UNIVERSITÄT KOBLENZ · LANDAU
Faculty 4: Computer Science

WeST People and Knowledge Networks
Institute for Web Science and Technologies

Implementation of the concept contraction in the Description Logic $\mathcal{EL}$

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.) in
Web Science and Technologies

submitted by
Fiorela Ciroku

First supervisor: Prof. Dr. Steffen Staab
Institute for Web Science and Technologies
Second supervisor: Dr. Tjitze Rienstra
Institute for Web Science and Technologies

Koblenz, October 2019

Statement

I hereby certify that this thesis has been composed by me and is based on my own work, that I did not use any further resources than specified – in particular no references unmentioned in the reference section – and that I did not submit this thesis to another examination before. The paper submission is identical to the submitted electronic version.

I agree to have this thesis published in the library. Yes □ No □

I agree to have this thesis published on the Web. Yes □ No □

The thesis text is available under a Creative Commons License (CC BY-SA 4.0). Yes □ No □

The source code is available under a GNU General Public License (GPLv3). Yes □ No □

The collected data is available under a Creative Commons License (CC BY-SA 4.0). Yes □ No □

.....

(Place, Date) (Signature)

Abstract

Belief revision is the subarea of knowledge representation which studies the dynamics of epistemic states of an agent [Ribeiro, 2012]. In the classical AGM approach, contraction, as part of the belief revision, deals with the removal of beliefs in knowledge bases. This master's thesis presents the study and the implementation of *concept contraction* in the Description Logic $\mathcal{EL}$. Concept contraction deals with the following situation. Given two concept C and D, assuming that $C \sqsubseteq D$, how can concept C be changed so that it is not subsumed by D anymore, but is as similar as possible to C? This approach of belief change is different from other related work because it deals with contraction in the level of concepts and not T-Boxes and A-Boxes in general. The main contribution of the thesis is the implementation of the concept contraction. The implementation provides insight into the complexity of contraction in $\mathcal{EL}$, which is tractable since the main inference task in $\mathcal{EL}$ is also tractable. The implementation consists of the design of five algorithms that are necessary for concept contraction. The algorithms are described, illustrated with examples, and analyzed in terms of time complexity. Furthermore, we propose an new approach for a selection function, adapt for the concept contraction. The selection function uses metadata about the concepts in order to select the *best* from an input set. The metadata is modeled in a framework that we have designed, based on standard metadata frameworks. As an important part of the concept contraction, the selection function is responsible for selecting the best concepts that are as similar as possible to concept C. Lastly, we have successfully implemented the concept contraction in Python, and the results are promising.

Acknowledgments

Firstly, I would like to thank my supervisors Prof. Dr. Staab and Dr. Rienstra. Prof. Staab, thank you for your immense support, thought my years as a master student at the University of Koblenz. The West Institute has welcomed me with open arms and introduced me to the beautiful world of the Semantic Web. Working for you has been a great experience and has made me realize my path in academic life. Dr. Rienstra, thank you very much for all the time that you have dedicated to me all these months. Our constructive discussions and your continued feedback have made this thesis what it is. Thank you for pushing me and helping me achieve my goals. Last but not least, I would like to thank Dr. Schon for all her valuable input and constant communication during the last year and a half.

***

A special thank you goes to my family for the unconditional support that they have given me during my studies and specifically during the work of the thesis. You are my motivation, and I could not have done it without you. This is for you mom and dad!

Contents

Chapter 1

Introduction and motivation

1.1 Introduction

With the passing of years, the *World Wide Web* has become an excellent means of providing and searching for information. At the same time, the vast amount of information available on the Web makes the search for information an overwhelming experience for the user. The reason is that most of the information that is currently in the Web is aimed to be human-readable and not machine-readable. The *Semantic Web* aims for machine-understandable Web resources, whose information can be shared and processed both by automated tools, such as search engines and by human users. One of the key concepts in the Semantic Web are ontologies, as they can be used to describe the semantics of information at various sites, overcoming the problem of implicit and hidden knowledge, and thus enabling content exchange[Berners-Lee et al., 2001]. Ontologies are not static, though. New information is introduced every second to the Web, and ontologies need to revise the knowledge that they already hold and change it with regards to the new information. In the classical approach of belief revision, to add new information that (usually) causes inconsistencies to the ontology, the ontology must give up the old information that is already in the knowledge base and that it contradicts the new information. This kind of operation is called belief revision, and it is realized by firstly contracting the old information and then adding the new information. The contraction of old information from a knowledge base results in the removal of the whole sentence that causes the inconsistency. Such an approach can be considered not too gentle for a knowledge base. Based on research on related works, to our knowledge, there are very few studies that tackle this problem. Most of the related work deal with contraction by working with T-Boxes or A-boxes. The T-Box is the part of a knowledge base that contains terminological knowledge about a domain. The A-Box of a Description Logic knowledge base contains assertional knowledge about the domain [Baader et al., 2003].

Based on the AGM theory [Alchourrón et al., 1985], we investigate belief change for ontologies by considering the problem of *concept contraction*: given two concepts C and D, we want to obtain a new concept that is as similar as possible to C, but that is not subsumed D. It is understandable that if concept C is not subsumed by concept D, the concept contraction does not apply, and C is left intact. We can see this as a form of contraction since C must be weakened in a way similar to how a knowledge base is weakened during belief contraction [Rienstra et al., 2018]. The basis of the concept contraction is Description Logics $\mathcal{EL}$, which forms the basis of the OWL 2 EL profile [Patel-Schneider, 2004]. The authors in [Rienstra et al., 2018] formalised concept contraction by adapting the AGM approach. The object of the study is a concept contraction operator $\ominus$ that takes as input two concepts C, D, and returns a new concept $C \ominus D$ representing the contraction of C by D. Associative to definitions of the operator are the postulates that determine classes of operators that are well-behaved in a precisely stated sense. The constructive definitions are linked with the postulates by representation theorems, which establish a precise correspondence between certain classes of operators and sets of postulates [Rienstra et al., 2018]. The goal of the master's thesis is the implementation of this concept contraction.

The overview of the thesis is as follows:

- **Chapter 1: Introduction and Motivation** We present the topic of the master's thesis and the motivation behind it. Later, we state the research questions and the methodology that we have used in this work. At the end of the chapter, we discuss the scope of the thesis.
- **Chapter 2: Concept contraction in the Description Logic $\mathcal{EL}$** In this chapter, we study in detail the concept contraction. Firstly, we provide an overview of the Description Logics family, in particular, $\mathcal{EL}$ language. We briefly discuss the contraction operation in the AGM theory, together with the respective postulates. Then, we analyse the concept contraction thoroughly in the setting of an empty T-Box. The situation where concepts are semantically dependable to a T-Box (cyclic or acyclic) is also treated but in less detail. Reformulations of the well-established AGM postulates for contraction are provided for the concept contraction as well. Lastly, we include a section concerning the limitations of the concept contraction.
- **Chapter 3: Designed algorithms and their complexity** Here, we present five algorithms that we have designed to implement the concept contraction operator. Each of the algorithms is described and analyzed in terms of time complexity.
- **Chapter 4: A selection function for concept contraction** This chapter tackles one of the leading research questions of the thesis: "How to create a reasonable selection function?". Firstly, we introduce the selection function in belief revision.

Then we present an approach to the concept and associative metadata representation. Based on the representation, we propose a theoretical implementation of a selection function, which makes use of metadata of concepts to choose the best remainders of a concept C with respect to a concept D.

- **Chapter 5: Implementation of $conceptContraction$ algorithm** In this chapter, we describe the implementation of the $conceptContraction$ algorithm in Python. We also include the results of the testing of the application.

- **Chapter 6: Conclusion and future work** As the last chapter of the thesis, we summarize our conclusions on the topic, and give an answer for each of the research questions. We also include a short discussion regarding future work.

1.2 Motivation

The classical approach for repairing a description logic ontology O in the sense of removing an unwanted consequence α is to delete a minimal number of axioms from O such that the resulting ontology O' does not have the consequence α, according to [Alchourrón et al., 1985]. However, we can argue that this approach may be too rough, meaning that it may also remove consequences that are wanted.

Example 1. Let us consider the case where $\mathcal{T}$ is a T-Box and $\mathcal{A}$ is an A-Box.

$$\mathcal{T} = \{Student \equiv Person \sqcap \exists isRegisteredAt.(University \sqcap Institute)\}$$
$$\mathcal{A} = \{Student(Anna)\}$$

Suppose that we want to contract the concept $\exists isRegisteredAt.Institute$ from the $Person \sqcap \exists isRegisteredAt.(University \sqcap Institute)$. If we apply the classical approach of AGM contraction [Alchourrón et al., 1985], the whole terminological axiom would be removed. This operation would cause the assertional axiom in A-Box to be removed as well, considering that there is no more a concept *Student*. Obviously, such contraction can lead to loss of information that might be valuable. With the means of the concept contraction, we could weaken the concept definition in the T-Box. A weaker definition would be $Student \equiv \exists isRegisteredAt.University$. In addition, the individual $Anna$ is assigned to a more *general concept*, and not permanently deleted. Following this approach, the definition of concept $Student$ and the information that $Anna$ is a $Student$ is not lost. In conclusion, we believe that it is interesting to study a gentle notion of repair in which axioms are not deleted but only weakened.

1.3 Research questions and methodology

This master's thesis aims to implement the concept contraction proposed in [Rienstra et al., 2018]. In the course of work, there are multiple questions that we raise. We believe that the following research questions are worth investigating:

- Suppose the selection function is given, how to implement a concept contraction operator?
  - How to compute the remainders of a concept C with respect to a concept D?
  - How to compute the least common subsumer, given a set of remainders?
- How to select/create a reasonable selection function for the concept contraction?
- What is the complexity of the algorithms developed to implement the concept contraction operator?

In order to provide a solution to the above research questions, our work will be divided in the following main tasks:

1. Implementation of the concept contraction operator

   The first research question is highly dependable from the two sub-questions. Finding a way to compute the remainders and the least common subsumer is the path to implementing the concept contraction operator. The computations of these elements of the operator will be described in pseudocode and include the computation of the weakenings of concepts and a subsumption algorithm for the Description Logic $\mathcal{EL}$.

2. Determining of a reasonable selection function

   The work regarding the selection function is divided into two paths: (i) Assuming that the selection function is given, and (ii) Creating/ Choosing a selection function. The first assumption is used in the design of the algorithms in the first part of the thesis. Later on, we shall find an approach to determine a selection function for the concept contraction.

3. Analysis of the complexity of the results

   As part of the master thesis, an analysis is included with the purpose of evaluating the algorithms that we have designed for the concept contraction. The analysis would give an insight into whether the operator is simple enough to generate an understandable output and complex enough to deal with different cases of contraction.

4. Prototype implementation

   A prototype implementation of the operator shall be created. The application is based on the algorithms that we designed for the concept contraction. The implementation of the operator will be completed in Python.

On completion, the result of this master thesis would be the study and implementation of the *concept contraction operator*, including a selection function, the remainders, and the least common subsumer. A final analysis is interesting to understand the results of the implementation.

1.4 Scope of the thesis

During the course of the thesis, we have made several assumptions, especially in the design of the algorithms for the computation of the weakenings of a concept, the remainders of a concept C with respect to a concept D, and the least common subsumer.

One of the main modeling choices of the concept contraction operator is the Description Logic language used to describe it, $\mathcal{EL}$. Although a limiting language in terms of expressivity, it is sufficient to define the concept contraction. Nonetheless, the lack of expressivity for the *union* operation forces us to use the notion of the least common subsumer.

Another assumption of the thesis is that in the discussion of the concept contraction operation, we work in the setting of an empty T-Box. This assumption is reflected in the design of the algorithms, where we deal directly with the concepts. Albeit, we do provide a short description of the behaviour of the concept contraction in the setting of acyclic and cyclic T-Boxes.

Lastly, in the first part of this work, where we discuss the concept contraction, we assume that the selection function is given. In this part, the way that the selection function selects the best remainders from a set is not our concern. In addition, the algorithms are built upon this assumption. This limits the analysis of the $conceptContraction$ algorithm, as we cannot determine the complexity of the selection function. In the second part of the thesis, we consider the development of a possible selection function for concept contraction, but the approach is not taken into account for analysis.

Chapter 2

Concept contraction in the Description Logics $\mathcal{EL}$

The chapter covers the theoretical background of the concept contraction in the Description Logics $\mathcal{EL}$. In section 2.1, we introduce the Description Logics family and the three basic ideas that have shaped the development of DLs. In section 2.2, we focus mainly on the description logic $\mathcal{EL}$, where we discuss the constructors of the language and its semantics. Later on, in section 2.3, we discuss in general the contraction operation in the AGM theory. We present the basic six associative postulates of the operation and provide a short description for each. In section 2.4, the concept contraction approach developed by Rienstra, Schon and Staab in [Rienstra et al., 2018] is analysed thoroughly. The concept contraction operator is discussed in great detail for different settings. Lastly, a summary of conclusions is provided for the chapter in section 2.6.

2.1 Description Logics

In this section, we briefly introduce Description Logics (DL) as a formalism for representing knowledge of an application domain. We state the main ideas that shape description logics and, the architecture of the knowledge representation system is presented. Basic definitions and the theory background included in this section are provided in 'The Description Logic Handbook: Theory, Implementation, and Applications' by [Baader et al., 2003].

Description logics (DLs) are a family of logic-based knowledge representation formalisms, which are employed in various application domains, such as natural language processing, configuration, databases, and biomedical ontologies. Their most notable

success so far is the adoption of the DL-based language OWL[1] as standard ontology language for the Semantic Web.

The following three ideas, first put forward in Brachman's work [Brachman, 1985], have largely shaped the development of DLs:

- The basic syntactic building blocks are atomic concepts, atomic roles, and individuals.
- The expressive power of the language is restricted in that it uses a rather small set of constructors for building complex concepts and roles.
- Implicit knowledge about concepts and individuals can be inferred automatically with the help of inference procedures. Subsumption relationships between concepts and instance relationships between individuals and concepts play an important role.

In figure 2.1, the architecture of a knowledge representation system based on description logics is presented. This knowledge representation system provides facilities to set up knowledge bases, to reason about their content, and to manipulate them. The source of the figure is [Baader et al., 2003]. The notions of T-Box and A-Box are described in the following section.

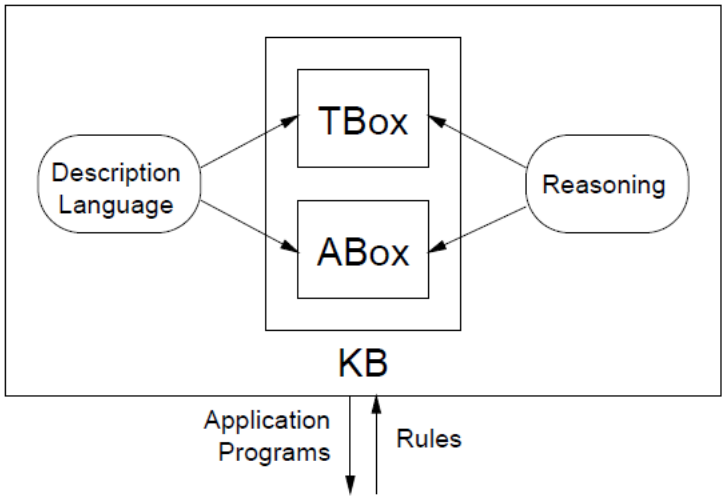

Figure 2.1: Architecture of a knowledge representation system based on description logics.

[1] See https://www.w3.org/TR/owl2-overview/ for its most recent edition OWL2.

A key component of a DL is the description language, which allows its users to build complex concepts (and roles) out of atomic ones. These descriptions can then be used in the terminological part of the knowledge base (TBox) to introduce the terminology of an application domain, by defining concepts and imposing additional constraints on their interpretation. In the assertional part of the knowledge base (ABox), facts about a specific application situation can be stated, by introducing named individuals and relating them to concepts and roles [Baader, 2017].

2.2 Preliminaries of Description Logic $\mathcal{EL}$

Having introduced the notion of description logics, we now focus on $\mathcal{EL}$ in particular. We will cover the basic definitions and semantics for this knowledge representation language. The definitions and the knowledge presented in this section are based on [Baader et al., 2005] and [Rienstra et al., 2018].

$\mathcal{EL}$ is a small description logic that allows only a limited set of concept constructors. Firstly, we define the signature of the language.

Definition 1 *An $\mathcal{EL}$ signature is a pair $\Sigma = (N_C, N_R)$ where N_C is the set of atomic concepts and N_R the set of atomic roles.*

Given a signature $\Sigma = (N_C, N_R)$, we use A to denote an atomic concept and R to denote an atomic role or the universal role u. The notations C and D are used to range over concepts, which are formulas inductively generated by the rules presented in the following definition. $\mathcal{EL}$ concept descriptions are defined as:

Definition 2 *Let N_C be a set of concept names and N_R a set of role names. The set of $\mathcal{EL}$ concept descriptions is the smallest set such that:*

- $\top$ *and all concept names $A \in N_C$ are $\mathcal{EL}$ concept descriptions;*
- *if C and D are $\mathcal{EL}$ concept descriptions, then so is $C \sqcap D$;*
- *if C is a $\mathcal{EL}$ concept descriptions and $r \in N_R$, then $\exists r.C$ is an $\mathcal{EL}$ concept description.*

Based on the definition, a concept in Description Logic $\mathcal{EL}$ can either be a universal concept, an atomic concept, an existential role restriction or a conjunction of concepts. Given a signature Σ, the set of concepts will also be denoted $\mathcal{C}(\Sigma)$. For example, the concept $Student$ in $\mathcal{EL}$ can be expressed as:

Example 1. A concept in $\mathcal{EL}$ description logics

$$Person \sqcap \exists isRegisteredAt.University$$

The semantics of $\mathcal{EL}$ is defined in terms of *interpretations*. An *interpretation* I is a pair $(\Delta^I, .^I)$, where Δ^I in a non-empty set called the *domain* and $.^I$ is an *interpretation function*. The interpretation maps each $A \in N_C$ to a set $A^I \subseteq \Delta^I$ and each $R \in N_R$ to a set $R^I \subseteq \Delta^I \times \Delta^I$. The semantics of the universal role u is given by $u^I = \Delta^I \times \Delta^I$. The universal role connects all the elements in the domain. The interpretation function maps every individual a to an element a^I of the domain $a^I \in \Delta^I$. The interpretation function is extended to concepts using the following definitions [Baader et al., 2003]:

- $\top^I = \Delta^I$
- $(C \sqcap D)^I = C^I \cap D^I$
- $(\exists R.C)^I = \{a \in \Delta^I | (a, b) \in R^I \text{ for some } b \in C^I\}$.

Definitions 3 and 6, included in [Heinz, 2018], provide a convenient way to implement the algorithms in chapter 3. Two additional definitions, 4 and 5, have been created with the purpose of defining the set of concepts that form a concept of type conjunction.

Definition 3 *A conjunction of concepts is said to be trivial iff it consists of exactly one conjunct. It is said to be non-trivial iff it is not a trivial conjunction.*

Definition 4 *If a non-trivial conjunction is constructed by the intersection of conjunction concepts, the non-trivial conjunction concept must be flattened. The flattening procedure consists on the removal of the parenthesis in the non-trivial conjunction concept.*

The construction of a non-trivial conjunction concept using the flattening procedure is expressed in example 2.

