Saturation-based Algebraic Reasoning for Description Logic $\mathcal{ALCHQ}$

Jelena Vlasenko

A Thesis

In the Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Computer Science) at

Concordia University

Montréal, Québec, Canada

April 2024

© Jelena Vlasenko, 2024

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Jelena Vlasenko

Entitled: Saturation-based Algebraic Reasoning for Description Logic $\mathcal{ALCHQ}$

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

Dr. Rabin Raut

______________________________ External Examiner

Dr. Weichang Du

______________________________ Examiner

Dr. Ferhat Khendek

______________________________ Examiner

Dr. Leila Kosseim

______________________________ Examiner

Dr. Juergen Rilling

______________________________ Supervisor

Dr. Volker Haarslev

Approved by ______________________________

Dr. Leila Kosseim,
Graduate Program Director

April 2, 2024 ______________________________

Dr. Mourad Debbabi, Dean,
Gina Cody School of Engineering and Computer Science

ABSTRACT

Saturation-based Algebraic Reasoning for Description Logic $\mathcal{ALCHQ}$

Jelena Vlasenko

Concordia University, 2024

In this work we present a novel calculus for the description logic $\mathcal{ALCHQ}$ implemented in a reasoner named Avalanche. $\mathcal{ALCHQ}$ permits intersection, disjunction, and negation of concepts. It also allows existential and universal restrictions, role hierarchy, and qualified number restrictions that are of particular interest to us. Avalanche incorporates a number of widely applied and well known optimization techniques such as saturation, resolution, and linear optimization. We apply saturation to create a compressed version of a saturation graph. As a result, the overall size of the constructed model can be kept reasonably small. We employ resolution techniques in order to reason on disjunctions that are part of our calculus. Finally and most importantly, we leverage linear optimization to handle qualified number restrictions. We transform qualified number restrictions into linear programs and then apply the Branch-and-Price algorithm to solve them in the most efficient way. This novel approach gives us a clear advantage over the other reasoners that implement a more traditional procedure to deal with qualified number restrictions as there are ontologies containing entailments caused by the presence of qualified number restrictions that can be classified only by Avalanche.

Acknowledgements

It took me a decade to finish my PhD journey. It was a big part of my life and now I cannot believe I am at the finish line. Despite the fact that it is time to move on I will certainly miss my academic life. If somebody asked if I would do it again my answer would be yes but I would have done it better. Unfortunately it is a part of a learning process and now I know how to do it better. I did not know it when I started this work.

I learnt a lot, I grew a lot, I met many wonderful people. First and foremost I would like to express my sincere gratitude to my supervisor Prof. Volker Haarslev. I was lucky to be supervised by him. Without his support this thesis would never be finished. Under his supervision I learnt things that I never hoped to master. For that I will be forever grateful as knowledge is one of the most wonderful gifts someone can offer.

I would like to thank my examiners for reading this thesis. I understand that probably it is not the most entertaining reading.

I would also like to mention my colleagues who helped me to become a better software developer which in turn helped me to implement the Avalanche reasoner. I wish I had this knowledge before I started working on the implementation.

Finally, I would like to thank my family for their patience and understanding. Without my husband's support I would never be able to go that far. As well as my two daughters and the third one who should arrive shortly after the final presentation, I hope this work will inspire them to do great things in life and believe that a woman can achieve anything she wants if she is ready to work hard. This work is dedicated to them.

Contents

List of Figures x

List of Tables xii

List of Abbreviations xiii

List of Symbols xiv

1 Introduction 1

1.1 Introduction to the Semantic Web 1

1.2 Relationship between Ontologies and Description Logics 5

2 Presentation of Description Logic $\mathcal{ALCHQ}$ Formal Semantics 7

2.1 Presentation of Description Logic $\mathcal{ALCHQ}$ Formal Semantics 7

2.2 An Example of DL $\mathcal{ALCHQ}$ Application 10

3 Related Works 12

3.1 Reasoning in Description Logics 12

3.2 Optimization Techniques 22

4 Motivation and Research Objectives 28

4.1 Motivation 28

4.2 Research Objectives 33

5 Description Logic Reasoner Avalanche **35**

5.1 Saturation-based reasoner Avalanche 35

5.2 Saturation Graph 37

5.2.1 Saturation Nodes 37

5.2.2 Static Node 38

5.2.3 Identified Node 39

5.2.4 Auxiliary node 39

5.2.5 Anonymous Node 40

5.2.6 Clone Node 41

5.2.7 Subsumption Clone 42

5.2.8 Disjointness Clone 42

5.2.9 Unfold Node 43

5.2.10 Connecting Edge 43

5.3 Normalization Process 43

5.3.1 Normal Form 43

5.3.2 Left-hand Side Normalization Rules 45

5.3.3 Right-hand Side Normalization Rules 46

5.3.4 Example of Normalization Rules Application 48

5.4 Calculus Presentation 50

5.4.1 Notation 50

5.4.2 Avalanche Saturation-based Rules 52

5.4.3 Implementation Details 62

5.5 Reasoning with Qualified Number Restrictions 63

6 Avalanche Implementation Details **66**

6.1 Overview of Avalanche 66

6.2 Communication between Avalanche and QMediator 68

7 Linear Programming Engine QMediator **71**

7.1 Interaction with Avalanche 71

7.2 Input Presentation 72

7.3 Branch and Price Approach 73

7.3.1 Column Generation 73

7.3.2 Generation and Interpretation of Constraints 76

7.3.3 Branch-and-Bound 80

7.3.4 Clash Set Detection 81

7.4 Examples and Result Interpretation 82

7.4.1 Simple Example 82

7.4.2 Branch and Bound Example 87

8 Proofs **98**

8.1 Termination 98

8.2 Soundness 101

8.3 Completeness 111

9 Complexity Analysis **117**

10 Performance Evaluation **119**

10.1 Canadian Parliament Benchmarks 120

10.1.1 Performance Evaluation for $\mathcal{ALCQ}$ Ontologies 121

10.1.2 Performance Evaluation for $\mathcal{ELQ}$ Ontologies 123

10.2 Satisfiable and Unsatisfiable $\mathcal{ALCHQ}$ Benchmarks 125

10.2.1 Performance Evaluation for Satisfiable $\mathcal{ALCHQ}$ Ontologies . 125
10.2.2 Performance Evaluation for Unsatisfiable $\mathcal{ALCHQ}$ Ontologies 127
10.3 Performance Benchmarks 127
10.3.1 Performance Evaluation for Satisfiable Performance Benchmarks 128
10.3.2 Performance Evaluation for Unsatisfiable Performance Ontologies 131

