information from multiple sources is very poor and often a tedious manual task.

Enter the Semantic Web. Rather than just a new way of storing and querying information, the authors at the World Wide Web Consortium attempt to change the way in which data on the web is thought of [20]. Instead of envisioning computer-stored information as a series of rows and columns *representing* "things", the information here *is* actually the "thing". This reversal of thought is crucial to understanding what the data actually is, both for users and machines. Objects can have types, inferences, semantic connections between other objects, and more. The paradigm is also interested in making *linked open data* - data which is publicly available and linked between different data sources and providers. A dataset of road accidents and a dataset of thunderstorms, then, can be accessed with one simple query, instead of perusing both individually.

In addition, the Semantic Web data formats and languages (RDF, SKOS, OWL) enable each object to be formally defined as a member of a certain type, which could automatically (inherently) give it other properties. A woman automatically becomes a wife when married without interventions from any software solution. It is possible to check if all buildings over 500 meters squared in area have at least 2 emergency exits.

2.1 Definitions of legal terms

These formal definitions and strict rule enforcement is useful in the case of various governmental projects, namely one lead by the Czech government's Ministry of the Interior[1]. The basic premise is that the government aims to increase transparency and data availability for the purposes of ease of communication between the government's own entities as well as public and private ones[12]. This is done through

- legislation enforcing certain data standards,

[1]This project is known as *KODI* (Full Czech name: *Rozvoj datových politik v oblasti zlepšování kvality a interoperability dat veřejné správy*) and will be referred to as such.

- meeting and convincing organizations to cooperate with submitting their data to the new standard,
- rigidly defining terms and types of things that the various data objects will be classified under,
- developing a software suite to make the transition and development process more efficient,
- delivering solutions so that data holders/providers can convert the data.

A particular sub-problem in this arena is the formal definitions of terms. As the project deals mainly with legislation, the meaning of terms within these can be (intentionally) vague or overlap with terms with the same name in different pieces of legislation. The need for a data structure which can properly represent definitions and meanings of terms within vocabularies assigned to pieces of legislation.

For this purpose, a Semantic Vocabulary of Terms (Czech: Sémantický slovník pojmů, SSP[2]) has been devised as a repository of glossaries, models and vocabulary data describing legal terms and associations. The terms are all defined with a base ontology, *Zs-Gov* ("Basic vocabulary"), partly created from the very popular standard OntoUML. The basics of the term representation are described in [2] and the detailed rationale and development of SSP is chronicled in [13].

2.2 OntoUML

OntoUML is a UML extension first envisioned in Giancarlo Guizzardi's Ph.D. thesis "Ontological foundations for structural conceptual models"[6]. It is based on the UFO (Unified Foundational Ontology), that he used to create an extension of UML (Unified Modeling Language). This extension enables the use of UML standards to create conceptual models of ontologies and, most importantly, visualize them and the relationships between them. Since its inception, OntoUML has been adopted by many public and private organizations all around the world.

[2]Can be seen at `https://github.com/opendata-mvcr/ssp`.

Chapter 3

Ontology Design Patterns

Ontology Design Patterns (ODPs) play a crucial role in ontological engineering and they are a field of study within ontological modeling that is growing in interest in the last few years[27]. ODPs help engineers create models more quickly, avoid certain frequently made inefficiencies and better prepare their models for any future extension or revision. This field is, however, only a few years old and therefore support of ODP usage with methodological approaches, classification, categorization, standardized distribution or implemented tools is still developing. Some particular advantages of an implemented toolset based on commonized standards for conceptual modeling would be in:

- providing a form of describing ODPs that can be used by other tools (i.e. standardized, machine-readable, and ideally in a commonly used format),
- modularizing patterns, allowing them to be separated into reusable sub-patterns which other patterns may be composed from,
- introducing an annotation framework for categorizing and classifying patterns. Such a framework would allow recording e.g. provenance data.

To accomplish this, we first need a solid pattern representation language, which would be expressive enough to capture key features of conceptual ontological models, yet able to efficiently fulfill the requirements above - along with another features such a language could enable:

- identification of patterns in models, even when they are not officially recognized by the designer;
- comparison and merging of patterns, whenever appropriate;
- modeling guidance via suggestion of more efficient patterns upon detection of certain anti-patterns (patterns that are comparably inefficient or unfavorable).

There have been suggestions as to what elements this language should have or what efforts should be made to make it viable, some of which are explored in Section 3.1. As of the time of writing, there are no available and commonly recognized implementations of such a language. In this work, a proposal is made for a language that aims to fulfill the aforementioned requirements. After research of current developments in the field and analyzing the various approaches within in Section 3.2, the Reasonable Ontology Template (OTTR) language is chosen. Emphasis will be put on evaluating whether this language is compatible with the Unified Foundational Ontology (UFO), as it is one of the most well-known high-level ontologies [8]. This will require implementing or at least allowing the implementation of constructs constituting OntoUML, an UFO-based extension of UML [6]. This evaluation of correctness of representation of OntoUML models and against the aforementioned requirements is done on several conceptual model examples in Chapter 4.

3.1 Motivation

In this section, the problem for which a solution is proposed in this work is introduced and discussed. Use-cases of ODP usage are described in which an ODP representation language could be used to either enable them or make performing them more efficient. To demonstrate these points more clearly, consider an example of an ODP that represents the relation between an identity card provider, identity card, and identity holder (e.g. Drivers' Registry, Driver's licence, Driver).

The Identity example (represented in Figure 3.1 and instantiated in Figure 3.2) could be a part of a larger model designating a relationship of persons between various governmental institutions - therefore, it could be instantiated as relationships between drivers and drivers' licences, citizens and identity cards, students and students' cards, etc.

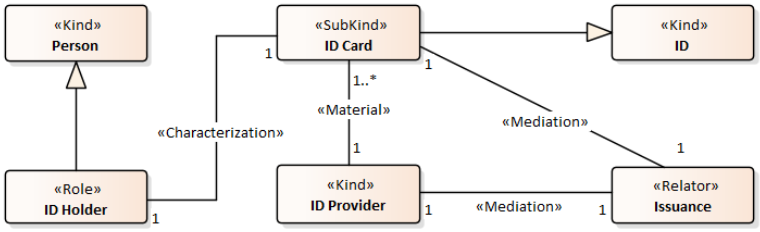


Figure 3.1: Conceptual diagram of the Identity ODP.

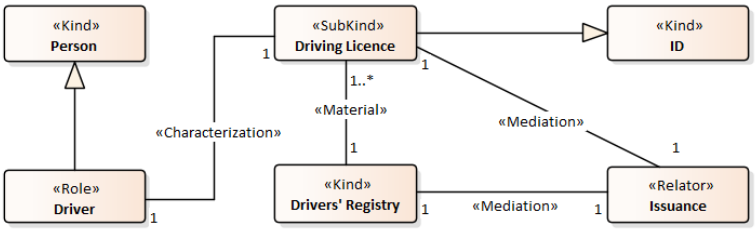


Figure 3.2: Conceptual diagram of a possible instantiation of the Identity ODP.

3.1.1 Standardization

A key aspect of design patterns in general is the ability to share best practices and efficient solutions for recurring problems with other engineers. These solutions, if they were expressed with a recognized and sufficient enough language, could be quickly implemented into existing ontologies, perhaps with developed supporting tools to help facilitate such a utilization. This collaboration can be either with the larger community or within an organization, where teams can create a library of ODPs commonly shared within the organization's ontologies tailored to enforce their ontology design philosophy.

3.1.2 Metadata

ODPs have a well established taxonomy, separating them into various categories by purpose or domain [5]. In addition, the creation of an ODP implies certain provenance and other metadata, for example

- whether the ODP belongs to a pattern library and/or serves a subdomain;
- if the ODP is an implementation of another pattern;
- description of the pattern and its recommended usage;

- information about anti-patterns.

This metadata can be useful to engineers and can help organize the ODPs in such a way that allows for easier sorting, searching, and sharing of ODPs and ODP libraries between users and team members.

Depending on its usage and implementation, the Identity ODP could be classified in many ways - e.g. as a Content/Reasoning ODP, as a Government/Personhood ODP, and so on. If the ODP is sufficiently annotated with examples and explanations, a user who perhaps is not as knowledgeable on ODP usage (or ontological modeling generally) can be aided in modeling, especially if the examples can be identified based on the models in which the ODP is used.

3.1.3 Modularization

Being able to formulate ODPs either individually, or together with similar ODPs into a shareable package is crucial for facilitating ODP usage and sharing; users, whether they are from the wider community or within a small team, can collaborate on ODP engineering. Having them packaged in a way that is accessible can help with implementing them into existing ontologies as seamlessly as possible, encouraging ODP usage and speeding up ontology engineering in recurring cases. Another feature of modularization is the possibility of using ODPs as part of other ODPs - along with standardization, this could further propel general ODP usage.

3.1.4 Algorithm support

As mentioned in Section 3.1, a language that satisfactorily and predictably describes ODPs and their metadata can be leveraged into the creation of certain algorithms that aid ontology engineers in modeling or ontology design. One such case is the ability to detect patterns, perhaps even anti-patterns, in models in which the given patterns are not explicitly identified; the user can then choose to re-engineer their model. Another case is ODP creation; a user can select a particular section of their model and modularize it for use in other models and/or share it with a wider selection of users.

For example, an ontology engineer can choose to refactor their model(s)

easily, if a function in modeling software could separate model parts of their choice into ODPs that they can then send to other models or other users. Perhaps the Identity ODP could be a part of a larger model, which concerns many aspects of personhood that a government is interested in enumerating and/or processing. Separating the model into ODPs can also help separate the models themselves in order to better fulfill the DRY (Don't Repeat Yourself) principle, among others.

3.2 Current research

In this section, existing proposals of ODP languages are analyzed for their potential as well as their shortcomings. Such analysis provides an understanding of the requirements the language must have, in addition to showcasing approaches to consider.

The individual proposals were chosen using a research methodology described in a 2008 paper by Petersen et al[17]. In it, the authors describe a process to acquire a representative article sample, the details of which are described in Appendix A.

3.2.1 Existing ODP languages

Of particular note for this paper were the ODP language proposals and presentations - the research captured two examples of each. These examples are detailed below.

The OPLa ontology

In the "Towards a simple but useful ontology design pattern representation language" paper from 2017, Hitzler et al. present an ontology for annotating ODPs and all their potential elements, which they call OPLa [10]. The ontology consists of two main classes: OntologicalCollection, describing types of patterns, and OntologicalEntity, denoting types of constructs that form a pattern. A classification of OntologicalEntity classes of note is into external or internal entities – where internal entities only connect to other individuals

within the ODP, external entities are only defined as a "connection point" between the ODP and other classes already present in the model. For example, a pattern that describes a complex relationship between Insurer, Insured, and InsuredItem classes would see these classes described as external entities in OPLa.

