Implementing the OWL Reasoner in the Virtual Knowledge Graph System Ontop

Bendik Mikal Kühlmann

Supervised by Guohui Xiao

Thesis for Master of Science Degree at the University of Bergen, Norway

Thesis date: 3rd June 2024

Year:	2024
Title:	Implementing the OWL Reasoner in the Virtual Knowledge Graph System Ontop
Author:	Bendik Mikal Kühlmann
Supervisor:	Guohui Xiao

Acknowledgements

I would like to extend my greatest thanks to my supervisor, Guohui Xiao, for his guidance. His expertise and valuable insights have been paramount in shaping the direction of this paper.

In addition, I am grateful for the support I have received from the development team of Ontop.

I would also like to extend my thanks to my fellow students Diluxan Sathalingam and Sigurd Loennechen. Over the course of this year, we have collaborated and discussed, which has helped me stay focused and on track.

Lastly, I would like to thank my family and friends for their moral support and encouragement. Their support has helped me stay focused and committed. Without them, this paper would not be possible.

Thank you all for your support.

Abstract

Ontop is a well-known framework designed to facilitate ontology-based data access. It supports two major APIs: RDF4J API for query answering and the SPARQL endpoint, and OWL API for editing ontology and mapping in the Protégé plugin. Ontop internally supports reasoning with the OWL2 QL ontology language using a Directed Acyclic Graph (DAG)-based algorithm, but this reasoning capability is only available through the RDF4J API, but not in OWL API, which limits its ability to handle complex inference tasks in Protégé. To close this gap, in this paper, we propose an enhancement to Ontop by properly implementing the `OWLReasoner` interface in the OWL API. This is achieved by leveraging the internal DAG-based ontology representation. We have validated our implementation through extensive testing and demonstrated the improvements in the Protégé plugin user interface.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

DAG Directed Acyclic Graph.

DB Database.

DL Description Logic.

FOL First-Order Logic.

KB Knowledge Base.

KG Knowledge Graph.

KR Knowledge Representation.

OBDA Ontology-Based Data Access.

OWL Web Ontology Language.

RDBMS Relational Database Management System.

RDF Resource Description Framework.

RDFS Resource Description Framework Schema.

SPARQL SPARQL Protocol and RDF Query Language.

SQL Structured Query Language.

VKG Virtual Knowledge Graph.

Chapter 1

Introduction

In this chapter we define VKG / OBDA systems (section 1.1). We introduce Ontop as such a system and detail the current limitation of the system (section 1.2) before we address how we will tackle the limitation (section 1.3). Lastly, an overview of the document and how it is organized is presented (section 1.4).

1.1 VKG and OBDA

Traditionally, we see data integrated and managed through the use of relational databases, where data is stored in tables that are linked together through primary- and foreign key relations. Access to this data is provided by some sort of relational database management system (RDBMS) - most of which utilize the SQL language for querying and updating the database. *Virtual Knowledge Graph* (VKG), also known in the literature as *Ontology-Based Data Access* (OBDA) [1], is a paradigm for data integration and access that takes a different approach. The paradigm enables the integration of distributed and heterogeneous data sources into a unified graph structure. What this means is that data, which can be stored in different locations and many different formats, is treated as if it were all in one central place or structure. The data can then be accessed through a higher-level representation of the domain.

The VKG paradigm allows us to perform reasoning on the information present in the system. Query answering is the main reasoning task in a system utilizing the VKG approach, which enables us to answer queries - i.e. retrieve information - from the knowledge graph without modifying the data sources [1]. We concern ourselves with a portion of **query answering** in this paper, which will be explained further in section 1.2.

We present some examples of systems [2] utilizing the VKG approach:

- **Ontop** [2]
- Mastro [3]
- Morph [4]
- Stardog[1]

Literature often refers to Ontology Based Data Access (OBDA). However - in recent times - the term Virtual Knowledge Graph (VKG) has been adopted more frequently. The two serve as synonyms to each other, each referring to the same paradigm. Therefore, onward in this paper, emphasis will be put on the term **VKG** as it is a more modern term. Keep in mind, that both terms may be used interchangeably, but they will refer to the same concept.

1.2 Ontop

Ontop is known as a Virtual Knowledge Graph (VKG) system. It is a platform that allows querying relational databases as virtual RDF[2] graphs [5] using SPARQL[3] - a semantic query language. This is done by exposing the contents of any arbitrary relational database as a knowledge graph. These graphs are virtual, meaning the data exposed by the semantic query is never migrated to another database. The data is kept in the respective relational data sources, thereby the term *virtual*. In order to achieve this, the SPARQL queries are rewritten over the knowledge graphs to SQL queries via mappings which map the concepts in the ontology to the data sources. These SQL queries can then be executed over the relational data sources [6].

The task of *query answering*, mentioned in section 1.1, can be split up into two different stages in the Ontop workflow: an off-line stage and an online stage [5]. The off-line stage handles ontology classification and processing of the mappings which map the ontology to the data sources. When this stage is complete we can initiate the online stage, which translates the SPARQL queries into SQL queries that can retrieve data from the relational sources [5]. In this paper, we concern ourselves with the former, *off-line* stage. More specifically, we will focus on the **ontology classification** (see Fig. 1.1) phase as we attempt to reimplement the *reasoner* which classifies ontologies in Ontop.

[1]https://www.stardog.com/
[2]https://www.w3.org/TR/rdf-primer/
[3]https://www.w3.org/TR/sparql11-query/

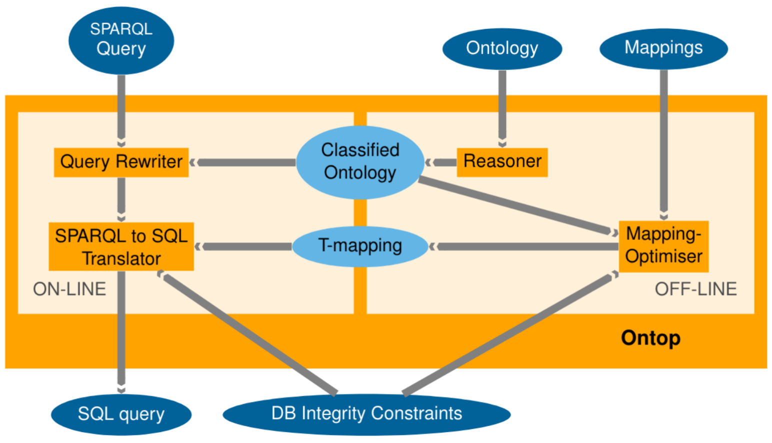


Figure 1.1: Ontop workflow [5]

Development and Use Cases The project was initiated by Diego Calvanese and Mariano Rodriguez-Muro in 2009 in the KRDB research center for Knowledge and Data at the Free University of Bolzano [7]. Nowadays, the community is still in active development of the Ontop project, both from a research perspective and a software development perspective - much of which has been funded by various research projects over the years.

The development of Ontop has been funded by several large research projects[4]. Optique is an example of such a past research project. Current research projects [7] include INODE[5] (data management infrastructures), PACMEL (VKG for process mining) and IDEE (data integration in the energy sector).

Ontop has been deployed in several significant use cases [1], e.g:

- Exploration geological data in the oil & gas industry by Equinor (formerly Statoil).
- Open and enterprise data in the domain of smart city by IBM Ireland.
- Appliance sensor & event data, analytical data, and other miscellaneous data used in machine diagnoses by the technology company Siemens.

Limitation of the OWLReasoner implementation There are two major APIs used in the community: RDF4J and OWL API. RDF4J is mostly used for SPARQL query answering, OWL API is more often used by OWL Reasoners and in Protégé. The RDF4J implementation is complete. However, the OWL

[4]https://ontop-vkg.org/research/#researchers
[5]https://www.inode-project.eu/

API implementation still needs to be implemented properly.

Currently, Ontop has an OWL API `OWLReasoner` compliant implementation[6] in place inside the `QuestOWL` class. However, this is only a dummy implementation where the reasoning methods are delegated to `StructuralReasoner` - a simple reasoner provided by the OWL API. This simple reasoner does have some capabilities, but it can not handle more complex reasoning tasks with complex expressions. We present a simple representation of the `QuestOWL` class and how it inherits functionality from the OWL API in Figure 1.2.

In the current implementation of the `QuestOWL` reasoner, simple reasoning tasks can be performed. However, when using Ontop as a plugin for Protégé, we can identify that for ontologies containing complex expressions and relations, the asserted and inferred hierarchies are identical. Potential inferred information and knowledge are lost during reasoning, something we will attempt to tackle in this paper.

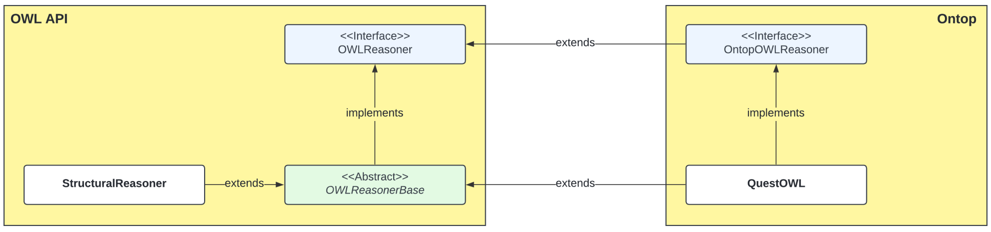

OWL API
<<Interface>> OWLReasoner
implements
StructuralReasoner
extends
<<Abstract>> OWLReasonerBase
extends
extends
Ontop
<<Interface>> OntopOWLReasoner
implements
QuestOWL

Figure 1.2: Simplified inheritance relation between `QuestOWL` and OWL API

Ontop already has another implementation in place which has complex reasoning capabilities for OWL 2 QL / DL-Lite [8]. The `ClassifiedTBox` class is a DAG-based reasoner which exposes more complex reasoning tasks. However, this implementation is used internally and is not ready for use with the OWL API yet. We can expose many reasoning features from this class into `QuestOWL`. This will expose the reasoning capabilities through Ontop's OWL API interface for use with external tools. **Protégé**[7] is an open-source ontology editor and framework, and an example of such a tool. Ontop has a *plugin* implementation for Protégé, which will be our focus area in this paper.

[6] https://github.com/ontop/ontop/issues/138
[7] https://protege.stanford.edu/

1.3 Contribution

The research part of this paper focuses on addressing the limitations mentioned in the previous section. The **objective** is to utilize the already implemented internal DAG-based reasoner in Ontop and see to which extent this component can support OWL API TBox reasoning. To address this limitation, we attempt to implement the methods in the `OWLReasoner` interface of OWL API in `QuestOWL` by utilizing the functionality of the internal reasoner already present in the Ontop codebase. The reasoner `ClassifiedTBox` serves as a "wrapper" over the DAG representations it creates of the ontology and allows us to interact with it by use of various methods. We will utilize these methods of the DAG reasoner, attempting to extract the correct information from the classified TBox, which we then can return through the OWL API of the `QuestOWL` reasoner.

This implementation is not trivial. Some of the OWL API methods can be simply delegated to the DAG in question. Other methods are more involved, with various edge cases to support and more extensive processing of the method arguments. We need to consider these edge cases in order to properly support reasoning. Importantly, we must also test the implementation.

We must also properly handle the different levels of APIs in the Ontop base. Since the DAG is an internal implementation, it uses internal types and classes (internal API). We must handle conversion between the OWL API and the internal API in order to interact with the DAGs in the `ClassifiedTBox` object.

We have implemented reasoning methods of the OWL API, focusing on getting the correct inferred hierarchies to display in Protégé first. We have tested the implementation by writing a multitude of various unit tests, testing small pieces of functionality. In addition, the implementation was compiled into an executable, which allowed us to further test the implementation inside Protégé. The code is currently available at Github[8], and it is scheduled to be included in the next stable release of Ontop.

[8]https://github.com/ontop/ontop/tree/feature/owlreasoner-methods

1.4 Organization of the Document

The paper is organized as follows.

- **Chapter 1**: Serves as an introductory. We cover the Virtual Knowledge Graph paradigm and approach and present some examples of systems using this approach. We present Ontop as such a system, give an overview of the current limitation of its reasoning capabilities, and cover how we will tackle the limitation.
- **Chapter 2**: A deep dive into the aspects needed to grasp the limitation. We give the reader an understanding of how knowledge can be formalized, represented, and used in a reasoning context, and present what is needed to tackle the limitation.
- **Chapter 3**: The implementation is presented in detail. We cover how we have tackled the limitation of Ontop, and present various examples and figures.
- **Chapter 4**: We evaluate our implementation and test it, accompanied with excepts of test cases.
- **Chapter 5**: The paper is summarized with a conclusion, and we discuss what future work is possible.

Chapter 2

Background

This chapter provides an overview of the Ontop framework and the necessary technologies to understand its functionality and objectives.

We explain ontologies (KBs), and how Description Logics (DLs) - specifically DL-Lite$_R$- underpin Ontop's reasoning capabilities. We discuss the Web Ontology Language (OWL), emphasizing OWL 2 QL, which Ontop uses for reasoning (section 2.1).