11 Conclusions and Future Work 133
11.1 Conclusions 133
11.2 Future Work 135

Bibliography 144

A Publications 145

B Example 146
B.1 Example of Rule Applications 146

C Evaluation 152
C.1 Performance Evaluation of $\mathcal{ALCQ}$ Ontologies 152
C.2 Performance Evaluation of $\mathcal{ELQ}$ Ontologies 153
C.3 Performance Evaluation of Satisfiable $\mathcal{SHQ}$ Ontologies 154
C.4 Performance Evaluation of Unsatisfiable $\mathcal{SHQ}$ Ontologies 158
C.5 Performance Evaluation of Satisfiable **P** Ontologies 160
C.6 Performance Evaluation of Unsatisfiable **P** Ontologies 165

List of Figures

1.1 Semantic Web Stack [48] 2

5.1 Clash Detection 64

10.1 $\mathcal{ALCQ}$ benchmark runtimes in seconds 123
10.2 $\mathcal{ELQ}$ benchmark runtimes in seconds 124
10.3 Sat-$\mathcal{ALCHQ}$ benchmark runtimes in seconds 126
10.4 Unsat-$\mathcal{ALCHQ}$ benchmark runtimes in seconds 128
10.5 P-Sat benchmark runtimes in seconds 131
10.6 P-Unsat benchmark runtimes in seconds 132

B.1 Initialization 148
B.2 Application of the Rule $R_{\sqsubseteq}$ 148
B.3 Creation of a Cloned Node 149
B.4 Creation of an Anonymous Node 150
B.5 Propagation of Subsumers to the Anonymous Node 151

List of Tables

2.1 DL $\mathcal{ALCHQ}$ Syntax and Semantics 8

6.1 Input parameters for QMediator 69

6.2 Input parameters for QMediator 69

8.1 Summary of the Saturation-based Rules (ϕ = any subsumer except for a negated one, τ = any subsumer, r = role, q = qualified number restriction, n = cardinality) 103

8.2 Summary of the Possible Subsumers Rules (*is_new* = does not exist in the label) 106

8.3 Summary of the Disjunction Rules (*res* = the result of the resolution, resolvent = function that resolves disjunctions) 109

10.1 Total CPU time and speedup factor for $\mathcal{ALCQ}$ benchmarks 122

10.2 Total CPU time and speedup factor for $\mathcal{ELQ}$ benchmarks 124

10.3 Total CPU time and speedup factor for Sat-$\mathcal{ALCHQ}$ benchmarks . . 126

10.4 Total CPU time and speedup factor for Unsat-$\mathcal{ALCHQ}$ benchmarks . 127

10.5 Total CPU time and speedup factor for Sat-$\mathcal{ALCHQ}$ benchmarks . . 131

10.6 Total CPU time and speedup factor for Unsat-$\mathcal{ALCHQ}$ benchmarks . 132

C.1 Benchmarks using $\mathcal{ALCQ}$ Ontologies 152

C.2 Benchmarks using $\mathcal{ELQ}$ Ontologies 153

C.3 Benchmarks for $\mathcal{ALCHQ}$-SAT Ontologies 157
C.4 Benchmarks for $\mathcal{SHQ}$-UNSAT Ontologies 160
C.5 Benchmarks for Performance Ontologies 165
C.6 Benchmarks for Unsatisfiable Performance Ontologies 169

List of Abbreviations

ALCHQ	**A**ttributive **L**anguage with **C**omplex Concept Negation, Role **H**ierarchy, and **Q**ualified Cardinality Restrictions
BnB	**B**ranch a**n**d **B**ound
BnP	**B**ranch a**n**d **P**rice
CNF	**C**onjunctive **N**ormal **F**orm
DL	**D**escription **L**ogic
DNF	**D**isjunctive **N**ormal **F**orm
GCIs	**G**eneral **C**oncept **I**nclusions
ILP	**I**nteger **L**inear **P**rogramming
ITR	**I**nitial **T**ransformation **R**ules
KR	**K**nowledge **R**epresentation
LP	**L**inear **P**rogramming
NF	**N**ormal **F**orm
NNF	**N**egation **N**ormal **F**orm
OWL	**W**eb **O**ntology **L**anguage
PP	**P**ricing **P**roblem
QCR	**Q**ualified **C**ardinality **R**estriction
RMP	**R**educed **M**aster **P**roblem

List of Symbols

$L(v)$	a node label that contains subsumers of the node
$L(v_A)$	a node label that contains subsumers of the node with the representative concept A
ϕ	a named concept, a qualified number restriction, or a disjunction
$\dot{\neg}\phi$	a negated concept in Negation Normal Form
τ	a named concept, a qualified number restriction, a disjunction, or a negated concept
$\langle r, q, n \rangle$	a tuple returned by the ILP module that we also call QMediator where r is a role, q is a qualification and n is a cardinality
q	a qualified number restriction
$\sigma(v)$	a function that extracts a cardinality, a role, and a role filler from a qualified number restriction that is a subsumer of the node v
V	graph nodes
$\#v_q$	a cardinality of a node v
$infeasible$	infeasible inequalities
$\bowtie n\, R.A$	either $\leq nR.A$ or $\geq nR.A$
is_new	a function that verifies that a qualified number restriction exists in a node label
add_to	a function that adds a qualified number restriction to a label of possible/non-possible subsumers of a given node

$L_P(v_B)$	a label of possible subsumers of a node with the representative concept B
$L_P^{\neg}(v_B)$	a label of non-possible subsumers of a node with the representative concept B
φ	a function that adds a new concept to preconditions of a tuple
Q	a qualified number restriction
$clone(v, B)$	a clone node that tests if node v is subsumed by B
$\mathsf{C}_{\mathcal{T}}^{\mathsf{Q}}$	qualified number restrictions and their negations in the TBox $\mathcal{T}$
P	possible subsumers

Chapter 1

Introduction

In this chapter we will present one the most common applications of description logics - the Semantic Web. We will give an overview of this technology and describe how this work could contribute to its development.

1.1 Introduction to the Semantic Web

In today's world, the Internet has become an indispensable tool of our everyday life despite the fact that it only became available to the general public in the 1990s. Even though people are capable of using the Web to carry out numerous tasks like paying bills or searching for information, the natural limitations of the interaction between a human and a computer combined with the extreme diversity of the data available on the Web present numerous challenges. Routine tasks such as finding the cheapest possible plane ticket to a given destination or the most relevant article on a given topic tend to be time-consuming and tedious for Internet users. Ideally, a potential computer program that has access to the same information that is available to a human Internet user should be able to accomplish such tasks much more efficiently. Such computer programs already exist and are known as "Intelligent

Agents". Unfortunately, these programs can not at present reliably accomplish very complex tasks. The reason for this is that most web pages today are designed to be read and comprehended by people and not by machines. This makes it extremely difficult for the intelligent agents to interpret the data available on the Internet. In order to overcome these obstacles, the concept of Semantic Web was introduced by Tim Berners-Lee, the inventor of the World Wide Web and director of the World Wide Web Consortium (W3C). Tim Berners-Lee proposed an idea of improving the existing network of hyperlinked human-readable web pages by augmenting them with machine-readable metadata. The metadata would serve to describe the content of these web pages and the ways they relate to each other. This enhancement would allow automated agents to access the Web more intelligently and perform certain tasks on behalf of humans. The realization of the Semantic Web would help the progress of the current Web by allowing human Internet users to access and share information more efficiently. Description logic was selected as the centrepiece of the implementation of the Semantic Web.