The advantage of an ontology approach is the simplicity of the design and its assured compatibility with other standards built on the Semantic Web paradigm. Extraction of ODPs and their individual parts from a model as well as their categorization is relatively straightforward. In addition, the inclusion of the ontology into existing data infrastructures can be done algorithmically, avoiding a manual and time-consuming task. Nevertheless, its simplicity also limits the use-cases the solution can satisfy. In the proposal, the ontology mainly serves an annotative function, as mentioned above; this means that problems such as pattern identification or reuse are not addressed. While the implementation of such functions is not prohibited outright by this approach, the amount of effort required for the development of an extension must be taken into account – in this case, it would be entirely from scratch. The authors themselves acknowledge these shortcomings – the focus of the paper is not a description of a feature-complete language proposal, but rather a foundation other researchers and engineers can take advantage of. An example of OPLa on the Identity ODP is shown below.

```
ex:Identity a opla:Pattern.
ex:IDHolder opla:isNativeTo ex:Identity;
opla:ofExternalType ex:externalClass.
ex:IDCard opla:isNativeTo ex:Identity;
opla:ofExternalType ex:externalClass.
ex:IDProvider opla:isNativeTo ex:Identity;
opla:ofExternalType ex:externalClass.
ex:IDIssuance opla:isNativeTo ex:Identity.
```

In a followup paper, OPLa is reorganized and expanded with two additional namespaces: OPLa-cp, a set of annotations covering ODP usage consequences with respect to scenarios, requirements, or competency questions addressed by the ODP, in addition to providing an annotation for unit tests, and OPLa-sd, which describes ODP entity positions (i.e. X, Y coordinates for a schema diagram visualization)[9].

Reasonable Ontology Templates

Skjæveland et al. describe Reasonable Ontology Templates (OTTR; spelled 'otter') within many papers, tutorials, and documentation articles available at `https://www.ottr.xyz/`. The central elements of the language are template definitions and instances; definitions describe a template, which has certain parameters and can have nested templates within its definition. These definitions are then instantiated with template instances.

The ability to use typed parameters in template definitions and nested templates results in a particularly flexible language; one that can easily be used for refactoring existing templates into reusable modules (as demonstrated in a real example in a paper by Skjæveland [22]). The templates and instances can be inputted and outputted in many forms as defined by related OTTR standards; for example, wOTTR can be used to publish the templates in an OWL-compliant graph; tabOTTR provides a way to import templates from tabular formats, and bOTTR allows engineers to map external sources to create templates via queries (therefore supporting an ETL approach). A clear disadvantage is the need for an interpreter for the language - the team developing OTTR has an implementation of the language, Lutra, available at the website mentioned above. The implementation can read and write templates between different formats, such as Turtle or tabular formats, and expand instances into RDF graphs.

Consider an example implementation of the Identity ODP. Note that as in the OPLa example, the way that a person can express the same ODP varies; an engineer might want to implement certain additional rules/inferences, or considers different things to be a "parameter" or "external entity" - this, as with any ODP, depends on the use-case. Other, more complex interpretations of this specific ODP will be explored in further sections.

```
ex:Identity[! ex:IDHolder ?holder, ! ex:IDCard ?id,
 ! ex:IDProvider ?provider] :: {
ottr:Triple(?holder, rdfs:subClassOf, ex:Person),
ottr:Triple(?id, rdfs:subClassOf, ex:ID),
ex:Connection(?holder, gufo:characterization, ?id, 1, 1, 1, 1),
ex:Connection(?id, gufo:material, ?provider, 1, ottr:none, 1, 1),
ex:Connection(?provider, gufo:mediation, ex:issuance, 1, 1, 1, 1),
ex:Connection(?id, gufo:mediation, ex:issuance, 1, 1, 1, 1)
}.

# Instantiation
ex:Identity(ex:Alice, ex:DriversLicence, ex:DriversRegistry).
```

The Distributed Ontology, Modeling and Specification Language

The Distributed Ontology, Modelling and Specification Language (DOL) is not a language for describing ontologies or ODPs per se, but rather a meta-language for connecting ontologies that have already been written in other languages - for example OWL, Common Logic or first-order logic - without the need to rewrite such ontologies[15]. As such, it provides syntax to perform various actions to one or more ontologies, such as:

- unification, combination, extension, or selection of a part of an ontology;
- translation or conversion of one ontology from one language to another;

among other actions. The paper presenting DOL also introduces Ontohub, a repository for organizing, sharing and collaborating on ontologies written in DOL or an ontology language supported by DOL. DOL has been accepted as an official OMG standard in 2018[1].

A paper by Krieg-Brückner and Mossakowski extends this language with generics, enabling the creation of general ODPs within DOL[14]. An example of the Identity pattern expressed in GDOL, is below:

```
pattern Identity
    [IDHolder: holder]
    [IDCard: card]
    [IDProvider: provider] =
holder SubClassOf: Person
card SubClassOf: ID
Connection
    [holder][Characterization][card][1][1][1][1]
Connection
    [card][Material][provider][1][1][1][1]
Connection
    [provider][Mediation][issuance][1][][1][1]
Connection
    [card][Mediation][issuance][1][1][1][1]

# Instantiation
ontology IdentityInstantiation =
    Identity
        [Alice][DriversLicence][DriversRegistry]
```

Notice the parameters in brackets and their subsequent use in the pattern definition itself; the generics introduced in this paper is what allows this syntax. The numbers in the Connection instantiations refer to minimum and maximum cardinalities for the subject and object, respectively. The empty argument results in no cardinality being applied - in this example, the `Connection [provider] [Mediation] [issuance] [1] [] [1] [1]` instantiation creates a `provider [1..*] -mediation-> [1] issuance` OntoUML relation.

The aforementioned authors and Codescu continued with GODPs in DOL with a proposal for extensions of generic DOL[3]. These mainly serve the purpose of easier ODP development, such as optional pattern parameters, list parameters, recursion, or local sub-patterns.

A general ontology pattern language formally defined with graph grammar

Giancarlo Guizzardi, the creator of OntoUML, wrote a paper with Eduardo Zambon with the intention to redefine OntoUML purely with graph transformation rules[26]. This has the advantages of (i) being independent from UML and therefore able to be implemented in other modeling languages, and (ii) a formal definition for OntoUML that explicitly notates the rules and ontological commitments of OntoUML constructs.

A graph transformation rule, here, is an action that takes an input graph and outputs a (presumably modified) different graph. In this way, modeling OntoUML models isn't done with using constructs (classes and relationships) individually, but with a set of patterns which ensure that the resulting model is always valid. For instance, instead of creating a SubKind class and subsequently connecting it via generalization to a Kind (or another Rigid Sortal), a "SubKind pattern" is used, creating both the SubKind class and the relationship (and the Rigid Sortal, if needed) at the same time.

Figure 3.3 describes an example use of the patterns. Note that since the patterns from the paper do not have a defined serialization, they provide a potential ODP use methodology rather than a formal language definition.

With these sets of patterns as "constructs", the user need not worry about validation, since constructing models solely through these patterns guarantees

[1]Note that in the original paper, some of these patterns are not defined despite being valid in OntoUML - this is because they were not relevant according to the paper itself.

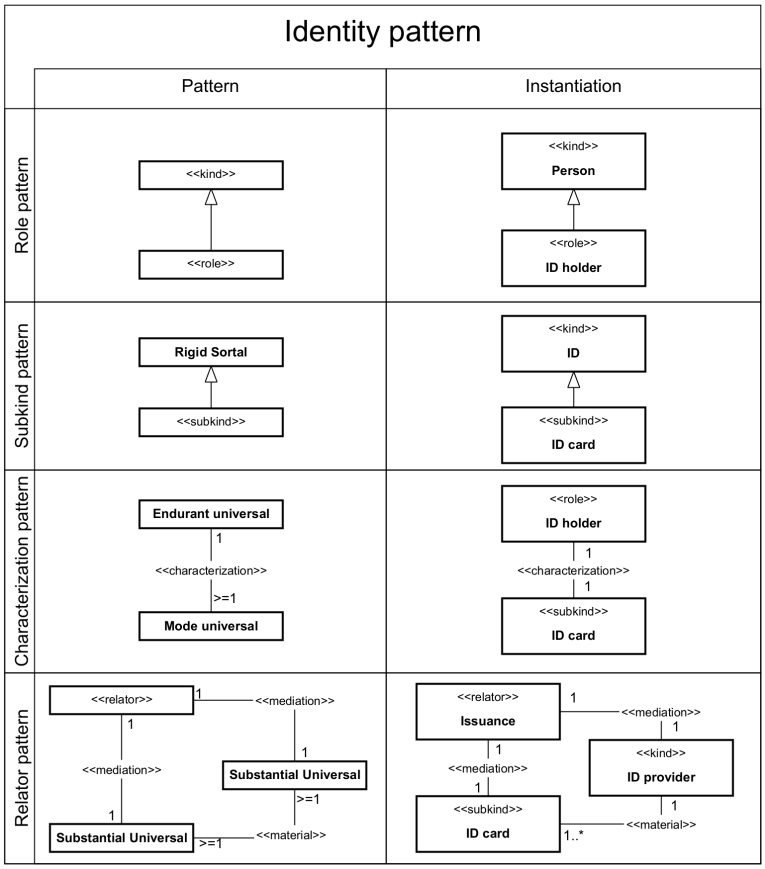


Figure 3.3: Example of an instantiation of the Identity pattern via available patterns within graph grammar. The left column describes the general pattern, the right instantiates it[1]. The instantiations describe individual patterns that are then pieced together to create the Identity pattern.

conformity and validity.

3.2.2 Discussion of approaches

The four proposals presented above showcase different approaches to packaging, modularizing, interconnecting, or annotating ontologies. While OPLa offers a competent annotation ontology w.r.t. describing patterns and subpatterns, OTTR offers one as well in its ODP catalogue. On the other hand, OTTR (when working with stOTTR, its proprietary format) conforms to an already established and popular standard. While the latter is appealing for reasons of faster adoption and compatibility, it lacks in comparison to the number of abilities the language can immediately provide for engineers. In addition, since wOTTR, a standard for OTTR enabling the use of RDF/OWL

for template definition/instantiation, a transition from an existing model that does not use ODPs to one that takes advantage of *OTTR's* refactoring potential can be made easier; especially with algorithms that enable such refactoring from an ontology or set of ontologies.