We outline the Virtual Knowledge Graph (VKG) paradigm (section 2.2) and elaborate on Ontop's architecture (section 2.3), highlighting the role of the OWL API, particularly the `OWLReasoner` interface (section 2.4). We describe the implementation of a DAG-based reasoner in Ontop's `ClassifiedTBox` (section 2.5). This sets the stage for understanding the proposed enhancements to Ontop.

2.1 Ontologies and Knowledge Bases

An ontology can be described as some way of formally representing knowledge of a given domain area. Concepts are modeled with their properties and relations between them in a structured manner by some primitives. These are typically *classes* (sets), *attributes* (properties), and *relationships* (relations among classes). Ontologies are usually specified by some language that allows implementation details and data structures to be abstracted away [9]. Thus, they can be utilized as a high-level, *conceptual* view over data, where a user would not need to familiarize themselves with how the data are structured - we say ontologies lie at the *semantic* level (conceptual separated from data).

Formally, an ontology O (or a Knowledge Base in DL) consists of two components, a TBox T and an ABox A: These components represent intensional and extensional knowledge respectively [10]. The conceptual schema (TBox) and data schema (ABox) combined will model domain knowledge. In the case of a VKG system such as Ontop, the ABox is (virtually) generated by *mappings* - a component that maps data from its sources to the instances of classes and properties in the ontology.

Below, in section 2.1.1, we recall the logic foundations for the ontology, and in section 2.1.2 we describe the W3C[1] standards of the ontology language.

2.1.1 Description Logics and DL-Lite$_R$

Description Logics - or DLs - are a family of languages known as formal knowledge representation languages [11]. Such languages provide a high-level description of the world, which can be used to create intelligent applications. "Intelligent" applications, in this context, mean that the explicit knowledge present in the system can be utilized to find implicit consequences and new knowledge.

Ontop is an example of such a system. Systems like these can be characterized as knowledge-based or knowledge representation (KR) systems [11]. Description Logics describe a domain in terms of some notions to construct an ontology, which in DL terms is referred to as a Knowledge Base (KB). As briefly mentioned in the previous section, an ontology comprises two components - a TBox and an ABox. The basic building blocks to construct these are atomic *concepts* (unary predicates), atomic *roles* (binary predicates), and *individuals* (constants), serving as the main underpinning for modeling ontological structures. DL ontologies also consist of a set of statements, called *axioms*, assumed to be true [12]. These "facts" model the asserted knowledge about the domain.

DLs are fragments of first-order logic (FOL) and come equipped with formal semantics that allow us to construct a precise specification of a DL ontology. These formal semantics allow us to logically deduct and infer additional information [12] from the facts that are stated explicitly in an ontology (or a DL KB).

DLs have a family of lightweight fragments. In this paper, we are focusing on

[1]https://www.w3.org/

Construct	**Syntax**	**Example**
atomic concept	A	Student
role	R	attendsCourse
top concept	$\top$	Thing
bottom concept	$\bot$	Nothing
inclusion	$\sqsubseteq$	Student $\sqsubseteq$ Person
equivalence	$\equiv$	Bachelor $\equiv$ Undergraduate
disjunction	$\sqcup$	Student $\sqcup$ Teacher
conjunction	$\sqcap$	Person $\sqcap$ Employee
negation	$\neg$	$\neg$Employed
exists restrictor	$\exists$	$\exists$attendsCourse.$\top$

Table 2.1: Syntax and examples of commonly used constructors in DL-Lite$_R$

the fragment DL-Lite$_R$, which is a decidable fragment of FOL and serves as the logical underpinning of the Virtual Knowledge Graph approach.

Terminology

We present the terminology and syntax [13] of DL-Lite$_R$.

We use the following letters:

- A and B for atomic *concepts*.
- C and D for concept *description*.
- R and S for *roles*.
- a and b for *individuals*.

We highlight the following symbols:

- $\top$ for the *universal* concept (contains all concepts. All individuals as instances)
- $\bot$ for the *bottom* concept (contains no concepts. No individuals as instances)
- $\sqsubseteq$ for concept *inclusion* (subsumption)
- $\equiv$ for concept *equivalence*
- $\sqcup$ for concept *disjunction*
- $\sqcap$ for concept *conjunction* (union)
- $\neg$ for concept *negation*
- $\exists$ for *existential restriction*

We present some examples of how these constructors can be used in Table 2.1.

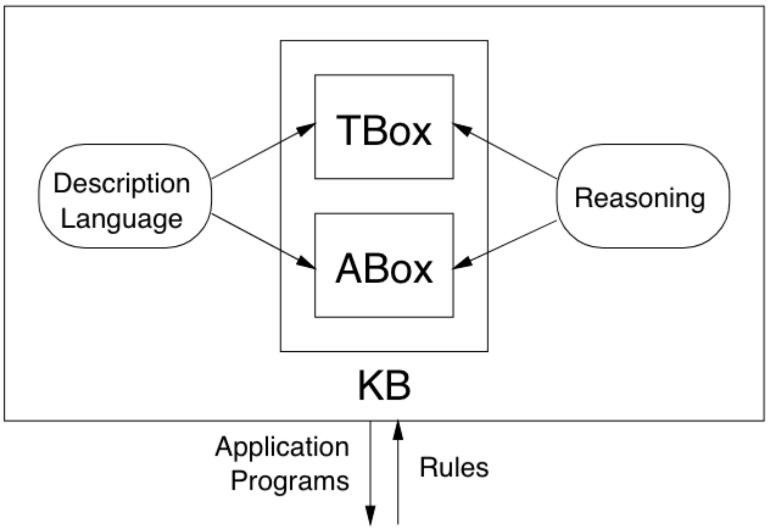


Figure 2.1: Architecture of a KR system comprising a KB, based on DL [11]

Formalism of DL-Lite$_R$

The formal semantics [14] of DL-Lite$_R$ are briefly presented in this section.

- Let N_C, N_R, and N_I be countably infinite sets of concept names, role names, and individuals, respectively.
- Let $N_R = \overline{N_R} \cup \{r^- \mid r \in N_R\}$ be the set of (complex) roles.
- For $R \in \overline{N_R}$, R^- denotes r^- if $R = r \in N_R$, and r if $R = r^-$.

ABox An *ABox* is a finite set of *assertions* of the forms $A(b)$ and $r(b,c)$, with $A \in N_C$, $r \in N_R$ and $b,c \in N_I$.

TBox A Tbox is a finite set of *axioms*, which in DL-Lite$_R$ are *concept inclusions* of the form $B_1 \sqsubseteq (\neg)B_2$, with B_1, B_2 of the form $A \in N_C$ or $\exists R$ with $R \in \overline{N_R}$.

Role inclusion TBoxes may additionally contain *role inclusions* $(DL-Lite_R)$ of the form $R_1 \sqsubseteq (\neg)R_2$ with R_1, $R_2 \in \overline{N_R}$.

Knowledge base A *knowledge base* (KB) consists of a TBox T and ABox A. Recalling from section 2.1, an ontology is formalized as a KB.

$$KB = \langle T, A \rangle$$

Interpretations The semantics of knowledge bases is defined in terms of *DL interpretations* $I = (\Delta^I, \cdot^I)$, where I is a *model* of a KB $K = \langle T, A \rangle$ if it satisfies every axiom in T and assertion in A, and we call K *consistent* if it admits some model. We use Ind(A) for the individuals occurring in A and let I_A be the interpretation with $\Delta^I = \text{Ind}(A)$ such that (i) $\in A^I$ iff $A(c) \in A$ and (ii) $(c,d) \in r^I$ iff $r(c,d) \in A$.

DL-Lite$_R$ ensures that reasoning tasks can be performed efficiently. Ontop supports ontologies and reasoning in the OWL 2 QL ontology language, which is designed such that relational data may be queried through an ontology without altering the data. The basic formalism of OWL 2 QL is supplied by the DL-Lite$_R$ fragment of Description Logics [8]. Reasoning services can be performed on both the TBox and the ABox, as presented in Figure 2.1 which outlines the components of a KR system. This paper focuses on the *reasoning* component. More specifically, we concern ourselves with **TBox reasoning**.

Considering a TBox T, the reasoning tasks [10] are:

- A concept C is *satisfiable* if there exist a model I of T such that C^I is not empty. We say that I is a model of C.
- A concept C is *subsumed* by a concept D if $C^I \sqsubseteq D^I$ for every model I of T. We write that $T \models C \sqsubseteq D$.
- Two concepts C and D are *equivalent* if $C^I = D^I$ for every model I of T. We write that $T \models C \equiv D$.
- Two concepts C and D are *disjoint* with respect to T if $C^I \sqcap D^I = \emptyset$ for every model I of T.

We provide some examples of OWL 2 QL ontologies formalized with DL-Lite$_R$ axioms (recall the constructors in Table 2.1):

Example 1 *TBox Simple*

$$C \equiv D$$
$$B \sqsubseteq A$$
$$C \sqsubseteq A$$

The example includes the classes A, B, C, and D. The axioms entail that C and D are equivalent to each other and that B and C are subsumed by A.

Example 2 *TBox Complex*

$$
\begin{aligned}
D &\sqsubseteq A \\
A &\sqsubseteq \exists r1.\top \\
\exists r1.\top &\sqsubseteq \exists r2.\top \\
\exists r2.\top &\sqsubseteq B \\
C &\sqsubseteq B
\end{aligned}
$$

The example includes the classes A, B, C and D, and the object properties r1 and r2. The axioms all entail some relation of subsumption, however, they also include some complex expressions with the existential restriction constructor.

2.1.2 Web Ontology Language

Web Ontology Language - often abbreviated by the acronym OWL - is a semantic web language designed to define ontologies on the web, based on Description Logic. OWL allows us to model and represent knowledge about various domains of interest concisely and is used to create ontologies, verify the consistency of such ontologies, as well as infer new knowledge from existing knowledge. It is a language that was designed to be compatible with the World Wide Web, thus it may also refer to online resources in addition to local domain knowledge [15].

OWL is available in two versions. The first proposed version OWL 1[2] became a W3C recommendation in the year 2004. It has since been superseded by OWL 2[3], an extension of the former OWL 1 with new functions and features [16]. This has been the current standard since it became a W3C recommendation in 2009 [17] - with the second edition published in 2012.

Web Ontology Language	**Description Logic**	**First-Order Logic**
individual	individual	constant
class	concept	unary predicate
property	role	binary predicate

Table 2.2: Comparison of terminology in OWL, DL, and FOL

An OWL Ontology describes a domain in terms of *classes*, *properties* and *individuals* (see Table 2.2 for DL equivalent terms), and can include rich descriptions of characteristics of those objects or Web resources [9]. As with

[2]https://www.w3.org/TR/owl-features/
[3]https://www.w3.org/TR/owl2-overview/

DLs, OWL comes in different variants for different use cases. We refer to these variants as “profiles”. These profiles sacrifice some expressive power for the efficiency of reasoning. Less expressive means increased efficiency. We summarize the profiles [17] of OWL 2:

- OWL 2 QL: For applications that use large volumes of instance data (*individuals*). Query answering is the most important reasoning task.

- OWL 2 EL: For applications utilizing ontologies that contain large numbers of *properties* and/or *classes*. Basic reasoning tasks can be performed in polynomial time with respect to the ontology size.

- OWL 2 RL: For applications requiring scalable reasoning without sacrificing too much expressive power.

In this paper, we will focus on the profile **OWL 2 QL** as this is the one Ontop implements for its reasoning services. This profile of OWL is formalized in the Description Logic language DL-Lite$_R$.

2.2 VKG Framework

The VKG approach for data integration presents an alternative to the relational model by combining the following three main ideas [1]:

- **Data virtualization**: Achieved by separating the end-user from the data sources. The data still sits in its respective sources, however, it is accessed through a higher-level representation of the domain.

- **Knowledge**: Domain knowledge enriches the data semantically, capturing important aspects of the data otherwise possibly lost. This enables one to perform inference on the enriched data, which can lead to new knowledge being derived from the explicitly stated knowledge. The OWL2 QL ontology profile [17], with the DL-Lite Description Logic [8] as the formal background, is the most popular ontology language for VKG.

- **Graph structure**: The data is structured in a graph form, where nodes represent domain data concepts, and edges represent properties and relations of these concepts. A graph structure is flexible and quite efficient for modeling entity relations. In addition, such a structure enables the use of various graph algorithms for analysis and manipulation.