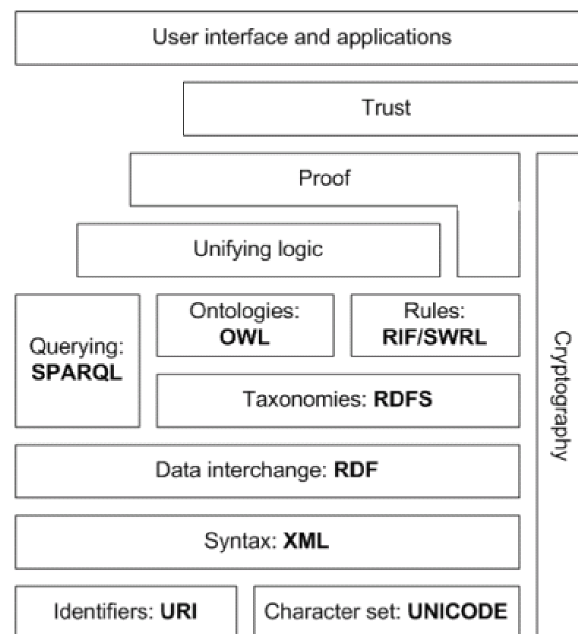

FIGURE 1.1: Semantic Web Stack [48]

A representation of the high-level architecture of the Semantic Web, as it was originally envisioned by Tim Berners-Lee, is presented in Figure 1.1. The Semantic Web Stack represents a hierarchy of languages and technologies that make up the various layers of the Semantic Web. It is the responsibility of each layer to provide services that are to be used by the layer above it. Note that some of these layers are crosscutting: a good example is the Cryptography layer depicted below, which is used by almost all other depicted layers. The technologies in the stack are divided into three broad groups based on the services they provide.

The first group consists of hypertext web technologies and serves as a foundation to the Semantic Web:

IRI Internationalized Resource Identifier (IRI) provides unique identifiers to the Semantic Web resources.

Unicode Unicode allows to process documents written in different languages.

XML Extensible Markup Language (XML) creates documents with structured data.

The second group consists of semantic web technologies which have been standardized by W3C and provides means to implement the Semantic Web applications:

RDF The Resource Description Framework (RDF) provides a framework for representing information about resources.

RDFS The basic vocabulary for RDF is provided by RDF Schema (RDFS). RDFS allows to create hierarchies of classes and object properties.

OWL The Web Ontology Language (OWL) is an extension of RDFS that adds more semantic features to RDF statements, e.g. cardinality restrictions, disjunction, intersection etc. OWL is based on description logics and allows us to reason about information available through the Semantic Web.

SPARQL SPARQL is a query language that is used to retrieve RDF-based data from the Semantic Web.

RIF/SWRL The Semantic Web Rule Language (SWRL)/Rule Interchange Format (RIF/SWRL) allows to implement rules. Rules allow to express an IF-THEN construct. Thus, if the conditions specified in the IF part hold then the conditions specified in the THEN part should hold as well.

Whereas the technologies in the first and second groups are already well known, the third group consists of technologies that have not yet been standardized or implemented:

Cryptography Cryptography should guarantee that the Semantic Web data are coming from a trusted source. This would assure that the derived statements obtained by applying logic are trustworthy.

Unifying Logic Formal logic should be applied to derive new statements.

Proof Proofs should be provided to ensure that the derived statements are correct.

Trust Trust should be developed to ensure that the derived statements obtained with use of the logic are trustworthy.

User Interface and Applications User Interfaces and Applications should enable human Internet users to employ the Semantic Web technology for their daily activities.

1.2 Relationship between Ontologies and Description Logics

The notion of ontology is fundamental in the fields of the Semantic Web, Artificial Intelligence, Systems Engineering, and many others. In the context of Computer Science, an ontology is a formal representation of knowledge as a set of concepts defined within a given domain, as well as the relationships that exist between these concepts. An ontology serves to describe a domain and to allow reasoning about the entities of that domain by application of Description Logic rules. Ontologies are often referred to as "knowledge bases". Further in the document these terms will be used interchangeably.

In the Semantic Web the main means of specifying ontologies is OWL: a knowledge representation language that is based on a description logic (DL). Description logics are a family of formal languages based on First Order Logic that are widely used in the area of Artificial Intelligence. The main benefit that DL brings to the Semantic Web is the reasoning power by means of specialized algorithms called "reasoners". Reasoners are deductive inference engines that perform logical inference over ontologies expressed as DL knowledge bases. As a result they can infer implicit knowledge from the information that is explicitly stated in the knowledge base.

There is a number of reasoners both free and paid that are currently available for usage but unfortunately all of them display a dramatic performance degradation as processed ontologies grow larger or the DL becomes more expressive. The lack of means of efficiently reasoning about ontologies could possibly be one of the main reasons why they have not seen wider adoption. The goal of this work is to remedy this by developing a reasoning algorithm that is more efficient than the ones

currently in existence. We achieved this by building a novel reasoner for the description logic $\mathcal{ALCHQ}$. $\mathcal{ALCHQ}$ is a subset of a more powerful description logic named $\mathcal{SROIQ}$ that is the basis of the OWL 2 ontology language.

Chapter 2

Presentation of Description Logic $\mathcal{ALCHQ}$ Formal Semantics

In this chapter we will provide a detailed presentation of the formal semantics of the description logic $\mathcal{ALCHQ}$ and demonstrate its application by means of a simple example.

2.1 Presentation of Description Logic $\mathcal{ALCHQ}$ Formal Semantics

Description logics are formal languages that are used to represent knowledge. They are usually classified by their set of supported logical operators. These logical operators define the expressivity of the description logic in question. The more operators are allowed, the more expressive is the resulting description logic. However, increased expressivity entails an increase of reasoning complexity. Therefore a reasoning algorithm designed for a more expressive a description logic will be more complex and will often suffer from slowdown in performance.