DOL, then, with all of its extensions, provides an alternative to OTTR, especially as far as (inter)compatibility with already existing languages goes. The authors of the paper introducing extensions to generic DOL even reference OTTR when comparing its capabilities of creating patterns to OTTR's, stating its superiority with regards to actions over list parameters specifically. However, DOL requires a custom interpreter as well - the one promoted by the authors is Hets. While support for UML is planned, it is not available at the time of writing, and the generic DOL with extensions is yet to be implemented. A custom conversion from UML to OWL/RDF is therefore still necessary. Hence DOL in our case does not offer significant advantages to OTTR.

Guizarrdi and Zambon's contribution presents an interesting way of treating ODPs. The text suggests treating the application of ODPs as a function with relevant classes as parameters and a modified graph (model) as output. This is what OTTR and GDOL essentially provide; with the exception of not being able to create patterns which delete or modify something in an existing graph. Regardless, the consequence of treating entire models as collections of ODPs and therefore, if the patterns have been designed correctly, not having to worry about model validity, provides an interesting approach with which ODPs can be treated - not as elements enhancing a model with other components, but the elementary basic blocks of models. Of course, not all models will only use formally defined ontologies such as UFO or be completely satisfied with existing ODPs - however, this is not the case for OntoUML, because of the theories presented in the discussed paper.

Chapter 4

Reasonable Ontology Templates

The two basic building blocks of the OTTR language are template definitions and instances. Both will be shown in the stOTTR format in this section, as that is the one intended by the authors for use in cases of demonstrations - Lutra can then be used to convert (in OTTR terminology expand), the template instances into Turtle.

Template definitions contain parameters and the content of the definition itself, which consists of other templates. The templates can be other predefined templates or base templates, which represent an RDF resource (e.g. a triple). Parameters can be typed, optional, and multiple (as a list). Cyclic dependencies are not allowed.

Instances are created using template definitions by referring to the template with the (mandatory) parameters filled out. Continuing with the Identity example from Section 3.2.1:

```
ex:Identity(ex:Alice, ex:DriversLicence, ex:DriversRegistry).
```

The resulting graph after expansion lists the following information:

```
ex:Alice rdfs:subClassOf ex:Person.
ex:DriversLicence rdfs:subClassOf ex:ID.
```

```
# ex:Connection(ex:Alice, gufo:characterization,
#    ?id, 1, 1, 1, 1),
ex:Alice rdfs:subClassOf
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:allValuesFrom ex:DriversLicence],
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:onClass ex:DriversLicence;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger],

# ex:Connection(ex:DriversLicence, gufo:material,
#    ex:DriversRegistry, 1, ottr:none, 1, 1),
ex:Alice rdfs:subClassOf
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:allValuesFrom ex:DriversRegistry],
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:onClass ex:DriversRegistry;
    owl:minQualifiedCardinality "1"^^xsd:nonNegativeInteger],

# ex:Connection(ex:DriversRegistry, gufo:mediation,
#    ex:issuance, 1, 1, 1, 1),
ex:Alice rdfs:subClassOf
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:allValuesFrom ex:issuance],
    [rdf:type owl:Restriction;
    owl:onProperty gufo:characterization;
    owl:onClass ex:issuance;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger],

# ex:Connection(ex:DriversLicence, gufo:mediation,
#    ex:issuance, 1, 1, 1, 1)
ex:DriversLicence rdfs:subClassOf
    [rdf:type owl:Restriction;
    owl:onProperty gufo:mediation;
    owl:allValuesFrom ex:issuance],
    [rdf:type owl:Restriction;
    owl:onProperty gufo:mediation;
    owl:onClass ex:issuance;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger],
```

4.1 Annotation ontology

In its template library, OTTR provides an official annotation ontology, docTTR, which is in the form of templates implementing the annotations[25]. With it, it is possible to:

- Version ODPs and describe changes from version to version;
- Establish provenance metadata;
- Describe the ODP, provide notes, examples, etc.

The list of available annotations is at `https://tpl.ottr.xyz/p/docttr/0.1/`.

4.2 Basic OntoUML representation

Because of OntoUML's wide adoption, there is some use in redefining OntoUML as a set of ODPs - at the very least it serves as an important test-case for ODP usage. Guizzardi himself with another colleague explore this possibility, which will be discussed in more detail in Section 3.2.1 describing their paper.

There is a lightweight thesaurus describing UFO, called gUFO, which could help in our effort if the ODPs were compatible with it. Such a design would need to also accept all the possible constructs of UML in order to be fully compatible with OntoUML.

There are certain UML-specific constructs that require a certain transformation in order for them to be implemented in an RDF graph. An advantage of the pattern approach is that difficult (or generally unobvious) translations can be used without having to understand the underlying conversions; similar in simplicity to modeling in UML in the first place.

4.2.1 Classes and connections

To show the modularization capabilities, the basic building blocks, which will be used in further examples, will be defined first. These building blocks are the Class and Connection templates, representing classes and the relationships between them as commonly used in OntoUML. The templates are defined first, followed by their description according to the ontology referenced in Section 4.1.

```
ex:Class[ottr:IRI ?iri, ? ottr:Literal ?name,
    ? NEList<ottr:IRI> ?customType] :: {
    ottr:Triple(?iri, rdf:type, owl:Class),
    cross | ottr:Triple(?iri, rdf:type, ++?customType),
    ottr:Triple(?iri, skos:prefLabel, ?name)
}.

ex:Connection[ex:Class ?subject, ottr:IRI ?predicate,
    ex:Class ?object, ? xsd:nonNegativeInteger ?minCardinality1,
    ? xsd:nonNegativeInteger ?maxCardinality1,
    xsd:nonNegativeInteger ?minCardinality2,
    ? xsd:nonNegativeInteger ?maxCardinality2] :: {
    o-owl-ax:SubObjectSomeValuesFrom(
        ?subject, ?predicate, ?object),
    o-owl-ax:SubObjectAllValuesFrom(
        ?subject, ?predicate, ?object),
    o-owl-ax:SubObjectMinCardinality(
        ?subject, ?predicate, ?object, ?minCardinality1),
    o-owl-ax:SubObjectMaxCardinality(
        ?subject, ?predicate, ?object, ?maxCardinality1),
    o-owl-ax:SubObjectMinCardinality(
        ?object, ?predicate, ?subject, ?minCardinality2),
    o-owl-ax:SubObjectMaxCardinality(
        ?object, ?predicate, ?subject, ?maxCardinality2),
}.

ex:Class a ottr:Pattern;
    ottr:status ottr:draft;
    owl:versionInfo 0.1;
    dct:description 'Pattern for creating classes
        with optional multiple types'@en;
    dct:creator ex:Alice;

ex:Connection a ottr:Pattern;
    dct:description 'Pattern for creating OntoUML connections
```

```
        with optional cardinalities'@en;
    dct:creator ex:Alice;
```

The `cross` operator performs a cartesian product with a given list (`?customType` in this example - a `?iri, rdf:type, ?customType` triple is generated for each `?customType` list member). OTTR also offers similarly used operators for convolution (also known as `zip` e.g. in the Python programming language) called `zipMin` and `zipMax`.

The patterns originating from prefix `o-owl-ax` are simple OWL patterns capturing class restrictions. For example, the `ax:SubObjectMinCardinality` pattern instantiation results in these triples in the RDF graph (?subject, ?predicate, ?object and ?minCardinality will be replaced with parameters):

```
    ?subject rdfs:subClassOf [ a owl:Restriction;
        owl:onProperty ?predicate;
        owl:onClass ?object;
        owl:minQualifiedCardinality ?minCardinality].
```

These prefixes can be then used for instantiations[1]:

```
ex:Class(ex:Person, "Person", (gufo:kind)).
ex:Class(ex:Patient, "Patient", (gufo:role)).
ex:Class(ex:Disease, "Disease", (gufo:quality)).
ottr:Triple(ex:Patient, rdfs:subClassOf, ex:Person).
ex:Connection(ex:Patient, gufo:characterization, ex:Disease,
    1,1,1,ottr:none).
```

This example describes a model shown in Figure 4.1. With these templates, a significant set of OntoUML models can be represented. Some special cases are shown in Section 4.3.

4.2.2 Cardinalities and restrictions

If patterns are to replace conventional modeling, special considerations have to be made towards any potential restrictions that the patterns' elements might

[1]Note that the template definition, instances, and metadata would each be in separate files; *stOTTR* and Turtle statements or definitions and instances cannot be in the same file.

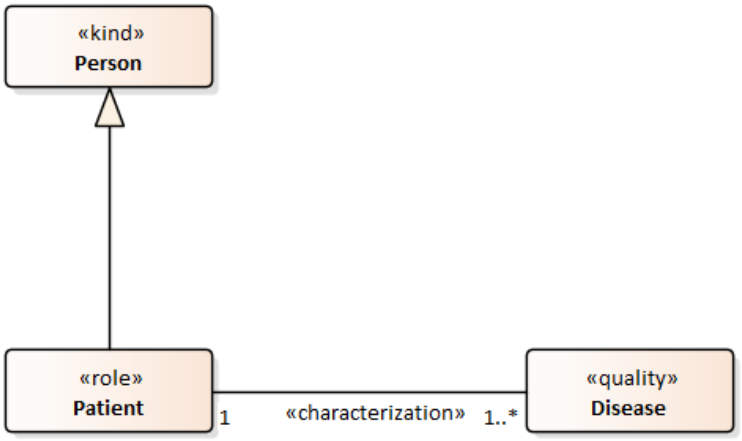


Figure 4.1: A patient-describing pattern described with instantiated class and connection templates.

require (e.g. cardinality requirements, data property minimum and maximum values, etc.). While the examples shown in the evaluation demonstrate the enforcement of parameter types and optional or mandatory parameters, there are no other checks performed on template instantiation (a facet shared by the DOL language introduced in Section 3.2.1). Therefore, any e.g. OWL restrictions have to be evaluated after instantiation by separate reasoner software.

4.3 Advanced model features

These examples were taken from 'Stability Patterns in Ontology-Driven Conceptual Modeling' by Guizzardi & Almeida [7]. The reason for this is that the models shown in the paper exemplify key features of OntoUML: n-ary relations, XOR pairwise constraints, etc. Such elements test potential flexibility of the proposed language. There are two notable constructs that present a considerable modeling challenge (such that require a specialized pattern of more than one restriction declaration):

- a XOR constraint between two relations or multiple XOR constraints in a pairwise distribution between three or more relations that are all connected with the same class.
- An N-ary (ternary or higher) relation.