Example 2. Flattening of a non-trivial conjunction concept

$$\begin{gathered} A \sqcap (C \sqcap \exists r.D) \\ \Downarrow \\ A \sqcap C \sqcap \exists r.D \end{gathered}$$

Definition 5 *The set Conjuncts(C) is defined to contain all concepts that occur within a non-trivial conjunction concept C.*

Example 3. $Conjuncts(C)$

$$Conjuncts(A \sqcap C \sqcap \exists r.D) = \{A, C, \exists r.D\}$$

Definition 6 *The notation InnerConcept(C) is defined to contain the concept that occurs within an existential role restriction concept, C.*

Example 4. $InnerConcept(C)$

$$InnerConcept(\exists r.(C \sqcap D)) = \{C \sqcap D\}$$

A description logic knowledge base (KB) is made up of two parts, a terminological part, called the T-Box and an assertional part, called the A-Box, each part consisting of a set of axioms. The most general form of T-Box axioms are the so-called general concept inclusions.

Definition 7 *A general concept inclusion (GCI) is of the form $C \sqsubseteq D$, where C, D are $\mathcal{EL}$ concepts.*

The terminological axioms can also be expressed by concept equivalence inclusion of the form $C \equiv D$, where C, D are $\mathcal{EL}$ concepts. We can now define a T-Box.

Definition 8 *A T-Box $\mathcal{T}$ with signature $\Sigma = (N_C, N_R)$ is a finite set of definitions of the form $A \equiv C$ or $A \sqsubseteq C$ with $A \in N_C$ and $C \in \mathcal{C}(\Sigma)$ such that no A appears more than once on the left-hand side of a definition in $\mathcal{T}$.*

An example of a T-Box that includes the definition of a $Student$ concept is illustrated in example 5. Part of the T-Box can be numerous terminological axioms in the domain of a university, such as definitions of a professor, class, examination, grade, etc.

Example 5. T-Box $\mathcal{T}$

$$\mathcal{T} = \{Student \equiv Person \sqcap \exists isRegisteredAt.University\}$$

An A-Box can contain two kinds of axioms, one for asserting that an individual is an instance of a given concept, and the other for asserting that a pair of individuals is an instance of a given role. The definition of the A-Box, provided in [Baader et al., 2008], is:

Definition 9 *An assertional axiom is of the form $x : C$ or $(x, y) : r$, where C is an $\mathcal{EL}$ concept, r is an $\mathcal{EL}$ role, and x and y are individual names. A finite set of assertional axioms is called an A-Box.*

An A-Box example, given that $Anna$ and $UniKoblenz$ are individual names that respectively represent a student and a university, can be:

Example 6. A-Box $\mathcal{A}$

$$\mathcal{A} = \{Anna : Student, (Anna, UniKoblenz) : isRegisteredAt\}$$

Now that the definitions of a T-Box and an A-Box are provided, we can define a knowledge base.

Definition 10 *A knowledge base (KB) is a pair $(\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a T-Box and $\mathcal{A}$ is an A-Box.*

We can develop further the definition by analysing inference problems w.r.t. a knowledge base consisting of an T-Box and an A-Box.

Definition 11 *Given a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a T-Box and $\mathcal{A}$ is an A-Box, $\mathcal{K}$ is called consistent if it has a model.*

A concept C is called specifiable with respect to $\mathcal{K}$ if there is a model I of $\mathcal{K}$ with

$$C^I \neq \varnothing.$$

Such an interpretation is called a model of C w.r.t. $\mathcal{K}$.

The concept D subsumes the concept C w.r.t $\mathcal{K}$ (written $\mathcal{K} \vDash C \sqsubseteq D$) if for all models I of $\mathcal{K}$ the condition holds:

$$C^I \subseteq D^I$$

Two concepts C, D are equivalent w.r.t $\mathcal{K}$ (written $\mathcal{K} \vDash C \equiv D$) if they subsume each other w.r.t $\mathcal{K}$.

$$C \sqsubseteq D \text{ and } D \sqsubseteq C$$

Furthermore, a concept C is equivalent to D w.r.t. $\mathcal{T}$, if for all models I of $\mathcal{T}$ the condition holds:

$$C^I = D^I$$

An individual a is an instance of a concept C with respect to $\mathcal{K}$ (written $\mathcal{K} \vDash a : C$) if for all models I of $\mathcal{K}$ the condition holds:

$$a^I \in C^I$$

A pair of individuals (a, b) is an instance of a role r with respect to $\mathcal{K}$ (written $\mathcal{K} \vDash (a, b) : r$) if for all models I of $\mathcal{K}$ the condition holds:

$$(a^I, b^I) \in r^I$$

Definition 12 *An interpretation I is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if I is a model of $\mathcal{T}$ and I is a model of $\mathcal{A}$.*

Example 7. Knowledge base $\mathcal{K}$

$$\mathcal{T} = \{Student \equiv Person \sqcap \exists isRegisteredAt.University\}$$
$$\mathcal{A} = \{Anna : Student, (Anna, UniKoblenz) : isRegisteredAt\}$$

It is important to emphasise that in section 2.4, where we discuss the concept contraction operator by [Rienstra et al., 2018], there are several assumptions on the description logics $\mathcal{EL}$. Firstly, the terminological axioms included in the T-Box can only be expressed by the concept equivalence inclusion ($C \equiv D$). Secondly, A-Boxes will not be used in the development of the concept contraction operator, as the operator deals with concepts in the T-Box, and not individuals in A-Box. Lastly, the concept contraction operator does not take the universal role u into account. Nonetheless, this knowledge is important to understand description logics, and thus is provided.

2.3 Contraction in the AGM Theory

In this section we introduce the epistemic changes of belief sets, based on the AGM theory. The attention is on the *contraction* operation where we will also introduce and discuss the associative postulates. The knowledge included in this section is based on [Alchourrón et al., 1985] and [Gärdenfors et al., 1995].

Belief revision is the subarea of knowledge representation which studies the dynamics of *epistemic states* of an agent. Belief sets, which are sets of sentences logically closed, are composed of three pieces: a representation of epistemic states, a set of epistemic attitudes, and types of belief change. An agent's *epistemic state* is the set of beliefs of the agent at a certain moment. The changes of attitudes are fired by an external trigger called epistemic input. This input can lead to several kinds of epistemic changes called:

- **Expansion** - A new sentence is added to a belief set $\mathcal{K}$ regardless of the consequences of the larger set so formed. The belief set that results from expanding $\mathcal{K}$ by a sentence $\mathcal{A}$ together with the logical consequences is denoted $\mathcal{K} + \mathcal{A}$.
- **Revision** - A new sentence that is (typically) inconsistent with a belief set $\mathcal{K}$ is added, but in order that the resulting belief system to be consistent some of the old sentences in $\mathcal{K}$ are deleted. The result of revising $\mathcal{K}$ by a sentence $\mathcal{A}$ is denoted $\mathcal{K} * \mathcal{A}$.

- **Contraction** - Some sentence in the belief set $\mathcal{K}$ is retracted without adding any new facts. The result of contracting $\mathcal{K}$ with respect to the sentence $\mathcal{A}$ is denoted $\mathcal{K} \div \mathcal{A}$.

Based on the short description of the epistemic changes, a similarity between revision and contraction is noted. Nonetheless, we should note that there is a distinctions between belief revision and belief contraction. The AGM approach to belief change states that while belief revision deals with consistently adding a new belief, contraction deals with removal of beliefs. Contraction is seen as a more fundamental type of change, since revision can often be defined in terms of contraction, by first removing beliefs inconsistent with the new belief, after which the new belief can be added without introducing inconsistency [Alchourrón et al., 1985].

AGM theory studies changes in epistemic states modelled as logically closed sets of sentences. AGM theory deals with *contraction*, with properties expressed by six postulates: (i) Closure, (ii) Inclusion, (iii) Vacuity, (iv) Success, (v) Recovery and (vi) Extensionality. The following definitions and explanations of the postulates can be found in [Gärdenfors et al., 1995] and [Hansson, 2017].

The first postulate is **Closure** which states that when a belief set $\mathcal{K}$ is contracted by a sentence $\mathcal{A}$, the outcome should be logically closed.

Closure ($\mathcal{K} \div 1$). For any sentence $\mathcal{A}$ and any belief set $\mathcal{K}$, $\mathcal{K} \div \mathcal{A}$ is a belief set.

According to postulate **Inclusion**, the contracted set is assumed to be a subset of the original set. Inclusion is usually considered to be a constitutive property of contraction. However, it has also been questioned with the argument that when the epistemic agent ceases to believe in $\mathcal{A}$, then this is usually because it receives some new information that contradicts $\mathcal{A}$.

Inclusion ($\mathcal{K} \div 2$). For any sentence $\mathcal{A}$ and any belief set $\mathcal{K}$, $\mathcal{K} \div \mathcal{A} \subseteq \mathcal{K}$.

If the sentence to be contracted is not included in the original belief set, then contraction by that sentence involves no change at all. Such contractions are vacuous operations, and they should leave the original set unchanged.

Vacuity ($\mathcal{K} \div 3$). For any sentence $\mathcal{A}$ and any belief set $\mathcal{K}$, if $\mathcal{A} \notin \mathcal{K}$, then $\mathcal{K} \div \mathcal{A} = \mathcal{K}$.

Contraction should be successful, i.e., $\mathcal{K} \div \mathcal{A}$ should not imply $\mathcal{A}$. However, it would be too much to require that $\mathcal{A} \notin \mathcal{K} \div \mathcal{A}$ for all sentences $\mathcal{A}$, since it cannot hold if $\mathcal{A}$ is a tautology. The success postulate has to be conditional on $\mathcal{A}$ not being logically true.

Success ($\mathcal{K} \div 4$). For any sentence $\mathcal{A}$ and any belief set $\mathcal{K}$, if not $\vdash \mathcal{A}$, then $\mathcal{A} \notin \mathcal{K} \div \mathcal{A}$.

Belief contraction should not only be successful, it should also be *minimal* in the sense of leading to the loss of as few previous beliefs as possible. The epistemic agent should give up beliefs only when forced to do so, and should then give up as few of them as possible. According to the **Recovery** postulate, so much is retained after $\mathcal{A}$ has been removed that everything will be recovered by expansion of $\mathcal{A}$.

Recovery ($\mathcal{K} \div 5$). For any sentence $\mathcal{A}$ and any belief set $\mathcal{K}$, $\mathcal{K} \subseteq (\mathcal{K} \div \mathcal{A}) + \mathcal{A}$.

Logically equivalent sentences should be treated alike in contraction. Extensionality guarantees that the logic of contraction is extensional in the sense of allowing logically equivalent sentences to be freely substituted for each other.

Extensionality ($\mathcal{K} \div 6$). For any sentence $\mathcal{A}$ and $\mathcal{B}$ and any belief set $\mathcal{K}$, if $\vdash \mathcal{A} \leftrightarrow \mathcal{B}$, then $\mathcal{K} \div \mathcal{A} = \mathcal{K} \div \mathcal{B}$.

The postulates ($\mathcal{K} \div 1$) to ($\mathcal{K} \div 6$) are called the *basic* set of postulates for contraction.

In the next section, we will see how these basic postulates have been adapted to the setting of the concept contraction. Firstly, let us introduce the notion of concept contraction.

2.4 Concept Contraction Operator

The foundation of the work in this master thesis is build on the **concept contraction** presented in [Rienstra et al., 2018]. In this section we describe in detail the operator and its properties. The basic postulates of belief contraction have been reformulated to fit the setting of the operator. In the end of the section, we describe the assumptions made for the development of the algorithms in chapter 3.

The concept contraction operator deals with the removal of single concepts from a certain knowledge base, instead of a whole axiom. The goal is, given two concepts C and D, the operator should be able to obtain a new concept that is as similar as possible to C but no longer subsumed by D. Contraction in this context is understood as finding a weakening of C (i.e., some concept C' such that $C \sqsubseteq C'$) that is no longer subsumed by D [Rienstra et al., 2018].

The authors model the process of concept contraction using the notion of a *concept*

contraction operator $\ominus$. The operator is defined by adapting the notion of AGM partial and full meet contraction. The definitions related to the concept contraction operator can be found in [Rienstra et al., 2018].

Definition 13 *A concept contraction operator* $\ominus$ *with signature* Σ *associates each pair of concepts* $C, D \in \mathcal{C}(\Sigma)$ *with a new concept denoted by* $C \ominus D \in \mathcal{C}(\Sigma)$.

2.4.1 LCS contraction

The authors in [Rienstra et al., 2018] present an explicit construction of a particular kind of contraction operator called *lcs contraction* operators. Initially the notion of *remainders* is introduced.

Definition 14 *A remainder of a concept* C *with respect to a concept* D *is a concept* C' *such that:*

1. $C \sqsubseteq C'$
2. $C' \not\equiv D$
3. $\nexists C''$ *s.t.* $C \sqsubseteq C'', C'' \not\equiv D$ *and* $C'' \sqsubseteq C'$.

Once the remainders C with respect to D have been identified, they can be considered candidate answers for contracting D from C. The set of remainders are closed under equivalence (i.e., if $C \equiv C''$ and C' is a remainder of C w.r.t. D then so is C''). Considering that a remainder of a concept C w.r.t. a concept D is a maximally specific generalization of C not subsumed by D, we can state that at least one remainder of C with respect to D always exists, provided that $D \not\equiv \top$. Otherwise, there can not be found a remainder of concept C that is not subsumed by the $\top$ concept. The sets of remainders are represented as sets of equivalent classes under $\equiv$. The equivalence class of a concept C is the set $\{D | C \equiv D\}$ and is denoted by $[C]$. Next, we introduce the notion of the $C \perp D$ set.

Definition 15 *Let* C, D *be concepts. We define* $C \perp D$ *by* $C \perp D = \{[C'] | C'$ *is a remainder of* C *w.r.t.* $D\}$.

In general, $C \perp D$ may contain more than one remainder. Here comes in play the role of a selection function σ that makes it possible to chose the 'most important' remainders. There are two limiting cases are (1) selection of single remainders (*maxi-choice*) and (2) selection of all remainders (*full meet*) that are considered in [Rienstra et al., 2018]. Let us define what a selection function is.

Definition 16 *A selection function σ selects, given every pair of concepts C, D a set $\sigma(C \perp D)$ such that:*

1. *If $C \perp D \neq \varnothing$ then $\sigma(C \perp D) \neq \varnothing$.*
2. *If $C \perp D \neq \varnothing$ then $\sigma(C \perp D) \subseteq C \perp D$.*
3. *If $C \perp D \neq \varnothing$ then $\sigma(C \perp D) = \{[C]\}$.*

A selection function σ is called *maxi-choice* iff σ selects exactly one elements of $C \perp D$ and *full meet* iff it selects all elements of $C \perp D$.

It is evident that the selection function may select more than one remainder. In the AGM contraction this fact is dealt with by intersecting all belief sets chosen by the selection function. This intersection is itself a belief set (i.e., is closed under logical consequence) and contains all beliefs that the chosen belief sets have in common [Rienstra et al., 2018]. In this setting of the concept contraction operator, this method corresponds to the union of the chosen remainders. The problem is that the union is not an allowed constructor in the description logic $\mathcal{EL}$. The authors, then, propose the use of the *least common subsumer* notion. Based on [Baader et al., 1999], the *most specific concept* (msc) of an individual b is the most specific concept description C (expressible in the given description logics) that has b as an instance, and the *least common subsumer* (lcs) of n concept descriptions $C_1, ..., C_n$ is the most specific concept description in the given description logic that subsumes $C_1, ..., C_n$.

The formal definition of the least common subsumer provided in [Rienstra et al., 2018] is:

Definition 17 *Let $C_1, ..., C_n$ be concepts. A concept C is a least common subsumer of $C_1, ..., C_n$ iff:*

1. *$C_1, ..., C_n \sqsubseteq C$,*
2. *For each C' s.t. $C_1, ..., C_n \sqsubseteq C'$ we have $C \sqsubseteq C'$.*

Assuming that the concepts $C_1, ..., C_n$ and $C'_1, ..., C'_n$ are given, such that $C_1 \equiv C'_1, ..., C_n \equiv C'_n$, we have that C is an *lcs* of $C_1, ..., C_n$ iff C is an *lcs* of $C'_1, ..., C'_n$. Considering this behaviour, the authors define lcs-s as a function of a set of equivalence classes under $\equiv$.

Definition 18 *Let $\mathcal{X} = \{[C_1, ..., C_n]\}$ be a set of equivalence classes under $\equiv$. We denote by lcs($\mathcal{X}$) the set of least common subsumers of $C_1, ..., C_n$.*

For any set of concepts, an lcs always exists and is unique up to equivalence. This implies that *lcs*($\mathcal{X}$) coincides with an equivalence class under ≡ [Rienstra et al., 2018]. Finally, the class of *lcs contraction* operators is defined.

Definition 19 *Let σ be a selection function. A contraction operator $\ominus$ is an lcs contraction operator defined by σ iff for all concepts C and D,*

$$C \ominus D \in lcs(\sigma(C \bot D)$$

The concept contraction operator $\ominus$ is called maxi-choice contraction operator if it is defined by a maxi-choice selection function and full meet if it is defined by a full meet selection function.

It is understandable that lcs contraction operators are dependable from the selection functions. Assuming that two lcs contraction operators $\ominus$ and $\ominus'$ are both defined from the same selection function σ, we can say that for all concepts C and D, the following holds:

$$C \ominus D \equiv C \ominus' D$$

For this reason, we refer to a lcs concept contraction operator defined by σ and denote this operator by $\ominus^{\sigma}$.

The concept contraction - like standard AGM contraction - is in general not uniquely determined. Only full meet contraction yields a uniquely determined operator. Furthermore, the principle of minimal change is best captured by a maxi-choice operator, but requires one to select single remainders, which may not be feasible [Rienstra et al., 2018]. Let us consider the following example.

Example 1. Let σ be a selection function. Suppose we want to determine the contraction:

$$Person \sqcap \exists isRegisteredAt.(University \sqcap Institute) \ominus^{\sigma} \exists isRegisteredAt.Institute$$

In figure 2.2, we present the lattice of the generalizations of a conjunction concept C formed by the intersection of an atomic concept and a role restriction concept (the arrows point to more general concepts). The remainder of $Person \sqcap \exists isRegisteredAt.(University \sqcap Institute)$ with respect to $\exists isRegisteredAt.Institute$ is enclosed in a dotted rectangle. Given that this a single remainder, it is also the lcs. Hence, the contraction operation result in example 1 equals $Person \sqcap \exists isRegisteredAt.University$ regardless of the selection function.
Now let us consider the following example.

Example 2. Let σ be a selection function. Suppose we want to determine the contraction:

$$Person \sqcap \exists isRegisteredAt.University \sqcap \exists attends.Lecture \ominus^{\sigma}$$
$$\exists isRegisteredAt.Institute \sqcap \exists attends.Lecture$$

In this case the remainders of the concept contraction are more than one. In figure 2.3, the remainders are enclosed inside a rectangle. The least common subsumer of these two concepts is the concept $Person$, also enclosed inside a rectangle. If the σ is a maxi-choice selection function, the result of the contraction is either of the remainders. If σ is a full meet selection function, then the contraction result is the least common subsumer, $Person$.