Name	**DL**	**Interpretation**
Concepts		
concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, A$ is concept name
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{x \| \forall y : (x, y) \in R^{\mathcal{I}} \to y \in C^{\mathcal{I}}\}$
universal restriction	$\forall R.C$	$\{x \| \forall y : (x, y) \in R^{\mathcal{I}} \to y \in C^{\mathcal{I}}\}$
at-least restriction	$\geq n\, R.C$	$\{x \| \# R^{\mathcal{I}}(x, C) \geq n\}$
at-most restriction	$\leq n\, R.C$	$\{x \| \# R^{\mathcal{I}}(x, C) \leq n\}$
Axioms		
concept subsumption	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
concept equivalence	$C \equiv D$	$C^{\mathcal{I}} \equiv D^{\mathcal{I}}$
Roles		
role definition	$R \in N_R$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$

TABLE 2.1: DL $\mathcal{ALCHQ}$ Syntax and Semantics

In this work, we focus on the description logic $\mathcal{ALCHQ}$. $\mathcal{ALCHQ}$ stands for $\mathcal{A}$ttributive $\mathcal{L}$anguage with $\mathcal{C}$omplex concept negation, role $\mathcal{H}$ierarchy, and $\mathcal{Q}$ualified cardinality restrictions. Its expressivity is summarized in Table 2.1. $\mathcal{ALCHQ}$ permits intersection, disjunction, and negation of concepts. It also makes use of qualified number restrictions (in this work we will use the terms qualified number restrictions and qualified cardinality restrictions interchangeably), existential and universal restrictions, and role hierarchy.

A description logic consists of concepts that can form axioms by combining allowed logical operators. An axiom is a logical statement that employs concepts and roles.

Let us assume that A is an atomic concept (i.e., it cannot be further unfolded) and R is a role. Then in the context of the DL $\mathcal{ALCHQ}$ complex concept expressions

C, D can be recursively defined as follows:

$$C, D = A|\top|\bot|\neg D|C \sqcap D|C \sqcup D|\exists R.C|\forall R.C| \geq nR.C| \leq nR.C$$

Using the definition above we can construct axioms by applying binary logical operators for subsumption denoted by $\sqsubseteq$ and equivalence denoted by $\equiv$.

Expressions in $\mathcal{ALCHQ}$ and in other DLs are divided into three groups: TBox, ABox, and RBox.

TBox or terminological box contains general concept inclusion axioms (GCIs) in the form of $C \sqsubseteq D$ or $C \equiv D$. For example, $F \sqsubseteq E$ can be interpreted as all the individuals who speak French must also speak English.

ABox or assertional box contains individuals and their relationship to classes and roles. A concept assertion is a statement of the form $a : C$ where a is an individual and C is a concept. For example, $john : E$ means that there is an individual $john$ who speaks $English$. Further, a role assertion is a statement of the form $(a, b) : R$ where a, b are individuals and R is a role. For example, $(john, jane) : knows$ means that an individual $john$ is related to an individual $jane$ via the role $knows$, i.e. $john$ $knows$ $jane$.

Finally, RBox or role box contains role properties and role hierarchies. In this work we focus only on role hierarchies. For example, if $hasFriend \sqsubseteq knows$ then it means that the role $hasFriend$ is a subrole of the role $knows$: $john$ $hasFriend$ $jane$. The latter will state that $john$ $hasFriend$ $jane$ and also $knows$ $jane$.

Formally, expressions in $TBox$, $ABox$, and $RBox$ describe sets of individuals that belong to explicitly stated concept descriptions. Concept descriptions are defined with the help of a set of concept constructors, such as conjunction, negation etc. The available constructors determine the expressive power of the DL in question. In this work we consider concept descriptions built from the constructors presented in the Table 2.1. We use the standard *Tarsky*-style semantics that define concept

descriptions in terms of an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. The domain $\Delta^{\mathcal{I}}$ of $\mathcal{I}$ is a non-empty set $\Delta^{\mathcal{I}}$ and the interpretation function maps each concept name A to a subset $\mathcal{A}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and each role R to a binary relation $\mathcal{R}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and every individual name a to an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

2.2 An Example of DL $\mathcal{ALCHQ}$ Application

DL $\mathcal{ALCHQ}$ is an expressive description logic that allows us to form various statements about an arbitrary domain. For example, we can describe some of the properties of a multilingual city like Montreal.

Let us assume that we have two sets people living in Montreal:

- Anglophones denoted by a concept A
- Francophones denoted by a concept F

Then, we will introduce two additional concepts that will denote two languages widely used in the city in question:

- French language: *French*
- English language: *English*

We will make them both subclasses of a concept *Language*:

- French is a language: $French \sqsubseteq Language$
- English is a language: $English \sqsubseteq Language$

Finally, we will introduce a role *speaks* and a role *understands*. Now with the help of our new vocabulary we can express the following information:

- Anglophones belong to a class Person: $A \sqsubseteq P$

- Francophones belong to a class Person as well: $F \sqsubseteq P$
- A person is a francophone or an anglophone: $P \sqsubseteq F \sqcup A$
- There is no one who can speak French: $\top \sqsubseteq \leq 0\, speaks.French$
- There is at least one person who speaks French: $\top \sqsubseteq \geq 1\, speaks.French$
- A bilingual is a person who can speak both English and French: $B \sqsubseteq P$ and $speaks.(English \sqcap French) \sqsubseteq B$
- There are at least 5 people who can speak English and at most 30 who can speak French: $\top \sqsubseteq \geq 5\, speaks.English$ and $\leq 30\, speaks.French$
- Someone who can speak English can also understand English: $\exists\, speaks.English \sqsubseteq \exists\, understands.English$

This is just a small example that does not even use all of the features of the DL $\mathcal{ALCHQ}$. However, it gives an idea of how the DL in question can be used in real life.

Chapter 3

Related Works

We introduced the Semantic Web and its potentials in Chapter 1. Then we discussed the description logic $\mathcal{ALCHQ}$ and presented its formal semantics in Chapter 2. In this chapter we will cover some of the studies conducted in the area of description logic specialized on Reasoner Optimization. These are the works that inspired and underpinned this study.

3.1 Reasoning in Description Logics

Description logics are a family of knowledge representation languages that provide formal semantics and reasoning power to the Web Ontology Language (OWL), which is in turn part of the Semantic Web architecture as it is depicted in Figure 1.1. Typically, reasoning is realized by implementing powerful tableau-based algorithms, known as semantic reasoners, that are able to process DL knowledge bases and to infer previously unknown knowledge from them. The main task of a DL reasoner is to classify its input ontology. The classification is a process of computing all subsumptions in a given ontology (i.e. a taxonomy). This task can be very time-consuming. The more expressive is the DL of the ontology in question, the

more time is generally needed to classify it. Consistency checking is another typical reasoning task. Its goal is to discover if an ontology can have at least one model. Other common reasoning tasks include testing concept satisfiability, subsumption checking between concepts, and checking if an individual could be an instance of a specific concept.