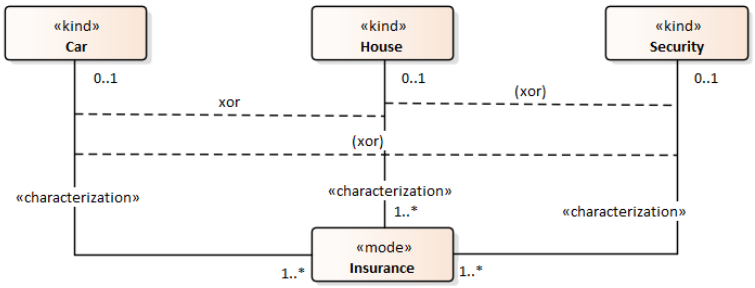


Figure 4.2: Example of pairwise XOR constraint use in a model of insured items.

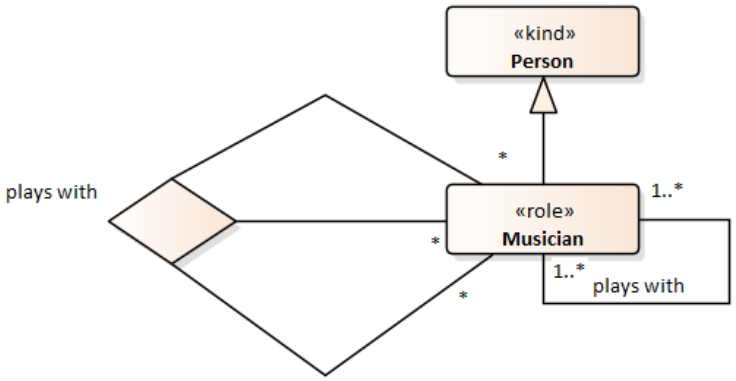


Figure 4.3: Conceptual diagram of a model with a binary and ternary relationship

In figure 4.3, a ternary relationship is used, which was emphasized in section *OntoUML constructs*. According to W3C's Working Group, a way to describe n-ary relationships is to reify the relationship [4]. An example of such reification for the model in the figure is expressed below.

```
ex:Class(ex:Musician, 'Musician', (gufo:role)).
ex:Class(ex:Person, 'Person', (gufo:kind)).
ottr:Triple(ex:Musician, rdfs:subClassOf, ex:Person).
ex:Connection(ex:Musician, ex:plays-with, ex:Musician, 1,
    ottr:none, 1, ottr:none).
ex:Class(ex:PlaysWith, 'Plays with', ottr:none).
ex:Connection(ex:Musician, ex:plays-with1, ex:PlaysWith,
    ottr:none, ottr:none, ottr:none, ottr:none).
ex:Connection(ex:Musician, ex:plays-with2, ex:PlaysWith,
    ottr:none, ottr:none, ottr:none, ottr:none).
ex:Connection(ex:Musician, ex:plays-with3, ex:PlaysWith,
    ottr:none, ottr:none, ottr:none, ottr:none).
```

Another feature of OntoUML is a pairwise XOR constraint expression. Looking at Figure 4.2, the appropriate pattern can now be constructed from it:

```
ex:XORpatternRestriction[ottr:IRI ?c, owl:ObjectProperty ?op,
        owl:Class ?object] :: {
ottr:Triple(?c, rdf:type, owl:Restriction),
ottr:Triple(?c, owl:onProperty, ?op),
ottr:Triple(?c, owl:allValuesFrom, ?object)
}.

ex:XORpatternUnionOf[ottr:IRI ?c, NEList<ottr:IRI> ?list] :: {
    ottr:Triple(?c, rdf:type, owl:Class),
    ottr:Triple(?c, owl:unionOf, ?list)
}.

ex:XORpattern[owl:Class ?c, owl:ObjectProperty ?op,
    NEList<ottr:IRI> ?blanks,
    NEList<owl:ObjectProperty> ?properties,
    NEList<owl:Class> ?objects] :: {
ottr:Triple(?op, rdf:type, owl:ObjectProperty),
ottr:Triple(?c, rdf:type, owl:Class),
ottr:Triple(?c, owl:disjointUnionOf, ?objects),
cross | ottr:Triple(++?objects, rdfs:subClassOf, ?c),
ottr:Triple([], owl:allDisjointProperties, ?properties),
zipMin | ex:XORpatternRestriction(++?blanks, ?op, ++?objects),
ex:XORpatternUnionOf(_:union, ?blanks),
ottr:Triple(?c, rdfs:subClassOf, _:union)
} .
```

The example is notable for several reasons:

- Note the use of blank nodes in the `ex:XORpattern` template definition. To build up the expression, they are used similarly to a variable; first, the list of restrictions is declared, then they are placed into the owl:unionOf range, and finally they are included in the subject class' subClassOf notation.
- A drawback of declaring these patterns manually is that the user needs to have already thought out all the IRIs that the pattern is going to require. Management of internal IRIs has to be done by an external tool if it is to be abstracted away from the end user.

An example of using this pattern library was shown in Section 3.2.1. An aspect of the ODP that one might find desirable is the ability to infer certain information about the person - if a person is assigned an identity card with "1996-02-06" as a recorded birth date, it is reasonable to assume that that person's birth date is "1996-02-06". Two ways of implementing this will be shown: (i) a pattern that eliminates redundancy by declaring the birth date the same on the card and for the person on creation of the card, and (ii) a pattern involving a SWRL [24] rule inferring the connection.

```
(i)

ex:Identity[! ex:IDHolder ?holder, ! ex:IDCard ?id,
! ex:IDProvider ?provider, ! ottr:Literal ? ?firstName] :: {
ottr:Triple(?holder, rdfs:subClassOf, ex:Person),
ottr:Triple(?id, rdfs:subClassOf, ex:ID),
ottr:Triple(?holder, ex:firstName, ?firstName),
ottr:Triple(?id, ex:firstName, ?firstName),
ex:Connection(?holder, gufo:characterization, ?id, 1, 1, 1, 1),
ex:Connection(?id, ex:material, ?provider, 1, ottr:none, 1, 1),
ex:Connection(?provider, ex:mediation, ex:issuance, 1, 1, 1, 1),
ex:Connection(?id, ex:mediation, ex:issuance, 1, 1, 1, 1)
}.

(ii)

@prefix swrl: <http://www.w3.org/2003/11/swrl#> .

ex:Identity[! ex:IDHolder ?holder, ! ex:IDCard ?id,
! ex:IDProvider ?provider] :: {
ottr:Triple(?holder, rdfs:subClassOf, ex:Person),
ottr:Triple(?id, rdfs:subClassOf, ex:ID),
ex:Connection(?holder, gufo:characterization, ?id, 1, 1, 1, 1),
ex:Connection(?id, gufo:material, ?provider, 1, ottr:none, 1, 1),
ex:Connection(?provider, gufo:mediation, ex:issuance, 1, 1, 1, 1),
ex:Connection(?id, gufo:mediation, ex:issuance, 1, 1, 1, 1),
ottr:Triple(ex:varX, rdf:type, swrl:Variable),
ottr:Triple(ex:varY, rdf:type, swrl:Variable),
ottr:Triple(ex:varZ, rdf:type, swrl:Variable),
ex:IdentityRuleBody(_:body, ex:varX, ex:varY, ex:varZ),
ex:IdentityRuleHead(_:head, ex:varX, ex:varZ),
ottr:Triple(_:rule, rdf:type, swrl:Imp),
ottr:Triple(_:rule, swrl:body, _:body),
ottr:Triple(_:rule, swrl:head, _:head)
}.
```

```
ex:IdentityRuleBody[?body, ?x, ?y, ?z] :: {
ottr:Triple(?body, rdf:type, swrl:AtomList),
ottr:Triple(?body, rdf:first, _:first),
ottr:Triple(_:first, rdf:type, swrl:IndividualPropertyAtom),
ottr:Triple(_:first, swrl:propertyPredicate, ex:hasDOB),
ottr:Triple(_:first, swrl:argument1, ?x),
ottr:Triple(_:first, swrl:argument2, ?y),
ottr:Triple(?body, rdf:rest, _:rest),
ottr:Triple(_:rest, rdf:type, swrl:AtomList),
ottr:Triple(_:rest, rdf:first, _:restFirst),
ottr:Triple(_:rest, rdf:rest, rdf:nil),
ottr:Triple(_:restFirst, rdf:type, swrl:DatavaluedPropertyAtom),
ottr:Triple(_:restFirst, swrl:propertyPredicate, ex:hasDOB),
ottr:Triple(_:restFirst, swrl:argument1, ?y),
ottr:Triple(_:restFirst, swrl:argument2, ?z),
}.

ex:IdentityRuleHead[?head, ?x, ?z] :: {
ottr:Triple(?head, rdf:type, swrl:AtomList),
ottr:Triple(?head, rdf:first, _:first),
ottr:Triple(?head, rdf:rest, rdf:nil),
ottr:Triple(_:restFirst, rdf:type, swrl:DatavaluedPropertyAtom),
ottr:Triple(_:restFirst, swrl:propertyPredicate, ex:hasDOB),
ottr:Triple(_:restFirst, swrl:argument1, ?x),
ottr:Triple(_:restFirst, swrl:argument2, ?z),
}.
```

The SWRL rule expressed in (ii) can be read as `hasID(?x, ?y), hasName(?y, ?z) -> hasName(?x, ?z)`. A clear advantage of this approach is that any changes on the ID card is immediately reflected on the record of the person themselves - in (i), the changes would have to be recorded in both instances separately, if for some reason queries are not a viable option for retrieving holder information.

An avenue of potential future development is also hinted at in (ii); in this instance, the SWRL rule patterns are created manually. However, since transformation of SWRL rules into RDF can be done algorithmically (as the RDF representation for SWRL already exists), an algorithm that can refactor these rules into the language seamlessly can multiply the refactoring potential of the proposed language. More specifically, if an algorithm had at its disposal a pattern library that represented SWRL's basic building blocks (in this case, atoms and their attributes) and could utilize those patterns seamlessly (perhaps even automatically create new ones when appropriate) from a more human-readable form of expressing SWRL rules, the task of refactoring SWRL representations in RDF could be largely automated. A

similar case could be made for OWL and its expressions.

Chapter 5

Pattern management server

As mentioned in Chapters 1 and 3, there is a need for an intermediary service simplifying the use of ODPs on actual data. Since Semantic Web data is meant to be stored on internet-accessible servers, the obvious choice is to create another server - a service providing users the endpoints needed to submit, search, expand, and instantiate patterns. This service would keep information about patterns and instantiations in an associated database. It is necessary for the server to be able to accept and return RDF documents.

5.1 Technologies

The server uses Spring as the base framework for deploying endpoints. RDF connections to the database, handling of statements and any updates/queries are handled by Apache Jena - a popular Java framework for Semantic Web implementations.

Technically, any triple store database (such as GraphDB, the store used for the KODI software suite) would be appropriate to serve as a database for the service, Apache Fuseki store is an easy choice simply because of almost no setup required for Jena to directly interact with the store. Figure 5.1 describes the basics of the configuration.