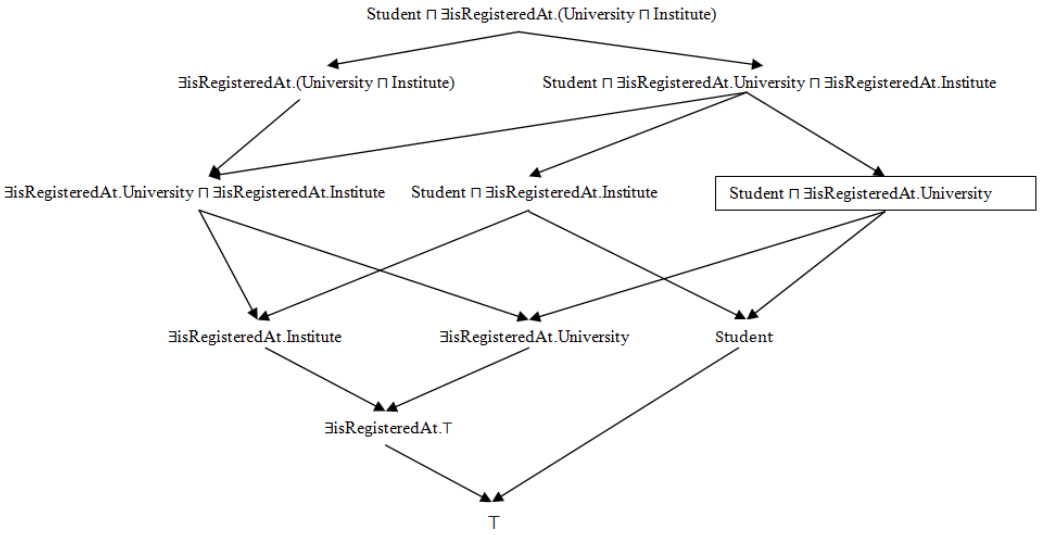

Figure 2.2: The generalization lattice for example 1

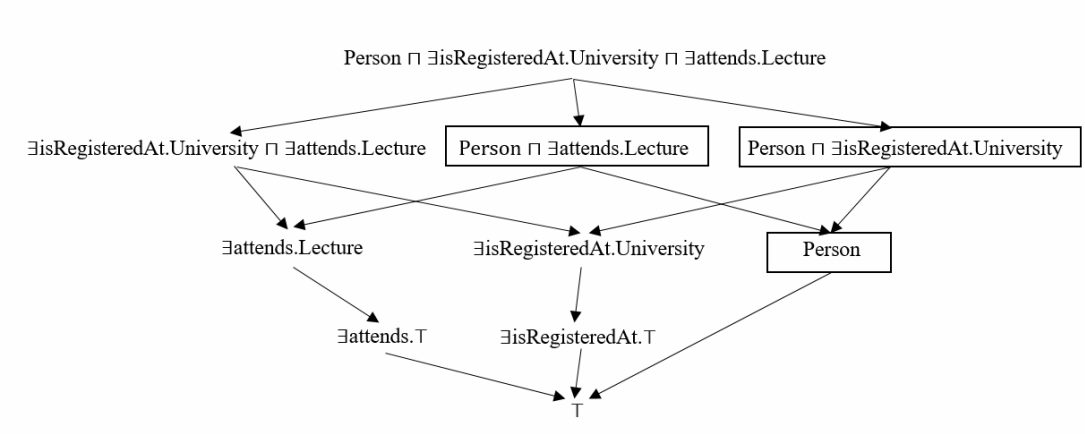

Figure 2.3: The generalization lattice for example 2

2.4.2 Postulates for Concept Contraction

In this section we present a set of reformulated AGM postulates for contraction adapted for the concept contraction operator. The postulates are described in the setting of the operator, where the modifications to the AGM theory are explained. We include two formulated theorems regarding the concepts contraction operator and the postulates that it satisfies.

The first four basic AGM contraction postulates are reformulated as follows and can be found in [Rienstra et al., 2018].

In the **Preservation** postulate is stated that given two equivalent concepts $D \equiv D'$, the results of the contraction of either of these concepts from concept C are equivalent as well.

Preservation: If $D \equiv D'$, then $C \ominus D \equiv C \ominus D'$.

As discussed, at least one remainder always exists unless $D \not\equiv \top$. Assuming that $D \not\equiv \top$, the contraction is successful, e.i. the result of the concept contraction $C \ominus D$ is not subsumed by D. This property is formulated in the **Success** postulate.

Success: If $D \not\equiv \top$ then $C \ominus D \not\sqsubseteq D$.

The **Inclusion** postulate states that the result of the concept contraction $C \ominus D$ is a weakening of concept C.

Inclusion: $C \sqsubseteq C \ominus D$

The fourth postulate **Vacuity** is rather straightforward and it is understandable that if concept C is not subsumed by concept D, then the concept contraction is not applicable.

Vacuity: If $C \not\sqsubseteq D$ then $C \ominus D \equiv C$

The sixth AGM postulate is Recovery and is formulated below. Like in the AGM setting, the converse of Recovery, i.e., $C \sqsubseteq ((C \ominus D) \sqcap D)$, follows from Inclusion, provided that we have $C \sqsubseteq D$. The postulate is not satisfied in the setting that we are discussing. The reason way be a consequence of the limited expressivity of description logics $\mathcal{EL}$. A suitable replacement of the **Recovery** postulate is needed [Rienstra et al., 2018].

Recovery: $(C \ominus D) \sqcap D \sqsubseteq C$.

Firstly, we present the **Failure** postulate. It is evident that in case when concept D is a $\top$ concept, the concept contraction $C \ominus \top$ is equivalent to concept C. According to [Rienstra et al., 2018], in the presence of Inclusion, Failure follows Recovery. However, Failure does not follow if Recovery is replaced with any of the postulates that are be discussed shortly below. Therefore, Failure needs to be considered explicitly.

Failure: $(C \ominus \top) \equiv C$.

The first replacement for Recovery considered is *Relevance*. It is a reformulation of the Relevance postulate in [Hansson, 1991] and can be thought of as expressing a principle of minimal change. It stated that, if we lose X after contracting D, then adding X to $(C \ominus D)$ gives D again.

Relevance: If $C \sqsubseteq X$ and $(C \ominus D) \not\equiv X$ then $(C \ominus D) \sqcap X \sqsubseteq D$.

The **Vacuity** postulate now becomes redundant. Therefore, the authors in [Rienstra et al., 2018] present the following preposition.

Preposition 1 *If $\ominus$ satisfies Inclusion and Relevance then it satisfies Vacuity.*

Together with basic postulates, Relevance fully characterises maxi-choice lcs concept contraction:

Theorem 1 *Let $\ominus$ be a concept contraction operator. The following are equivalent:*

1. *$\ominus$ is a maxi-choice lcs concept contraction operator.*
2. *$\ominus$ satisfies Preservation, Inclusion, Success, Failure and Relevance.*

The proof of theorem 1 can be found in [Rienstra et al., 2018]. This theorem considers only the case when $\ominus$ is a maxi-choice lcs concept contraction operator, and does not cover the lcs contraction in general. The following weakening of Relevance, which is a reformulation of what is called Core-Retainment in [Hansson, 1991], characterises lcs concept contraction in general [Rienstra et al., 2018].

Retainment: If $C \sqsubseteq X$ and $(C \ominus D) \not\equiv X$, then there is a Y s.t. $C \sqsubseteq Y \sqsubseteq C \ominus D$ and $Y \not\equiv D$ and $Y \sqcap X \sqsubseteq D$.

Finally, we can introduce a theorem that represents the lcs concept contraction operator as a whole, and not specific types of operators. Akin to theorem 1, the proof of this theorem can be found in [Rienstra et al., 2018].

Theorem 2 *Let $\ominus$ be a concept contraction operator. The following are equivalent:*

1. $\ominus$ *is an lcs concept contraction operator.*

2. $\ominus$ *satisfies Preservation, Inclusion, Success, Failure and Retainment.*

This section marks the end of the discussion of the concept contraction operator's construction and properties. So far, we have considered concept contraction involving concepts whose semantics do not depend on a T-Box. We have to accentuate the fact that the concept contraction operator is proved to work when the T-Box is considered. A short summary of the theory about this case is presented in the next section.

2.4.3 Contraction modulo acyclic and cyclic T-Boxes

In this section, we describe the concept contraction when there exists a non-empty T-Box. Considering that for the developing of the algorithms we assume that the T-Box is empty, the information provided is summarized and not explained in great detail. The definitions and arguments included in this section are based on [Rienstra et al., 2018]. Further information about this case, together with proofs of the theorems can be found in the fore-mentioned literature.

Primarily, we will define what cyclic and acyclic T-Boxes are. The definition is based on [Baader et al., 2003].

Definition 20 *A T-Box $\mathcal{T}$ contains a cycle iff there exists an atomic concept in $\mathcal{T}$ that uses itself in a concept definition. Otherwise, $\mathcal{T}$ is called acyclic.*

Let us analyse the setting where there is an acyclic background T-Box. An example of such a T-Box is presented in example 3.

Example 3. An acyclic T-Box $\mathcal{T}$

$$\mathcal{T} = \{Student \equiv Person \sqcap \exists isRegisteredAt.University, \\ Professor \equiv Person \sqcap \exists teaches.Lecture\}$$

For every acyclic TBox $\mathcal{T}$, we can effectively construct an equivalent acyclic TBox $\hat{\mathcal{T}}$ such that the right-hand sides of concept definitions in $\hat{\mathcal{T}}$ contain only primitive concepts. Given an acyclic T-Box $\mathcal{T}$, we define $\hat{\mathcal{T}}(\cdot)$ by:

- $\hat{\mathcal{T}}(A) = A$, if A does not appear on the left-hand side of a definition in $\mathcal{T}$;
- $\hat{\mathcal{T}}(A) = \hat{\mathcal{T}}(C)$, if $A \equiv C$ appears in $\mathcal{T}$;

- $\hat{\mathcal{T}}(C \sqcap D) = \hat{\mathcal{T}}(C) \sqcap \hat{\mathcal{T}}(D)$;
- $\hat{\mathcal{T}}(\exists R.C) = \exists R.\hat{\mathcal{T}}(C)$.

$\hat{\mathcal{T}}(\cdot)$ is well-defined only if $\hat{\mathcal{T}}$ is acyclic. When we define contraction with respect to a T-Box $\hat{\mathcal{T}}$ we need to determine the signature of $\hat{\mathcal{T}}$ consisting only of *base concepts*.

Definition 21 *Given a T-Box $\hat{\mathcal{T}}$ with signature $\Sigma = (N_C, N_R)$, we say that an atomic concept $A \in N_C$ is a base concept of $\hat{\mathcal{T}}$ iff A does not appear on the left-hand side of a definition in $\hat{\mathcal{T}}$.*

The set of base concepts of $\hat{\mathcal{T}}$ is denoted by $\mathcal{B}_{\mathcal{T}}$. The *base signature* of $\hat{\mathcal{T}}$ is the signature (B_T, N_R) and is denoted by $\Sigma_{\downarrow\mathcal{T}}$. If $\mathcal{T}$ is an acyclic T-Box with signature Σ then for every concept $C \in \mathcal{C}(\Sigma)$, we have $\mathcal{T}(C) \in \mathcal{C}(\Sigma_{\downarrow\mathcal{T}})$.

Given a T-Box $\mathcal{T}$ with signature (N_C, N_R) and a concept contraction operator $\ominus$ with signature $(B_{\mathcal{T}}, N_R)$, an operator $\ominus^{\uparrow\mathcal{T}}$ with signature (N_C, N_R) can be defined as follows.

$$C \ominus^{\uparrow\,\mathcal{T}} D \equiv^{\mathcal{T}} \hat{\mathcal{T}}(C) \ominus \hat{\mathcal{T}}(D)$$

To account for a T-Box $\mathcal{T}$ during contraction, the **Inclusion** postulate can be adapted by requiring that $C \sqsubseteq^{\mathcal{T}} C \ominus D$; and **Success** by requiring that if $D \not\equiv^{\mathcal{T}} \top$, then $C \ominus D \not\equiv^{\mathcal{T}} D$. All the postulates discussed in the above section can be adapted in a similar way. That is, they are defined relative to a T-Box $\mathcal{T}$ and replace $\sqsubseteq$ with $\sqsubseteq^{\mathcal{T}}$. All these postulates are equivalent to the original versions when we assume $\mathcal{T}$ is empty. The postulates in this case are named *$\mathcal{T}$-postulate name*.

Preposition 2 *$\ominus$ satisfies Inclusion (resp. Success, Failure, etc.) if and only if $\ominus^{\uparrow\mathcal{T}}$ satisfies $\mathcal{T}$-Inclusion (resp. $\mathcal{T}$-Success, $\mathcal{T}$-Failure, etc).*

Theorem 3 *Let $\mathcal{T}$ be a T-Box and $\ominus'$ be a contraction operator, both with signature (N_C, N_R). The following are equivalent:*

1. *$\ominus' = \ominus^{\uparrow\mathcal{T}}$ for some maxi-choice lcs contraction operator $\ominus$ with signature $(\mathcal{B}_{\mathcal{T}}, N_R)$.*
2. *$\ominus'$ satisfies $\mathcal{T}$-Inclusion, $\mathcal{T}$-Success, $\mathcal{T}$-Failure and $\mathcal{T}$-Relevance.*

The following are also equivalent:

1. *$\ominus' = \ominus^{\uparrow\mathcal{T}}$ for some lcs contraction operator $\ominus$ with signature $(\mathcal{B}_{\mathcal{T}}, N_R)$.*
2. *$\ominus'$ satisfies $\mathcal{T}$-Inclusion, $\mathcal{T}$-Success, $\mathcal{T}$-Failure and $\mathcal{T}$-Retainment.*

The theorem concludes that a concept contraction operator can perform in the same way in the setting when there exists an acyclic T-Box, as to the setting where the T-Box is empty. The concept contraction is well-defined and proved for this scenario.

Now, let us consider the case when the T-Box $\mathcal{T}$ is cyclic. If $\mathcal{T}$ is not acyclic then $\hat{\mathcal{T}}(\cdot)$ is not well-defined, so the approach taken in the acyclic T-Box case does not work. There are fundamental problems preventing from modelling contraction modulo cyclical T-Boxes. In [Rienstra et al., 2018] it is argued that the fact that a remainder of C with respect to D always exist (unless $D \equiv \top$) does not hold if $\mathcal{T}$ contains cycles. It has been shown that the least common subsumer may also not exist in the presence of cycles. Thus, the notion of lcs contraction cannot be applied to cyclical T-Boxes in a straightforward way.

2.4.4 Limitations of concept contraction

The theory of a concept contraction operator as a means of a more gentle approach to AGM contraction analysed in this chapter comes with its limitations. We will briefly discuss limitation/modeling choices caused by the Description Logic $\mathcal{EL}$ and the behaviour of the concept contraction operation in specific settings. This section will shortly summarise the arguments provided in the above sections.

Starting with limitations caused by $\mathcal{EL}$, it is predictable that the restricted number of concept constructors of the language could create inconveniences in modeling of the concept contraction operator. The main problem is the lack of expressivity of $\mathcal{EL}$ for the *union* constructor. This problem forces the authors in [Rienstra et al., 2018] to use the least common subsumer notion, which is weaker than the union. There might be solutions that can be considered as more reasonable, but not expressible in $\mathcal{EL}$. It seems that this is the price to pay for a uniquely determined concept contraction in $\mathcal{EL}$. If this is unacceptable, then it means that a well-behaved uniquely determined concept contraction in $\mathcal{EL}$ is impossible [Rienstra et al., 2018].

Another evident limitation is that the theory cannot be applied in the case of cyclic T-Boxes. In this setting, the statement that a remainder of C with respect to D always exists, unless $D \equiv \top$, does not hold. Furthermore, it is shown in [Baader, 2003] that a least common subsumer may not exist in this setting. Thus, the notion of lcs contraction cannot be straightforwardly applied to cyclic T-Boxes, as in acyclic T-Boxes. According to [Rienstra et al., 2018], a possible solution is to compute approximations of remainders and the least common subsumer, and using this as a basis for an "approximate lcs contraction" operator. Approximating the lcs in the presence of cycles has already been investigated by [Baader, 2003].

2.5 Related work

In this section we will present several works that are related to concept contraction. These works differ from concept contraction as they focus on changes of T-box and A-box and not in the concept level.

An important work concerning *belief contraction* can be found in [Baader et al., 2018]. The authors introduce a framework for repairing Description Logic-based ontologies that is based on weakening axioms rather than deleting them. This approach is different from the classical one where for repairing a Description Logic ontology in the sense of removing an unwanted consequence is to delete a minimal number of axioms from such that the resulting ontology does not have the consequence [Alchourrón et al., 1985]. In [Baader et al., 2018], it is shown how to instantiate this framework for the Description Logic $\mathcal{EL}$ using appropriate weakening relations. In [Qi and Du, 2009], a similar approach is described with the purpose of making contraction/revision a more gentle operation. In this paper, three model-based revision operators to revise terminologies in description logics are proposed. It is shown that one of them is more rational than others by comparing their logical properties and this operator is the focus of the work. The authors consider the problem of computing the result of revision by the operator with the help of the notion of concept forgetting. The computational complexity of the revision operator is analyzed.

In [De Giacomo et al., 2006], the authors study the notion of update of an ontology expressed as a Description Logic knowledge base, constituted by a TBox and an ABox. The focus of the paper is in the case where the update affects only the instance level of the ontology, i.e., the ABox. They provide a general semantics for instance level update in description logics, with a focus on DL-Lite, a family of description logics that underlie the tractable fragment OWL 2 QL [2] of the Web Ontology Language OWL 2. The authors provide an algorithm that computes the result of an update in DL-Lite, and it is shown that it runs in polynomial time with respect to the size of both the original knowledge base and the update formula. In the paper, it is concluded that the result of an update is always expressible as a new DL-Lite A-Box. The authors in [Liu et al., 2006] operate in a similar setting as the previous work. They consider the problem of updating A-Boxes. The assumption that they make is that changes are described at an atomic level, i.e., in terms of possibly negated ABox assertions that involve only atomic concepts and roles. Basic ABox updates are analyzed in several standard description logics by investigating whether the updated ABox can be expressed in these DLs and, if so, whether it is computable and what is its size. They have devised algorithms to compute updated A-Boxes in several expressive DLs and show that an exponential blowup in the

[2] For more information: `https://www.w3.org/TR/owl2-profiles/#OWL_2_QL`

size of the whole input (original ABox + update information) cannot be avoided unless every PTIME problem [3] is LOGTIME [4] parallelizable. This result can be considered similar to the conclusions of the work [De Giacomo et al., 2006], although their work is based only on DL-Lite.

The work of [Qi et al., 2006] focuses on knowledge base revisions. As we have previously mentioned, there is a clear distinction between contraction and revision in the AGM theory [Alchourrón et al., 1985]. Nonetheless, the contraction operation is part of the revision and thus this work is of interest to us. In this paper, the authors first generalize the AGM postulates on revision to description logics. Then, they define two revision operators in DLs, from which one is the weakening-based revision operator which is defined by weakening of statements in a DL knowledge base and the other is its refinement. The results of the work are promising and they show that both operators capture some notions of minimal change and satisfy the generalized AGM postulates for revision. Another work that focuses in this direction is [Zheleznyakov et al., 2019]. Here, the authors investigate knowledge expansion and contraction for knowledge bases expressed in DL-Lite. A novel knowledge evolution framework and natural postulates that evolution should respect are presented, and a comparison of the postulates to the well-established AGM postulates is drawn. The authors propose a formula-based approach that respects their principles and for which evolution is expressible in DL-Lite. In addition, they propose polynomial time deterministic algorithms to compute evolution of DL-Lite knowledge bases when evolution affects only factual data.

2.6 Conclusions

In this chapter, we have introduced the description logics family, with a special focus on the Description Logic $\mathcal{EL}$. This particular language is used in the work of [Rienstra et al., 2018], where an approach to a more gentle AGM contraction, *concept contraction*, is introduced. An overview of the AGM contraction and its postulates are discussed. We define the notion of lcs concept contraction and represent it with a set of postulates. The postulates are reformulations of the original AGM postulates for contraction. From the basic AGM postulates, only the *Recovery* postulate did not apply and had to be replaced with the weaker *Relevance* and *Retainment* postulates. Concept contraction is described in three different settings: (i) an empty T-Box, (ii) an acyclic T-Box, and (iii) a cyclic T-Box. In the first two cases, concept contraction is proved to work, whereas in the third case, lcs concept contraction cannot be applied.