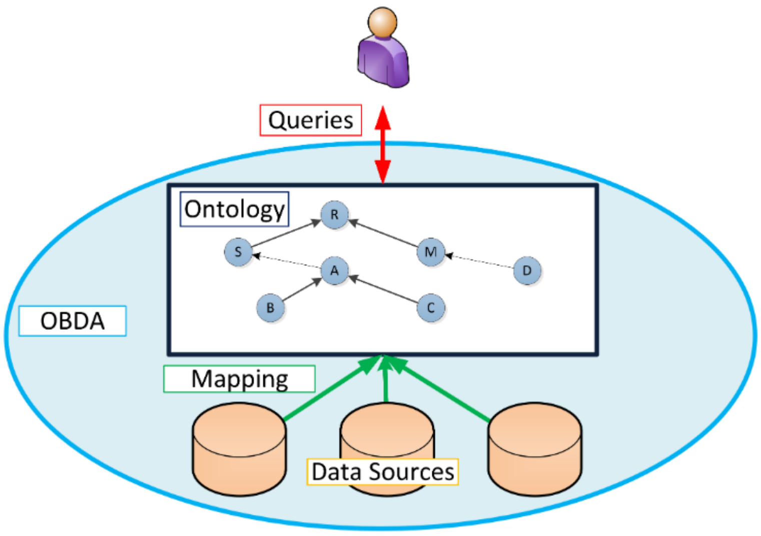


Figure 2.2: VKG / OBDA framework [10]

Framework

A VKG system comprises multiple components, or layers, which enable the ideas mentioned briefly earlier. These are needed for the system to function and are presented in Figure 2.2. Since the Virtual Knowledge Graph paradigm is a way of accessing data, the main functionality of such a system is query answering. The flow of information starts from the user, who makes a query (i.e. requests something from the system). The query is done on the ontology, transformed through mappings, and data is retrieved. We summarize the layers of a VKG system:

Conceptual layer (ontology) Represents the domain knowledge and schema that describes the data at a high level. Provides a unified, abstract view of the data. Often formalized as an RDF(S) or OWL ontology. We call this the *intensional* knowledge - or a DL *TBox*. In this paper, we concern ourselves with the *conceptual* layer (the **ontology**) of a VKG system.

Mapping layer Contains rules that map the high-level ontology terms to the underlying data sources. Ensures that data from different sources can be accessed and integrated consistently.

Data layer The actual storage where data resides, typically relational databases. CSV files or other data formats are also supported. Provides the foundational data that the VKG system queries and integrates.

Query layer The interface through which users interact with the VKG system, typically using a query language like SPARQL. Allows users to formulate queries against the ontology, which are then rewritten and executed against the data sources.

Formalism

A VKG system [10] is a triple

$$VKG = \langle T, S, M \rangle$$

where:

- T is a DL TBox. In the case of this paper, T is a DL-Lite$_R$ TBox (*intensional knowledge*).
- S is a relational database representing the sources.
- M is a set of mapping assertions between S and T, each one of the form

  $$\phi(\vec{x}) \rightsquigarrow \psi(\vec{x})$$

  where:

  - $\phi(\vec{x})$ is a query over S, returning tuples of values for $\vec{x}$.
  - $\psi(\vec{x})$ is a query over T, whose free variables are from $\vec{x}$.

M is used to populate the elements of T with the data in S [10]. The source query $\phi(\vec{x})$, which retrieves values from S, is taken to generate triple patterns that refer to concepts in T, by using $\psi(\vec{x})$.

2.3 Ontop

Ontop is an Open-Source software system for Virtual Knowledge Graphs released under the Apache 2.0[4] license. More specifically, it is a platform used to query relational databases as Virtual RDF Knowledge Graphs using SPARQL. Platform, in this regard, refers to Ontop being available as a set of software, with the ability to connect to external tools and processes. We divide Ontop's architecture into four layers [5], which we detail in the following.

[4] `https://www.apache.org/licenses/LICENSE-2.0`

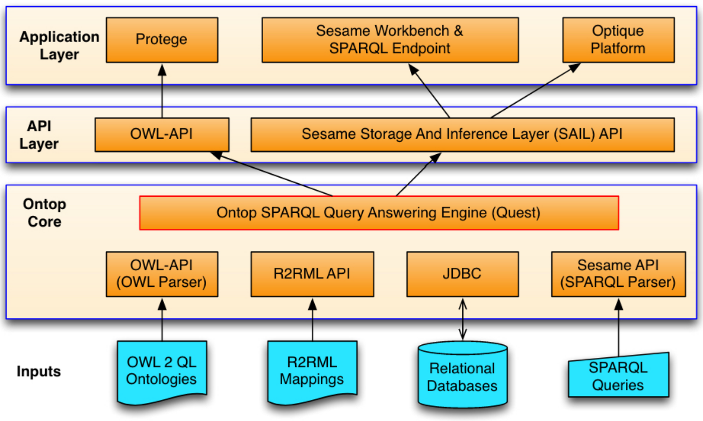


Figure 2.3: Ontop architecture [5]

Inputs

The input layer for the VKG system is where we handle domain-specific data. This data serves as input to the VKG system. The inputs are (*i.*) ontology, (*ii.*) mappings, (*iii.*) data sources, and (*iv.*) queries. As a VKG system, Ontop supports the most important W3C recommendations[5] for the Semantic Web and linked data: OWL, R2RML, SPARQL, SWRL and SPARQL OWL 2 QL regime [7]. Ontop also has major support for commercial and free databases [18]. Recalling the components of a VKG system in section 2.2, Ontop supports the widely used standards in the following manner[6]:

i. Ontop fully supports OWL 2 QL and RDFS ontologies, while also being extended to support the linear recursive fragment of SWRL (Semantic Web Rule Language). OWL 2 QL is based on the $DL-Lite$ family [18] of lightweight description logics. This subset language guarantees query rewriting - i.e. decideability.

ii. Ontop supports two mapping languages: its own native mapping language and the RDB2RDF Mapping Language (R2RML) [18]. A tool is included in Ontop for converting between these two mapping languages.

iii. Most standard relational database engines are supported in Ontop via the Java Database Connectivity (JDBC) API[7] - providing a universal way of accessing data. This also enables Ontop to support federated databases,

[5]https://www.w3.org/TR/?status%5B0%5D=standard
[6]https://ontop-vkg.org/guide/#main-features
[7]https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

allowing for multiple types of data sources such as XML and CSV files, or even live Web Services [5]. Ontop supports the following main database systems[8]: PostgreSQL, MySQL, MariaDB (since 5.0.0), SQL Server, Oracle, DB2, Snowflake (since 5.0.0), Databricks (since 5.0.0), Google BigQuery (since 5.0.2), AWS Redshift (since 5.0.2), DuckDB (since 5.0.2), and AWS DynamoDB (since 5.1.0).

iv. Ontop supports the majority of SPARQL 1.1 features[9], including the main SPARQL aggregation functions (since 4.0.0) and GeoSPARQL functions (since 4.1.0).

Together, *i.*, *ii.* & *iii.* form a Virtual Knowledge Graph, and *iv.* is used to query and interact with the knowledge graph.

Ontop core

At the core of Ontop is the SPARQL engine *Quest* [18], which is in charge of rewriting SPARQL to SQL. It handles query translation, optimization, and execution as queries over the virtual RDF graph and ontology are transformed into queries over the data sources.

$$VKG_{sparql} \rightsquigarrow DB_{sql}$$

API layer

APIs exposing standard Java interfaces to users of the system. System developers can use Ontop as a *Java library* [18]. Two widely-used Java APIs are implemented by Ontop: (*i.*) OWL API and (*ii.*) RDF4J / Sesame.

i. OWL API is a reference implementation for creating, manipulating, and serializing OWL ontologies. The `OWLReasoner` Java interface is extended in Ontop to support SPARQL query answering. We will be interacting directly with the **OWL API** in this paper, specifically through the `OWLReasoner` interface.

ii. RDF4J (formerly Sesame) is a framework for processing RDF data. The OpenRDF Sesame project was rebranded[10] in May 2016 for the purpose

[8] https://ontop-vkg.org/guide/#main-features
[9] https://ontop-vkg.org/guide/compliance.html
[10] https://projects.eclipse.org/projects/technology.rdf4j

of migrating[11] the library to the Eclipse foundation. Ontop implements the Sesame Storage And Inference Layer (SAIL) API supporting inferencing and querying over relational databases [5], as well as the newer RDF4J for internal handling of RDF data.

Application layer

Applications that allow end-users to execute SPARQL queries over databases. Ontop can be used as (*i.*) a command-line interface (CLI), or (*ii.*) its functionality can be exposed via APIs to support external applications such as Protégé or SPARQL Endpoint [18].

i. Ontop ships a shell script and a bat file. These expose the core functionality, along with several utilities, through the command-line interface (CLI). Without a graphical interface, but allows for quick set-up. Easily test execution, query or materialize[12].

ii. Ontop's API support allows it to be easily integrated. Through the OWL API, Ontop implements a plugin [18] for Protégé[13] - an open-source ontology editor and framework. Along with a graphical user interface, the functionality of Ontop is exposed for use in this external application.

    SPARQL endpoint is another application facilitated by the use of such interfaces. It is a deployable HTTP endpoint that allows for query answering. Available either through the CLI or as a Docker[14] image. End-points such as these are a popular and standard way for applications to exchange data between them. Data is transmitted as payload through the HTTP protocol and received in a standardized format. This enables Ontop to be integrated into most existing applications smoothly.

Ontop implements the OWL API through its `OWLReasoner` Java interface. The result of reasoning from query answering is exposed through this API so that applications such as Protégé can interact with the result. This can be seen in the transition from the Ontop core through its API to the application layer in Figure 2.3.

In this paper, we concern ourselves with the **plugin** that comes with Ontop for use with *Protégé*. This plugin provides Protégé with Ontop's reasoning

[11] https://rdf4j.org/documentation/reference/migration/
[12] https://ontop-vkg.org/guide/cli.html
[13] https://protege.stanford.edu/
[14] https://hub.docker.com/r/ontop/ontop

capabilities. As shown in Figure 2.4, Protégé includes a visualization of the tree of the hierarchy of classes and properties. There are two trees, the asserted one and the inferred one. The asserted one follows the structure of the input ontology directly, and the inferred one uses the `OWLReasoner` to show a (potentially richer) hierarchy.

Currently, `ontop-protege-plugin` uses the dummy OWL reasoner, which means the asserted and the inferred ones are identical. We aim to improve the situation by implementing the `OWLReasoner` used in the `ontop-protege-plugin` properly.

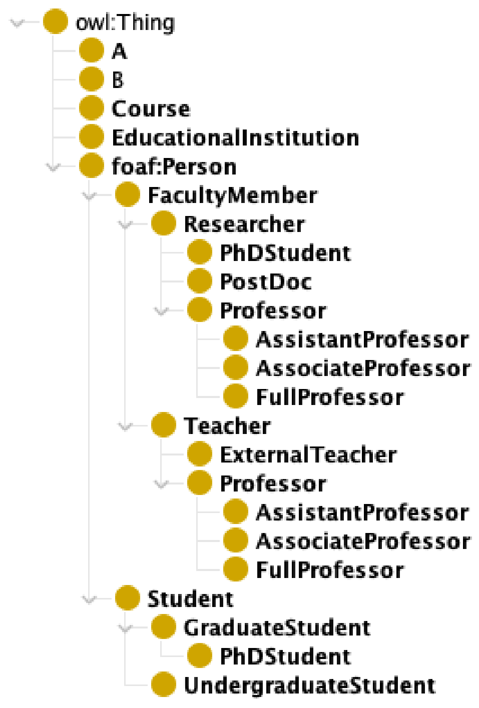


Figure 2.4: Example class hierarchy in Protégé

2.4 OWL API

The OWL API is a Java API and reference implementation for creating, manipulating, and serializing OWL Ontologies [19]. It is open source and available under either the LGPL or Apache Licenses. The API includes the following components[15]:

[15]`https://owlapi.sourceforge.net/`

- **Programming interface**: Interface for OWL 2 and an efficient in-memory reference implementation.
- **Reasoner interface**: Interfaces for working with and implementing compliant external reasoners.
- **Parsing & writing**: Supports RDF/XML, OWL/XML, OWL Functional Syntax, and TTL (Turtle).
- **Parsing**: Support KRSS and OBO Flat file format.

In this paper, we concern ourselves with the **reasoner interface** of the OWL API. In particular, the `OWLReasoner` Java interface provides extensive detailed documentation and functionality relating to the process of reasoning with OWL ontologies [19]. The interface allows Ontop to support common reasoning tasks such as consistency checking, computation of class or property hierarchies, and axiom entailment.

As of the date of this thesis, version 4.3.1 and 5.1.0 are the current main versions of the OWL API[16]. The main codebase of Ontop is targeting OWL API 5, but the Protege plugin requires OWL API 4. Internally, Ontop is using the shading[17] strategy to make sure different modules use the right version of Ontop.

OWLReasoner Interface

As mentioned briefly, the OWL API provides access to common reasoning tasks such as checking an ontology's consistency, computing class- or property hierarchies, as well as axiom entailment. Implementation of the semantics of a reasoner is not a trivial task. The OWL API makes this job less cumbersome by separating reasoning functionality into manageable pieces. Reasoning functionality is provided to us by the `OWLReasoner` Java interface [19]. We present some of the reasoning methods provided by the OWL API `OWLReasoner` interface:

- `getSubClasses(ce, direct)`: Gets the set of named classes that are the strict subclasses of the specified class expression *ce* with respect to the reasoner axioms. The boolean value *direct* dictates whether all

[16]https://github.com/owlcs/owlapi/wiki
[17]Involves packaging the classes of one or more dependencies within a JAR file under a different namespace to avoid conflicts with other dependencies.

descendants or only the immediate descendants should be retrieved. Returned as a `NodeSet`.