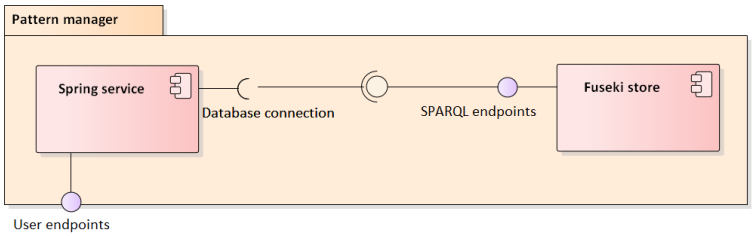


Figure 5.1: Component diagram for the manager. The only functions meant for users are exposed by the Spring service, as denoted by the *User endpoints* port. The Fuseki store interacts with the service only.

5.2 Requirements

The basic read/update requirements can be itemized as follows:

- Retrieve an ODP.
- Query the ODPs (e.g. return and ODPs satisfying some metadata requirements).
- Submit an ODP into the database.
- Retrieve instantiations of an ODP.

5.2.1 Lutra inclusion

There is a potential benefit for Lutra, the OTTR expander, to be also included in the overall package, namely the ease of setting up the whole environment. However, there is no benefit for Lutra requests to go through the service first: Lutra takes an input, desired format, and returns a translated or expanded result. Therefore, an endpoint in the Spring service to return Lutra results would just function as a simple passthrough.

Lutra is released as both a .jar and a .war solution, so the inclusion of it into an existing framework should not be difficult.

5.3 Algorithms

Along with the aforementioned basic endpoints, the service also implements three algorithms that process both inputted model data and its own database to find and/or create appropriate data.

5.3.1 The suggestion algorithm

Consider a situation where an ontology engineer is tasked with modifying certain models, but is unaware of the complete range of the library of patterns available to them. One solution is to use a search function (taking advantage of the querying endpoint), but is still unsure of what pattern would fit their particular use-case.

The suggestion algorithm takes a (part of a) model and processes it to find possible combinations of patterns and applicable parameters to create instantiations with. This takes advantage of the fact that OTTR patterns oftentimes have defined types of parameters, so when that information is cross-referenced with the types of subject present within the input, the matches are evident. The premise of the algorithm is the following:

1. Create pairs of subjects and their rdf:type values.
2. Query patterns to extract pairs of patterns and parameter type requirements from their rOTTR representation in the database.
3. Combine the two pairs in such a way that results in a map of patterns and sets of subjects that could populate the patterns' parameters.
4. (Optional enhancement) Provide a ranking of the assignments from step 3 based on perfect matches, usage statistics from the same model, etc.

The automatic applicability of the results is heavily predicated on the heuristics employed in step 4 and the input. It is often the case, especially with larger inputs, that the number of possible combinations can rise exponentially. Because of this consequence, the algorithm does not replace the need to know the general ins and outs of the pattern library entirely, but merely supplements the search for the right pattern to use.

5.3.2 The refactor/create algorithm

The remaining two algorithms share basic principles of operation, and as such they will be discussed together. The general underlying algorithm is described in a companion online demonstration[18] for a paper about using OTTR to automatically generate patterns and use them to refactor an entire database of a corporation with significantly less user intervention[22].

The basic structure of the algorithms rests on *dependency pairs*: a mapping of "patterns" (or sets of predicates, as will be explained more precisely below) and subjects de facto including that pattern.

Example 5.1. Consider a model in which we have several concepts with some, but not all repeating predicates.

The first step is to convert all predicates to patterns returning a simple statement - this is so they can be more easily processed later. For example, `rdf:type` turns into:

```
rdfType(?subject, ?object) ::
    ottr:triple(?subject, rdf:type, ?object).
```

Then, using these predicate patterns, all subjects with their predicates are turned into patterns as well. This way, the statements

```
ex:Car a owl:Class;
     skos:prefLabel "Auto"@cs,
     skos:inScheme ex:exampleScheme.
```

turn into a pattern:

```
ex:Car(?prefLabel, ?rdfType, ?skosInScheme) :: {
     ottr:triple(ex:Car, rdf:type, ?rdfType).
     ottr:triple(ex:Car, skos:prefLabel, ?prefLabel).
     ottr:triple(ex:Car, skos:inScheme, ?skosInScheme).
}
```

This set of patterns is then constructed into the aforementioned dependency pairs: each subject is correlated with the predicate patterns needed to create an expansion that would exactly reproduce the given data of the subject. The pairs take the form of <I,T>, where

- I are the predicate patterns which could form a potential pattern, and
- T are the subjects that can be created by instantiating and expanding the pattern from I.

Any pairs that have only one entry in I or T are removed, as that does not signify a repeated pattern.

The difference between the refactoring and creating patterns begin after the dependency pairs have been formed. In the case of refactoring, the existing database is checked whether an exact match resembling the resulting pattern candidates is found - if so, the subjects from the pair are refactored into instantiations of the found pattern. This is useful for record keeping of the usage of patterns, which can then be studied for further analysis (usage correctness, pattern improvements, metadata clarification, etc.). However, if the pattern candidates do not yet exists, there is an option to automatically create them and instantly populate them with the subject found in the other part of the pair. The effects of both algorithm variants can be combined.

This algorithm has a shortcoming in that it only works with subjects and not any further model constructs. In an environment where the model, a glossary connected with semantic links, has possible redundancies that encompass both subjects and subject relationships, this algorithm will not find them.

5.4 Implementation

The endpoints are therefore implemented as follows:

Request ODP and its info The service is passed an IRI of a supposed pattern in the database. The service returns all related statements pertaining to the pattern.

Update the database The service is passed an RDF document and attempts to insert it into the database.

Query the database The service is passed a SPARQL query and attempts to ask the database with it.

Request instantiations of an ODP The service is passed an IRI of a supposed pattern in the database. The service returns all instantiations pertaining to the pattern.

Suggest patterns The service is passed an RDF document and runs the aforementioned suggestion algorithm on it.

Refactor the model The service is passed an RDF document and runs the aforementioned refactoring algorithm on it.

Create new ODPs The service is passed an RDF document and runs the aforementioned pattern creation algorithm on it.

The reason the inputs are not more tightly defined is that the rOTTR specification of patterns and instances is formally defined; it would be limiting to resolve queries or updates through a JSON formatted input, for example, and it would also create an obligation to continually update the accepted format in case the specification changes.

Evaluation of the implementation is described in Chapter 7.

Chapter 6

Ontographer integration

Ontographer is a web application used within the KODI project to provide conceptual modeling functionality to the vocabulary-creating process, taking glossaries and turning them into models via creating semantic connections between various terms in a graphical environment. The tool does not necessarily require understanding of underlying Semantic Web concepts - it is intended for clerks who have the domain knowledge to create accurate semantic connections. Figure 6.1 shows a screenshot of Ontographer during ordinary operation.

6.1 Standard workflow

The usual use loop of Ontographer is that a user loads a *workspace*, a list of vocabularies that are set to be edited, the terms of which can then be displayed in the main center window, or the *canvas*. Users can manipulate the terms' information, their positions within the canvas, as well as the relationships that connect them. At any time, the validation service can be called to potentially catch invalid attributes or connections w.r.t. the SSP specifications. The application supports multiple *diagrams* - essentialy different views of the entire model, sectioning it into specific chunks should the user desire for purposes of presentation or easier manipulation. The diagrams, term and relationship positions and attributes are persisted to the database as soon as a change is made - this allows quick changes and prevents loss of work. After the desired changes have been made, the workspace is

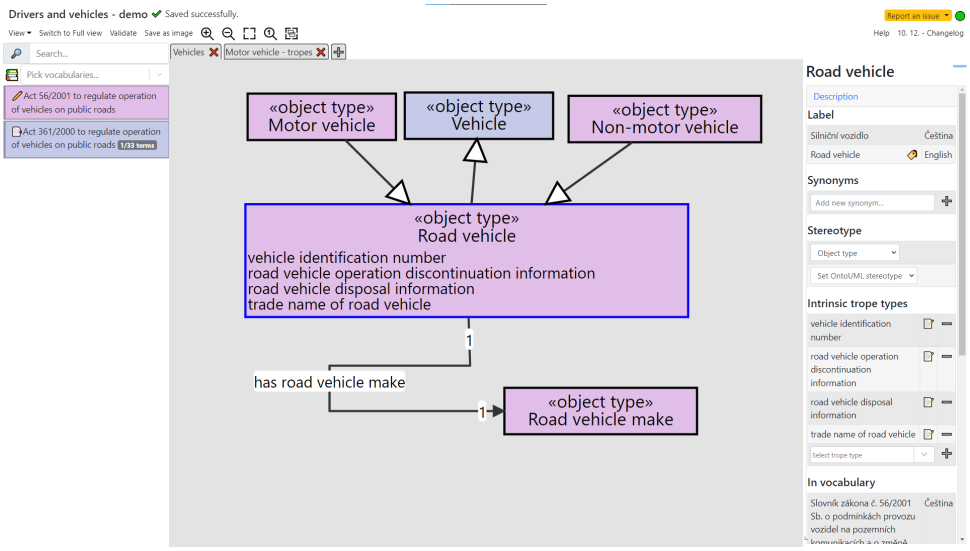


Figure 6.1: Screenshot of OntoGrapher without any pattern enhancements with the "Drivers and vehicles" workspace open. The left panel lists all vocabularies and concepts in the model, while the right panel details a single selected concept. The tabs above the work area list diagrams. Concepts are color-coded to their respective vocabularies. The first vocabulary in order of appearance on the left panel is write-enabled and the second is read-only (as depicted by pictographs next to their titles).

marked as a candidate for publishing, i.e. merging the changes made into the original repository, making the changes canon.

In SSP, terms are represented as owl:Classes and skos:Concepts with various labels and descriptions. Relationships can be represented in two ways: in the case of a specialization, as a simple rdfs:subClassOf predicate or another Class with an intricate set of owl:Restrictions otherwise.