Later, we highlight the limitations of the concept contraction such as lack of expres-

[3] Problems that can be solved in polynomial amount of time.

[4] Problems that can be solved in logarithmic amount of time.

sivity of $\mathcal{EL}$ for the *union* constructor, and the problems that the concept contraction encounters in the case of cyclic T-Boxes.

A summary of related works to the concept contraction is provided in the last section. These works differ in the description logics that the authors have used in their approaches, the assumptions that are made and in which part of the knowledge base they focus on.

Chapter 3

Designed algorithms and their complexity

This chapter presents the main work of the thesis, which consists of the design of the algorithms to implement the concept contraction operator. The focus of the chapter is on five key algorithms: (i) $computeWeakenings$ in section 3.1, (ii) $computeRemainder$ in section 3.2, (iii) $subsumedBy$ in section 3.3, (iv) $computeLCS$ in section3.4, each expressed in pseudocode. These algorithms are the foundations of the fifth algorithm, $conceptContraction$, which is described in section 3.5. For each of the algorithms, we provide a description, a discussion of the best and worst-case scenarios, and a straightforward analysis of the time complexity.

3.1 $computeWeakenings$ Algorithm

This section covers the first algorithm that we have designed, which is the algorithm $computeWeakenings$. We discuss the four basic cases of the weakening procedure for concepts in the Description Logics $\mathcal{EL}$. This discussion is the main theoretical background of the algorithm. The purpose of this algorithm is to compute the weakenings of a concept. It is important to design this algorithm because $computeRemainder$ algorithm, which is introduced in the next section, uses the weakenings of a concept C to find remainders with respect to a concept D.

Firstly, we define the notion of the weakening as follows:

Definition 22 *A weakening of a concept C is a concept C' such that:*

- $C \sqsubseteq C'$
- $\nexists C''$ *s.t.* $C \sqsubseteq C''$, *and* $C'' \sqsubseteq C'$.

Considering the limitations of the expressivity of the description logics $\mathcal{EL}$, there are only four possible ways that a concept can be constructed in this language, namely: (i) atomic, (ii) top, (iii) existential role restriction, and (iv) conjunction. Each of these types of concepts have a different procedure of weakening. To build an algorithm that computes the weakenings of concepts in $\mathcal{EL}$, there are four cases that need to be taken into consideration.

Case 1 *C is an atomic concept. Then:*

$$computeWeakenings(C) = \{\top\}$$

Case 2 *C is a $\top$ concept. Then:*

$$computeWeakenings(\top) = \varnothing$$

Case 3 *C is an existential role restriction of the form $C = \exists r.C_1$, where r is a role. Then:*

$$computeWeakenings(C) = \bigsqcap\{\exists r.W | W \in computeWeakenings(C_1)\}$$

Case 4 *C is a non-trivial conjunction of concepts C_i such that $C_1 \sqcap C_2 \sqcap ... \sqcap C_n$ and every C_i is a trivial conjunction. Then:*

$$\begin{aligned} computeWeakenings(C) = \{W \sqcap ... \sqcap C_n | W \in computeWeakenings(C_1)\} \sqcup \\ \{C_1 \sqcap W \sqcap ... \sqcap C_n | W \in computeWeakenings(C_2)\} \sqcup ... \sqcup \\ \{C_1 \sqcap ... \sqcap C_{n-1} \sqcap W | W \in computeWeakenings(C_n)\} \end{aligned}$$

To compute the weakenings of concepts expressed in $\mathcal{EL}$, we have designed an algorithm $computeWeakenings$ and an embedded function $computeWeak\ Conjunction$. The $computeWeakenings$ algorithm requires as input a concept and produces as an output a set of weakenings of the concept considering the procedure described above.

The algorithm firstly considers the basic cases when the concept is a $\top$ concept or an atomic concept. For these situations, there is no need to compute the weakenings as the output is straightforward. If concept C is an atomic concept, then the algorithm returns a set that includes only the $\top$ concept, as its weakening. If concept C is $\top$ concept, an empty set is returned as there cannot be a weakening of a top concept.

If the parameter, concept C, does not fall into these two types of concepts, the algorithm checks if it is an existential role restriction. If this is the case, a variable $weak\_role$ is assigned with the return value of $computeWeakenings$ algorithm, using as a parameter the concept that forms the existential role restriction, $InnerConcept(C)$ stores as C_1.

For example, if concept C is $\exists R.(A \sqcap B)$, the value of the variable $weak\_role$ is the return value of $computeWeakenings(A \sqcap B)$. The algorithm creates an empty list $role\_weakening$ that is populated with new existential role restriction concepts formed by the concepts in the weak_role list. The variable $weakenings$ is assigned with the return value of the $conjunctionOf()$ function. This function creates a conjunction of each of the elements in the given list. Lastly, the variable $weakenings$ is returned.

In the last clause, when concept C is of a conjunction form, the computation of the weakening is more complex and requires the call of $computeWeakConjunction$ function. When the algorithm enters in this **if** condition, the concept is firstly flattened, and then a variable conjunction_list is assigned the set $Conjuncts(C)$. A **for** loops starts with an index i = 0 and ends when the index reaches the length of $Conjunction\_list$. For each of the concepts in the list $computeWeakConjunction$ is called and the returned value is extended in a variable $weakenings$ of type list. The extend() function adds elements of a list into another list individually. Once the iteration has terminated, the algorithm returns the variable $weakenings$.

The $computeWeakConjunction$ function requires as an input the list $Conjuncts(C)$ and the counter i of the loop in $computeWeakenings$ algorithm. The output is one single weakening of the conjunction concept. When the function is called, an empty list named $weakening$ is created. A **for** loop, starting from j = 0 to the length of the conjunction_list initiates. The loop checks is the index j is equal to index i, given as a parameter. If the indexes point to the same element in the conjunction_list, the $computeWeakenings$ algorithm is called. The return value is added to the weakening list. If the indexes i and j point to different concepts in the conjunction_list, then no computations are needed and the concept that index j refers to is added to the list weakening. When the loop terminates, the conjunctionOf() function is called to form a conjunction of all the elements in the weakening list and its value is stored in $weak$ variable. This variable is the return value of the function.

In figure 2.2 in chapter 2, is shown the lattice of the generalizations of the concept $Person \sqcap \exists\, isRegisteredAt.(University \sqcap Institute)$. This concept is of type conjunction and it is formed by two concepts: $Person$, and $\exists isRegisteredAt.(University \sqcap Institute)$. Based on algorithm 1, there are two ways on how to weaken this concept. The first weakening can be computed by calling the $computeWeakenings$ algorithm with parameter the first concept, $Person$, and leaving the second concept intact. Concept $Person$ is an atomic concept, therefore its weakening is the top concept $\top$. After this value is returned, a new conjunction of the concepts $\top$ and $\exists isRegisteredAt.(University \sqcap Institute)$ is created. Since the intersection of any concept with the top concept is the concept itself, only the concept has been added in the lattice. For

the second weakening of the initial concept, the first concept is left intact and the $computeWeakenings$ algorithm is called on the second concept. Considering that this is an existential role restriction concept, initially the algorithm computes the weakening of the concept that forms it, $University \sqcap Institute$ and then returns the intersection of the weakenings. After the second weakening is returned, $computeWeakenings$ algorithm is called for each of the weakenings, until all the concepts are weakened to the top concept $\top$.

The analysis of the $computeWeakenings$ algorithm consists of the discussion about the best and worst-case scenarios and the complexity of the algorithm in the worst-case scenario. We have chosen to discuss the complexity of the worst-case scenario in order to predict how bad the algorithm can perform in terms of time. For the computation of the time complexity, we have not followed the standard procedures for recursive algorithms. The analysis is done in terms of the size of the input and the size of the output of the algorithm. Further calculations are necessary to get a more accurate time complexity function.

Clearly, the best cases of this algorithm are when concept C is either an atomic concept or a top concept. In both cases, the number of steps to compute the weakenings is very small, and it can be considered constant.

The worst-case scenario is when concept C is an existential role restriction of a conjunction. An essential factor in the complexity of this case is the number of concepts that form the conjunction. Let's assume that the size of the number of conjuncts in the conjunction concept is n. Given that this concept is in an existential role restriction, the input size of the whole concept is $n+1$. Because there is always more than one weakening of a conjunction concept, the algorithm will create a new conjunction of existential role restriction concepts formed by each of the weakenings. For example, if the input is $\exists r.(A \sqcap B)$, the output is $\exists r.A \sqcap \exists r.B$. The size of the set of weakenings of a conjunction concept is equal to the number of concepts that form the conjunction. A key issue is that each of the weakenings is of size bigger than one, precisely $n+1$. Hence, the size of the output is $n*(n+1)$. We can conclude that the complexity of the $computeWeakenings$ algorithm is $O(n^2)$, a quadratic function. Taking into consideration the instruction on calculating the Big O notation in [Cormen et al., 2009], only the highest order term is kept. Although, it is arguable that for small sizes of input, the removed parts of the function can influence the result.

An average-case scenario is challenging to analyse because it is not apparent what constitutes an 'average' input for this algorithm. Although, it is possible that it would lean

towards a worst-case scenario. The analysis of this case will not be discussed for any of the algorithms that we have designed.

Algorithm 1 computeWeakenings(C)

```
if C is Atomic then
    return {⊤}

else if C is Top then
    return ∅

else if C is Existential Role Restriction then
    C_1 ← InnerConcept(C)
    weak_role ← computeWeakenings(C_1)
    role_weakening ← ∅
    for i = 0 to length[weak_role] do
        role_weakening.add(∃r.weak_role[i])
    weakenings ← conjunctionOf(role_weakening)
    return weakenings

else if C is Conjunction then
    C ← Flatten(C)
    conjunction_list ← Conjuncts(C)
    weakenings ← ∅
    for i = 0 to length[conjunction_list] do
        weakenings.extend(computeWeakConjunction(conjunction_list, i)
    return weakenings

function computeWeakConjunction(conjunction_list, counter)
weakening ← ∅
for j = 0 to length[conjunction_list] do
    if j == counter then
        weakening.extend(computeWeakenings(conjunction_list[j]))
    else
        weakening.add(conjunction_list[j])
weak ← conjunctionOf(weakening)
return weak
```

3.2 $computeRemainders$ Algorithm

In this section, we will describe the design and the analysis of the $computeRemainders$ algorithm. Computing a remainder of a concept C with respect to another concept D is one of the main goals of the thesis. We have designed $computeRemainders$ algorithm with the purpose of classifying weakenings of a concept as remainders with respect to the concept D that is being contracted. The remainders are found by going through the elements in the weakening lattice explained in 2.2. This algorithm retrieves all the weakenings of a concept C that are not subsumed by concept D.

The $computeRemainders$ algorithm requires two parameters, concept C and concept D. These concepts can be of any form allowed in description logics $\mathcal{EL}$. An empty list, $remainder$, is created to store all the remainders that are found. Another list, $level$, initially stores concept C. Taking into consideration that unless C is a top concept, a remainder always exists, the algorithm should be able to find at least one remainder.

For this reason, a **while** loop is initiated with the condition that the level list is not empty. Inside this loop, there are three blocks of **for** loops that have different objectives.

The first **for** loop removes concepts from the level list that are weaker than the concepts of the remainder list. This goal is realized by creating a nested **for** loop that iterates over the concepts in the level list. For each pair of concepts between the level and remainder list, the $subsumedBy$ algorithm is called. If a concept in the level list subsumes a concept in the remainder list, the concept is removed from the level list and cannot be considered as a candidate for a remainder.

The second **for** loop removes concepts from the level list in case they are remainders, and adds them to the remainder list. The loop iterates over the concepts of the level list and for each of the concepts, the $subsumedBy$ algorithm is called with parameter the concept and concept D. If the concept in the level list is not subsumed by concept D, the concept is added to the remainder list and removed from the level list. The removal assures that the concept will not be considered as a candidate in the later iterations of **while** loop.

Before the third **for** loop can start the iteration, an empty list new_level is created. The loop's purpose is to create a new list that contains the weakenings of the elements in the existing level list. The loops call the $computeWeakenings$ algorithm for each concept in the level list. The weakenings are added to the new_list. Once the loop has terminated, the level list is assigned with the values of the new_level list. This loop guarantees that the algorithm reaches an end.

The process continues until the **while** loop reaches its condition to terminate the iteration. The output of the algorithm is a list of remainders.

Algorithm 2 computeRemainders(C, D)

```
remainder ← ∅
level ← C

while level is not empty do:

    for i=0 to length[remainder] do
        for j=0 to length[level] do
            if subsumedBy(remainder[i], level[j]) then
                level.remove(level[i])

    for i = 0 to length[level] do
        if not subsumedBy(level[i], D) then
            remainder.add(level[i])
            level.remove(level[i])

    new_level ← ∅
    for i = 0 to length[level] do
        weak ← computeWeakening(level[i])
        new_level.add(weak)
    level ← new_level

return remainder
```

Considering the example in figure 2.2, let's assume that $D \equiv \exists isRegisteredAt.Institute$. The algorithm $computeRemainders$ takes $Person \sqcap \exists isRegisteredAt.\ (University \sqcap Institute)$ and $\exists isRegisteredAt.Institute$ as parameters. Firstly, concept C is checked if it is subsumed by D. This condition is true, so the weakening of concept C is computed by calling $computeWeakenings$ algorithm. With the first iteration of the **while** loop, the remainder list is empty and the weakenings retrieved do not subsume any remainders. In the next **for** loop, both weakenings are subsumed by concept $\exists isRegisteredAt.Institute$. Therefore, the algorithm calls again the $computeWeakenings$ on the existing weakenings, in the third **for** loop. In this new computation, only one of the weakenings is not subsumed by concept D and is classified as a remainder. As for the remaining weakenings, the $computeWeakenings$ algorithm is called on them and the algorithm is now checking weakenings of the fourth level of the lattice. The

algorithm concludes that the weakenings in this level are weaker than the remainder that has already been found. For this reason, they are not categorized as remainders and they are discarded as candidates. The algorithm terminates and the return value is $[Person \sqcap \exists isRegisteredAt.University]$.

The analysis of the algorithm follows on the same logic as in the previous section for the $computeWeakenings$ algorithm. The best-case scenario of the $computeRemainder$ algorithm is if concept C is a top concept. In this case, the algorithm would take a limited number of steps to return the remainder of a top concept with regards to any other concept. The $computeRemainders$ algorithm does not enter the **while** loop as the weakening of a top concept is an empty list. The returned remainder list would also be empty.

The worst-case scenario is the case when concept C is the existential role restriction of a conjunction and concept D is a top concept. In this case the algorithm would have to go through each element of the weakening lattice of concept C because it cannot find a weakening that is not subsumed by D. The complexity of this algorithm is not easy to compute because we cannot determine the size of the weakening lattice. The classification of a weakening as a remainder highly depends primarily from concept D and secondly from the concepts that are already in the remainder list. In this simple analysis, we cannot predict how many times the algorithm iterates on the **while** loop. At this point we can speculate that the complexity of the algorithm is polynomial, considering that there are at most two nested **for** loops inside the **while** loop. In this calculation of the complexity, the complexity of the $subsumedBy$ algorithm should also be taken into consideration.

3.3 $subsumedBy$ Algorithm

A fundamental part of the $computeRemainders$ algorithm is the subsumption check between concepts. We have designed the $subsumedBy$ algorithm taking into consideration the subsumption properties for the Description Logics $\mathcal{EL}$. In this section, we discuss the different cases that need to be taken into account to design this algorithm. We describe the algorithm and provide an analysis of its complexity. The properties of subsumption discussed in this section are adapted from a technical report by [Suchanek et al., 2016]. In the referenced report, it is also included the proof of each of the properties.

Properties 1-4 are the bases of the subsumption properties for $\mathcal{EL}$. In the context of the thesis, a concept is considered to be *basic* if it is an atomic, $\top$, or an existential concept. These four properties are the fundamental components of the $subsumedBy$ algorithm.

The algorithm is recursive, where the base cases are stated in the *Top* and *Atomicity* property. The *Existential Atomicity* can be solved using the *Top* and *Atomicity* property and *Distribution* can be solved using the base cases and *Existential Atomicity* property.

Property 1 *Atomicity: For an atomic concept $C \in N_C$ and a basic concept D, the subsumption $C \sqsubseteq D$ holds iff $C = D$.*

Property 2 *Existential Atomicity: For an existential concept $\exists r.C$ and a basic concept D, the subsumption $\exists r.C \sqsubseteq D$ holds iff $D = \exists r.C'$, with $C \sqsubseteq C'$.*

Property 3 *Distribution: For two conjunctions concepts C and D, the subsumption $C \sqsubseteq D$ holds iff for every conjunct D_j of D there must exist a conjunct C_i of C such that $C_i \sqsubseteq D_j$.*

Property 4 *Top: For a top concept C and a basic concept D, the subsumption $C \sqsubseteq D$ holds iff $C = D$.*

The *Atomicity* property has been changed to include the case when D is a top concept. As an extension of this property, it is assumed that an existential role restriction can not subsume an atomic concept. This assumption leaves the subsumption of an atomic concept only by another atomic concept or a top concept $\top$. The *Existential Atomicity* has been adapted not to consider the sub-case of the universal role mentioned in [Suchanek et al., 2016] and to consider the case when concept D is a top concept. From this property, it is understood that an existential role restriction can be subsumed only by another concept of the same type or $\top$ concept. For this reason, other types of concepts have not been taken into account in the algorithm. Lastly, a new property *Top* has been added to consider the cases when concept C is a top concept.

Once that theoretical base for the design of the algorithm has been set, we can present the $subsumedBy$ algorithm. The algorithm requires two parameters, a concept C and a concept D. The two first cases considered are when D or C is a top concept. If D is a top concept, it does not matter what type of concept C is because the return value of the subsumption algorithm is always `True`. If C is a top concept, then return value is `False`, because no other concept can subsume a top concept, besides itself. This condition was tested in the previous situation.

In the next case, the algorithm deals with the situation when C is an atomic concept. An atomic concept is only subsumed by another atomic concept or the top concept. Given that it has already been checked if concept D is a top concept, the algorithm checks if C is equal to D. The value of this comparison is returned.

If concept C is not an atomic, it is checked if it is an existential role restriction concept.

If this is the case, the algorithm controls if concept D is of the same type. The value of the $subsumedBy$ algorithm with parameters, respectively, $InnerConcept(C)$ and $InnerConcept(D)$, is returned. If D is not an existential role restriction, the value `False` is returned because an existential role restriction can not be subsumed by any other type of concept besides another existential role restriction and the top concept based on the *Existential Atomicity* property.

The last case of the algorithm is when concept C is a conjunction concept. Initially a variable, $list\_c$ that stores the value of the $Conjuncts(C)$ and an empty list $list\_d$ are created. The algorithm checks what type of concept is concept D. In case it is a conjunction concept list_d get the value of $Conjuncts(D)$. Otherwise, the concept is added to the list_d. An empty list, $check$, is initialized to store the returned values of the $subsumedBy$ algorithm call for each of the combinations of the concepts in list_c and list_d. Two **for** loops make sure that each concept in list_d is checked if it subsumes at least a concept in list_c. After the subsumption check has been completed for an element in list_d, the algorithm checks if there are any `True` values in the check list. If this is the case, the check list is emptied, and the first **for** loop can continue to the next concept in list_d. Otherwise, the value `False` is returned. If all the concepts in list_d subsume at least an element in list_c, the algorithm returns `True`.

Lastly, there are no more cases to consider for the subsumption algorithm. If the parameters do not classify in any of the fore-mentioned cases, the algorithm returns `False`.