- `getEquivalentClasses(ce)`: Gets the set of named classes that are equivalent to the specified class expression *ce* with respect to the set of reasoner axioms. Returned as a Node.
- `getSubObjectProperties(pe, direct)`: Gets the set of simplified object property expressions that are the strict (potentially direct) subproperties of the specified object property expression *pe* with respect to the imports closure of the root ontology. The boolean value *direct* dictates whether all descendants or only the immediate descendants should be retrieved. Returned as a `NodeSet`.
- `getObjectPropertyDomains(pe, direct)`: Gets the named classes that are the direct or indirect domains of the property expression *pe* with respect to the imports closure of the root ontology. Returned as a `NodeSet`.
- `getDisjointDataProperties(pe)`: Gets the data properties that are disjoint with the specified data property expression *pe*. The data properties are returned as a `NodeSet`.

2.5 DAG for DL-Lite$_R$ Reasoning

The `ClassifiedTBox` implementation in `QuestOWL` is a DAG-based TBox reasoner. It constructs optimized directed acyclic graph (DAG) representations of an ontology, with relation to the ontology axioms. Graphs are a very natural way of modeling real-world relations and concepts and therefore are a good fit for knowledge-based reasoners. The following section covers Directed Acyclic Graphs (DAG), as well as how the DAGs in `ClassifiedTBox` are implemented.

A graph consists of a set of vertices, or nodes, and a set of edges. Pairs of vertices are connected via edges. We can define a graph G as containing a set of vertices V and edges E. The set of edges in set E is each a pair of vertices (v_1, v_2), representing a connection between them [20].

$$G = (V, E)$$

A *directed* graph (digraph) is a graph where the edges E have a given direction of traversal, usually indicated with an arrow. A *cycle* in a graph is a path in which only the first and last vertices are equal. We call a directed graph without directed cycles a *directed acyclic graph* - or DAG.

EquivalencesDAG

`ClassifiedTBox` implements three DAGs for its reasoning capabilities. It constructs a class DAG, an object properties DAG, and a data properties DAG from a common `EquivalencesDAG` object. For each DAG, the vertices V of the DAG are sets of equivalences, while the edges E form the minimal set where transitive and reflexive closures coincide with the transitive and reflexive closures of the ontology graph. The DAGs contain properties that map the different equivalences. Equivalence objects act as nodes that store all nodes which are deemed *equivalent*, with one acting as a representative for the other, equivalent nodes. This helps to optimize the DAG, removing redundant nodes.

Breadth-First Search

Consider a graph $G = (V,E)$ [10].

1. Start a list by choosing a vertex $v \in V$
2. Add all the vertexes adjacent to v to the list and keep track of them.
3. Consider all the next vertexes on the list. Add each of the vertexes to the list and keep track of all their adjacent vertexes.
4. When the last vertex with no new adjacent vertex is reached, terminate the process.

Our implementation includes using the BFS algorithm, which will be covered in more detail in section 3.2.3.

Chapter 3

Implementation of the OWLReasoner Interface in Ontop

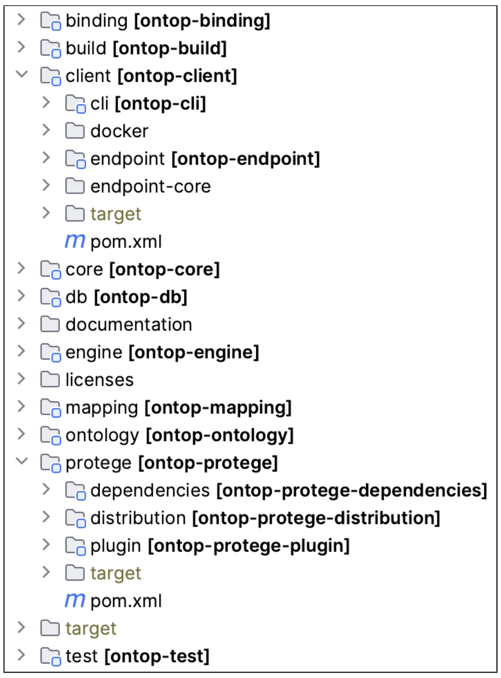
The purpose of this thesis was to properly implement the reasoning methods of `OWLReasoner` interface in the `QuestOWL` class by utilizing an already implemented DAG-based TBox reasoner, replacing the dummy reasoner implementation which was present.

Recall that `ClassifiedTBox` is a DAG-based TBox reasoner which is made available to `QuestOWL` as a field variable. When `QuestOWL` is instantiated, the TBox reasoner is *saturated*. It classifies the different classes & object- and data properties, as well as generates Directed Acyclic Graph (DAG) representations of classes and properties.

Currently, QuestOWL implements `StructureReasoner` which is a simple, incomplete reasoner and has been the dummy reasoner implementation of `QuestOWL`. It is part of the OWL API, and therefore a fitting temporary solution for Ontop to achieve OWL API compliance for its reasoning capabilities.

In this chapter, we go through in detail how the OWL API methods were implemented in the `QuestOWL` Java class of the Ontop project. In particular, what was done to replace `StructuralReasoner`, and how `ClassifiedTBox` can be used to expose its capabilities to `QuestOWL`.

Before introducing the new implementation of QuestOWL (section 3.2), we start by describing the structure of the codebase of Ontop (section 3.1).



```
> binding [ontop-binding]
> build [ontop-build]
v client [ontop-client]
    > cli [ontop-cli]
    > docker
    > endpoint [ontop-endpoint]
    > endpoint-core
    > target
      m pom.xml
> core [ontop-core]
> db [ontop-db]
> documentation
> engine [ontop-engine]
> licenses
> mapping [ontop-mapping]
> ontology [ontop-ontology]
v protege [ontop-protege]
    > dependencies [ontop-protege-dependencies]
    > distribution [ontop-protege-distribution]
    > plugin [ontop-protege-plugin]
    > target
      m pom.xml
> target
> test [ontop-test]
```

Figure 3.1: Ontop project structure

3.1 Codebase of Ontop

The Ontop codebase is hosted on the popular repository management site GitHub[1], and uses the version control system Git to handle collaborative software development. Development of Ontop is carried out openly in GitHub, where our implementation is developed under a separate branch[2] of the project. As of the date of this thesis, the most recent stable release of Ontop is version 5.1.2 - released January 17, 2024[3].

The code of Ontop is organized as a multi-module Java Maven project. Each module is in charge of some functionality to the system, each with its own dependencies and *pom* file. This allows us to only test and build out the modules we need. The codebase is split into mainly two parts (modules) which act as separate entry points for separate usages. These are `ontop-cli` and `ontop-protege`, and are the two main ways of running Ontop. We concern ourselves with the latter.

As we can see in Figure 3.1, the `ontop-protege` module also has its own

[1] https://github.com/ontop/ontop
[2] https://github.com/ontop/ontop/tree/feature/owlreasoner-methods
[3] https://github.com/ontop/ontop/releases

set of dependencies `ontop-protege-dependencies`. This effectively works as a "black box", where a *shaded jar* will merge the required dependencies into one dependency, essentially creating one uber jar. This allows it to avoid the consequences of having multiple dependencies of different versions in the codebase. This is important since Protégé utilizes OWL API 4 while Ontop uses OWL API 5.

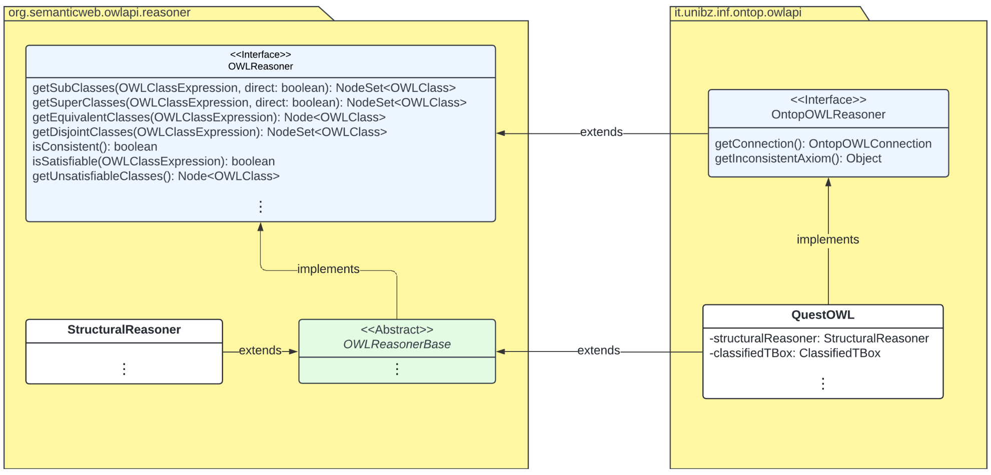

org.semanticweb.owlapi.reasoner
<<Interface>>
OWLReasoner
getSubClasses(OWLClassExpression, direct: boolean): NodeSet<OWLClass>
getSuperClasses(OWLClassExpression, direct: boolean): NodeSet<OWLClass>
getEquivalentClasses(OWLClassExpression): Node<OWLClass>
getDisjointClasses(OWLClassExpression): NodeSet<OWLClass>
isConsistent(): boolean
isSatisfiable(OWLClassExpression): boolean
getUnsatisfiableClasses(): Node<OWLClass>
⋮
implements
StructuralReasoner
⋮
extends
<<Abstract>>
OWLReasonerBase
⋮
it.unibz.inf.ontop.owlapi
extends
<<Interface>>
OntopOWLReasoner
getConnection(): OntopOWLConnection
getInconsistentAxiom(): Object
implements
QuestOWL
-structuralReasoner: StructuralReasoner
-classifiedTBox: ClassifiedTBox
⋮
extends

Figure 3.2: Inheritance relation between `QuestOWL` and OWL API

3.2 QuestOWL

QuestOWL implements the OWL reasoner interface through `OWLReasonerBase`. Figure 3.2 illustrates how QuestOWL inherits its methods and functionality from the OWL API. Visible in the figure is also how `QuestOWL` includes `StructuralReasoner` and `ClassifiedTBox`. Since Ontop already models OWL 2 QL ontologies internally, it has its own API which is based on RDF4J data structures. This API models most counterparts of OWL API objects. This means that, in order to interact with the internal workings of Ontop, we must convert values to and from the OWL API and the Ontop API, since the OWL API is implemented on top of the Ontop API.

$$\texttt{OWL API} \rightarrow \texttt{Ontop API} \rightarrow \texttt{RDF4J}$$

To properly implement our reasoner, we must interact with the different APIs and their data structures. To perform this, we utilized "factories"[4] which were

[4]A design pattern that simplifies object creation.

provided to us by the OWL API and RDF4J framework. These object factories allowed us to easily create and instantiate the various objects we needed.

```
    OWLDataFactory owlDataFactory = getOWLDataFactory();
```

```
    RDF rdfFactory = new RDF4J();
```

These factories were used plentifully in our implementation and thus will be mentioned in the upcoming sections.

3.2.1 Ontop API and OWL API

We detail some of the OWL API objects and the Ontop API equivalent objects in this section.

Class expression A class expression in the OWL API represents an OWL 2 QL specification class expression. The interface covers both named and anonymous classes. It contains many helpful methods, a subset of which are represented below. These can convert the expression to a class, or check if the class expression is the built-in class `owl:Thing` or `owl:Nothing`. We utilize these methods in our implementation.

```
  public interface OWLClassExpression
    extends OWLObject, OWLPropertyRange, SWRLPredicate {
      ClassExpressionType getClassExpressionType();
      OWLClass asOWLClass();
      boolean isOWLThing();
      boolean isOWLNothing();
    ...
    }
```

Ontop models the OWL 2 QL class expression from extending from another internal `Description` interface, which is an interface that represents partially ordered classes. This interface contains methods for top ($\top$) and bottom ($\bot$), which extends to all types of expression objects.

```
  public interface ClassExpression extends DescriptionBT {
    // NO-OP
  }

  public interface DescriptionBT extends Description {
    boolean isTop();
    boolean isBottom();
  }
```

Equivalences Important for optimizing the DAG, objects that model entities that are equivalent to each other are implemented in both Ontop and OWL API. The OWL API models this as a `Node`, which represents a set of entities. The `Node` interface inherits from the Java `Iterable` type, enabling us to perform various procedures on it. Ontop has a counterpart `Equivalences` object, also inheriting from `Iterable`. The equivalent entities are in this implementation represented by a *representative* and a set of *members*. Since the set of entities are equivalent, and therefore refer to the same thing, we can instead choose one entity that represents the whole set of entities (the representative).

```
public class Equivalences<T> implements Iterable<T> {
  final private ImmutableSet<T> members;
  private T representative;
  private boolean isIndexed;
...
}
```

The `Node` interface is accompanied by the `NodeSet` interface, representing a set of nodes. Ontop does not have a counterpart for a set of `Equivalences`. We instead use Java APIs like `ImmutableSet` and `Stream` to process and handle such sets.

```
public interface Node<E extends OWLObject> extends
    Iterable<E> {
  boolean isTopNode();
  boolean isBottomNode();
  Set<E> getEntities();
...
}

public interface NodeSet<E extends OWLObject> extends
    Iterable<Node<E>> {
  boolean containsEntity(@Nonnull E e);
  Set<Node<E>> getNodes();
...
}
```

IRI International Resource Identifiers (IRIs) are used to uniquely identify resources, such as classes, properties, and individuals in an ontology. Ontop utilizes the RDF4J framework to handle these identifiers while the OWL API utilizes its own implementation.