Example 6.1. An example of a connection between two terms with the latter representation of a relationship looks like this:

```
# "Documents birth event" Relator type
gov-birth-registry:documents-birth-event a skos:Concept,
    z-sgov-pojem:typ-vztahu;
    skos:inScheme gov-registry-office:scheme;
    skos:prefLabel "documents birth event"@en;
    skos:altLabel "documents"@en.

# Connection from "Documents birth event" to "Record of birth"
gov-birth-registry:documents-birth-event rdfs:subClassOf
    [rdf:type owl:Restriction;
```

```
    owl:onProperty z-sgov-pojem:má-vztažený-prvek-1;
    owl:allValuesFrom gov-birth-registry:record-of-birth],
    [rdf:type owl:Restriction;
    owl:onProperty
        [owl:inverseOf z-sgov-pojem:má-vztažený-prvek-1];
    owl:allValuesFrom gov-birth-registry:documents-birth-event],
    [rdf:type owl:Restriction;
    owl:onProperty z-sgov-pojem:má-vztažený-prvek-1;
    owl:onClass gov-birth-registry:record-of-birth;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger],
    [rdf:type owl:Restriction;
    owl:onProperty
        [owl:inverseOf z-sgov-pojem:má-vztažený-prvek-1];
    owl:onClass gov-birth-registry:documents-birth-event;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger].

# Connection from "Documents birth event" to "Birth of person"
gov-birth-registry:documents-birth-event rdfs:subClassOf
    [rdf:type owl:Restriction;
    owl:onProperty z-sgov-pojem:má-vztažený-prvek-2;
    owl:allValuesFrom gov-civil-law:birth-of-person],
    [rdf:type owl:Restriction;
    owl:onProperty
        [owl:inverseOf z-sgov-pojem:má-vztažený-prvek-2];
    owl:allValuesFrom gov-birth-registry:documents-birth-event],
    [rdf:type owl:Restriction;
    owl:onProperty z-sgov-pojem:má-vztažený-prvek-2;
    owl:onClass gov-civil-law:birth-of-person;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger],
    [rdf:type owl:Restriction;
    owl:onProperty
        [owl:inverseOf z-sgov-pojem:má-vztažený-prvek-2];
    owl:onClass gov-birth-registry:documents-birth-event;
    owl:qualifiedCardinality "1"^^xsd:nonNegativeInteger].
```

Ontographer creates these data by essentially employing (currently) hard-coded informal patterns: a creation or editing of a term of relationship is done through various forms present throughout the user interface; the application then translates its representation of the data into RDF, which is then sent into the main database via SPARQL Update requests. The data mentioned in example 6.1 can be created by the user in a couple of seconds during normal operation.

Conceptually, then, the implementation of standardized patterns into On-

tographer makes sense; replacements of the fundamental hard-coded patterns would allow the application to represent multiple variants of term specifications outside SSP, but the main point is the creation of patterns that encapsulate multiple terms and patterns. Consider the Identity ODP from Chapter 3; similar designs are commonly present throughout many vocabularies of SSP.

6.2 Pattern integration

Patterns and instances are implemented into Ontographer via Pattern and Instance *TypeScript* types. These types are responsible for all interactions between them and other parts of the application. The conversions are done on communication with the service and are described in Section 6.4. On application load, the patterns and relevant instances (those of the terms present in the workspace) are loaded through the service like the terms, connections, and diagrams themselves.

6.3 User interface

The enhancements have been implemented as to not significantly alter the current user experience. The user is normally able to select multiple terms at the same time to remove them from the canvas or perhaps move them together. Also, right clicking of the mouse on empty space on the canvas triggers a form to create a new term. In the enhancement, right clicking when terms are selected brings up the Pattern or Instance creation forms. Figure 6.2 shows the creation of a pattern, whereas Figure 6.3 show the creation of an instance. Instances can also be created by right clicking the mouse on an empty canvas without any selection - the difference is that Ontographer won't try to autofill the fields for parameters and relationships with the user's selection. The pattern values can either be new terms and relationships that are created along with the instance itself or can be existing terms that the instance associates with the parameter.

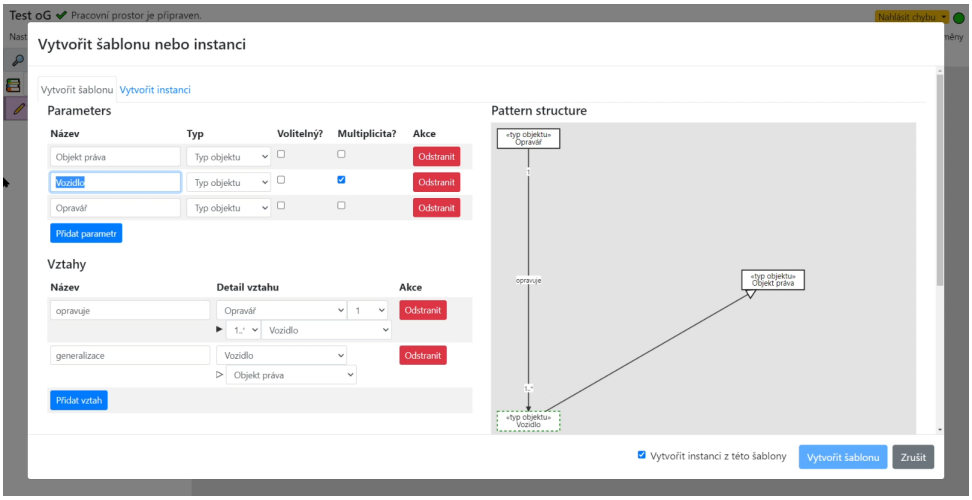


Figure 6.2: A creation of a new pattern. On the left are forms for editing the features of the pattern, while on the right the vizualization shows the structure of the pattern in the process of creation.

6.3.1 Instance view

Once the patterns or instances have been created, however, they can seem kind of invisible, prompting the question if anything was even done in the first place. The user can see the instantiations of patterns with the "Show instantiations" button on the top panel. Clicking this option replaces terms and relationships encapsulated with instances into similar looking "boxes" describing the pattern the instance instantiates and its parameter names and values. A link between instances demonstrates a pattern value that is shared between the given instances. Clicking on an instance "box" opens up the Detail panel familiar to users of Ontographer, but with information about the instance instead, containing links to view pattern statistics or the internal structure of the instance (i.e. terms and relationships that are "hidden" in the instance). Figure 6.4 shows this on an example with two connected instances. Another button on the top panel is the "Pattern statistics" button. That button shows another dialogue with selections of patterns - if one is selected, the right column provides information about the pattern, along with a list of instantiations of this particular pattern. Figure 6.5 shows this on an example.

6.4 Server integration

. The server is used for basic operations, such as fetching or posting updates, the results of which are then converted from the rOTTR form stored in the

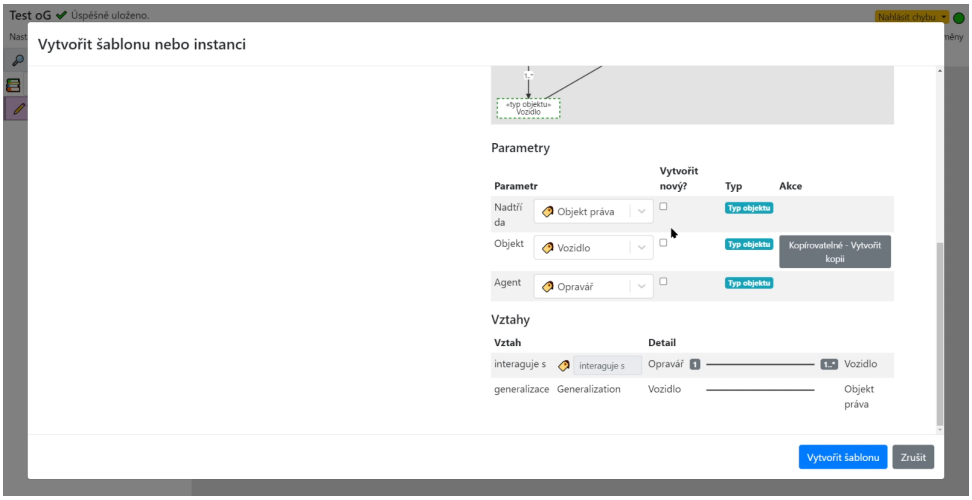


Figure 6.3: A creation of a new instance. The tag pictogram next to parameter and relationships names signifies that the relationship or term mentioned already exists, and therefore nothing new (except the instance) will be created.

Fuseki database to Ontographer's internal type representations. The design of the patterns and instances Ontographer sends and receives is worth detailing, as there are several enhancements to the specification-mandated forms of the rOTTR types[1].

Instead of producing and consuming patterns that could expand into data useful for the underlying dataset (e.g. the one described in example 6.1), the patterns describe the *actions* Ontographer should take to deliver the actual data.

Example 6.2. Ontographer might push a pattern such as

```
ex:Example(ottr:IRI ?t1, ottr:IRI ?t2, ottr:IRI ?r3) :: {
    ottr:triple(?t1 a og:term),
    ottr:triple(?t1 a z-sgov:typ-objektu),
    ottr:triple(?t2 a og:term),
    ottr:triple(?t2 ottr:modifier ottr:optional),
    ottr:triple(?r3 a og:conn),
    ottr:triple(?r3 og:from ?t1),
    ottr:triple(?r3 og:to ?t2),
}
```

This simple example tells Ontographer that there is a pattern with two terms,

[1]This does not break the database entries for other users, however, because all Ontographer does is add additional entries other applications won't think to look for or will ignore

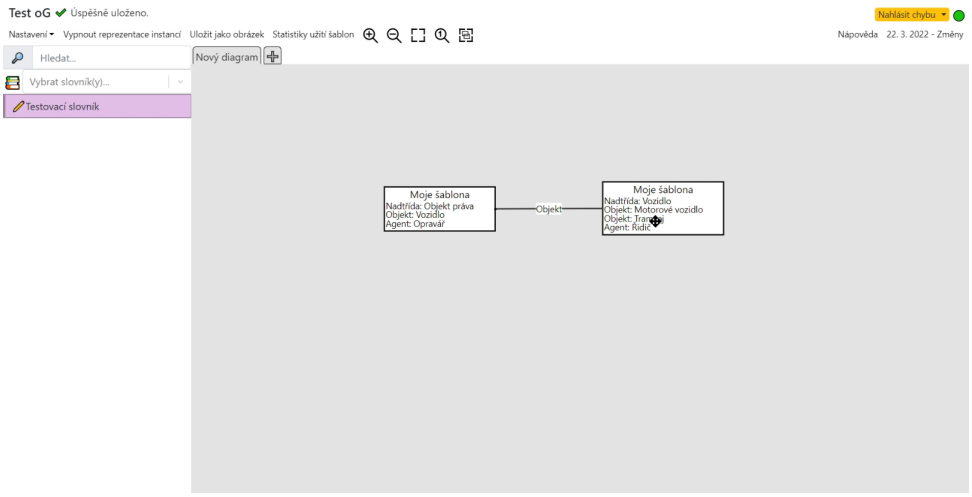


Figure 6.4: Ontographer with Instance view enabled. There are two instances shown, each with a list of parameters and its values in the format `name: value`. Since the instances share the Objekt term, a link is shown demonstrating just that.

one relationship connecting the two, the first term has to be *at least* an Object Type and the other's existence when instantiating is optional.