To better illustrate the behaviour of the $subsumedBy$ algorithm, let's understand its execution when concept C is $\exists isRegisteredAt.(University \sqcap Institute)$ and D is $\exists isRegisteredAt.Institute$. Firstly, the algorithm will try to categorize concept C in one of the types of concepts of $\mathcal{EL}$. Once the C is categorized as an existential role restriction, it is checked if concept D is of the same type. For this example, this is the case and the algorithm calls itself with new parameters, respectively $University \sqcap Institute$ and $Institute$. The process starts again and, concept C is classified as a conjunction concept. The check variable is initiated as an empty list. The **for** loop iterates over the elements in list_d, [$Institute$] and in each iteration the algorithm $subsumedBy$ is called again with parameters the concept in list_d and list_c, [$University$, $Institute$]. This call is fairly easy as both concepts provided in the parameter are atomic concepts. A `False` and a `True` value are added to the check list as $University$ is not subsumed by $Institute$, and $Institute$ is subsumed by $Institute$. The $any()$ function returns `True` confirming that $University \sqcap Institute$ is subsumed by $Institute$. This value is returned by the initial call of the algorithm.

As for the analysis of the algorithm, the best-case scenario of $subsumedBy$ algorithm is when concept D is a top concept. The algorithm does not have to check for concept

Algorithm 3 subsumedBy(C, D)

```
if D is Top then
    return True

else if C is Top then
    return False

else if C is Atomic then
    if D is Atomic then
        return C == D
    else
        return False

else if C is Existential Role Restriction then
    if D is Existential Role Restriction then
        return subsumedBy(InnerConcept(C), InnerConcept(D))
    else
        return False

else if C is Conjunction then
    list_c ← Conjuncts(C)
    list_d ← ∅
    if D is Conjunction then
        list_d ← Conjuncts(D)
    else
        list_d.add(D)
    check ← ∅
    for i = 0 to length[list_d] do
        for j = 0 to length[list_c] do
            if subsumedBy(list_c[j], list_d[i]) then
                check.add(True)
            else
                check.add(False)
        if any(check) then
            check ← ∅
        else
            return False
    return True

else
    return False
```

C, and a `True` value is returned. Concept C being a top concept can also classify for a best-case scenario, as in both cases the number of steps can be considered as constant.

The worst-case scenario is when an existential role restriction of a conjunction is subsumed by another existential role restriction of a conjunction. In the simplest case of this scenario when $C = D = \exists R.(A \sqcap B)$, the algorithm needs to call itself with the parameters $A \sqcap B$ and $A \sqcap B$. Afterward, the $subsumedBy$ algorithm is called for each combination of the concepts in the conjunction list [A, B]. Let's consider the input size to be $n + 1$, where n is the size of the conjunction list. In the worst-case scenario, the algorithm will make $n * n$ calls to itself for each combination of the concepts. The extra steps executed from the algorithm until it reaches the nested **for** loops are excluded from the function calculation. The time complexity of the $subsumedBy$ algorithm is $O(n^2)$, which is also confirmed in [Suchanek et al., 2016]. According to [Baader et al., 1999], the subsumption between $\mathcal{EL}$ concepts can be decided in polynomial time.

3.4 $computeLCS$ Algorithm

In this section, we will describe the fourth algorithm that we have designed, which is the $computeLCS$ algorithm. Akin to the previous sections, we provide a discussion about the best and worst-case scenarios, together with an analysis of the complexity of the $computeLCS$ algorithm. The purpose of the algorithm is to find the least common subsumer of a list of remainders.

The algorithm requires as an input a list of remainders, that can be retrieved by executing the $computeRemainders$ algorithm with parameters the concepts C and D. An empty list lcs is created to store all the candidates for the least common subsumer, while the $level$ list is initially assigned the value of the parameter.

The structure includes a **while** loop that is separated into two sections. The first **for** loop, with the help of a nested **for** loop, checks for each pair of concepts in the parameter R and level list, if there is a concept in the level list that subsumes all concepts in R. If such a concept is found, it is added to the lcs list. The control is realized with the help of a list $check$ that keeps track of the $subsumedBy$ algorithm results. If the list contains all `True` values, the concept is considered an lcs; otherwise, the check list is emptied.

The second section of the **while** loop checks if the lcs list is empty or not. If lcs is not empty, the strongest concept in the list is returned. Two **for** loops feed the $subsumedBy$ algorithm with combinations of concepts in the lcs list. A pair of the same concept is not allowed, as the algorithm would always return true. If there is a concept in the lcs

that subsumes another concept of the list, the weaker concept is removed from the lcs. When the iteration terminates, the lcs list is returned.

In case that the lcs list is empty, a **for** creates a new level list by computing the weakenings of the elements in the lcs. The loop iterates over the concepts in the lcs list and calls the $computeWeakenings$ algorithm for each of the concepts. After the loop terminates, duplicates of concepts are removed from the new_level list and, its elements are assigned to the level list.

The termination condition for the **while** loop is the return of the strongest lcs. This condition is bound to happen, as in the worst-case, it would have to reach the top concept in the weakening lattice.

Let's go back to the example shown in picture 2.2 and assume that concept D is $\exists isRegisteredAt.Institute$. The $computeRemainder$ algorithm returns as a remainder only $Student \sqcap \exists isRegisteredAt.University$. In consequence, the $computeLCS$ algorithm has an input a list that contains only this remainder. The level list is not empty, so the $computeLCS$ algorithm firstly checks if the concept subsumes any concept in the lcs, which is initially empty. This is not true, and it enters in the second **for** loop. Here the $subsumedBy$ algorithm call returns true because the concept is subsumed by itself. The remainder is added to the lcs list and removed from the level list. Since the level list is now empty, the algorithm does not enter in the third **for** loop, and it terminates. The return value of the algorithm is the list $[Student \sqcap \exists isRegisteredAt.University]$.

The best-case scenario of the $computeLCS$ algorithm is when the parameter, R is an empty list. In this case, the level list is empty as well, and the algorithm does not enter the **while** loop. The output is an empty list.

The worst-case scenario happens when the parameter is a list that contains a considerable number of remainders, and the $\top$ concept is the least common subsumer. Initially, the algorithm will have to go through all combinations of concepts in the level list and the parameter list and call the $subsumedBy$ algorithm for each combination. If no least common subsumer is found in the level list, the weakenings of all the concept of the list are computed. This process increases the size of the level list, and therefore increases the number of combinations in the first nested **for** loop. As stated in the analysis of the $computeRemainders$ algorithm, the size of the lattice is not determined, and as a consequence, we cannot determine the size of the set of weakenings of the remainders. At most, we can hypothesize that the time complexity of the algorithm is polynomial, considering that the algorithm deals with nested loops.

Algorithm 4 computeLCS(R)

```
lcs ← ∅
level ← R
check ← ∅

while level is not empty do:

    for i = 0 to length[level] do
        for j = 0 to length[R] do
            if subsumedBy(R[j], level[i]) then
                check.add(True)
            else
                check.add(False)
        if all(check) then
            lcs.add(level[i])
        else
            check ← ∅

    if lcs is not empty then
        for i=0 to length[lcs] do
            for j=0 to length[lcs] do
                if i == j then
                    pass
                else
                    if subsumedBy(lcs[j], level[i]) then
                        lcs.remove(level[i])
        return lcs
    else
        new_level ← ∅
        for i = 0 to length[level] do
            weak ← computeWeakenings(level[i])
            new_level.extend(weak)
        Remove duplicates of concepts in new_level
        level ← new_level
```

3.5 $conceptContraction$ Algorithm

In the last section of this chapter, we present the $conceptContraction$ algorithm. This algorithm can perform the concept contraction operation on concept C with respect to concept D. It is simple in structure and makes use of all the algorithms described in the above sections. We need to emphasize that the selection function mechanism included in this algorithm has not been designed and is considered as given.

The $conceptContraction$ algorithm requires two parameters, concepts C and D. The first step is to find the remainders of concept C with respect to concept D by calling the $computeRemainders$ algorithm. The list of remainders retrieved from this step is stored in the $remainder$ variable, and it is used as a parameter of a selection function algorithm. The remainder list now has assigned only the concepts that are chosen by the selection function. The last step is to find the least common subsumer of the selected remainders, and that can be done by executing $computeLCS$ algorithm. The least common subsumer found, stored in the lcs list, is returned.

Algorithm 5 conceptContraction(C, D)

remainder ← $computeRemainders(C, D)$

remainder ← $selectionFunction(remainder)$

lcs ← $computeLCS(remainder)$

return lcs

To compute the complexity of the algorithm, we have to sum all the complexity functions of the algorithms that are called. Considering that we do not know how the selection function is built, we will consider its complexity as a constant that does not influence the end result. The $computeRemainders$ and the $computeLCS$ algorithms both have the time complexity of a polynomial function. If the two polynomials have the same degree, the degree of the sum is at most this common degree. If the polynomials have different degrees, the degree of the sum is the maximum of the degrees of each polynomial. The degree of the sum defines the time complexity of the $conceptContraction$ algorithm.

3.6 Conclusions

In this last section of the chapter, we finalize the completion of the design of five algorithms: (i) $computeWeakenings$, (ii) $computeRemainders$, (iii) $subsumedBy$, (iv)

$computeLCS$, (v) $conceptContraction$. These algorithms ensure that all the computations needed to implement the concept contraction operator are realized. Respectively, the algorithms complete the following objectives:

- Computation of the weakenings of a concept C
- Computation of the remainders of a concept C with respect to D
- Implementation of the subsumption between concepts in $\mathcal{EL}$
- Computation of the least common subsumer
- Implementation of the concept contraction operator, consisting of all the forementioned algorithms.

Regarding the analysis, for all the algorithms, the best-case scenario is when concept C or concept D is a $\top$ concept. The worst-case scenario for all the algorithms is when concept C is an existential role restriction of a conjunction. As for concept D, the values vary from an existential role restriction of a conjunction to a $\top$ concept. In all cases, the time complexity of the algorithms is polynomial. Considering that concepts of large size are not very common, this kind of complexity can be acceptable. Although, scientific analysis is needed to conclude the time complexity functions of each of the algorithms. This work lacks the mathematical proof of the time complexity of the algorithms.

Chapter 4

A selection function for concept contraction

One of the main assumptions of the concept contraction is that the selection function is considered as extraneously given. In this chapter, firstly, we introduce the notion of the selection function in belief revision. In the second section, we discuss the possibility of associating metadata in the concept level in order to help a selection function choose the best candidates. Lastly, we propose an approach of how to represent concepts and its associative metadata and a theoretical implementation of a selection function using this representation.

4.1 Selection function, an integral part of the concept contraction

Choosing or developing a reasonable selection function is not an easy task. The general idea is to create a selection mechanism that selects the 'best' candidates from a remainder set. The term *best* can be interpreted in different ways from humans and machines, as well. Manually choosing the best candidates, based on logic, is not an option because the information on how to take this decision is missing. Both, we need additional information in order to decide rationally which sentences to give up and which to keep [Gärdenfors et al., 1995].

So far, we have discussed the concept contraction by assuming that the selection is given. The role of a selection function in the concept contraction is to select the best candidates from a list of remainders, as it can be understood from the $conceptContraction$ algorithm in section 3.5. The method of how the selection function chooses these best remainders from a candidate list can be considered as a black-box approach. Such assumption is illustrated in figure 4.1. In this chapter, we uncover this black box and

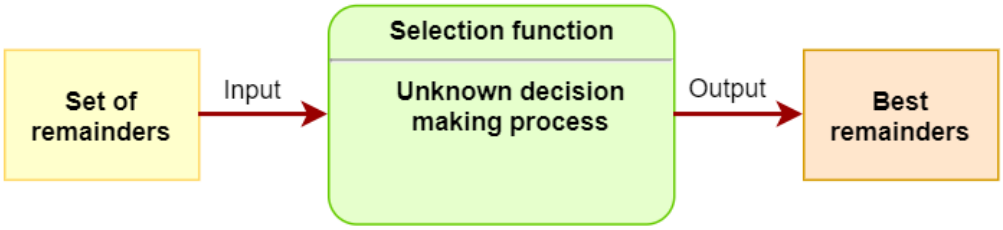

Figure 4.1: A black box approach of the selection function

propose an approach to the implementation of the selection function in the setting of concept contraction.

As it is understood from figure 4.1, the input of the selection function is a set of remainders. This set is retrieved from the execution of the $computeRemainder$ algorithm. The selection function must scan each item of the set of remainders for additional 'features'. We propose to provide the selection function with metadata about concepts. Metadata that provide information about the source, creation date, version can influence the decision-making process of the selection function.

The main problem with this approach is that almost all the metadata standard frameworks that exist are created for ontologies in general and do not give enough information about concepts. A merge and modification of the different standards for ontology description can provide a solution to choosing metadata that are specific to concepts. The question that we raise now is: How to represent the concepts and how to attach this set of metadata to individual concepts?

4.2 Metadata standards for ontology description

In this section, we discuss two metadata standards for the ontology description. We make a comparison of these two standards in terms of completeness, implementation, and usage. Based on the comparison, we have created a new small framework of metadata to help the selection function to choose the best candidates.

Firstly, let us define the notion of metadata. Metadata is information about a given document, such as data, author, and publisher. In [Breitman et al., 2007], metadata is defined as: "Metadata is data about data. The term refers to any data used to aid the

identification, description, and location of networked electronic resources. Many different metadata formats exist, some quite simple in their description, others quite complex and rich." An ontology metadata model provides a method to characterize terminology resources richly [Min et al., 2016]. This method enables the discovery of ontologies, and the possibility to be reused is increased. Having metadata information can as well help in the comparison of ontologies, and choosing the most suitable for a certain domain based on several aspects such as author, domain, design, etc. We aim to create a selection function for the concept contraction operator by using these advantages.

4.2.1 Ontology Metadata Vocabulary

Ontology Metadata Vocabulary (OMV) is a proposed metadata standard reflecting the most relevant properties of ontologies for supporting their reuse. The OMV metadata schema is formally represented as an ontology and includes two separate modules: the OMV Core and various OMV Extensions. The OMV Core contains the most relevant information about the ontology, for the purpose of reuse. The OMV Extensions provide application-specific information for ontology developers and users [(OEG), 2015]. For creating the OMV Core, the authors have used two classes: (i) an ontology base and, (ii) an ontology document. An ontology base represents the abstract or core idea of an ontology. An ontology document represents a specific realization of an ontology base [Palma et al., 2006].

In figure 4.2, an overview of the framework of OMV is presented. This figure can be found in [Palma et al., 2006]. As it can be easily understood, two classes, *Person* and *Organization* have been created as subclasses of the *Party* class. These three classes as a whole provide information on the authors/creators of the ontology. A Party can create, contribute, review an *OntologyDocument* and an *OntologyBase*. From the framework, the distinction between the *OntologyBase* and *OntologyDocument* is clear. In the *OntologyBase*, as mentioned above, is included important information about the ontology such as name, description, documentation, subject, and keywords. On the other hand, in the *OntologyDocument* is included more specific information like: status, creation date, modified date, language, number of classes, properties, individuals, and axioms. Further, the *Party* can develop the language, syntax, type, methodology, license, and engineering tools of the ontology.

The presented OMV tries to model as much information about ontologies and the important aspects for ontology reuse as possible and at the same time, intends to stay as simple as possible [Palma et al., 2006].

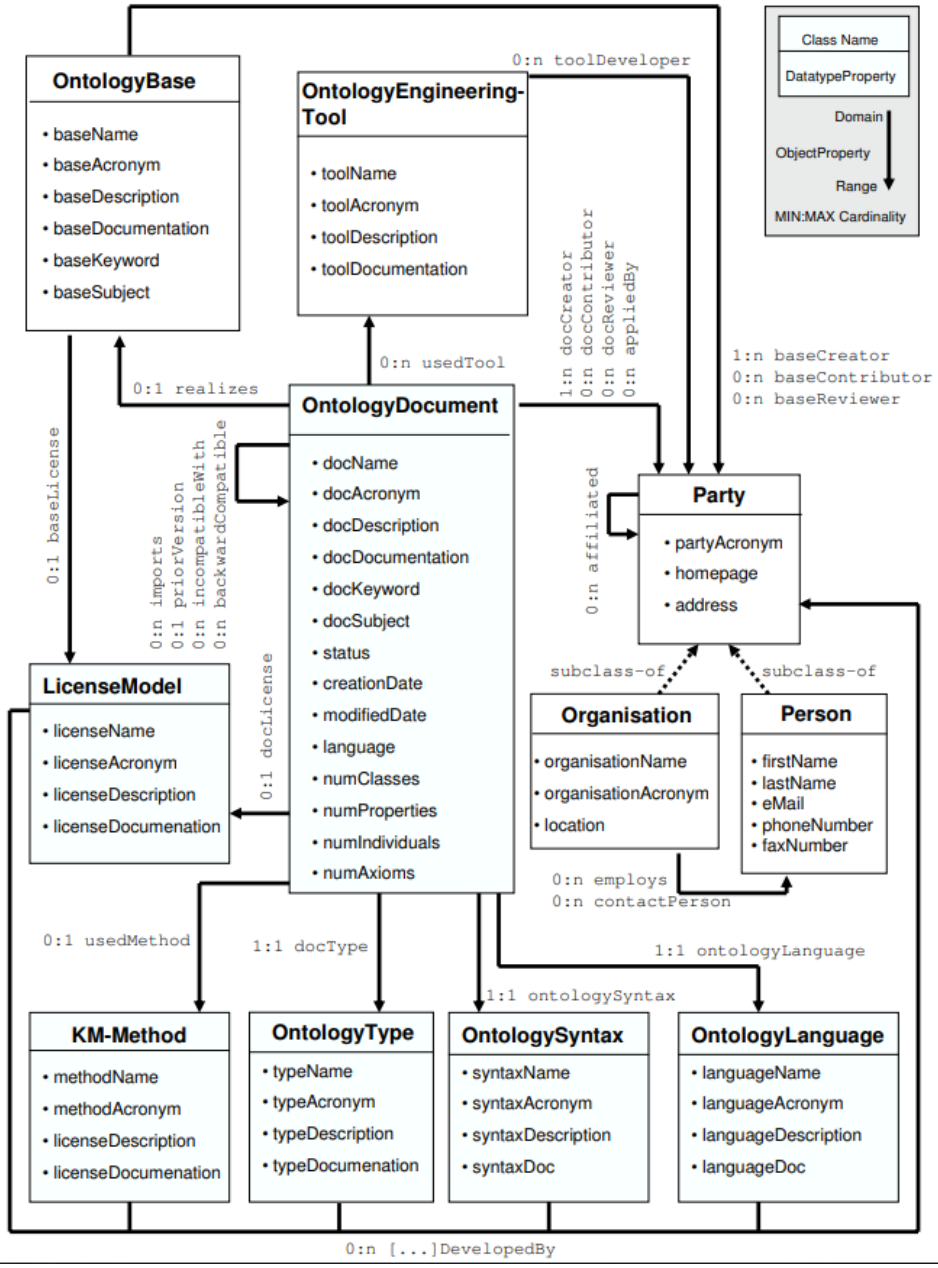

Figure 4.2: OMV Vocabulary Overview

4.2.2 Metadata for Ontology Description and Publication Ontology

Metadata for Ontology Description and Publication Ontology (MOD) is a project that consists of building an OWL ontology and application profile to capture metadata information for ontologies, vocabularies, or semantic resources [Dutta and Jonquet, 2019a]. To make the MOD vocabulary interoperable and conform to the major representation languages currently being used for the Semantic Web applications, MOD is expressed using OWL. The ontology is available at [Dutta and Jonquet, 2019b].