3.2.2 Helper Methods

To facilitate readable and maintainable code, and to make development easier, multiple helper methods were created. Helper methods are internal, `private`[5], methods which are further used again in other methods. Procedures that were done repeatedly at multiple places in the code were extracted into their own method and given a descriptive name. The majority of these methods are conversion methods that convert object types to and from Ontop's internal types and the OWL API-provided types. Because the `QuestOWL` class inherits from the OWL API `OWLReasoner` interface, every reasoning method of this interface has its parameters and return types from this API. Since the task was to utilize the `ClassifiedTBox`, which has its parameters and return types from Ontop's internal code, these conversion methods proved important to implement. In addition, helper methods for handling Ontop's `Equivalences` object also proved useful. These include functions for retrieving equivalence objects from the `ClassifiedTBox` DAGs, as well as the more complex task of converting these equivalences back to OWL API.

Conversion of classes, data properties, and object properties

The main object types of the reasoning methods of the `OWLReasoner` interface are those of `OWLClass`, `OWLDataProperty`, and `OWLObjectPropertyExpression`. For each of these, Ontop has corresponding objects for internal use - we concern ourselves with conversion methods to and from these objects first.

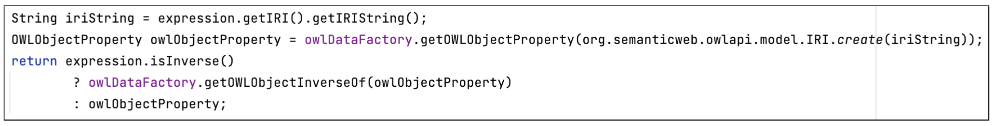
```
private OClass owlClassAsOClass(OWLClass owlClass) {
    String iriString = owlClass.getIRI().toString();
    IRI iri = rdfFactory.createIRI(iriString);
    return classifiedTBox.classes().get(iri);
}

private OWLClass oClassAsOWLClass(OClass oClass) {
    String iriString = oClass.getIRI().getIRIString();
    return owlDataFactory.
        getOWLClass(IRI.create(iriString));
}
```

`OWLClass` represents a *class* in the OWL 2 specification - Ontop represents this through `OClass`. Converting from `OWLClass` to `OClass` requires the following steps. First, we must retrieve a string representation of the `IRI` of the class. This string can then be provided to the `rdfFactory`, an instance of the

[5]A method that can only be accessed and used within the class it is defined in, hiding it from other classes.

RDF4J framework (as mentioned at the start of section 3.2), to create an `IRI` object. This `IRI` object is used to retrieve the `OClass` with that identifier from `classifiedTBox.classes()`.

The process of going from `OClass` to `OWLClass` is quite similar. The string representation of the `IRI` of the class is retrieved, which again must be used to construct an `IRI` object. However, since Ontop and the OWL API concern themselves with `IRI` objects from two different APIs, we must take caution to use the correct one. This `IRI` object can then be passed to `owlDataFactory.getOWLClass(IRI iri)`, provided by the OWL API, to construct the corresponding `OWLClass`.



```
String iriString = expression.getIRI().getIRIString();
OWLObjectProperty owlObjectProperty = owlDataFactory.getOWLObjectProperty(org.semanticweb.owlapi.model.IRI.create(iriString));
return expression.isInverse()
        ? owlDataFactory.getOWLObjectInverseOf(owlObjectProperty)
        : owlObjectProperty;
```

Figure 3.3: Converting an Ontop object property expression to its OWL API counterpart in `QuestOWL`

We now present a more complex conversion method. When we convert an object property expression, we must cover the case if the expression is an *inverse* type of expression. An inverse object property expression is used to refer to the inverse of a property, without actually naming the property. In the OWL API, this is modeled by the `OWLObjectInverseOf` interface, which extends from the `OWLObjectPropertyExpression` interface. A snippet of the code used to convert an object property expression from the Ontop API to the OWL API is provided in Figure 3.3. Here we can see how we first get the object property, before checking if it is an inverse property and then retrieving the correct expression.

Checking anonymous nodes

It proved important to create a method to check if all members of an equivalence set were anonymous nodes. This method was then further implemented in the BFS implementation needed to handle *direct* relations (we cover this implementation in detail in section 3.2.3). The `isAllAnonymous` method was implemented with *overloading*[6]. Two constructors were developed:

```
    return equivalences.getMembers()
        .stream()
        .noneMatch(x -> x instanceof OClass);
```

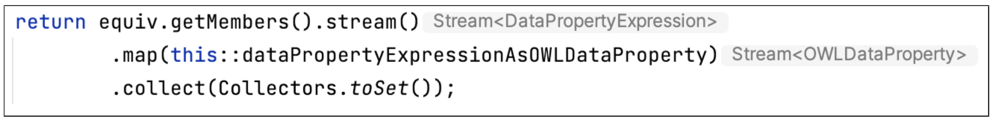
[6]Multiple constructors of the same name with different parameters.

```
    return equivalences
        .stream()
        .allMatch(this::isAllAnonymous);
```

The method utilizes the Java Stream API and checks the members of the equivalence set for any instance that is not an instance of `OClass` (Ontop's `OWLClass` counterpart).

Converting Ontop equivalences to OWL API

Recalling section 3.2.1, `Equivalences` is Ontop's counterpart of the OWL API `Node` interface and is much used internally in the DAG-based `ClassifiedTBox` reasoner. We handle equivalence objects in many instances of the reasoning methods that we implement, and therefore we must handle converting these to the OWL API.



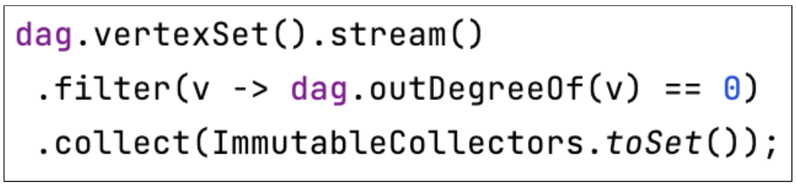
```
return equiv.getMembers().stream() Stream<DataPropertyExpression>
        .map(this::dataPropertyExpressionAsOWLDataProperty) Stream<OWLDataProperty>
        .collect(Collectors.toSet());
```

Figure 3.4: Converting an Ontop equivalences of data property expression to Java Set

We present our helper method for converting an `Equivalences` object of type `DataPropertyExpression` to a Java Set containing the expressions translated to `OWLDataProperty` objects in Figure 3.4.

Handling direct sub of top concept

It became apparent to us that `ClassifiedTBox` implementation did not include the top and bottom concepts in the DAGs it constructed. Seemingly stemming from a conscious design decision, this led to some difficulties in covering these cases in the appropriate reasoning methods. In particular, when handling the case of retrieving the subclasses of the built-in `owl:Thing`, it was not feasible to only retrieve the correct vertex of the DAG since no vertex of this built-in class was included. We overcame this limitation by extending the implementation of the `EquivalenceDAG` object with a new method `getDirectSubOfTop`, presented in Figure 3.5. This method filters out all vertices that an out-degree larger than zero, leaving only vertices that are at the top of the DAG.



```
dag.vertexSet().stream()
 .filter(v -> dag.outDegreeOf(v) == 0)
 .collect(ImmutableCollectors.toSet());
```

Figure 3.5: Implementation in `EquivalencesDAG` retrieving the direct vertices of the top of the DAG

3.2.3 Reasoning Methods

In the following, we will detail the implementation of the `getSubClasses` method. This method is one of the most complex of the `QuestOWL` reasoner and is the main method called by the Protégé plugin for displaying the inferred class hierarchy. Other methods are less complex and more akin to just delegating. An overview of the implementation of these other methods will also be provided, however, less detailed.

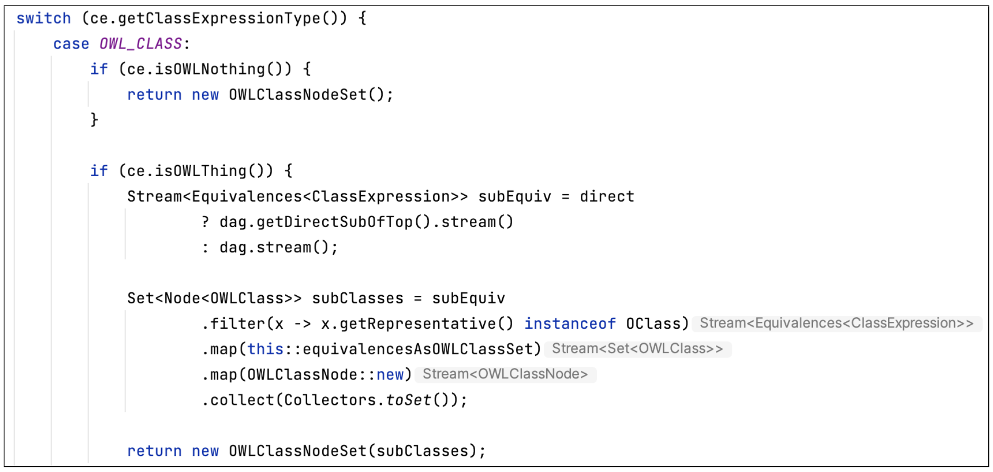
```
    public NodeSet<OWLClass> getSubClasses
        (@Nonnull OWLClassExpression ce, boolean direct)
```

getSubClasses

This method gets all classes that subsume the input class expression, with respect to the reasoner axioms. The method takes two arguments, a class expression *ce* of type `OWLClassExpression` and a boolean value *direct*. The boolean value determines whether only the direct or all descendants of the class expression should be retrieved. Returned from the method is a `NodeSet` containing the correct subclasses. It is important to keep in mind that `NodeSet` is a collection of individual Nodes, each of which can contain multiple entities. So, the returned value is a set of Nodes where each entity in the node represents classes that are equivalent.

We consider the following cases:

1. The expression is part of the *top* concept ($\top$ in DL).
2. The expression is part of the *bottom* concept ($\perp$ in DL).
3. The class entity is an *anonymous* class (without IRI).
4. The *type* of class expression. The expression can be either a class entity or it can be a complex class expression. Complex expressions must be handled differently from classes.
5. If the *direct* sub classes are to be retrieved. We must traverse the DAG to find the closest named classes.



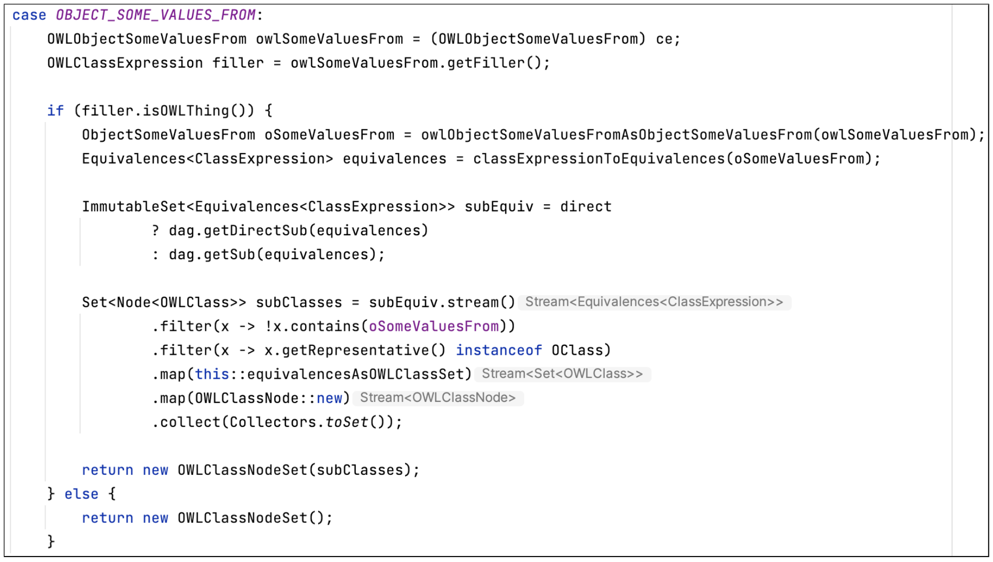
```
switch (ce.getClassExpressionType()) {
    case OWL_CLASS:
        if (ce.isOWLNothing()) {
            return new OWLClassNodeSet();
        }

        if (ce.isOWLThing()) {
            Stream<Equivalences<ClassExpression>> subEquiv = direct
                    ? dag.getDirectSubOfTop().stream()
                    : dag.stream();

            Set<Node<OWLClass>> subClasses = subEquiv
                    .filter(x -> x.getRepresentative() instanceof OClass) Stream<Equivalences<ClassExpression>>
                    .map(this::equivalencesAsOWLClassSet) Stream<Set<OWLClass>>
                    .map(OWLClassNode::new) Stream<OWLClassNode>
                    .collect(Collectors.toSet());

            return new OWLClassNodeSet(subClasses);
```

Figure 3.6: Top and bottom concepts in the `getSubClasses` method of `QuestOWL`

Expression type We first check the type of the class expression, which we do by using the built-in `.getClassExpressionType()` method on the argument *ce*. This method will return a string representation of the given expression type, which we can then check using the `ClassExpressionType` enum[7] object provided by the OWL API. This can be either just a class, which is the simplest form of a class expression or it can be some complex class expression[8]. In the case of OWL 2 QL, the complex class expression can be an existential class expression.