A problem found during development is that docTTR does not offer a way to label the parameters. The `?t1 ?t2 ?r3` parameter identifiers are present only in pattern definitions, but in actual use are invisible, so the choice is between generating the name somehow or to include another sub-pattern describing the name of the parameter. Ontographer's implementation chooses the latter.

This issue is also present in the instantiations - in OTTR normally, the instantiation parameters are present as an ordered rdf:List without context, since the context is provided through the actual pattern definition. Therefore, Ontographer's instantiations adds another ordered list to rOTTR instantiation representations, this time documenting the names of the parameters so that they may be cross-referenced by the names in the pattern definition.

Unfortunate omissions are those of the Automatic generation and refactoring algorithms. As mentioned in Section 5.3.2, these algorithms as implemented in the service are usable for patterns describing only one subject. In Ontographer, it would be very useful to be able to analyze patterns spanning multiple terms and relationships - therefore, those algorithms are of little value here.

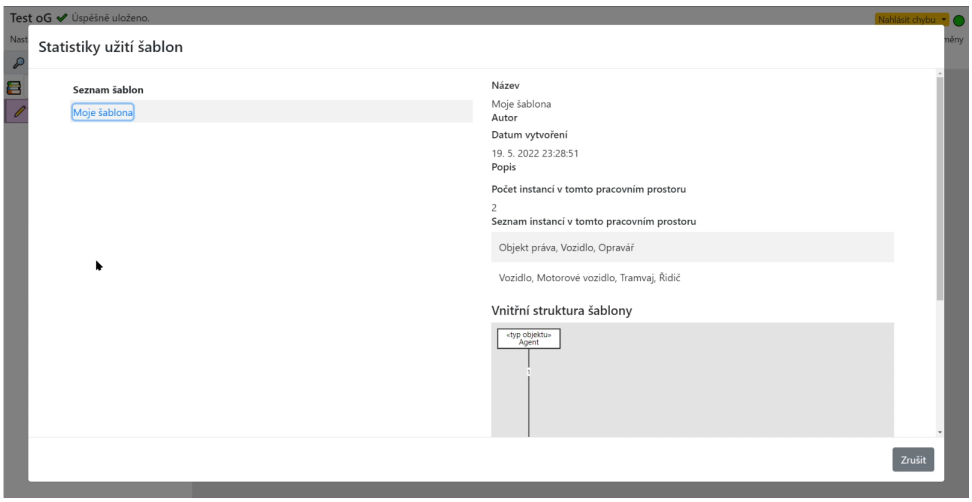


Figure 6.5: The Pattern Statistics dialogue with a pattern selected.

Chapter 7

Evaluation

There are many aspects of the overall solution which need to be evaluated: the language, server implementation and Ontographer implementation. This evaluation will be based on the capability to bring value to the KODI process of enhancing SSP.

7.1 OTTR

Evaluations have already been done with OTTR w.r.t. its ability to conform itself to OntoUML in Chapter 4, which, while demonstrating the ability to emulate OntoUML constructs, indicated some issues. Because of the language's adoption into the core of the package, these issues propagate through the rest of the components.

Namely, it is the inability to have multiple types required for a single parameter. A workaround exists with creating two parameters in which the input is the same, but that is not ideal, given the realities of SSP design. In SSP, the terms have at least one, but ideally two stereotypes: one OntoUML-like stereotype and one Type-like stereotype. This configuration is common there and so an enhancement there would be appreciated if the developers were to include it.

During the implementation of integration of Ontographer (see Section

6.2), it became obvious that even with docTTR, OTTR's annotation pattern library, the description of e.g. parameters and other parts of the patterns and instances were adequate and needed to be supplemented by additional statements.

Another, but quite understandable, limitation is the inability of the language to validate the parameters. The parameters will not accept subjects with a certain type if the parameter forbids it, but the parameter is then willing to expand into semantically incorrect or impermissible statements, even if the limitations are defined right in the pattern definition, for example via owl:Restrictions. However, this would imply the inclusion of a reasoner ran on every expansion, which could be impractical w.r.t. compounded computational complexity.

Lastly, it is the lack of ability to generate IRIs - this is something that software in the middle (in this thesis, that falls on Ontographer) to provide IRIs for subjects. If one were to generate a new instance of a Car, for example, the IRI has to be asked for in a parameter.

7.2 Server

The server, more specifically its basic functions and communication with Fuseki, did not overtly disappoint in its application, but that is perhaps because its application, as prescribed in the thesis, has been rather limited. This is due to the fact that SSP as a sample is rather repetitive and user testing, along with further exploration of possibilities is necessary to fully analyze the potential of such a server.

7.2.1 Algorithms

The algorithms provide further analysis than the rest of the server. A general note on all of the algorithms presented are that heuristic analyses performed during or after the basic algorithm run are necessary for the analysis to become truly useful in everyday use. Currently, due to the general results they return, they at the time of writing, unless the dataset is as repetitive as SSP, function more as a potential enhancement rather than a realized one. As with the server itself, more testing is needed with users and with more datasets to fully appreciate what is (or isn't) possible with these algorithms.

Suggestion algorithm

This algorithm is adept for quickly filtering unusable patterns w.r.t. the model given as an input, however it is as of yet inadequate for automatically filling out the pattern values, since often there are so many combinations of available values, especially when the types for the parameters are not enforced. Currently, it serves well for filtering patterns that can then be filtered even more with additional search criteria.

Refactor/creation algorithm

The usefulness of the algorithm, without factoring Ontographer into the equation, highly varies on the provided dataset and the order of operations. The first testing occurred with a single vocabulary, with which it was able to detect a repeating pattern of

```
ex:Example(?iri, ?rdfType1, ?rdfType2, ?skosPrefLabel, ?rdfsSubClassOf)
```

and variations differing in the number of requested rdf:types or rdfs:subClassOfs. Connections in the form of owl:Restrictions as is the common case in SSP could not be identified. Applying the newly created patterns to refactor other vocabularies, however, bore fruit; since SSP's structures are highly repetitive, the algorithm was able to pick up lots of terms into pattern instantiations. This is useful in the case of when one wants to get started quickly with new patterns without having to define every single pattern from scratch and then apply those patterns to gain insight into the statistics of pattern usage.

7.3 Ontographer

Integration of pattern functionality into the Ontographer user experience was fairly smooth, given the nature of the experience being suitable to creation forms, element visualisations, and database connections already. The user experience of the additions, while functional, is not the most appealing; forms take up almost the entire screen with information that requires explanations before using them for the first time. Nevertheless, in cases of repeated patterns

that exist within vocabularies (as is often the case in SSP), the ability to create new terms and connections quickly greatly enhances the experience when creating a new or redoing an existing workspace.

7.4 Summary

Clearly, there is potential in the technology and implementations presented in this thesis. However, due to various circumstances, from time constrains to the fact that what is developed here presents a new approach to modeling in the Semantic Web that is currently just starting to be explored, the true meaning (or meaninglessness) of this proposal remains to be fully seen in future research. Section 8.1 describes the various enhancements that the project could take moving forward.

Chapter 8

Conclusion

In this thesis, the current processes for data creation, processing and management using the proposals of the Semantic Web have been introduced. The concept of Ontology Design Patterns was presented to hypothetically improve data manipulation and collection with several motivational examples to implement them. Research for the most suitable ODP language was conducted, where OTTR (Reasonable Ontology Templates) was finally selected. This selection was then evaluated with attempts to translate various OntoUML model examples into OTTR, which provided an entry point to try using it for the Czech government's SSP, a repository of vocabularies containing legal terms from various pieces of legislation. Since the government's team (KODI) use a tool suite to produce and publish these vocabularies, a proposal for a service mediating the connection between KODI's RDF datasets and OTTR's own pattern and instance architecture was devised, along with plans to demonstrate the usage of the service in a visual tool that serves KODI to connect terms with semantic connections called OntoGrapher. The implementation of both the server and Ontographer enhancement was then evaluated on SSP, which showed current limitations of the implementation, but also potential to be explored in the concept in the future.

8.1 Future work

Based on the experience gained during the development of this thesis, there are some enhancements of various parts of the solution that are worth discussing in future development. However, as mentioned in Chapter 7, user testing and

different datasets are needed to have a complete picture about the current state of the thesis.

8.1.1 OTTR limitations

As the OTTR language is in current development, there is no doubt that the specification is going to improve. In future versions of OTTR (or perhaps docTTR), a way of naming parameters instead of just giving them descriptions would be welcome. Additionally, there is the single type enforcement only parameter issue mentioned many times throughout the thesis.

8.1.2 Server and Ontographer interactions

One disappointment has been the inability to implement the (or at least some variation of) the pattern creation algorithm, the reason of which is in Section 6.2. Otherwise, the development of heuristics into searching, creating, and refactoring patterns is an obvious example of an enhancement.

8.1.3 Ontographer user interface

The pattern additions have been implemented into Ontographer without disturbing much of the usual user experience. The experience with patterns, however, is full of forms filling the entire screen and lack of clarity in the forms themselves. A redesign of the pattern-specific user interface to a perhaps more visual rather than form oriented experience could prove more pleasant for potential users.

When it comes to specific feature additions, the implementation could go in various ways; there could be an approval system for patterns, where a user could only create draft patterns until their supervisor (or other persons responsible) approve them into production use. In Ontographer, the usage of docTTR metadata is highly limited; pattern descriptions and search options could be extended with examples, parameter descriptions, pattern classifications, subgroups, etc.

Appendix A

Ontology Design Pattern Papers Research Methodology

The chosen methodology from [17] describes the following process to gather research and evaluate their relevancy:

1. Define research questions that the research aims to answer.
2. Conduct a search across chosen resources with search terms relevant to the research questions.
3. Screen the results to discard irrelevant papers.
4. Define a classification scheme which sorts the papers into different, but still relevant, categories.
5. Extract data from each of the categories using the sorted papers.

A.1 Research Questions

As mentioned above, the main topic of interest is current research in the ODP field; more specifically, it is the state of the art with respect to ODP usage methodologies, software implementations, and language proposals. Therefore, the research questions were defined as follows:

1. Which journals feature papers on Ontology Design Patterns?
2. What methodologies and tools for developing, evaluating, reengineering, and deploying ODPs are currently (in the last 5 years) in use?
3. What are the most frequently identified underdeveloped areas in ODP usage?

A.2 Paper search

The website used for searching for the papers is `http://scopus.com` with the following search term:

```
TITLE-ABS-KEY('ontology design pattern' or
'ontology design patterns' or 'ontology template' or
'ontology templates' or 'ontology template') and
ALL(('methodology' or 'approach') or
(('case' or 'case study') or ('example' or 'examples')) or
('tool' or 'tooling'))
PUBYEAR > 2015
```

This search term returned 222 results, of which 182 were discarded as irrelevant, leaving 37 papers to classify.