In this vocabulary, the class *Agent* refers to a person or an organization that created and/ or manages an ontology. *Person* and *Organization* are considered as classes and connected to class *Agent* by a subclassOf relationship. An Agent can create, contribute, endorse, and evaluate an ontology. Differently from the OMV ontology, in this vocabulary, is it not explicitly evident who develops the design language, syntax, methodology, license, design tool of the ontology. Nonetheless, this information is available and is connected to the *Ontology* class. This class contains the most important information of the ontology, including name, version, creation date, accessibility, language, description, status, number of classes, properties, individuals, axioms, etc.

Figure 4.3 provides an overview of MOD vocabulary. The MOD terms are standardised by using equivalent terms that are available in the existing metadata standards. Some of the metadata standards that are used are Friend Of A Friend (FOAF), Dublin Core (DC), and Simple Knowledge Organization System (SKOS) [Dutta and Jonquet, 2019b].

4.2.3 Comparison of metadata standards

Ontology Metadata Vocabulary and Metadata for Ontology Description and Publication Ontology are only two of the metadata standards used to describe ontologies. We chose these metadata standards for the ontology description for different reasons. Firstly, these standards include metadata about a wide range of aspects for ontology design. They provide information not only for the authorship of the ontology but also for the implementation choices of the ontology in general. Many of the classes included in these frameworks are not useful for concept description. Nonetheless, the core information can be adapted for the concept level of the ontology.

When comparing the frameworks, it is clear in the first glace that the similarities are multiple. The main class, *Ontology*, for both frameworks defines crucial information about the ontology such as name, label, identifier, language, number of classes, number of properties, number of individuals, number of axioms, language, creation date, version, etc. The same correspondence can be found in *Agent* and *Party* classes, even

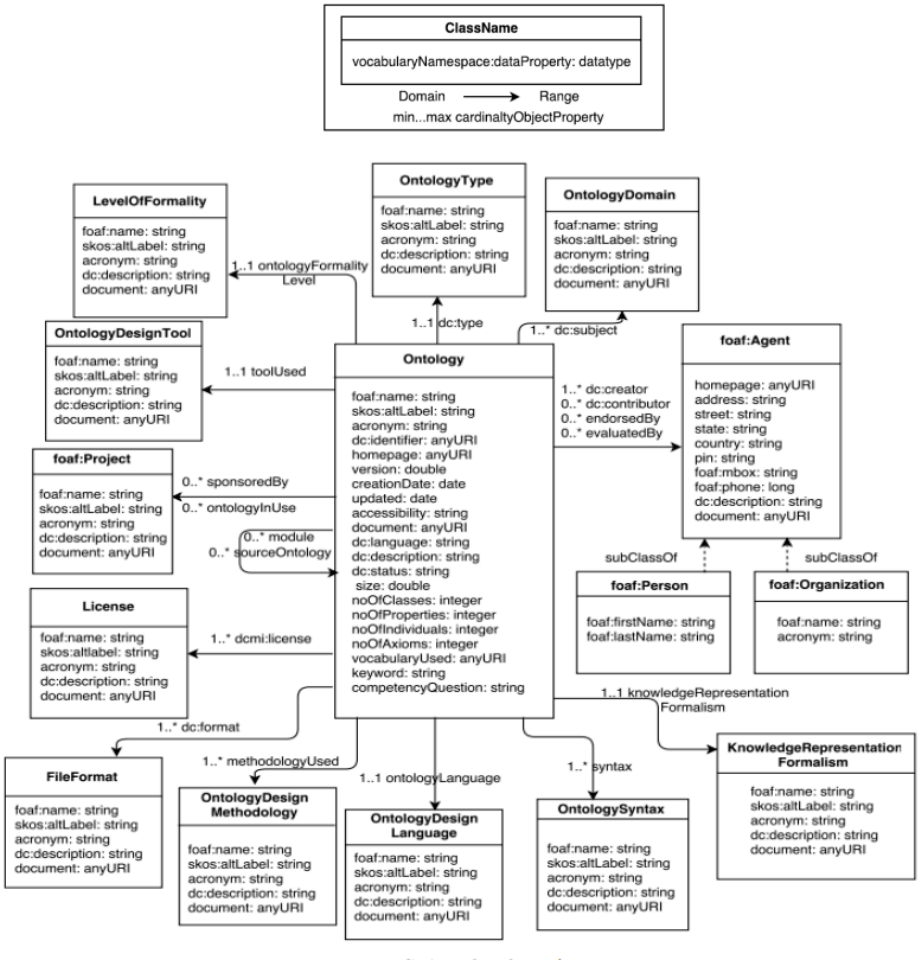

Figure 4.3: MOD Vocabulary Overview

though the organization is different. Classes such as *OntologyType*, *OntologyDomain*, *FormalityLevel/LevelOfFormality*, *License/LicenseMode* and so on, are almost identical in both frameworks, with the only difference that MOD framework has added the *label* metadata. The only class that is not found in MOD framework is *Location*. Nonetheless, the metadata in that class is included in class *Agent*. In table 4.1, we have listed the classes of OMV and by its side are the analogue classes in MOD.

According to [Palma et al., 2006], among the main limitations of OMV are: (i) it does not reuse any other existent relevant metadata vocabularies, (ii) it has not been included in a common ontology editor like Protégé, (iii) the metadata properties were never used by ontology libraries, (iv) after the year 2009, there was no update. On the other hand, the team behind MOD has used existing relevant metadata vocabularies, as it can be proved in figure 4.3. MOD is already implemented as an ontology in Protégé, and in

OMV	MOD
Ontology	Ontology
Party	foaf:Agent
Organization	foaf:Organization
Person	foaf:Person
OntologyType	OntologyType
KnowledgeRepresentationParadigm	KnowledgeRepresentationFormalism
LicenseModel	License
FormalityLevel	LevelOfFormality
OntologyTask	foaf:Project
OntologyDomain	OntologyDomain
OntologyEngineeringMethodology	OntologyDesignMethodology
OntologyEngineeringTool	OntologyDesignTool
OntologySyntax	OntologySyntax
OntologyLanguage	OntologyDesignLanguage
Location	-

Table 4.1: Comparison of OMV and MOD frameworks

their documentation are also included basic queries that can be run for the ontology. Most importantly, the work on Metadata for Ontology Description continues on MOD 1.4, with the extension of new URIs, description of the properties and mappings, documentation, extended number of properties [Dutta and Jonquet, 2019a].

It is worth mentioning that both of these frameworks have been implemented in projects. There are two prototypical applications for decentralized (Oyster [1]) and centralized (ONTHOLOGY [2]) sharing of ontology metadata based on OMV [Palma et al., 2006]. As for MOD, there is a knowledge base consisting of metadata about agronomic ontologies selected from AgroPortal [Jonquet et al., 2018] and defined as an instance of omv: Ontology [Dutta et al., 2015].

4.3 A selection function for concept contraction operator

As discussed in the previous sections of the chapter, metadata can help the selection function to decide which candidate is more accurate and relevant. For this purpose, we

[1] http://timm.ujaen.es/recursos/oyster/

[2] https://kmi.open.ac.uk/events/eswc06/poster-papers/FP50-Hartmann.pdf

chose a small set of core metadata to create a framework that would be sufficient to describe concepts. A theoretical approach to the representation of concepts and their metadata is proposed based on the work of [Schueler et al., 2008]. Furthermore, we propose a reasonable selection function mechanism based on the representation of the fore-mentioned. The goal of this section is to demonstrate that existing theories can be adapted to create a selection mechanism for concept contraction. We have to point out that this proposal is not complete, and it is a mere presentation of the technologies that can be used to achieve the objective of creating a reasonable selection function.

4.3.1 Concept description framework

Based on the description of the OMV and MOD ontologies, it is obvious that both vocabularies are specific to describing an ontology in general, whereas in this thesis, the focus is on concepts. Therefore, only a group of metadata is needed to describe a specific concept. The inclusion of the metadata information for each individual concept is a more complex task compared to adding metadata for the whole ontology. Based on the comparison of the frameworks, most of the metadata is mapped from the MOD framework. Figure 4.4 represents a group of metadata that has been chosen to describe a concept.

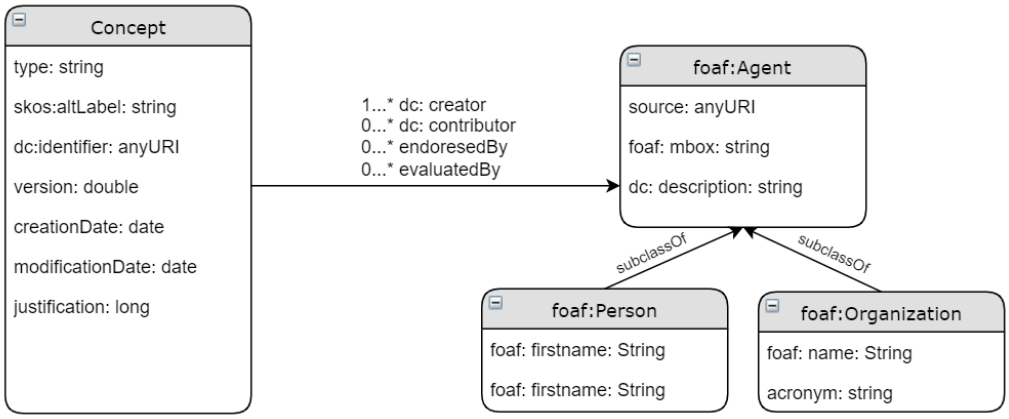

Figure 4.4: Concept metadata framework

In class *Concept* are included important metadata that can help to identify a concept and give information on the version, creation, modification date. The first metadata is used to identify the type of concept, such as atomic, existential role restriction, top, conjunction, etc. The last metadata in this class is justification. Justification can be used to store information about the reason for modification of the concept. The *Concept*

class is connected to the *Agent* class via the following properties: *creator*, *contributor*, *endorsedBy* and *evaluatedBy*. A concept must have at least one creator, while the other properties are not a hard requirement. The *Agent* class can be a person or a organization, information which is shown by the classes *Person* and *Organization*. The entity is identified by first name, last name in case it is a person or name and acronym in case it is an organization. Extra information that is valid for the two kinds of entities are source, email, and description.

The metadata has been selected considering the principles of design of the MOD and OMV frameworks, based on [Dutta et al., 2015] and [Palma et al., 2006]:

- Brevity and expressiveness: The vocabulary should consist of a minimal set of elements maintaining a balance between necessity and sufficiency.
- Clarity: The metadata elements must be well-defined and clear descriptions should be provided.
- Simplicity: The vocabulary should be easy to use.
- Standardization: To confirm the standardization, the individual elements should be mapped with the existing standard vocabularies.
- Interoperability: The vocabulary should be interoperable. It should conform to the major knowledge representation languages currently in use for Semantic Web applications.

4.3.2 Theoretical implementation of the concept description framework

After creating a framework of metadata for concept description, we go back to the main question asked in this chapter. How do we represent the concepts and how to attach the metadata for each concept in an ontology? In this subsection, we present an approach to achieve this goal.

Inspired from the work in [Schueler et al., 2008], we can define a theoretical structure for the implementation of attaching metadata to concepts. In the referenced paper, the authors present an original approach to manage dimensions of meta knowledge. The meta knowledge is modeled in existing RDF structures. The authors define an abstract syntax for RDF$^+$, which is more expressive than RDF, and they have created an extension for SPARQL. The initial results of the implementation are promising. It is understandable that the purpose and the domain of the paper are not the same with the domain

of this thesis, but we can use parts of their approach to reach our goal of attaching metadata to concepts. The following definitions are drawn from [Schueler et al., 2008]. The notations U, L, $G \subseteq U$ and $P \subseteq U$ are respectively Uniform Resource Identifiers(URIs), all RDF literals, the set of graph names and the set of properties.

Definition 23 *The set of all RDF$^+$ literal statements, $\mathcal{K}$, is defined as quintuples by:*

$$\mathcal{K} := \{(g, s, p, o, \theta) | g \in G, s \in U, p \in P, o \in U \cup L, \theta \in \Theta\}$$

A literal interpretation of RDF statements in RDF$^+$ is presented in terms of graph, subject, predicate, object, and an identifier θ. We can adjust this definition in terms of concepts as follows. The notation c defines a singular concept that can be only a URI.

Definition 24 *The set of all concept statements, $\mathcal{K}$, is defined as triads by:*

$$\mathcal{K} := \{(g, c, \theta) | g \in G, c \in U, \theta \in \Theta\}$$

The following definition determines the representation of selected RDF statements as RDF$^+$ meta knowledge. The representation is done using a separate structure of RDF$^+$.

Definition 25 *Let $\Pi \subseteq P$ be the set of meta knowledge properties. Let Ω_π, with $\pi \in \Pi$, be sets providing possible value ranges from the meta knowledge properties $\pi \in \Pi$. Then the set of all RDF$^+$ meta knowledge statements, $\mathcal{M}$, is defined by:*

$$\mathcal{M} := \{(\theta, \pi, \omega) | \theta \in \Theta, \pi \in \Pi, \omega \in \Omega\}$$

Definition 26 *A RDF$^+$ theory of literal statements and associated meta knowledge statements is a pair (K, M) referring to a set of literal statements $K \subseteq \mathcal{K}$ and a set of meta knowledge statements $M \subseteq \mathcal{M}$.*

The definitions of meta knowledge statements and the RDF$^+$ theory can be left intact. The meta knowledge statement uses θ to identify the concept and then provides the meta knowledge available for it. In the paper [Schueler et al., 2008], the authors state three additional definitions, namely: Standard interpretation and Model, Π-Interpretation and Model, and Meta knowledge Interpretation and Model in order to prevent ambiguities caused by definitions 23, 25, and 26, and to produce interpretations for RDF.

Based on the brief description of the theory, the concept $Person \sqcap \exists isRegisteredAt.$ $University \sqcap \exists attends.Lecture$ can be represented in three graphs: G1, G2, and G3. Each graph contains only one concept, as required in the definition. The representation of the metadata that are attached to the concept is graph G4, G5, and G6.

G1	{ Person }
G2	{ ∃isRegisteredAt.University }
G3	{ ∃attends.Lecture }

G4 {	G1 skos:altlabel ”person”. G1 ex:type ”atomic”. G1 ex:creationDate ”10/10/1999”. G1 ex:source <http://uni-koblenz.org/ciroku.owl>. }
G5 {	G2 skos:altlabel ”RegUni”. G2 ex:type ”existentialRoleRestriction”. G2 ex:creationDate ”08/10/2017”. G2 ex:source <http://uni-koblenz.org/ciroku.owl>. }
G6 {	G3 skos:altlabel ”AttLecture”. G3 ex:type ”existentialRoleRestriction”. G3 ex:creationDate ”10/10/2019”. G3 ex:source <http://uni-koblenz.org/ciroku.owl>. }

4.3.3 Theoretical implementation of the selection function

Now that we have defined how a concept and its metadata are represented, we present an approach to how a selection function can make use of the metadata. In order to make the process of the selection function easier, we propose to use all the metadata about a given concept to calculate a $certainty$ degree of the concept. The procedure on how to calculate such a degree is not decidable without further research, but different possibilities can be taken into consideration. For example, a concept that has been created by an agent that is trustworthy can have a higher degree than a concept that is created by an agent who is not. Metadata such as source, creation date, version, justification, can be taken into account to calculate a certainty degree for a concept. The choice of the selection function is much easier when it is fed with concepts and their respective degrees than with their group of metadata. Therefore, we define the notion of

a ***certainty*** mapping. The ***certainty*** mapping represents a concept that can contain one or more conjuncts with the respective degrees of certainty.

Definition 27 *A **certainty** mapping for a concept C is a mapping (C_1:V_1, ..., C_n:V_n) where for each i, $C \sqsubseteq C_i$ and V_i is a non-negative integer named "certainty degree".*

The **certainty** mapping of concept $Student \sqcap \exists isRegisteredAt.University \sqcap \exists attends.Lecture$ is shown in example 1. The scores associated with each of the concepts are arbitrary, considering the type of concepts and creation date.

Example 1. Certainty mapping of a concept

$$\begin{pmatrix} Student: & 99 \\ \exists isRegisteredAt.University: & 56 \\ \exists attends.Lecture: & 48 \end{pmatrix}$$

With a complete mapping of the concept, we can now define a **relevance** degree for concept C. The relevance degree is the sum of all the certainty degrees of all concepts that form concept C.

Definition 28 *The **relevance** degree of a concept C w.r.t the **certainty** mapping (C_1:V_1, ..., C_n:V_n) is:*

$$\sum_{i=1}^{n} \{V_i | C \sqsubseteq C_i\}$$

This relevance degree can be provided to the selection function together with the remainder candidate concepts. This information about the concepts, based on its metadata, can provide us with a reasonable selection function mechanism. Thus, we extend the properties of the selection function, as expressed below. This property enables the selection function with a working mechanism specific to the concept contraction.

Property 5 *A selection function σ selects, given every pair of concepts C, D, the concept with the maximum **relevance** degree.*

In figure 4.5, we illustrate the transparent mechanism of the selection function based on the approach that we propose. In this case, the input of the selection function is a set of remainders and also a set of associative relevance degrees. The decision of the selection is based on the relevance degree of the concepts, and it chooses the remainders with the maximum degree. The output is a set of best remainders, which is the objective of the selection function.

The approach described in this section can be further developed and implemented in

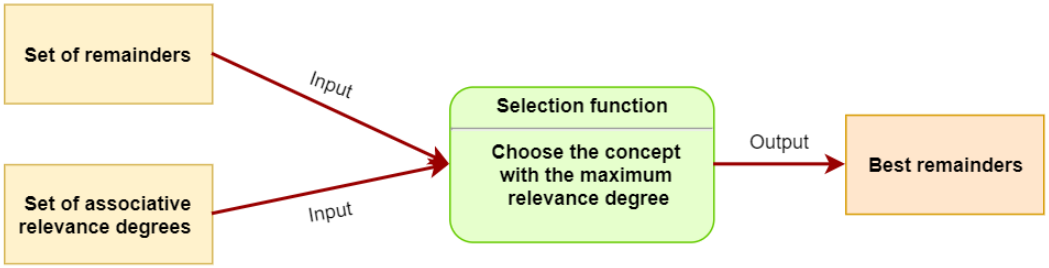

Figure 4.5: A glass-box approach of the selection function

future work. The important issue that we need to emphasise is that there is an approach to model concepts and its associative metadata, and to use this metadata to define a selection function. Considering that the development of the approach is not the main focus of the thesis, the discussion will not go into further detail. More thorough research on this topic can conclude on an optimized approach, and the role of the metadata in the selection function process could be determined.

4.4 Conclusions

In this chapter, we have answered the research question of how to create a reasonable selection function. As discussed in chapter 2, the input of the selection function for concept contraction is a set of remainders. We have proposed an approach that includes several elements into creating a selection function, such as:

- Creation of a framework of metadata specific for a concept description. The set of metadata that we have chosen is based on the merge and adaption of two different ontology description frameworks.
- An approach for the representation of the concepts and their respective metadata based on [Schueler et al., 2008]
- An approach for the computation of a certainty degree of a concept based on its metadata.

Certainly, this proposed approach needs to be studied thoroughly in order to achieve a working selection function. The connection between different technologies is not clear enough, and most importantly, the computation of the certainty degree for each concept has not been investigated.

Chapter 5

Implementation of $conceptContraction$ algorithm

In this chapter, we present the implementation of the $computeWeakenings$, $computeRemainders$, $subsumedBy$, $computeLCS$ and $conceptContraction$ algorithms. In the first section, we introduce the environment of the implementation, including a short description of the programming language and specific libraries. In the second section, we describe the implementation of the concepts as classes and their methods. In section 5.3, we present the implementation of the $computeRemainder$, $subsumedBy$, $computeLCS$ algorithms. Each of the classes, their methods and functions are tested, and the results can be found in section 5.4.