Top and bottom If it is the case that the class expression is of type **Class**, we can continue with checking two edge cases. If the class `IRI` is either *owl:Thing* or *owl:Nothing*, then they are built-in classes in OWL 2 and need to be handled differently. Designed to represent the top and bottom concepts from DL, `owl:Thing` represents the set of all individuals while `owl:Nothing` represents the empty set. We present the code for checking these cases in Figure 3.6. In the case of retrieving the subclasses of a class expression, we can simply return an empty `NodeSet` if the class `IRI` is `owl:Nothing`, since the bottom concept contains no individuals. However, if the class `IRI` is `owl:Thing`, or if the `IRI` is neither of the two (i.e. not a built-in class), we can proceed.

[7]A special type used to define a collection of constants.
[8]`https://www.w3.org/TR/owl2-syntax/#Class_Expressions`

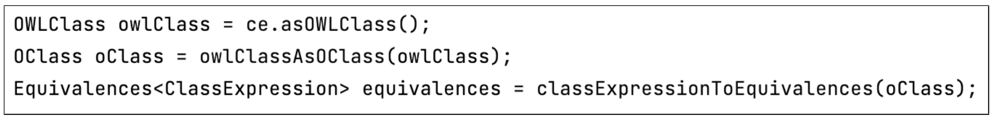


```
case OBJECT_SOME_VALUES_FROM:
    OWLObjectSomeValuesFrom owlSomeValuesFrom = (OWLObjectSomeValuesFrom) ce;
    OWLClassExpression filler = owlSomeValuesFrom.getFiller();

    if (filler.isOWLThing()) {
        ObjectSomeValuesFrom oSomeValuesFrom = owlObjectSomeValuesFromAsObjectSomeValuesFrom(owlSomeValuesFrom);
        Equivalences<ClassExpression> equivalences = classExpressionToEquivalences(oSomeValuesFrom);

        ImmutableSet<Equivalences<ClassExpression>> subEquiv = direct
                ? dag.getDirectSub(equivalences)
                : dag.getSub(equivalences);

        Set<Node<OWLClass>> subClasses = subEquiv.stream() Stream<Equivalences<ClassExpression>>
                .filter(x -> !x.contains(oSomeValuesFrom))
                .filter(x -> x.getRepresentative() instanceof OClass)
                .map(this::equivalencesAsOWLClassSet) Stream<Set<OWLClass>>
                .map(OWLClassNode::new) Stream<OWLClassNode>
                .collect(Collectors.toSet());

        return new OWLClassNodeSet(subClasses);
    } else {
        return new OWLClassNodeSet();
    }
```

Figure 3.7: Handling a complex class expression in `QuestOWL` `getSubClasses` method

Anonymity Class expressions are always anonymous, so we do not have to check these. However, classes can be either named (they have an IRI) or they can be anonymous (without IRI). We easily perform this check with the OWL API built in `.isAnonymous()` method on the class expression *ce*.

Complex expression If the expression is a complex expression, we must check which type of complex expression we allow. In our case, we must support existential restriction, which in the OWL API is represented by `OWLObjectSomeValuesFrom`. This object consists of an object property expression and a class expression. We must further check the class expression - the *filler* of the existential restriction. A filler value of anything other than a class with `IRI owl:Thing` would be too complex, and is not in the scope of this paper. However, if the filler value is a class with `IRI owl:Thing` we can proceed (see Fig. 3.7).

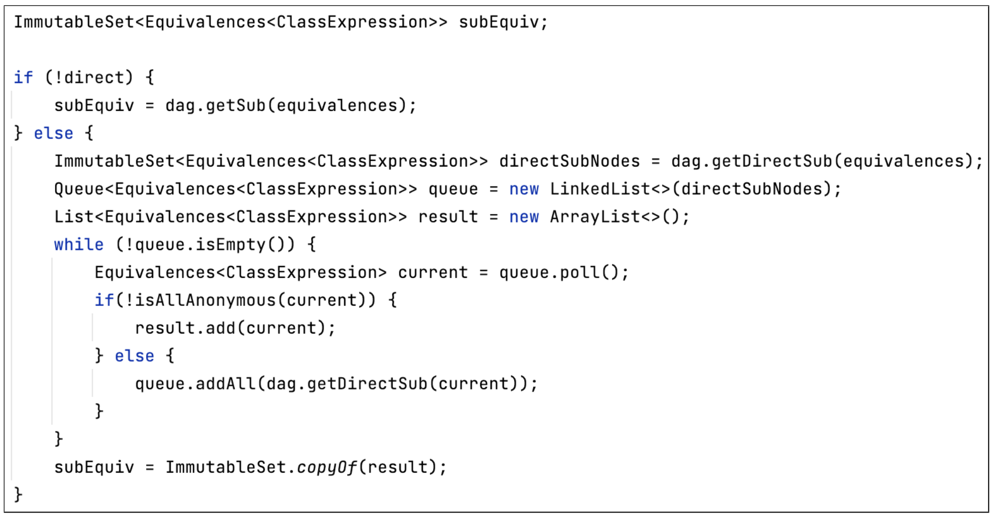


```
OWLClass owlClass = ce.asOWLClass();
OClass oClass = owlClassAsOClass(owlClass);
Equivalences<ClassExpression> equivalences = classExpressionToEquivalences(oClass);
```

Figure 3.8: Converting an `OWLClass` to Ontop `Equivalences` object from an `OWLClassExpression` in `QuestOWL`

Converting After we have identified which type of class expression the argument is and verified that the expression can be supported, we continue with converting the expression from the OWL API to Ontop's internal API. This is necessary in order to interact with the `ClassifiedTBox` implementation and retrieve the `Equivalences` for the given class expression. Recalling section 3.2.2, we utilize our helper methods for much of this work. The equivalences are retrieved from the class DAG `EquivalencesDAG<ClassExpression>` of the `ClassifiedTBox`, by using the converted class expression to find the correct vertex of the graph. This gives us a corresponding equivalences object `Equivalences<ClassExpression>`. We present in Figure 3.8 how we first use the OWL API `.asOWLClass()` method to convert the class expression to an `OWLClass` before we utilize our helper methods to eventually get an `Equivalences` object.

Now that we have acquired the correct equivalences object, which contains all classes that are equivalent to the input class expression, we can use this to retrieve the corresponding sub-classes from the DAG. This is done by utilizing methods that are already exposed by the DAG: `getSub` and `getDirectSub`, which leads us to the next case we have to handle.



```java
ImmutableSet<Equivalences<ClassExpression>> subEquiv;

if (!direct) {
    subEquiv = dag.getSub(equivalences);
} else {
    ImmutableSet<Equivalences<ClassExpression>> directSubNodes = dag.getDirectSub(equivalences);
    Queue<Equivalences<ClassExpression>> queue = new LinkedList<>(directSubNodes);
    List<Equivalences<ClassExpression>> result = new ArrayList<>();
    while (!queue.isEmpty()) {
        Equivalences<ClassExpression> current = queue.poll();
        if(!isAllAnonymous(current)) {
            result.add(current);
        } else {
            queue.addAll(dag.getDirectSub(current));
        }
    }
    subEquiv = ImmutableSet.copyOf(result);
}
```

Figure 3.9: Handling direct (using BFS algorithm) vs indirect in `QuestOWL`

Direct Recall at the beginning of this section, the `getSubClasses` method takes two arguments, a class expression *ce* of type `OWLClassExpression` and a boolean value *direct*. The boolean value determines whether we should only retrieve the direct relations or exhaustively travel until all relations have been retrieved. This notion of direct vs indirect is different in terms of the OWL API compared to the DAG. Direct in the OWLAPI sense means we only consider the sub-nodes that are connected to the given node via a (possibly empty) path of **only** non-anonymous nodes. However, in the sense of the DAG, the direct sub-nodes can refer to nodes which are anonymous, i.e. not named classes. This proves no challenge if we want to find *all* sub-classes, as we can easily utilize the `getSub` method of the DAG. However, this is not the case with the `getDirectSub` method. We provide an example to further understand this notion.

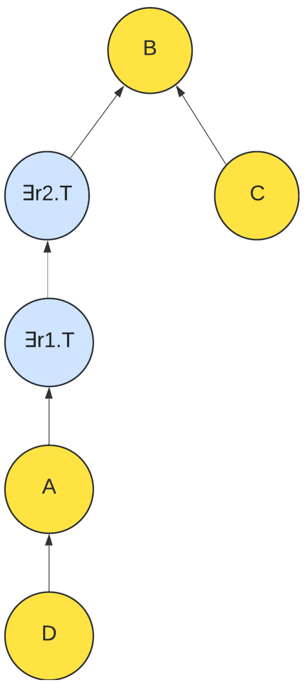
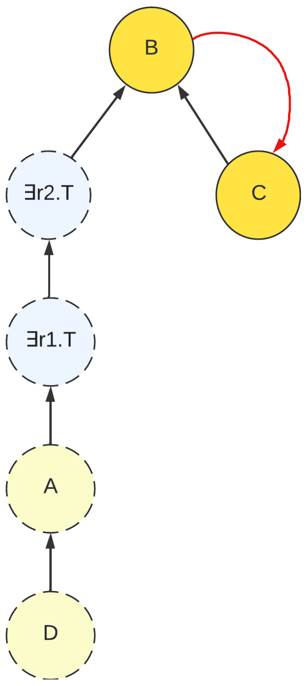


Figure 3.10: Resulting DAG from processing an ontology with complex class expressions

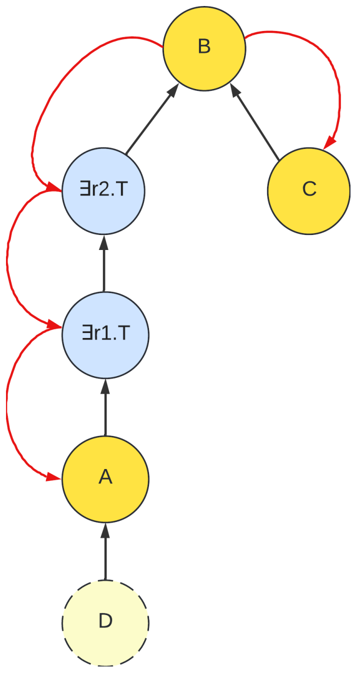
Example 3 *Handling direct*

Expanding on the ontology in example 2 found in section 2.1.1, we provide a visual of the resulting DAG of the `ClassifiedTBox`. As we can see in Figure 3.10, the *nodes* of the optimized DAG consist of both complex class expressions (anonymous nodes) and classes (non-anonymous). The `getSubClasses` method must traverse the DAG to find the subclasses of a given class expression *ce*. Therefore, if we are tasked with finding the *direct* subclasses of some class expression, it is not sufficient to travel only to the directly connected nodes of a given start node. In this example we will cover both ways of traversing the

DAG found in Fig. 3.10. We will find the subclasses of the top node of the DAG - a node containing the class B. If we attempt the simple approach of traveling out from each of the edges to the directly connected named nodes, we find that we only get one node C as a result. We can see this in Fig. 3.11a, where the red arrow indicates which node was visited. The two directly connected nodes to node B are $\exists r2.\top$ and C. From these two, only C is a named class. We are interested in finding the closest direct (named) subclasses, and must therefore employ a different strategy to achieve this.