The main and the most expressive DL that is currently recognized by OWL is $\mathcal{SROIQ}$. $\mathcal{SROIQ}$ is composed of different language constructs that have partially been presented in Chapter 2. One challenge that it presents is that reasoning on very expressive knowledge bases tends to be computationally expensive.

In order to improve reasoner performance the language can be made less complex by leaving out certain logical constructs. Hence, striking the right balance between the expressivity of an ontology language and performance of reasoning algorithms remains an open question. This issue has been widely studied by different researchers and various optimization techniques have been proposed in order to speed up reasoning algorithms. These techniques will be discussed further in this chapter. In this section we will discuss some of the general studies conducted in the area of description logic reasoning.

Tableau-based algorithms are the most commonly applied reasoning algorithms to date according to [29]. In [29] the authors give a comprehensive overview of the tableau algorithms. They say that as the tableau-based algorithms do not have a formal definition they are identified by their distinctive features. A tableau-based algorithm constructs a completion tree or a graph in the presence of nominals or inverse roles that represents an abstraction of a model for the knowledge base in question. To construct the tree the algorithm creates an initial node and then expands it by applying completion rules to the axioms in the knowledge base. The completion rules describe the preconditions that must be satisfied in order to apply a corresponding

rule and the side effects of the rule application. In many cases, the completion rules only operate on a node and its direct neighbours but occasionally they also affect the nodes that are arbitrarily far apart or they transform global information that is applicable to all the nodes. As multiple rules may be applicable to a given node at the same time it is essential to design and implement a suitable rule application strategy. An inconsistency in the completion tree is detected with the help of clash triggers. A typical example of a clash is presence of a concept and its complement in a node, i.e. $C, \neg C$. It indicates that the current tree cannot be transformed into a model. In the case when disjunctions are present in the knowledge base other paths should be explored. This should be done until either a clash-free completion tree can be constructed or until all possible paths have been explored and no clash-free completion tree can be found. Thus, if a knowledge base has at least one model then its completion tree should be clash-free. Otherwise, the knowledge base does not have a model.

In [8] the authors present an extensive literature review on the studies conducted in the area of tableau-based reasoning algorithms since the year 1990. In this paragraph we will talk about some of them. The authors of [8] first introduce the standard tableau algorithm for a basic description logic language $\mathcal{ALC}$ and then they present possible extensions to the algorithm that allow to integrate more expressive description logics. These extensions include the following language constructs: number restrictions, terminological axioms, and role constructors. For the $\mathcal{ALC}$ language the tableau algorithm contains only four basic consistency preserving completion rules: the $\sqcup$-rule, the $\sqcap-$rule, the $\exists$-rule, and the $\forall$-rule. Then in order to handle number restrictions the algorithm was first extended by unqualified number restrictions $\mathcal{N}$ to represent at-most and at-least number restrictions - $\leq n\, R.\top$ and $\geq n\, R.\top$, or simply $\leq n\, R$ and $\geq n\, R$ respectively. The qualifying concept is the top

concept which is denoted in DL by the symbol $\top$. $\top$ is the concept that is satisfied at each node of the completion tree. In the $\mathcal{ALCN}$ logic it is not possible to impose any restriction on the class of the role fillers of the number restrictions above. According to the semantic of $\mathcal{N}$, the role fillers can belong to any class in the domain. $\mathcal{ALCQ}$ is an extension of $\mathcal{ALCN}$ that allows to impose a restriction on the class of the role fillers - $\leq n\,R.C$ and $\geq n\,R.C$. Thus, while in $\mathcal{ALCN}$ it would only be possible to formulate a restriction such as "a mother has at least 4 children", $\mathcal{ALCQ}$ enables us to express a restriction such as "a mother has at least 4 children that are female". Using formal DL syntax , the former would be denoted as $mother \sqcap \geq 4\, hasChild.\top$ (i.e. $mother \sqcap \geq 4\, hasChild$) while the latter would be $mother \sqcap \geq 4\, hasChild.Female$. However, this extension brings additional complexity to the algorithm. To check that the qualified number restrictions are satisfiable, the standard tableau algorithm first creates a number of role fillers to satisfy at-least number restrictions and then tries to merge the role fillers if any one of the at-most number restrictions has been violated. If it is impossible to satisfy all at-most restrictions, then the knowledge base becomes unsatisfiable. Thus, in $\mathcal{ALCN}$, the satisfiability algorithm requires an additional clash rule to verify that the number restrictions can be satisfied. In $\mathcal{ALCQ}$ the algorithm proceeds in a similar manner but it also adds qualifying classes to the role fillers. The algorithm also introduces an additional choose-rule that selects role fillers. Then another extension to the $\mathcal{ALC}$ language introduces features or functional roles. Functional roles allow having only one role filler per individual, e.g., $(a, b) : hasBirthMother$. With functional roles we can express that a person can have only one birthmother and several persons can have the same birthmother. The resulting language would be $\mathcal{ALCF}$. Another extension that adds expressivity to the $\mathcal{ALC}$ is the introduction of role hierarchies $R \sqsubseteq S$, inverse roles R^-, and transitive roles R^+. The role hierarchy allows to express that the individuals that belong to the

role R would also belong to a role S if R is a subrole of S, i.e, $R \sqsubseteq S$. For example, $(a,b) : knows$ and $knows \sqsubseteq likes$, then $(a,b) : likes$. If $(a,b) : likes$ and $likedBy$ is an inverse role of $likes$, then $(b,a) : likedBy$. Finally, transitive roles allow to build chains of linked individuals. For example, if $(a,b) : likes$ and $(b,c) : likes$, and $likes$ is a transitive role, then as a result $(a,c) : likes$ as well.