5.1 Environment of implementation

In this section, we will describe *Python*, the language chosen for the implementation of the algorithms that we have designed. In the description, we include some features of the language that support our choice of implementation language. In addition, we introduce a package, *Owlready2*, supported by Python, which is specific to the creation of ontologies and work with concepts and roles.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It provides high-level built-in data structures, combined with dynamic typing and dynamic binding [pyt, 2019]. As a programming language, Python delivers both the power and complexity of traditional compiled languages along with the ease-of-use of simpler scripting and interpreted languages [Swamy, 2014]. Python has many features that make it desirable to work with. The features listed below are the reason why this language has been chosen for the implementation of the $conceptContraction$ algorithm.

- **Easy-to-use**: This feature includes the 'Easy-to-learn' and 'Easy-to-read' described in [Swamy, 2014]. The simple syntax of the language, variable's behaviour compared to other programming languages makes Python easy to understand when one reads the code and also easy to use. Moreover, the high level of abstraction of data structures reduces the framework development time [Swamy, 2014].

- **Expressiveness**: For most programming tasks, Python requires fewer lines of code than most programming languages. The features make the code easier to maintain and debug [Ceder, 2010].

- **Completeness**: When Python is installed, there is no need to install additional libraries. The Python standard library comes with modules for handling email, web pages, databases, operating system calls, GUI development, and more [Ceder, 2010].

- **Cross-platform**: Python is available on a wide variety of platforms, which contributes to its surprisingly rapid growth in today's computing domain.

Owlready2 is a package for ontology-oriented programming in Python. Owlready2 allows transparent access to OWL ontologies. Owlready2 has been created at the LIMICS research lab, by Jean-Baptiste Lamy. According to [Lamy, 2017], Owlready2 can:

- Import ontologies in RDF/XML, OWL/XML, or NTriples format.

- Manipulate ontology classes, instances, and annotations as if they were Python objects.

- Add Python methods to ontology classes.

- Re-classify instances automatically, using the HermiT reasoner [1].

In order to better understand the purpose of Owlready2 in the implementation of the $conceptContraction$ algorithm, the conversion of formulas between Description Logics and Owlready is of interest. In table 5.1, the analogy between Description Logics and Owlready2 syntax is presented. The table has been adapted from [Lamy, 2018], by selecting only the concepts and operations allowed in Description Logics $\mathcal{EL}$.

No additional external libraries have been used for the implementation of the algorithms. We make use of the power of Python and Owlready2. Further information about the library can be found in [Lamy, 2018].

[1] HermiT is reasoner for ontologies written using the Web Ontology Language (OWL). Given an OWL file, HermiT can determine whether or not the ontology is consistent, identify subsumption relationships between classes, and much more[her, 2019].

	DL Syntax	Python + Owlready2
Top	$\top$	`Thing`
Subsumption	$A \sqsubseteq B$	`issubclass(A, B)`
Equivalence	$A \equiv B$	`A.equivalent_to.append(B)`
Instanciation	$A(i)$	`isinstance(i, A)`
Relations	$R(i, j)$	`i.R = j`
Intersection	$A \sqcap B$	`And([A, B])`
Existential role restriction	$\exists R.B$	`R.some(B)`

Table 5.1: Description logics and formal ontology notation

5.2 Building an ontology for $\mathcal{EL}$ concepts

This section covers the implementation of concepts of the Description Logics $\mathcal{EL}$. For all concepts allowed in the language, we describe the class that we have created and the methods included. For each of the classes, the methods initialize an instance of the class, create a representation of the concept and compute the weakenings of the said concept. The methods of weakening are based on the procedure discussed in section 3.1, $computeWakenings$ Algorithm.

To implement the concept contraction algorithm, firstly, we have designed an ontology named **onto**. The URI of the ontology, `http://uni-koblenz.org/ciroku.owl`, is given as a parameter. In this ontology, six classes are included, from which five are concept classes, and one is a property class. A new class can be created in an ontology by inheriting the owlready2.Thing class. The ontology class attribute can be used to associate the class to the given ontology. The **Concept** class has the role of the superclass in this ontology. Based on the $\mathcal{EL}$ language expressivity, a concept can be of the form: (i) *atomic*, (ii) *top concept*, (iii) *existential role restriction*, or (iv) *conjunction*.

Class **Atomic**, a subclass of class Concept, includes atomic concepts. The **Top** class has only the $\top$ concept. Class **Role_Restriction** comprises concepts of the form of existential role restriction. Class **Conjunction** is built to include conjunction concepts. The concepts that form the existential role restriction or the intersection form can be of type Concept. Lastly, the property class **role** is created for the purpose of initiating an instance of the Role_Restriction class. In figure 5.1, we present the hierarchy of the ontology.

The superclass **Concept** is created to be a subclass to the $Thing$ class provided from the $owlready2$ package. We have created an $\_\_init\_\_()$ method for Concept class. This method not only initializes an instance of the class, but whenever an instance is created,

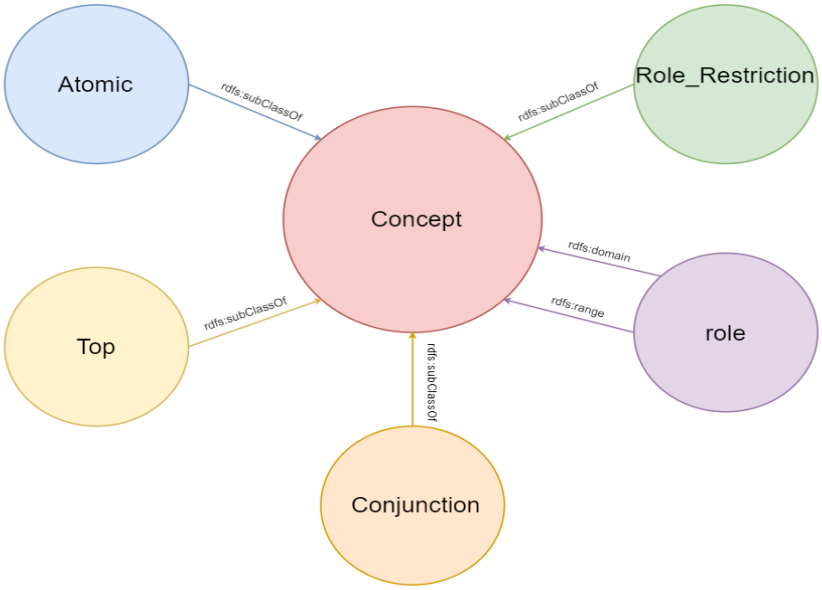

Figure 5.1: Ontology of $\mathcal{EL}$ concepts

the latter is stored in a variable **instance** of type list. This method is called each time that an instance of the subclasses is generated. This action is necessary to ensure that the instance of the subclass is also of type Concept.

To initialize an instance of class **Atomic**, we use the $\_\_init\_\_()$ method. This method requires a String as a parameter. Upon initialization, the method calls another method, $create\_concept()$, that creates the representation of an atomic concept with the String parameter as the name of the concept. In case that one needs to print an atomic concept, this method can be used, as it returns the name of the concept. The $weak\_atomic()$ method, that returns the weakening of an atomic concept, is a static method, and therefore it can be called without an object for that class. Taking into account that the weakening of an atomic concept is the $\top$ concept, the function creates a class Top instance and returns the instance.

In example 1, a new instance of class Atomic is created, given the string 'A' as an argument. In line 4, it is printed the return value of $weak\_atomic()$ method. As shown, the weakening of an atomic concept is, in fact, the $\top$ concept.

Example 1. Execution of methods of Atomic class

```
1 # Create instance of Atomic class
```

```
A = Atomic('A')
# Print the weakening of atomic concept A
print('The weakening of atomic concept', A.create_atomic(), 'is',
    A.weak_atomic())

>>>The weakening of atomic concept owl.A is [owl.Thing]
```

The **Top** class also includes three methods to deal with its instances. The $\_\_init\_\_()$ method does not require any parameters to initialize. Once an instance has been created, $create\_top()$ method is called. This method adds the value of $owlready2.Thing$ to the instance and returns the instance. The method that returns the weakening of a $\top$ concept is $weak\_top()$. Similar to $weak\_atomic()$, this is also a static method. The weakening of the $\top$ concept does not exist, an empty list is returned.

In example 2, the *conceptT* variable is initialized as an instance of the Top class. In line 4, the weakening of the Top concept is printed, using the method described in the above paragraph. The results of the calling of the methods assure that the methods return the correct value of the weakening of a Top class instance.

Example 2. Execution of methods of Top class

```
# Create instance of Top class
conceptT = Top()
# Print the weakening of Top concept conceptT
print('The weakening of concept',conceptT.create_top(), 'is',
    conceptT.weak_top())

>>>The weakening of concept owl.Thing is []
```

The **Role_Restriction** class, also, has three methods. Akin to the previous classes, the methods are $\_\_init\_\_()$, $create\_role\_concept()$, and $weak\_role()$. The $\_\_init\_\_()$ method requires as a parameter an instance of Concept class to initialize a new instance. The given parameter is stored, and then the $create\_role\_concept()$ method is called. To create an instance of a class that includes the existential role restriction, the $role$ property class comes in help. This class has been created as shown in listing 5.2. The property is used to build an instance of Role_Restriction class in $create\_role\_concept()$ method. The method does not require any parameters, although it uses the argument given in $\_\_init\_\_()$ method and adds the existential role restriction to that argument. The return value is a concept in a role restriction form. Lastly, the $weak\_role()$ method

is used to compute the weakening. This method does not require extra parameters to compute the weakening of an existential role restriction concept. $weak\_role()$ calls function $weak\_concept()$, described in listing **??**, with the initialization parameter as its parameter. After the return value of $weak\_concept()$ method is retrieved, it is checked if there are one or more weakenings. In the case that there is only one weakening, the method creates a new Role_Restriction instance with the weakening as a parameter. Otherwise, multiple new instances of Role_Restriction class are created with parameters the weakenings. Afterwards, a new Conjunction instance is created using all the newly created Role_Restriction instances. The final result is returned.

The listing 5.2 presents the creation of the property class **role**. A new property can be created by subclassing the ObjectProperty or DataProperty class from the owlready2 package. This property is functional, meaning that it has a single value for a given instance. Domain and range must be given in the list, since OWL allows to specify several domains or ranges for a given property. The domain and the range of the role property is Concept class. This allows creating the existential role restrictions of an atomic, top, existential role restriction or conjunction concept.

```
1 #Create a property named 'role' with class Concept as domain and
      range
2 with onto:
3      class role(owlready2.ObjectProperty, owlready2.
      FunctionalProperty):
4           owlready2.domain = [Concept]
5           owlready2.range = [Concept]
```

Listing 5.1: Creation of role property

The creation of a new instance of Role_Restriction class is shown in example 3. As an argument for the initialization, an atomic concept A from example 1 has been used. Line 4 prints the weakening of the new instance, after the methods $create\_role\_concept()$ and $weak\_role()$ have been called. The results are correct as the weakening of a concept $\exists r.A$ is, in fact, $\exists r.\top$.

Example 3. Execution of Role_Restriction class methods

```
1 # Create instances of Role_Restriction class
2 conceptR = Role_Restriction(A)
3 # Print the weakening of conceptR
4 print('The weakening of concept', conceptR.create_role_concept(),'
      is', conceptR.weak_role())
```

```
5
6 >>>The weakening of concept role.some(owl.A) is role.some(owl.
    Thing)
```

Class Conjunction, includes two methods $\_\_init\_\_()$ and $create\_conjunction()$, to initialize and create a new instance, and two classes $weak\_conjunction()$ and $weak\_conjunct()$ to compute the weakening of a conjunction of concepts. Contrasting the fore-mentioned classes, the $\_\_init\_\_()$ method of this class required a list of concepts as an argument to initialize an object. The list is stored and then $create\_conjunction()$ method is called to create a proper form of conjunction given the parameter. The $create\_conjunction()$, although it does not require other arguments, uses the list to create a conjunction form of the concepts included. The conjunction is the return value of the method.

Given the complexity of the weakening of conjunction compared to the other weakening methods that were described above, two methods are needed to reach such a goal. The $weak\_conjunction()$ method does not require any arguments and can be called to return a list of weakenings. Inside the method, a variable $weakmatrix$ of type list is created to store the weakenings of the conjunction. $weak\_conjunction()$ makes sure that $weak\_conjunct()$ method is executed as many times that there are concepts in the list. The return values are stored in the variable $weakmatrix$. The $weak\_conjunct$ method requires as parameters only the index of the loop in $weak\_conjunction()$. A new variable $weak\_row$ of type list is initiated. It calls $weak\_concept()$ function as many times that there are concepts in the list. If the return value of $weak\_concept()$ is an instance of Top class, e.g.$\top$ or instance of Role_Restriction class with inside concept of Top class, e.g. $\exists r.\top$, the value is ignored. If the above situations are not the case, then the value is added to the $weak\_row$ list. If the list has only one element, the element is returned as is, otherwise a new Conjunction instance is created using the $weak\_row$ list as a parameter. The method returns a single weakening of the conjunction. After all the single weakenings are computed by $weak\_conjunct()$ method, the $weak\_conjunction$ method adds all the return values in $weakmatrix$ list and returns the latter.

The mentioned methods of Conjunction class are illustrated in example 4. Variable $conceptA\_B$ represents an instance of Conjunction class, that is created by two atomic concepts such as A and B. The print statement in line 4 calls the methods $create\_conjunction()$ and $weak\_conjunction$ for the instance. As observed in line 6, the return values of both methods are correct regarding $conceptA\_B$.

Example 4. Execution of Conjunction class methods

```
# Create instances of Conjunction class
conceptA_B = Conjunction([A, B])
# Print the weakening of conceptA_B
print('The weakening of conjunction', conceptA_B.
    create_conjunction(), 'is', conceptA_B.weak_conjunction())

>>>The weakening of conjunction owl.A & owl.B is [owl.B, owl.A]
```

Additionally to these methods, a function $weak\_concept()$ has been created, as mentioned in the description of the methods of Role_Restriction class and Conjunction class responsible for the weakening of the concepts. Both these classes are formed by other concepts, that may be of any form allowed in $\mathcal{EL}$ Description Logics. If not specified, the program needs to know which weakening method to perform. Taking into consideration that the parameters needed to create objects of these classes are concepts, such a function is necessary. The function checks if the instance falls into one of the categories of the classes and returns the weakening of the concept.

5.3 Implementation of the remainders and LCSs

At this point of the implementation, we have created classes for each type of concept of $\mathcal{EL}$ Description Logics and methods to initialize, represent and compute the weakenings of the concepts. Before we implement the function for $computeRemainders$ algorithm, another function of great relevance is needed. The function is $subsumed\_by()$, based on the $subsumed\_by$ algorithm. In this section, we primarily describe the implementation of this algorithm and then the implementation of the $computeRemainders$ and $computeLCS$ algorithms.

5.3.1 Implementation of $subsumedBy$ algorithm

The $subsumedBy()$ function aims to test an important condition for a weakening to be officially chosen as a remainder. The function is the implementation of the $subsumedBy$ algorithm which is based on the subsumption properties for $\mathcal{EL}$. Based on the algorithm, there are several cases of subsumption that need to be implemented.

The function requires two parameters, namely $concept\_c$ and $concept\_d$, both instances of Concept Class. The function deals initially with the case when concept_c or concept_d is a $\top$ concept. As discussed for the $subsumedBy$ algorithm, this case is straightforward and it can be implemented as shown in listing 5.3.1.

```
1 if isinstance(concept_d, Top):
2     return True
3 elif isinstance(concept_c, Top):
4     return False
```

Listing 5.2: subsumedBy(Top, Concept), subsumedBy(concept, Top)

The next case is when concept_c is an atomic concept. Based on the properties of subsumption discussed in section 3.3, an atomic concept can only be subsumed by ⊤ concept or another atomic concept. Listing 5.3.1 shows the implementation of this case.

```
1 if isinstance(concept_c, Atomic):
2     if isinstance(concept_d, Atomic):
3        return concept_c == concept_d
4     else:
5         return False
```

Listing 5.3: subsumedBy(Atomic, Concept)

In $\mathcal{EL}$ Description Logics, a concept of the form of existential role restriction can be subsumed only by a concept of the same form or a top concept. This makes the implementation of this case fairly easy, as there is only one condition to be fulfilled for a Role_Restriction instance to be subsumed by a Concept instance. In listing 5.3.1, is checked whether both concept_c and concept_d are instances of Role_Restriction class. If it is true, then the $subsumedBy()$ function is called with new arguments. $my\_role\_concept$, used in line 2, is an attribute of a Role_Restriction instance, which stores the concept that forms the instance. This is the equivalent of the $InnerConcept$ used in the $subsumedBy$ algorithm.

```
1 if isinstance(concept_c, Role_Restriction) and isinstance(
    concept_d, Role_Restriction):
2     return subsumedBy(concept_c.my_role_concept, concept_d.
    my_role_concept)
```

Listing 5.4: subsumedBy(Role_Restriction, Concept)

The last case to be implemented for the $subsumedBy()$ function is when $concept\_c$ is an instance of Conjunction class. For the implementation of this case, we have to

mention that we consider conjunction concepts only non-trivial conjunction. Meaning a conjunction concept is a concept that is formed by the intersection of at least two concepts. This assumption falls into contradiction with the $Distribution$ property for the subsumption in $\mathcal{EL}$. For this reason, the implementation is divided in two sub-cases.

The first sub-case is when $concept\_d$ is a Conjunction instance. Initially, the lists that contain the concepts that form both conjunctions are retrieved. This action is reached by using $conjuncts$ attribute of the Conjunction class that stores the parameter used to create an instance. The $conjuncts$ attribute has the same function as $Conjuncts$ set mentioned in 2.2. The the implementation is very similar to the design of the $subsumedBy$ algorithm for this case. The code is included in listing 5.3.1.

```
if isinstance(concept_c, Conjunction):
list_c = concept_c.conjuncts
    if isinstance(concept_d, Conjunction):
        list_d = concept\_d.conjuncts
        mat = []
        for i in range(len(list_c)):
            for j in range(len(list_d)):
                if subsumedBy(list_c[i], list_d[j]):
                    mat.append(True)
                else:
                    mat.append(False)

            if any(mat):
                pass
            else:
                return False
        return True
```

Listing 5.5: subsumedBy(Conjunction, Conjunction)

The second sub-case is $concept\_d$ being an Atomic or a Role_Restriction instance. In this case, any of the concepts that form $concept\_c$ have to be subsumed by $concept\_d$. The implementation includes only one **for** loop that iterates over the concepts of the conjunction. This only difference in implementation between these to sub-cases is that in this case there only one **for** loop that iterates over $concept\_c$. Again, this implementation is a clear translation of the pseudocode in Python. This sub-case is illustrated in listing 5.3.1.

```
if isinstance(concept\_c, Conjunction):
```

```
list\c = concept_c.conjuncts
    if isinstance(concept\_d, Atomic) or isinstance(concept\_d,
    Role_Restriction):
        value_check = []
        for i in range(len(list_c)):
            if subsumedBy(list_c[i], concept_d):
                value_check.append(True)
            else:
                value_check.append(False)

        if any(value_check):
            pass
        else:
            return False
        return True
```

Listing 5.6: subsumedBy(Conjunction, Atomic/ Role_Restriction)

If as parameters of the function is used any other combination of types of concepts not mentioned above, the function returns `False`.