(a) Direct named subclasses in the DAG



(b) Direct named subclasses skipping the anonymous nodes in the DAG

Figure 3.11: Finding direct subclasses of B

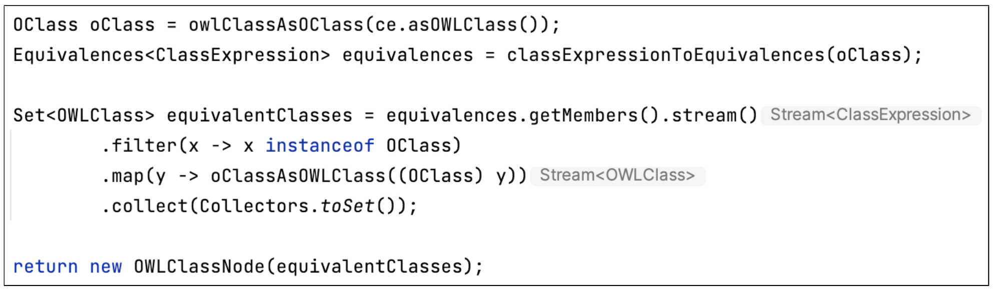
To combat this, a Breadth-First Search (BFS) is implemented to traverse the graph and find the closest direct nodes of named subclasses (BFS algorithm covered in section 2.5). This algorithm is implemented in the code using a queue and is presented in Figure 3.9. We initialize an empty *result set* and initialize the queue with the direct sub-nodes of the given node, which is the equivalence object. We use `getDirectSub` to initialize. When a node is found that is not anonymous, it is added to the result set. If not, we recursively proceed with adding its direct sub-nodes to the queue using `getDirectSub`.

When the queue is empty, the algorithm terminates. The result of this algorithm on node *B* of the DAG in Fig. 3.10 will give us two nodes, node *A* and node *C*. As can be seen with the red arrows in Fig. 3.11b, the algorithm traverses recursively through the nodes containing anonymous classes. It does this until it either reaches the end of the DAG or until all named classes are found.

getEquivalentClasses

```
    public Node<OWLClass> getEquivalentClasses(@Nonnull
        OWLClassExpression ce)
```

This method gets the set of named classes that are equivalent to the specified class expression with respect to the set of reasoner axioms. The classes are returned as a single Node.



```
OClass oClass = owlClassAsOClass(ce.asOWLClass());
Equivalences<ClassExpression> equivalences = classExpressionToEquivalences(oClass);

Set<OWLClass> equivalentClasses = equivalences.getMembers().stream() Stream<ClassExpression>
        .filter(x -> x instanceof OClass)
        .map(y -> oClassAsOWLClass((OClass) y)) Stream<OWLClass>
        .collect(Collectors.toSet());

return new OWLClassNode(equivalentClasses);
```

Figure 3.12: Retrieving equivalent classes with Java Stream in `getEquivalentClasses` method

We consider the following cases:

1. The expression is part of the *top* concept ($\top$ in DL).
2. The expression is part of the *bottom* concept ($\bot$ in DL).
3. The class entity is an *anonymous* class (without IRI).
4. The *type* of class expression.

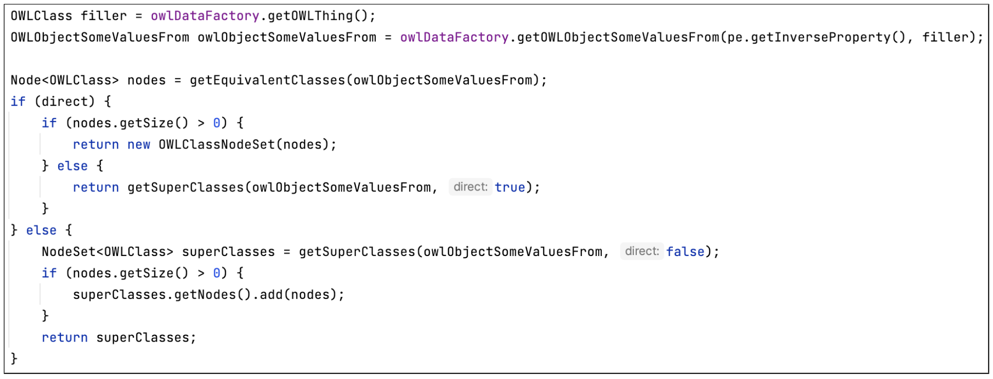
We handle these cases the same as we did in the previous section (`getSubClasses`). Presented in Figure 3.12, we show how we first retrieve the `Equivalences` object, which we then iterate through by using the Java Stream API. We collect the result in a `Set` and return this set as an `OWLClassNode`.

getObjectPropertyRanges

This method gets the named classes that are the direct or indirect ranges of this property expression with respect to the import closure of the root ontology. The classes are returned as a NodeSet. The method was implemented by following an algorithm declared in the documentation of the OWL API method. We present the algorithm in pseudocode:

```
           PE = PropertyExpression

    let invPE = ObjectInverseOf(PE) owl:Thing
    let   OPE = ObjectSomeValuesFrom(invPE)
    let     N = getEquivalentClasses(OPE)

    If direct is true:
        then if N is not empty:
            then the return value is N,
        else the return value is
        the result of getSuperClasses(OPE, true)

    If direct is false:
        then the result of getSuperClasses(OPE, false)
        together with N if N is non-empty
```

The Ontop implementation is presented in Fig. 3.13. We simply follow the procedure specified in the OWL API documentation.



```
OWLClass filler = owlDataFactory.getOWLThing();
OWLObjectSomeValuesFrom owlObjectSomeValuesFrom = owlDataFactory.getOWLObjectSomeValuesFrom(pe.getInverseProperty(), filler);

Node<OWLClass> nodes = getEquivalentClasses(owlObjectSomeValuesFrom);
if (direct) {
    if (nodes.getSize() > 0) {
        return new OWLClassNodeSet(nodes);
    } else {
        return getSuperClasses(owlObjectSomeValuesFrom,  direct: true);
    }
} else {
    NodeSet<OWLClass> superClasses = getSuperClasses(owlObjectSomeValuesFrom,  direct: false);
    if (nodes.getSize() > 0) {
        superClasses.getNodes().add(nodes);
    }
    return superClasses;
}
```

Figure 3.13: Implementation of `getObjectPropertyRanges`

Chapter 4

Evaluation

This chapter serves to evaluate and test our implementation. We test the implementation in two main ways. We compile our implementation and run it in the Protégé application, testing if the hierarchies are as we expect (section 4.1) before we perform unit tests on the reasoning methods, testing them in isolation (section 4.2).

4.1 Protégé Test

We test if the reasoner implementation works as we expect by loading the Ontop plugin into the Protégé ontology editor. Here we concern ourselves with comparing the asserted and inferred hierarchies. We load a test ontology file with some complex expressions into the ontology editor. The asserted hierarchies will not pick up on these complex expressions - this is what our reasoner is for. We initialize the Ontop reasoner and compare the two hierarchies.

To test Ontop in Protégé we first need to build and compile. This can be done in two ways: (*i.*) compile the Ontop *plugin* and manually replace the `.jar` file in Protégé's file structure with our plugin, or (*ii.*) compile the Protégé software and Ontop plugin together which will yield an executable we can directly run. We utilize the provided Maven bash script (`mvnw`) to build and compile. We specify which Maven profile to utilize in the following manner:

```
./mvnw -Passet-protege-plugin
```

```
./mvnw -Passet-protege
```

In our case it proved easier to compile Protégé and the Ontop plugin bundled together, so we ended up mainly using the latter command. Protégé allows us to easily construct ontologies with a graphical user interface. We present the

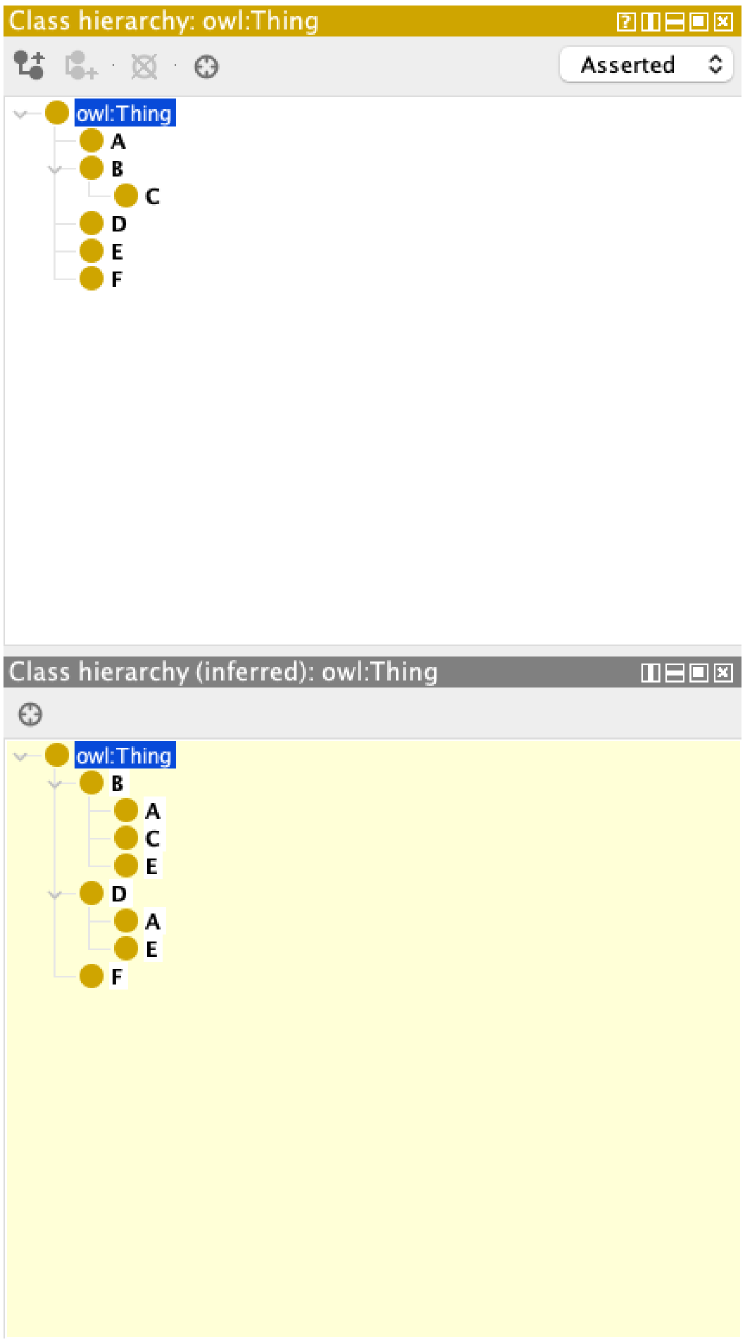

Class hierarchy: owl:Thing
Asserted
owl:Thing
A
B
C
D
E
F
Class hierarchy (inferred): owl:Thing
owl:Thing
B
A
C
E
D
A
E
F

Figure 4.1: Protégé asserted (see top) and inferred (see bottom) hierarchies with our reasoner implementation

asserted and inferred hierarchies of an ontology created for test purposes. This ontology contains the following axioms:

- SubClassOf(D, A)
- SubClassOf(A, someR1)
- SubClassOf(someR1, someR2)
- SubClassOf(someR2, B)
- SubClassOf(C, B)
- SubClassOf(E, someR2)
- ObjectPropertyDomain(someR2, D)
  where:
  - someR1 = ObjectSomeValuesFrom(r1, OWLThing())
  - someR2 = ObjectSomeValuesFrom(r2, OWLThing())

We select Ontop as the reasoner of choice in Protégé and initialize it. The ontology and axioms are processed by `QuestOWL`, and we can inspect the resulting hierarchy. Both the asserted hierarchy and the resulting hierarchy are presented in Figure 4.1. As we expect, the two are not identical. Our implementation has successfully inferred new knowledge from the already asserted knowledge in the system.

4.2 Unit Testing

We test the reasoner implementation through unit testing, the basic idea is to break down the complex system into small manageable, and testable units. We have created 43 test cases in total, we illustrate some representative ones below. Some selected interesting test cases are also provided in the appendix A.

In the test environment, we utilize the `OWLFunctionalSyntaxFactory`, provided to us by the OWL API. This allows us to easily construct test cases. Constructing an ontology is done by using the `Ontology` constructor class in the `OWLFunctionalSyntaxFactory`. Classes, properties, and axioms are also defined by utilizing the object factory. We detail this here:

```
String prefix = "http://www.example.org/";

OWLClass A =
    Class(IRI.create(prefix + "A"));
OWLObjectProperty r1 =
    ObjectProperty(IRI.create(prefix + "r1"));
OWLDataProperty d1 =
    DataProperty(IRI.create(prefix + "hasAge"));
```

These are some ways to create classes, object properties, and data properties with the `OWLFunctionalSyntaxFactory` provided by the OWL API. Keep this in mind as we detail a subset of test cases in the following sections.

Case 1a.