In [36] the authors focus on the the small DL language $\mathcal{EL}$ that offers conjunctions and existential restrictions. However, this fairly inexpressive language exhibits some very interesting computational properties. Namely, it is polynomial time decidable, i.e. tractable. By decidability we understand that it is sound, complete, and terminating. In addition, due to the inexpressiveness of $\mathcal{EL}$ it is relatively easy to implement an efficient classification procedure for this language. For example, in contrast to Tableau-based algorithms [8] that we will discuss later, in $\mathcal{EL}$ it is not needed to construct a counter model to test if one concept is subsumed by another one. In a Tableau-based algorithm to test if $A \sqsubseteq B$ we need to add a concept that is subsumed by $A \sqcap \neg B$ and to check if the subsumption holds. If it results in a clash then the algorithm concludes that it has discovered a new subsumption. Thus, we need to test all possible pairs of concepts in a given ontology if we want to compute its taxonomy. This is not the case in $\mathcal{EL}$ as all we need to do is to unfold axioms. For example, having that $A \sqsubseteq B$ we know that B is a subsumer of A. We would never test whether A is subsumed by C if was not directly stated in of the axioms in the ontology. Moreover, computing subsumptions in $\mathcal{EL}$ can be done in "one pass" which clearly has positive effects on performance and simplicity of the implemented algorithm. Reasoning algorithms for the DL $\mathcal{EL}$ are often referred to as saturation algorithms due to their nature of saturating knowledge. The authors of [36] present ELK - a reasoner that can efficiently process $\mathcal{EL}$ knowledge bases. They claim that at the time of the publication ELK was the most competitive existing reasoner for

$\mathcal{EL}$ knowledge bases. The goal of ELK is to provide scalability of the reasoning algorithm, as opposed to expressivity of the input language. The authors introduced a number of optimization techniques in order to make their reasoner perform well and efficiently process ontologies that contain hundreds of thousands axioms. These techniques are mostly architecture-dependent and are specifically tailored for ELK. In this work, we aim to implement a reasoning algorithm for a DL richer than $\mathcal{EL}$, namely $\mathcal{ALCHQ}$. However, our approach differs from [36] since we do not follow their reasoner implementation strategy.

Expressivity of a description logic might negatively affect the time needed by a semantic reasoner to process a knowledge base expressed in that description logic language. Thus, it is important to know which are the language constructs that create the most significant computational overhead by adding them to a smaller DL fragment. In [25] the authors explore how adding role conjunctions to different description logics affects complexity of standard reasoning tasks. Role conjunction can be expressed in a DL syntax as $\exists(R \sqcap S).A$, where A is a concept and R and S are roles, and interpreted as $\{x|\forall y : (x,y) \in R^{\mathcal{I}} \wedge \forall y : (x,y) \in S^{\mathcal{I}} \rightarrow y \in A^{\mathcal{I}}\}$, where x and y are variables. The authors find that adding role conjunctions to the description logics $\mathcal{SHI}$ and $\mathcal{SHOIF}$ causes a significant increase in the computational complexity from ExpTime-complete to 2ExpTime-complete and from NExpTime-complete to N2ExpTime-hard respectively. Moreover, they also argue that this increase is caused by the presence of inverse roles.

Another interesting observation is presented in [2]. The authors identify a set of expressive means that can be added to $\mathcal{EL}$ without sacrificing tractability. These DL constructs are the bottom concept, nominals, a restricted form of concrete domains, a restricted form of role-value maps. Presence of the bottom concept allows us to express disjointness of concepts, i.e. $A \sqcap B \sqsubseteq \bot$ would mean that A is disjoint to

B. Then, they show that all other additions of other typical DL constructors to $\mathcal{EL}$ with General Concept Inclusions (GCIs) make subsumption intractable, and in most cases even ExpTime-complete. Thus, it has been proven that adding qualified cardinality restrictions $\mathcal{Q}$ makes $\mathcal{ELQ}$ non-tractable and adding inverse roles $\mathcal{I}$ makes it PSpace-hard. In [4] the authors continue searching for balance between expressivity and tractability and prove that it is possible to extend $\mathcal{EL}$ with range restrictions and reflexive roles and still remain tractable.

The authors of [54] exploit the fact that saturation-based reasoners perform better on the OWL $\mathcal{EL}$ ontologies than tableau-based reasoners. The authors state that a saturation-based reasoner is able to very efficiently process the SNOMED ontology [53] that is written in the $\mathcal{EL}$ description logic and that it would not be possible to achieve this level of efficiency with a standard tableau-based reasoner. However, it is not possible to apply saturation-based rules alone to ontologies implemented in a more expressive description logic, for example to the ontologies that conform to the description logic $\mathcal{SHIQ}$. Thus, based on this observation, the authors propose to combine tableau- and saturation-based rules to build a reasoner that would achieve a better performance than existing tableau-based reasoners. The proposed reasoning algorithm generates data structures that could be later used as input for an extended tableau algorithm that supports more expressive DLs. Similar to the tableau completion rules, the saturation rules generate nodes that are labelled with sets of concepts. One concept serves as the representative concept of that node and the rest are the subsumers of that representative concept. Thus, it is possible to directly transfer results from the constructed saturation graph to a completion graph as the saturated labels can be used to initialize the labels of new nodes of the completion graph. The authors implement their approach in a reasoner named Konclude [56]. According to the results presented at the OWL Reasoner Evaluation

2014 (ORE2014), at the moment of publication Konclude showed significantly better performance than other well-known reasoners such as Hermit [49] and Fact++ [59]. The main advantage of the saturation technique is that no new nodes will be created unless they are needed. Whenever a new edge requires an individual of a specific class to be created, the algorithm links the edge to the node that contains this concept as a subsumer. Thus, application of the saturation technique allows to create significantly smaller models. Moreover, higher level reasoning tasks such as classification often exploit information that can be extracted from the constructed completion graphs [24]. A saturated graph can be used for this purpose as well. For example, if a node with a representative concept A is neither clashed nor critical (i.e. possibly incomplete and requires tableau rules to be applied to it), then the concept A is satisfiable and the label of that node contains the subsumers of A. In particular, if no node is critical (which is the case for many $\mathcal{EL}$ ontologies) then only a transitive reduction is necessary to classify the ontology in question. Thus making it possible to automatically get a one-pass classification for simple ontologies.

In [40] the authors present a novel reasoning calculus for the description logic $\mathcal{SHIQ}$. They aim to solve two of the main problems that exist in the traditional tableau-based algorithms - or-branching and and-branching. The or-branching is caused by the presence of disjunctions in knowledge bases. Clearly, it is possible to avoid disjunctions but in this case the knowledge bases will not be very expressive even though there exist knowledge bases that are described in such a way. For example, the standard description logic language that disallows disjunctions is $\mathcal{EL}$ that we have already discussed.

Disjunctions cause nondeterminism, making the reasoning process complicated. The problem is that there is no known technique for selecting the correct disjunct as

the first choice when building a completion tree. For example, when we have a logical expression such as $A \sqcup B \sqcup C \sqcup D \sqcup ...$, we need to choose only one concept and ignore the rest. However, we have no way of knowing choosing which concept will result in a logical clash and which will not. Therefore, the standard tableau-based algorithm usually picks any of the disjuncts and tries to build a completion tree. Then, if a logical clash is discovered as a result of the selection, the algorithm has to roll back and pick one of the remaining alternatives. The algorithm will continue its execution in this manner until no more rules can be applied. Consequently, we can assert that the knowledge base is satisfiable if the right disjunct has been found. In the worst case the algorithm might have to try all the disjuncts. Clearly, this procedure adds an undesirable computational overhead because there is no limit on the number of disjuncts in an expression.