In the following example, the function is called with different parameters. X and T are respectively an atomic concept X and a $\top$ concept. $roleX$ and $roleY$ are existential role restrictions formed with atomic concepts X and Y. Lastly, XYZ is a conjunction of atomic concepts X, Y and Z and XY is also a conjunction of the atomic concepts X and Y. The results, depending on the various combinations of the parameters, prove that the behaviour of the function is correct. The call in line 1 returns True because an atomic concept X is subsumed by $\top$ concept. The reverse combination of these parameters returns False, as a $\top$ concept cannot be subsumed by an atomic concept. The result for the call in line 3 is also False because the concept that forms $roleX$ is not subsumed by the concept that forms $roleY$. The last two calls deal with the case when the first parameter is a conjunction. The $subsumedBy()$ call in line 4 returns True because all the concepts in XY subsume a concept in XYZ. The last call returns True, as there is a concept in the conjunction that is subsumed by the atomic concept X.

Example 1. Execution of the $subsumedBy()$ function

```
print(subsumedBy(X, T))
print(subsumedBy(T, X))
print(subsumedBy(roleX, roleY))
print(subsumedBy(XYZ, XY))
print(subsumedBy(XY, X))
```

```
6
7 >>> True
8 >>> False
9 >>> False
10 >>> True
11 >>> True
```

5.3.2 Implementation of $computeRemainders$ algorithm

Subsequently, the implementation of the $computeRemainders$ algorithm is in order. The algorithm's goal is to select remainders from the list of weakenings of a concept with regards to the concept that is to be contracted and return the remainders. In this subsection we describe the implementation of the algorithm with the help of snippets of code and an example of execution of the implementation.

The function requires two arguments, a concept C, and a concept D. An empty list, $remainder$, is created to store the weakenings that classify as remainders. Another list, $level$, is initialized with the weakenings of C. This list is designed to store candidates for the $remainder$ list. A **while** loop begins, with the condition to run as long as there are elements in $level$ list. Inside the while loop, three loops need to run in order to find remainders.

The goal of the first **for** loop is to check if there is a concept in the level list that is stronger then concepts in remainder list. The weak candidates are removed from the level list.

```
1 new_level = []
2 for i in range(len(remainder)):
3     mark = []
4     for j in range(len(level)):
5         if subsumedBy(remainder_found[i], level[j]):
6             mark.append(True)
7         else:
8             mark.append(False)
9         if not any(mark):
10            new_level.append(level[i])
11 level = new_level
```

Listing 5.7: Removing weak candidates

The second **for** loop classifies candidates in the level list as remainders and stores them in the remainder list.

```
new_level = []
for i in range(len(level)):
    if subsumedBy(level[i], concept_d):
        new_level.append(level[i])
    else:
        remainder_found.append(level[i])
level = new_level
```

Listing 5.8: Find remainders

The last **for** loop, replaces the candidates in the level list with their weakenings. So far, the elements in the level list are weakenings that are subsumed by concept_d. If the list is empty, the process has reached an end, and the remainder list is returned. Otherwise, a new **while** loop begins.

```
new_level = []
for i in range(len(level)):
    new_level.append(weak_concept(level[i]))
level = new_level
```

Listing 5.9: Create next level of weakenings

In example 2, the $computeRemainders()$ function is called with parameters XYZ and XY, both conjunction concepts. As seen in line 3, the result of the call is a list of remainders. The remainders are the weakenings of XYZ, not subsumed by XY, specifically conjunction concepts $Y\&Z$ and $X\&Z$.

Example 2. Execution of the $computeRemainders()$ function

```
print(computeRemainders(XYZ, XY))

>>> [Y & Z, X & Z]
```

5.3.3 Implementation of $computeLCS$ algorithm

The last part of the implementation of the $conceptContraction$ algorithm is computing the least common subsumer. The function $computeLCS()$ is designed for this purpose and is the implementation of the $computeLCS$ algorithm.

As described in the algorithm, the parameter of the function is a remainders list that is retrieved from the $computeRemainders$ algorithm. The algorithm includes a **while** loop that is divided in two sections, two nested **for** loops and an **if/else** clause. The first section controls if there is a concept in the level list that subsumes all the concepts in the parameter list. If such concept is found, it is added to the lcs list. The second section, firstly checks if there are concepts in the lcs list. If this is the case, the strongest of the concepts is returned. Otherwise the **else** clause computes the weakenings of the concepts in the level list and the iteration in the **while** loop continues.

The example 3, illustrates the execution of the $computeLCS()$ function. The parameter used is the return list of $computeRemainders()$ function, $[Y\&Z, X\&Z]$. From the result in line 4, the lcs is the atomic concept Z.

Example 3. Execution of the $compute\_LCS()$ function

```
1 rem = compute_remainder(XYZ, XY)
2 print(compute_LCS(rem))
3
4 >>> [Z]
```

5.3.4 Implementation of the $conceptContraction$ algorithm

The final stage of the implementation is the creation of a function for the $concept\ Contraction$ algorithm. Given that we have already implemented all the components required for the algorithm to work, the function is a wrap up of these functions. A selection function has not been implemented, so it will not be included even though it is part of the $conceptContraction$ algorithm.

The function requires a concept C and a concept D as parameters. The first call is to the $computeRemainders$ function where the remainders of C with respect to D are computed. The list that is retrieved is given as a parameter to the $computeLCS$ function. The return value is a least common subsumer of the remainders.

```
def conceptContraction(concept_c, concept_d)

remainders = computeRemainders(concept_c, concept_d)
lcs = computeLCS(remainders)

return lcs
```

Listing 5.10: $conceptContraction(C, D)$ function

5.4 Testing of the implementation

With the completion of the implementation of all the algorithms, the next necessary step is to test the code and make sure that all the possible cases are caught and treated correctly. In this section we describe the test cases that we have executed for all the classes and their methods, and for all the functions.

Firstly, the methods of Atomic, Top, Role_Restriction, and Conjunction class are tested. Class Atomic requires an input of type string for the method $\_\_init\_\_()$. Meanwhile for method $create\_atomic()$ and $weak\_atomic()$ no input is required. As for the Top class, no input is required, and the testing is pretty straightforward. Contrarily, class Role_Restriction requires one of four possible types of concepts as an initializing parameter. The input can be an Atomic, Top, Role_Restriction, or Conjunction instance. Regarding Class Conjunction, there are several possible combinations of concepts that need to be tested. Considering the fact that the input required is a list of at least two elements, there are at least six possible combinations to be tested. For each of these possible inputs for the classes, all the methods have been tested. The results of the tests suggest that the methods created, given any possible parameter, are able to initialize and represent an instance of a certain class, and most importantly to compute the weakenings of that concept.

Next, the testing of the $subsumedBy()$ function is designed and executed. For the testing, it is necessary to provide the function with different combinations of parameters. The combinations should represent all the examined cases in section 3.3. Based on the results, it is confirmed that the $subsumedBy$ function returns the correct `True` or `False` values for each of the considered cases.

We continue with the testing of the $computeRemainders()$ function. For this function, the testing coincides in considering different examples that should return different results. The situations are as follows: (i) there exists at least one remainder, (ii) there

exist no remainders. For the first case, example 1 can be taken into consideration. The return list is proven to be correct, and therefore this test is successful. The second case can be by testing the case that $concept\_c$ or $concept\_d$ is a $\top$ concept. If $concept\_c$ is $\top$ concept, there would be no weakenings, and therefore the function should return an empty list. If concept_d is $\top$ concept, then there would be no weakenings of concept_c that are not subsumed by concept_d. Based on the testing, the handling of these cases is as predicted and is considered succesful.

Lastly, we have designed the testing of $computeLCS()$ function. The cases that we take into consideration are: (i) the input is an empty list, (ii) the input is a list containing at least one element. If the input is an empty list, the function should return an empty list as well. In case that the input list contains one or more remainders, the return list of the function should in the best case include weakenings of the next level and in the worst case it should include $\top$ concept. The testings prove that the results retrieved are correct, considering the cases mentioned above. For the first test, example 3 has been used.

As for the $conceptContraction$ function, we have not designed any specific tests. Considering the fact that all its component have successfully passed their testing, it is acceptable to state that this function is successful as well.

5.5 Conclusions

At the end of this chapter, we can state that we have successfully implemented all the algorithms that we have designed for the concept contraction. It is understandable that the implementation choices that we have made are always open for optimizations. Nonetheless, the testing of the implementation proves to be successful for each of the methods of the classes that we have created to represent specific types of concepts allowed in Description Logic $\mathcal{EL}$. The testing of the functions $subsumedBy$, $computeRemainders$ and $computeLCS$ also is proved to be successful as we have run several test cases to understand if the functions treat them gracefully.

Chapter 6

Conclusions and future work

In this chapter we draw a final conclusion on the work that we have presented in this master's thesis. In addition, we include a short discussion about potential future work.

6.1 Conclusions

In this work, we have given a short introduction to the Description Logics family, with a special focus on the Description Logic $\mathcal{EL}$. This is the language that is used for the development of the *concept contraction* operator, which is discussed in chapter 4. Beforehand, we present contraction in the AGM theory and the postulates that describe this operation. Having provided this introduction to belief change, we define the notion of concept contraction proposed in [Rienstra et al., 2018] and represent it with a set of postulates, which are reformulations of the original AGM postulates for contraction. From the six basic AGM postulates, only the *Recovery* postulate did not apply to the setting of the concept contraction and had to be replaced with the weaker *Relevance* and *Retainment* postulates. Concept contraction is described in three different settings: (i) an empty T-Box, (ii) an acyclic T-Box, and (iii) a cyclic T-Box. In the first two cases, concept contraction is proved to work, whereas in the third case, lcs concept contraction cannot be applied. In addition, we have highlighted the limitations of the concept contraction such as lack of expressivity of $\mathcal{EL}$ for the *union* constructor causing the authors to chose the least common subsumer as a solution, and the problems that the concept contraction encounters in the case of cyclic T-Boxes.

We have designed five algorithms: (i) $computeWeakenings$, (ii) $computeRemainders$, (iii) $subsumedBy$, (iv) $computeLCS$, (v) $conceptContraction$, valuable for the implementation of the concept contraction operation. These algorithms ensure that all the computations needed to implement the operator are completed. Respectively, the algorithms complete the following objectives:

- Computation of the weakenings of a concept C
- Computation of the remainders of a concept C with respect to D
- Implementation of the subsumption between concepts in $\mathcal{EL}$
- Computation of the least common subsumer
- Implementation of the concept contraction operator, consisting of all the forementioned algorithms.

Regarding the analysis of the algorithms, we have primarily discussed the best and worst-case scenarios for each of the algorithms. An average-case scenario cannot be discussed without extensive research because it is not clear what classifies as average for a concept. For all the algorithms, the best-case scenario happens when concept C or concept D is a $\top$ concept. The worst-case scenario for all the algorithms is when concept C is an existential role restriction of a conjunction. As for concept D, the values vary from an existential role restriction of a conjunction to a $\top$ concept. Regarding the complexity of the algorithms, the analysis is based on the worst-case scenario as it is of interest to know how bad can the algorithms perform. In the design of the $computeContraction$ algorithm, we have considered the selection function is a given. This assumption has caused the analysis of the computation to not been complete as we have ignored the complexity of the selection function algorithm. In all cases, the complexity of the algorithms is polynomial. Assuming that concepts of large size are not very common, this kind of complexity can be acceptable.

To answer the research question of how to determine a reasonable selection function, we have proposed an approach. The approach consists of several elements into creating a selection function, such as:

- Creation of a framework of metadata specific for a concept description. The set of metadata that we have chosen is based on the merge and adaption of two different ontology description frameworks.
- An approach for the representation of the concepts and their respective metadata based on [Schueler et al., 2008]
- An approach for the computation of a certainty degree of a concept based on its metadata.

Lastly, we have implemented all the algorithms that we have designed for the concept contraction operator. The testing of the implementation proves to be successful for each of the methods of the classes that we have created to represent specific types of

concepts allowed in description logic $\mathcal{EL}$. The testing of the functions $subsumedBy$, $computeRemainders$ and $computeLCS$ also is proved to be successful as we have run several test cases to understand if the functions treat them gracefully.

The aim of the thesis was the implementation of the concept contraction in the Description Logic $\mathcal{EL}$. We believe that we have answered all the research questions that we raised in the beginning of the thesis. Nonetheless, there is always room for improvement and in the next section we discuss about possible future work based on the thesis.

6.2 Future work

In this section we will describe possible improvements of the thesis. These improvements can be a starting point for future work.

- The Description Logic language chosen for the concept contraction operator is $\mathcal{EL}$. Even though it is a powerful language, its limitations in expressivity force us to find alternative solutions for the modeling of the operator. It would be interesting to study how concept contraction and its properties can be redefined in other languages.

- When one designs algorithms, there is always room for optimization. Defining an algorithm to compute the size of a lattice of generalizations can also be of interest. A further study of the topic could help in this direction and possibly affect the complexity of the algorithms.

- Throughout the thesis, we have dealt with concept contraction in the setting of an empty T-box. The adaption of the algorithms that we have designed to apply in the setting of an acyclic T-Box can be part of future work. The research can also try to find alternative solutions for the case when the T-Box is cyclic.

- As stated before, the analysis of the algorithms is not officially complete. Therefore, future work focusing on this aspect can help to understand the behaviour of the algorithms better.

- The implementation of the approach of the selection function can be an exciting topic to study. We have only presented bits of existing theories and technologies, but it is evident the connection of the components is not realized. The discussion on how to compute a certainty degree is always interesting. The implementation of the selection function would make the analysis of the algorithm $conceptContraction$ complete.

- We have implemented the concept contraction operator in Python, but even though it was successful, this is only a prototype of the operator. The implementation of the operator as an extension to the Ontology editor Protégé [1] would be the next step for a full and usable implementation.

This section concludes the work of this master's thesis. The thesis is always open for improvement and the fore-mentioned ideas tackles some of the main assumptions and limitation of the thesis.

[1] https://protege.stanford.edu/

List of Algorithms

List of Tables

List of Figures

Literature

[Alchourrón et al., 1985] Alchourrón, C. E., Gärdenfors, P., and Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic*, 50(2):510–530.

[Baader, 2003] Baader, F. (2003). Computing the least common subsumer in the description logic $\mathcal{EL}$ w.r.t. terminological cycles with descriptive semantics. In *International Conference on Conceptual Structures*, pages 117–130. Springer.

[Baader, 2017] Baader, F. (2017). Basic description logics. In *[11]*, pages 43–95.

[Baader et al., 2005] Baader, F., Brandt, S., and Lutz, C. (2005). Pushing the EL envelope. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 364–369.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

[Baader et al., 2008] Baader, F., Horrocks, I., and Sattler, U. (2008). Description logics. *Foundations of Artificial Intelligence*, 3:135–179.

[Baader et al., 2018] Baader, F., Kriegel, F., Nuradiansyah, A., and Peñaloza, R. (2018). Making repairs in description logics more gentle (extended abstract). In *Proceedings of the 31st International Workshop on Description Logics co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), Tempe, Arizona, US, October 27th - to - 29th, 2018.*

[Baader et al., 1999] Baader, F., Küsters, R., and Molitor, R. (1999). Computing least common subsumers in description logics with existential restrictions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 96–103.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.

[Brachman, 1985] Brachman, R. J. (1985). Schmolze. *JG An overview of the KL-ONE representation system. Cognifive Science*, 9(2):171–216.

[Breitman et al., 2007] Breitman, K. K., Casanova, M. A., and Truszkowski, W. (2007). *Semantic Web: Concepts, Technologies and Applications*. NASA Monographs in Systems and Software Engineering. Springer.

[Ceder, 2010] Ceder, N. R. (2010). The quick python book. pages 3–7. Manning Publications Co.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press.

[De Giacomo et al., 2006] De Giacomo, G., Lenzerini, M., Poggi, A., and Rosati, R. (2006). On the update of description logic ontologies at the instance level. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1271–1276.

[Dutta et al., 2015] Dutta, B., Nandini, D., and Shahi, G. K. (2015). MOD: metadata for ontology description and publication. In *Proceedings of the 2015 International Conference on Dublin Core and Metadata Applications, DC 2015, São Paulo, Brazil, September 1-4, 2015*, pages 1–9.

[Gärdenfors et al., 1995] Gärdenfors, P., Rott, H., Gabbay, D., Hogger, C., and Robinson, J. (1995). Belief revision. *Computational Complexity*, 63:6.

[Hansson, 1991] Hansson, S. O. (1991). Belief contraction without recovery. *Studia logica*, 50(2):251–260.

[Hansson, 2017] Hansson, S. O. (2017). Logic of belief revision. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2017 edition.

[Heinz, 2018] Heinz, H. J. (2018). How i lost my owl: Retracting knowledge from el concepts. pages 8–9.

[Jonquet et al., 2018] Jonquet, C., Toulet, A., Dutta, B., and Emonet, V. (2018). Harnessing the power of unified metadata in an ontology repository: The case of agroportal. *J. Data Semantics*, 7(4):191–221.

[Lamy, 2017] Lamy, J. (2017). Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80:4–7.

[Liu et al., 2006] Liu, H., Lutz, C., Milicic, M., and Wolter, F. (2006). Updating description logic aboxes. In *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 46–56.

[Min et al., 2016] Min, H., Turner, S., de Coronado, S., Davis, B., Whetzel, T., Freimuth, R. R., Solbrig, H. R., Kiefer, R. C., Riben, M., Stafford, G. A., Wright, L. W., and Ohira, R. (2016). Towards a standard ontology metadata model. In *Proceedings of the Joint International Conference on Biological Ontology and BioCreative, Corvallis, Oregon, United States, August 1-4, 2016.*

[Palma et al., 2006] Palma, R., Hartmann, J., and Gómez-Pérez, A. (2006). Towards an ontology metadata standard. *European Semantic Web Conference*.

[Patel-Schneider, 2004] Patel-Schneider, P. F. (2004). Owl web ontology language semantics and abstract syntax, w3c recommendation. *http://www. w3. org/TR/2004/REC-owl-semantics-20040210/.*

[Qi and Du, 2009] Qi, G. and Du, J. (2009). Model-based revision operators for terminologies in description logics. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 891–897.

[Qi et al., 2006] Qi, G., Liu, W., and Bell, D. A. (2006). Knowledge base revision in description logics. In *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, pages 386–398.

[Ribeiro, 2012] Ribeiro, M. M. (2012). *Belief revision in non-classical logics*. Springer Science & Business Media.

[Rienstra et al., 2018] Rienstra, T., Schon, C., and Staab, S. (2018). Concept contraction in the description logic $\mathcal{EL}$.

[Schueler et al., 2008] Schueler, B., Sizov, S., Staab, S., and Tran, D. T. (2008). Querying for meta knowledge. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 625–634.

[Suchanek et al., 2016] Suchanek, F. M., Menard, C., Bienvenu, M., and Chapellier, C. (2016). Can you imagine... A language for combinatorial creativity? In *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, pages 532–548.

[Swamy, 2014] Swamy, H. K. (2014). Core python applications programming (third edition) by wesley j. chun. *ACM SIGSOFT Software Engineering Notes*, 39(3):24.

[Zheleznyakov et al., 2019] Zheleznyakov, D., Kharlamov, E., Nutt, W., and Calvanese, D. (2019). On expansion and contraction of dl-lite knowledge bases. *J. Web Semant.*, 57.

Internet Literature

[pyt, 2019] (2019). About python. https://www.python.org/.

[her, 2019] (2019). Hermit reasoner. http://www.hermit-reasoner.com/.

[Dutta and Jonquet, 2019a] Dutta, B. and Jonquet, C. (2019a). Mod-ontology. https://github.com/sifrproject/MOD-Ontology.

[Dutta and Jonquet, 2019b] Dutta, B. and Jonquet, C. (2019b). Repository of mod-ontology. https://www.isibang.ac.in/~bisu/.

[Lamy, 2018] Lamy, J.-B. (2018). The great ontology table.

[(OEG), 2015] (OEG), O. E. G. (2015). Omv - ontology metadata vocabulary. http://mayor2.dia.fi.upm.es/oeg-upm/index.php/en/downloads/75-omv/index.html.