This case is to test basic reasoning of `getSubclasses` and the cardinality of the `NodeSet`.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(C, false);
assertTrue(subClasses.containsEntity(A));
assertTrue(subClasses.containsEntity(B));
assertEquals(1, subClasses.getNodes().size());
```

Input The ontology contains the following three axioms:

- SubClassOf(A, C)
- SubClassOf(B, C)
- EquivalentClasses(A, B)

Expected `getSubClasses(C, false)` will return exactly one node, which contains both *A* and *B*.

Case 1b.

This case is to test reasoning of a complex expression of `getSubclasses`.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(B, false);
assertTrue(subClasses.containsEntity(A));
assertTrue(subClasses.containsEntity(C));
assertTrue(subClasses.containsEntity(D));
```

Input The ontology contains the following four axioms:

- SubClassOf(A, ObjectSomeValuesFrom(r1, OWLThing()))
- SubClassOf(ObjectSomeValuesFrom(r1, OWLThing()), B)
- SubClassOf(D, C)
- SubClassOf(C, B)

Expected `getSubClasses(B, false)` will return three nodes, *A*, *C* and *D*.

Case 1c.

This case is to test reasoning of `getSubclasses` when the *class expression* is the built-in `owl:Thing`, with *direct* argument set to true.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(OWLThing(), true);
assertFalse(subClasses.containsEntity(Male));
assertFalse(subClasses.containsEntity(Female));
assertTrue(subClasses.containsEntity(Person));
```

Input The ontology contains the following two axioms:

- SubClassOf(Male, Person)
- SubClassOf(Female, Person)

Expected `getSubClasses(OWLThing(), true)` will return one node, *Person*.

Case 1d.

The following case replicates the ontology from example 2 mentioned earlier in section 2.1.1. This case is to test more complex reasoning of `getSubclasses` when *direct* is true. Test retrieving direct sub-classes from the DAG. Done by utilizing BFS to search for the nearest named class.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(B, true);
assertEquals(2, subClasses.getNodes().size());
assertTrue(subClasses.containsEntity(A));
assertTrue(subClasses.containsEntity(C));
```

Input The ontology contains the following five axioms:

- SubClassOf(D, A)
- SubClassOf(A, someR1)
- SubClassOf(someR1, someR2)
- SubClassOf(someR2, B)
- SubClassOf(C, B)
  where:
  - someR1 = ObjectSomeValuesFrom(r1, OWLThing())
  - someR2 = ObjectSomeValuesFrom(r2, OWLThing())

Expected `getSubClasses(B, true)` will return two nodes, *A* and *C*.

Case 2a.

This case is to test reasoning of `getEquivalentClasses` with a complex expression.

```
Node<OWLClass> equivalentClasses =
    reasoner.getEquivalentClasses(someR1);
assertTrue(equivalentClasses.contains(A));
assertTrue(equivalentClasses.contains(B));
```

Input The ontology contains the following three axioms:

- EquivalentClasses(A, someR1)
- EquivalentClasses(B, someR2)
- EquivalentClasses(someR1, someR2)
  where:
  - someR1 = ObjectSomeValuesFrom(r1, OWLThing())
  - someR2 = ObjectSomeValuesFrom(r2, OWLThing())

Expected `getEquivalentClasses(someR1)` will return two nodes, *A* and *B*.

Case 3a.

This case is to test reasoning of `getDisjointClasses`.

```
NodeSet<OWLClass> disjointClasses =
    reasoner.getDisjointClasses(A);
assertTrue(disjointClasses.containsEntity(B));
assertTrue(disjointClasses.containsEntity(C));
assertTrue(disjointClasses.containsEntity(G));
assertFalse(disjointClasses.containsEntity(D));
assertFalse(disjointClasses.containsEntity(E));
assertFalse(disjointClasses.containsEntity(F));
```

Input The ontology contains the following three axioms:

- DisjointClasses(A, B, C)
- DisjointClasses(D, E, F)
- DisjointClasses(A, G)

Expected `getDisjointClasses(A)` will return three nodes, *B*, *C* and *G*.

Case 4a.

This case is to test simple reasoning of `getObjectPropertyDomains`.

```
NodeSet<OWLClass> domains =
    reasoner.getObjectPropertyDomains(r1, false);
assertTrue(domains.containsEntity(A));
assertTrue(domains.containsEntity(B));
assertFalse(domains.containsEntity(C));
assertFalse(domains.containsEntity(D));
```

Input The ontology contains the following four axioms:

- ObjectPropertyDomain(r1, A)
- ObjectPropertyDomain(r1, B)
- ObjectPropertyRange(r1, C)
- ObjectPropertyRange(r1, D)

Expected `getObjectPropertyDomains(r1, false)` will return two nodes, *A* and *B*.

Chapter 5

Conclusion and Future Work

This chapter concludes the paper. We restate the research objective and limitation, reiterating what has been done to address the limitation (section 5.1) before we discuss what future work is possible (section 5.2).

5.1 Conclusion

We have proved that it is possible to use the functionality present in the internal DAG-based reasoner implementation in Ontop. The `ClassifiedTBox` of `QuestOWL` was utilized to provide complex TBox reasoning capabilities, improving on the dummy reasoner implementation which was already present. We developed methods for converting necessary objects between the OWL API and Ontop's internal API, such that they could be used to interact with the internal DAG-based reasoner implementation in `QuestOWL`. With a focus on reasoning methods important for displaying proper class- and property hierarchies in Protégé, we implemented the TBox reasoning methods inside `QuestOWL` and made them OWL API compliant. The implementation was tested thoroughly through unit testing and application testing through Protégé and its plugin interface. We proved our implementation to infer and display a potentially richer hierarchy than the explicitly asserted one, proving our research objective.

5.2 Future Work

More Testing

It could prove useful to further test the implementation to verify its integrity. This should be done both with unit testing and testing in Protégé as a plugin.

Other cases

There are some other methods that require not only the DAGs that were exposed to us by the `ClassifiedTBox`. These methods require accessing both the TBox and ABox, and cannot be simply delegated to the corresponding DAGs. Since Ontop's internal DAG implementation only concerns itself with the TBox, we cannot implement any OWL API methods that require ABox reasoning. Examples of such methods are `getTypes` and `getInstances`.

Caching

Minor performance benefits might be gained from implementing a caching solution in the reasoner implementation. More specifically, this could be implemented in the conversion layer between the OWL API and Ontop's internal API. The internal reasoning capabilities in `QuestOWL` are exposed through the `ClassifiedTBox` which uses Ontop's internal classes and types. QuestOWL reasoner in Ontop is OWL API compliant - which means that the parameters and return values of the Java methods must adhere to this API. The consequence of this is that to utilize the internal implementation of the `ClassifiedTBox`, argument and return types must be translated to and from these two API's. Currently, these translations are done on the fly as they are needed and subsequently "thrown away" when the active reasoning task is finished. Depending on the size of the ontology in question, there may be an influx of translations between these two API's.

It could prove useful to cache the results of such a conversion and save it to some key-value store or map. This way, already translated values do not have to go through the same process again. Instead, the value in question can just be retrieved from the cache.

More research needs to be done to verify whether there is any performance to be gained from this. In theory, performance gains should increase in tune with ontology size. However, this must be tested in practice.

Appendix A

Additional test cases

Case 5a.

This case is to test reasoning of `getSubClasses` when the *class expression* is the built in `owl:Thing`, with *direct* argument set to false.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(OWLThing(), false);
assertTrue(subClasses.containsEntity(Male));
assertTrue(subClasses.containsEntity(Female));
assertTrue(subClasses.containsEntity(Person));
```

Input The ontology contains the following two axioms:

- SubClassOf(Male, Person)
- SubClassOf(Female, Person)

Expected `getSubClasses(OWLThing(), false)` will return three nodes, each containing one entity, *Male*, *Female*, and *Person*.

Case 5b.

This case is to test reasoning of `getSubClasses` with object property domain. *Direct* is set to true.

```
NodeSet<OWLClass> subClasses =
    reasoner.getSubClasses(B, true);
assertTrue(subClasses.containsEntity(A));
assertEquals(1, subClasses.getNodes().size());
```

Input The ontology contains the following two axioms:

- SubClassOf(A, ObjectSomeValuesFrom(r1, OWLThing()))
- ObjectPropertyDomain(r1, B)

Expected `getSubClasses(B, true)` will return one node, containing *B*.

Case 6a.

This case is to test reasoning of `getDisjointObjectProperties`.

```
NodeSet<OWLObjectPropertyExpression>
   disjointProperties =
    reasoner.getDisjointObjectProperties(r1);
assertTrue(disjointProperties.containsEntity(r2));
```

Input The ontology contains the following axiom:

- DisjointObjectProperties(r1, r2)

Expected `getDisjointObjectProperties(r1)` will return one node, containing *r*2.

Case 7a.

This case is to test reasoning of `getSubObjectProperties` with the top object property and *direct* set to true.

```
NodeSet<OWLObjectPropertyExpression> subProperties =
    reasoner.getSubObjectProperties(
       owlTopObjectProperty, true);
assertFalse(subProperties.containsEntity(r1));
assertTrue(subProperties.containsEntity(r2));
assertTrue(subProperties.containsEntity(r2.
   getInverseProperty()));
```

Input The ontology contains the following axiom:

- SubObjectPropertyOf(r1, r2)

Expected `getDisjointObjectProperties(r1)` will return two nodes, containing *r*2 and $r2^{-1}$.

Case 7b.

This case is to test reasoning of `getSubObjectProperties` with the top object property and *direct* set to false.

```
NodeSet<OWLObjectPropertyExpression> subProperties =
    reasoner.getSubObjectProperties(
       owlTopObjectProperty, false);
assertTrue(subProperties.containsEntity(r1));
assertTrue(subProperties.containsEntity(r1.
   getInverseProperty()));
assertTrue(subProperties.containsEntity(r2));
assertTrue(subProperties.containsEntity(r2.
   getInverseProperty()));
```

Input The ontology contains the following axiom:

- SubObjectPropertyOf(r1, r2)

Expected `getDisjointObjectProperties(r1)` will return four nodes, containing $r1$, $r1^{-1}$, $r2$ and $r2^{-1}$.

Case 8a.

This case is to test reasoning of `getSuperClasses` with the bottom concept and *direct* set to false.

```
NodeSet<OWLClass> superClasses =
    reasoner.getSuperClasses(OWLNothing(), false);
assertTrue(superClasses.containsEntity(A));
assertTrue(superClasses.containsEntity(B));
```

Input The ontology contains the following axiom:

- SubClassOf(A, B)

Expected `getSuperClasses(OWLNothing(), false)` will return two nodes, containing A and B.

Bibliography

[1] Guohui Xiao, Linfang Ding, Benjamin Cogrel, and Diego Calvanese. Virtual Knowledge Graphs: An Overview of Systems and Use Cases. *Data Intelligence*, 1(3):201–223, June 2019.

[2] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-Based Data Access: A Survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 5511–5519, Stockholm, Sweden, July 2018. International Joint Conferences on Artificial Intelligence Organization.

[3] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The MASTRO system for ontology-based data access. *Semantic Web*, 2(1):43–53, January 2011.

[4] Freddy Priyatna, Oscar Corcho, and Juan Sequeda. Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 479–490, New York, NY, USA, April 2014. Association for Computing Machinery.

[5] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, December 2016.

[6] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalaycı, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. The Virtual Knowledge Graph System Ontop. In Jeff Z. Pan, Valentina Tamma, Claudia d'Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and

Lalana Kagal, editors, *The Semantic Web – ISWC 2020*, volume 12507, pages 259–277. Springer International Publishing, Cham, 2020.

[7] Ontop. https://ontop-vkg.org/.

[8] Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The DL-Lite Family and Relations. *Journal of Artificial Intelligence Research*, 36:1–69, October 2009.

[9] Tom Gruber. Ontology. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2574–2576. Springer, New York, NY, 2018.

[10] Sarah Seyenam Adjoa Komla-Ebri. DL-Lite reasoning using Directed Acyclic Graphs. October 2013.

[11] Franz Baader, Diego Calvanese, Deborah Mcguinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, January 2007.

[12] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A Description Logic Primer, June 2013.

[13] F. Baader. Appendix: Description Logic Terminology. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook*, pages 525–536. Cambridge University Press, 2 edition, August 2007.

[14] Meghyn Bienvenu, Magdalena Ortiz, Mantas Šimkus, and Guohui Xiao. Tractability Guarantees for DL-Lite Query Answering. *CEUR Workshop Proceedings*, 1014:41–52, January 2013.

[15] Sean Bechhofer. OWL: Web Ontology Language. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2640–2641. Springer, New York, NY, 2018.

[16] OWL 2 Web Ontology Language Document Overview (Second Edition). https://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

[17] OWL 2 Web Ontology Language Profiles (Second Edition). https://www.w3.org/TR/owl2-profiles/.

[18] Timea Bagosi, Diego Calvanese, Josef Hardi, Sarah Komla-Ebri, Davide Lanti, Martin Rezk, Mariano Rodríguez-Muro, Mindaugas Slusnys, and Guohui Xiao. The Ontop Framework for Ontology Based Data Access. In Dongyan Zhao, Jianfeng Du, Haofen Wang, Peng Wang, Donghong Ji, and Jeff Z. Pan, editors, *The Semantic Web and Web Science*, volume 480, pages 67–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[19] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

[20] Hans Hinterberger. Graph. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1635–1636. Springer, New York, NY, 2018.