In [42] the authors extend their previous work [40] by presenting a reasoning calculus for the description logic $\mathcal{SHOIQ}^{+}$. In particular, the authors add a rule to handle nominals $\mathcal{O}$. This approach leverages the hypertableau [12] and hyperresolution [45] calculuses in an attempt to reduce nondeterminism. In order to ensure termination, a specialized blocking condition is introduced - the "anywhere" pairwise blocking. The authors claim that this blocking condition decreases size of the constructed model. They also present an improved nominal introduction rule that ensures termination in the presence of nominals, inverse roles, and number restrictions - a combination of DL constructs that has been proven notoriously difficult to handle. The implementation shows significant performance improvements over other existing reasoners on several well-known ontologies.

According to [40, 41, 42] in order to apply hypertableau to a $\mathcal{SHIQ}$ knowledge base, it needs to be preprocessed and transformed into DL clauses. A DL clause is

a universally quantified implication that contains DL concepts and roles as predicates. The main inference rule for DL clauses is hyperresolution. Hyperresolution resembles lazy unfolding: an atom from the right-hand side of a DL clause is derived only if all atoms from the left-hand side have already been derived. Left- and right-hand sides of the DL clause are separated by an implication sign. On Horn clauses this calculus is deterministic, thus, it eliminates all the or-branching. The algorithm can be viewed as a hybrid of resolution and tableau. It is also related to the hypertableau and hyperresolution calculuses.

Hyperresolution is able to resolve many first-order logic fragments. However, $\mathcal{SHIQ}$ allows us to express cyclic GCIs in the form of $C \sqsubseteq \exists R.C$ thus making hyperresolution generate infinite paths of successors. Therefore, to ensure termination the authors use the pairwise blocking technique from [33] to detect cyclic computations. To limit and-branching, the authors extend the blocking condition from [33] to anywhere pairwise blocking: an individual can be blocked by an individual that is not necessarily its ancestor. This significantly reduces the size of the constructed models. [50] is a more recent work dedicated to exploring potentials of hypertableau, however, in this work the authors explore only $\mathcal{SHI}$. In [10] the previously mentioned work has been further extended to $\mathcal{SHIQ}$ and later in [9] to $\mathcal{SRIQ}$, however, the way the authors deal with $\mathcal{Q}$ is different from the approach that will be proposed in this work.

In [51], the authors further discuss consequence-based reasoning. They postulate that this reasoning procedure can only be applied to ontologies expressed using Horn logic. The before-mentioned work describes a consequence-based procedure for the description logic $\mathcal{ALCH}$. The authors claim that this approach performs well on non-Horn ontologies.

In [11], the authors present a consequence-based calculus for description logic

$\mathcal{ALCHIQ}^+$ - a very expressive logic that also allows qualified number restrictions which is our topic of interest. As a proof of concept they implemented the calculus in a reasoner called Sequoia. The reasoner shows very promising results but it does not implement Integer Linear Programming to efficiently deal with qualified number restrictions. In [58], the authors further extend Sequoia by adding support for nominals thus being able to classify ontologies expressed in $\mathcal{SROIQ}$.

To summarize, the tableau algorithm is the most prominent reasoning algorithm that is widely used to determine satisfiability of knowledge bases and to implement reasoners. However, there are other techniques that can be used for the same purpose. One of them is based on automata theory [5]. The algorithm is based on a technique that translates a concept into an automaton that accepts all models for that concept. However, it appears that the tableau based algorithms are more suitable for reasoner implementation and optimization than automata-based algorithms, therefore the automaton-based reasoners have not found a wide adoption [5].

3.2 Optimization Techniques

In the previous section we discussed the most prominent reasoning algorithms in existence today. A reasoner represents an implementation of a calculus - a set of rules for a specific description logic. However, implementing the rules in a naive way will most likely result in poor performance. Most modern reasoners implement various optimization techniques. Many of the latter are implementation-dependent but some of them could be considered as guidelines for semantic reasoner implementation [60]. In this section we will look at some of the existing optimization techniques.

The tableau-based algorithms provide reasoning services to knowledge bases expressed as one of the possible subsets of the description logic $\mathcal{SROIQ}$. Despite the fact that there are other algorithms that could provide similar functionality, the tableau-based algorithms found wide recognition due to their simplicity of implementation [5]. However, these algorithms do not perform well on complex knowledge bases if appropriate optimization techniques have not been implemented. For instance, it is a known fact that there are two main sources of computational overhead - nondeterminism caused by presence of disjunctions in knowledge bases and construction of very large models caused by existential restrictions.

Many optimization techniques have emerged in order to address the above mentioned problems. Saturation is a technique that originally comes from the $\mathcal{EL}$ description logic language. It can be integrated in semantic reasoners for more expressive knowledge bases in order to addresses the problem of constructing large models. There are other optimization techniques that aim at mitigating the effects of applications of disjunctive clauses, however, no groundbreaking technique has been proposed yet.

Apart from the computational overhead caused by disjunctions and existential restrictions, there are also other DL constructs that can negatively affect reasoner performance. Modern reasoners are not able to effectively handle large numbers of qualified cardinality restrictions. This problem was explored in [20], [19], and [22]. The authors propose to translate qualified number restrictions into systems of linear inequalities and then solve these systems using a specialized Integer Linear Programming module. The proposed approach shows very promising results and it will be explained in detail in the following chapter as it serves as motivation for this work.

As it has already been mentioned earlier, a tableau-based algorithm builds a

completion tree by applying predefined completion rules. Then, the completion tree can be transformed into a model if it is satisfiable (i.e. it is clash-free). Despite the fact that the order of application of the completion rules does not affect the satisfiability/unsatisfiability of the completion tree, an imprudent application of the rules may result in building an infinite tree. In order to ensure that the algorithm could handle this situation and eventually terminate, an appropriate blocking technique should be implemented. The main idea of blocking [31] is to terminate the algorithm when the completion tree does not acquire new information by applying the completion rules. Most importantly, blocking should also ensure that the model reconstructed from the completion tree is sound and complete. Blocking too early will prevent the algorithm from inferring all possible information from a knowledge base.