A.3 Paper content classification

The remaining papers were sorted into two groups of classifications:

- What is being discussed, i.e. a language, a tool, or a methodology.
- In what stage is the thing that is being discussed.

The second classification can be explained like so:

- Overviews - summaries or meta-analyses of solutions (e.g. none are offered by the authors of the overview)
- Proposals - solution description
- Presentations = solution description with a real-life case study (examples / demos for purpose of clarification are not sufficient)

At most 3 papers for each classification combination are selected (as some combinations were not able to be fulfilled with this number of papers).

A.4 Research results

Listed below are the at most 3 papers selected for each classification combination. The papers analyzed in Section 3.2 are from the Language presentation and proposal classes.

A.4.1 Methodology overviews

- Zhu, Q., Kong, X., Hong, S., Li, J. and He, Z. (2015), Global ontology research progress: a bibliometric analysis, Aslib Journal of Information Management, Vol. 67 No. 1, pp. 27-54.
- Presutti V., Lodi G., Nuzzolese A., Gangemi A., Peroni S. and Asprino L. (2016) The Role of Ontology Design Patterns in Linked Data Projects. In: Comyn-Wattiau I., Tanaka K., Song IY., Yamamoto S., Saeki M. (eds) Conceptual Modeling. ER 2016. Lecture Notes in Computer Science, vol 9974. Springer, Cham.
- Kindermann, C., Parsia, B., and Sattler, U. (2019), Comparing Approaches for Capturing Repetitive Structures in Ontology Design Patterns. Workshop on Ontology Design and Patterns 2019.

A.4.2 Methodology presentations

- He, Y., Xiang, Z., Zheng, J., Lin, Y., Overton, J. and Ong, E. (2018). The eXtensible ontology development (XOD) principles and tool imple-

mentation to support ontology interoperability. Journal of Biomedical Semantics.

- Ławrynowicz, A., Potoniec, J., Robaczyk, M. and Tudorache, T. (2017). Discovery of Emerging Design Patterns in Ontologies Using Tree Mining. Semantic Web 9(1):1-28.

- Forssell, H., Lupp, D.P., Skjæveland, M.G. and Thorstensen, E. (2017). Reasonable Macros for Ontology Construction and Maintenance. 30th International Workshop on Description Logics.

A.4.3 Methodology proposals

- Esposito, A. and Di Martino, B. (2015). A rule-based procedure for automatic recognition of design patterns in UML diagrams. Software Practice and Experience 46(7).

- Dorodnykh, N. and Yurin, A. (2019). Towards Ontology Engineering Based on Transformation of Conceptual Models and Spreadsheet Data: A Case Study. Intelligent Systems Applications in Software Engineering (pp.233-247).

A.4.4 Tool presentation

- Yasvi M.A. and Mutharaju, R. (2019), Comparing Approaches for Capturing Repetitive Structures in Ontology Design Patterns. Workshop on Ontology Design and Patterns 2019.

- Jupp, S., Burdett, T., Welter, D., Sarntivijai, S., Parkinson, H. and Malone, J. (2016). Webulous and the Webulous Google Add-On - a web service and application for ontology building from templates. Journal of Biomedical Semantics.

- Souza, A.K., Guizzardi, R., Campos, M.L. and Guizzardi, G. (2016). Extending an Ontology Editor for Domain-related Ontology Patterns Reuse: An Application in the Collaboration Domain. ONTOBRAS.

A.4.5 Tool proposals

- Shimizu, C. (2018). Towards a Comprehensive Modular Ontology IDE and Tool Suite. DC@ISWC.

A.4.6 Language presentations

- Skjæveland, M., Lupp, D., Karlsen, L. and Forssell, H. (2018). Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part I.
- Quirino, G., Barcellos, M., Falbo, R. (2017). OPL-ML: A Modeling Language for Representing Ontology Pattern Languages. ER Workshops, pp. 187-201.
- Zambon, E., Guizzardi, G. (2017). Formal Definition of a General Ontology Pattern Language using a Graph Grammar. 2017 Federated Conference on Computer Science and Information Systems (FedCSIS), 1-10.

A.4.7 Language proposals

A.5 Used prefixes

Here is a list of all prefixes used in the paper:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix ottr: <http://ns.ottr.xyz/0.4/> .
@prefix ax: <http://tpl.ottr.xyz/owl/axiom/0.1/> .
@prefix rstr: <http://tpl.ottr.xyz/owl/restriction/0.1/> .
@prefix ex: <http://example.com/ns#> .
@prefix dct: <http://purl.org/dc/terms/> .
```

```
@prefix gufo: <http://purl.org/nemo/gufo#> .
```

Appendix B

Guide to the attachments

- `readme.txt` - a guide on how to compile and run the server and the client
- `srcClient/` - the client's source code
- `srcServer/` - the server's source code

Appendix C

Bibliography

[1] *About data.europa.eu | data.europa.eu*. URL: https://data.europa.eu/en/about/about-dataeuropaeu (visited on 05/19/2022).

[2] Alice Binder and Petr K˘remen. "OntoGrapher: a Web-based Tool for Ontological Conceptual Modeling". en. In: (), p. 15.

[3] Mihai Codescu, Bernd Krieg-Brückner, and Till Mossakowski. *Extensions of Generic DOL for Generic Ontology Design Patterns*. June 14, 2019.

[4] *Defining N-ary Relations on the Semantic Web*. URL: https://www.w3.org/TR/swbp-n-aryRelations/ (visited on 11/26/2021).

[5] Aldo Gangemi and Valentina Presutti. "Ontology Design Patterns". In: *Handbook on Ontologies*. Ed. by Steffen Staab and Rudi Studer. International Handbooks on Information Systems. Berlin, Heidelberg: Springer, 2009, pp. 221–243. ISBN: 978-3-540-92673-3. DOI: 10.1007/978-3-540-92673-3_10. URL: https://doi.org/10.1007/978-3-540-92673-3_10 (visited on 11/26/2021).

[6] Giancarlo Guizzardi. "Ontological Foundations for Structural Conceptual Models". PhD thesis. Jan. 1, 2005.

[7] Giancarlo Guizzardi, João Almeida, and A Almeida. "Stability Patterns in Ontology-Driven Conceptual Modeling". In: Brazilian Seminar on Ontology Research (Ontobras). Nov. 2020.

[8] Giancarlo Guizzardi et al. "Towards Ontological Foundations for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story". In: *Applied ontology* 10 (Dec. 1, 2015). DOI: 10.3233/AO-150157.

[9] Quinn Hirt, Cogan Shimizu, and P. Hitzler. "Extensions to the Ontology Design Pattern Representation Language". In: *WOP@ISWC*. 2019.

[10] P. Hitzler et al. “Towards a Simple but Useful Ontology Design Pattern Representation Language”. In: *WOP@ISWC*. 2017.

[11] *Implementace strategií v oblasti otevřených dat II - Projekty - www.esfcr.cz*. URL: https://www.esfcr.cz/projekty-opz/-/asset_publisher/ODuZumtPTtTa/content/implementace-strategii-v-oblasti-otevrenych-dat-ii (visited on 05/19/2022).

[12] *KODI - Ministerstvo vnitra České republiky*. cs. URL: https://www.mvcr.cz/clanek/kodi.aspx (visited on 05/20/2022).

[13] Petr Kremen and Martin Necasky. *Improving Discoverability of Open Government Data with Rich Metadata Descriptions Using Semantic Government Vocabulary*. SSRN Scholarly Paper ID 3303148. Rochester, NY: Social Science Research Network, Dec. 18, 2018. DOI: 10.2139/ssrn.3303148. URL: https://papers.ssrn.com/abstract=3303148 (visited on 12/11/2021).

[14] Bernd Krieg-Brückner and Till Mossakowski. “Generic Ontologies and Generic Ontology Design Patterns”. In: *WOP@ISWC*. 2017.

[15] Till Mossakowski et al. “The Distributed Ontology, Modelling and Specification Language – DOL”. In: *WoMO*. 2013.

[16] *OWL - Semantic Web Standards*. URL: https://www.w3.org/OWL/ (visited on 05/20/2022).

[17] Kai Petersen et al. “Systematic Mapping Studies in Software Engineering”. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering* 17 (June 26, 2008).

[18] *Practical Ontology Pattern Instantiation, Discovery, and Maintanence with Reasonable Ontology Templates*. URL: https://www.ottr.xyz/event/2018-10-08-iswc/ (visited on 05/20/2022).

[19] *RDF - Semantic Web Standards*. URL: https://www.w3.org/RDF/ (visited on 05/20/2022).

[20] *Semantic Web - W3C*. URL: https://www.w3.org/standards/semanticweb/ (visited on 05/20/2022).

[21] *Série Praxe otevřených dat v ČR: Data Královéhradeckého kraje a interoperabilita - Portál otevřených dat České republiky*. cs. URL: https://data.gov.cz/%C4%8Dl%C3%A1nky/s%C3%A9rie-praxe-otev%C5%99en%C3%BDch-dat-v-%C4%8Dr-data-kr%C3%A1lov%C3%A9hradeck%C3%A9ho-kraje-a-interoperabilita (visited on 05/19/2022).

[22] Martin G. Skjæveland et al. “Practical Ontology Pattern Instantiation, Discovery, and Maintenance with Reasonable Ontology Templates”. en. In: *The Semantic Web – ISWC 2018*. Ed. by Denny Vrandečić et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 477–494. ISBN: 978-3-030-00671-6. DOI: 10.1007/978-3-030-00671-6_28.

[23] *SKOS Simple Knowledge Organization System - home page*. URL: https://www.w3.org/2004/02/skos/ (visited on 05/20/2022).

[24] *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. URL: https://www.w3.org/Submission/SWRL/ (visited on 11/26/2021).

[25] *The OTTR Template Library.pdf*. Google Docs. URL: https://drive.google.com/file/d/1Mgyq5g3NFSXFXyOJdEbl3V8uMm-_yun8/view?usp=sharing&usp=embed_facebook (visited on 11/26/2021).

[26] Eduardo Zambon and Giancarlo Guizzardi. "Formal Definition of a General Ontology Pattern Language using a Graph Grammar". In: Sept. 24, 2017. DOI: 10.15439/2017F001.

[27] Qiaoli Zhu et al. "Global ontology research progress: A bibliometric analysis". In: *Aslib Journal of Information Management* 67 (Jan. 19, 2015), pp. 27–54. DOI: 10.1108/AJIM-05-2014-0061.