There is a number of well-known and widely applied blocking techniques [8, 23, 40]. One of these techniques is known as subset blocking. This is an approach that allows an algorithm to block and thus prevent an existential restriction rule from creating infinitely many role successors, while at the same time ensuring that the resulting model is sound and complete. An individual x may be safely blocked by its direct role successor an individual y in an ABox $\mathcal{A}$ if $\{C \mid C(x) \in \mathcal{A}\} \subseteq \{C' \mid C'(y) \in \mathcal{A}\}$. The main underlying idea of the subset blocking technique is that the blocked individual can use the role successors of y instead of generating new successors. However, if the conditions of the subset blocking are to be violated by further application of the completion rules, the block can be safely broken and the node can be expanded. Subset blocking has been extended with anywhere blocking [13]. Still, in the presence of inverse roles this strategy cannot be applied. Instead, the pairwise blocking [33] should be chosen. The main idea of pairwise blocking is to look at pairs of nodes. A natural extension of this strategy is known as anywhere pairwise

blocking when the witness of the blocked node is not required to be its direct successor [40]. In [23], the authors introduce a new blocking mechanism that is called core blocking. This algorithm employs a very strict blocking condition that can suspend a model from being constructed much earlier than other existing blocking techniques. However, as the authors have mentioned, this is technique is very "aggressive" and if it is used alone then the completion tree might not be complete and therefore it might not be possible to transform it into a model. In order to ensure the validity of each block the algorithm incorporates an additional checking mechanism. When necessary the algorithm can release the block and continue expanding the completion tree. Thus, the checking mechanism ensures that the complete model will eventually be constructed. However, this blocking condition works only on knowledge bases expressed as Horn clauses.

There are also optimization techniques that address indeterminism in the tableau algorithm. One of these techniques is known as lazy unfolding. Since the main source of indeterminism are general inclusion axioms (GCIs) of the form of $C \sqsubseteq D$, it is possible to avoid transforming such axioms into $\neg C \sqcup D$ and unfold them *lazily* instead. By lazy unfolding it is simply meant that the axioms of this form will be applied only to those nodes that contain C in their label. Thus these nodes will also obtain D as their subsumer. A technique that facilitates lazy unfolding [6, 7, 39] is absorption. By applying absorption it becomes possible to rewrite GCIs of the knowledge base as $B \sqsubseteq C$ with B an atomic concept and C any concept expression. Then, during the reasoning process the algorithm derives C only if the node already contains B in its label. Absorption has been further extended to binary absorption, which rewrites a GCI to $B1 \sqcap B2 \sqsubseteq C$, and to role absorption that rewrites a GCI to $\exists R.\top \sqsubseteq C$ [55]. However, the axiom $\exists R.A \sqsubseteq A$ cannot be absorbed directly. To be absorbed, the axiom has to be rewritten as $A \sqsubseteq \forall R^{-}.A$. It is important to mention

that there is no clear technique to identify which combination of transformation and absorption techniques will yield the best results. Therefore, implemented absorption algorithms are guided primarily by heuristics.

In [57], the authors discuss the problem of a very high worst-case complexity that is present in modern reasoning systems such as FaCT++ [59], HermiT [49], or Pellet [52] that implement the tableau algorithm. The reason for that is the number of modelling constructs supported by the expressive DL $\mathcal{SROIQ}$. Therefore, it has been longstanding challenge in the DL community to propose novel optimization techniques to improve the performance of the tableau based reasoner. A very effective and widely implemented optimization technique is caching. The standard implementation of caching is satisfiability caching [17]. The main purpose of it is to cache the satisfiability status for a set of concepts. This information is stored in a so called cache and can be easily accessed during the reasoning process. If the set of concepts stored in the cache appears again in a completion tree then instead of applying tableau rules it is possible to obtain the necessary information from the cache. However, with increasing expressivity of the used DL naively caching might become unsound, for instance, due to the possible interaction of inverse roles with universal restrictions. The authors of [57] propose to improve the existing caching technique so that it would be possible to process the expressive DL $\mathcal{SROIQ}$. They develop the unsatisfiability caching method that is based on sophisticated dependency management that further enables better informed tableau backtracking and more efficient pruning. In the proposed extension the algorithm stores and maintains information about the sets of concepts that are known to be unsatisfiable. Based on that any superset of the cashed unsatisfiable concept sets would also be unsatisfiable. Thus, when expanding the completion tree, one encounters a node label that contains a superset of an unsatisfiable cache entry, then it is safe to stop expanding the branch

of the completion tree. This novel technique is integrated in the reasoning system Konclude. The results prove that the proposed approach improves the overall performance of the reasoner.

The unsatisfiability caching is closely associated with the dependency tracking optimization technique. Dependency directed backtracking is a technique that can effectively prune irrelevant alternatives of non-deterministic branching decisions caused by the presence of the disjunctive clauses in knowledge bases. If branching points are not involved in clashes it will not be necessary to compute other alternatives of these branching points because the other alternatives cannot eliminate the cause of the clash. To identify involved non-deterministic branching points, all facts in a completion graph are labelled with information about the branching points they depend on. Thus, the united information of all clashed facts can be used to identify involved branching points. A typical realization of dependency directed backtracking is backjumping [60, 1], where the dependent branching points are collected in the dependency sets for all facts. The dependency tracking stores all necessary information to exactly trace back the cause of the clash in a node making it possible to identify all involved non-deterministic branching points for the dependency directed backtracking and also to identify small unsatisfiable sets of concepts that can be used to create new entries in the unsatisfiability cache.

To conclude, in this section we have covered some of the existing optimization techniques that are commonly used in modern reasoner implementations. In this work we focus on leveraging Integer Linear Programming in order to address performance issues that arise from having entailments caused by presence of qualified cardinality restrictions in ontologies. We will discuss this approach in detail in the next chapter.

Chapter 4

Motivation and Research Objectives

4.1 Motivation

Before stating our research objectives we would like to present the potential benefits of applying linear optimization to description logic reasoners. We will use an example from [22] to demonstrate that even a simple problem expressed in number restrictions can significantly increase the complexity of the reasoning process.

The authors of [22] model a simple situation of a university student who has to take courses in different departments. For each completed course the student receives credits. Restrictions are imposed on the number of credits that the student is allowed to get for the courses taken from each department.

This problem is modelled as follows:

$$\forall hasCredit.(Science \sqcup Engineering \sqcup Business) \quad (4.1)$$

$$\geq 140\, hasCredit \quad (4.2)$$

$$\geq 120\, hasCredit.(Science \sqcup Engineering) \quad (4.3)$$

$$\leq 32\, hasCredit.(Science \sqcup Business) \quad (4.4)$$

$$\leq 91\, hasCredit.Engineering \quad (4.5)$$

It can be interpreted in a natural language as follows:

(1) Students must take courses only from Science, Engineering, or Business departments.

(2) Students must earn at least 140 credits to complete the study program.

(3) Students must earn at least 120 credits from the department of Science or from the department of Engineering.

(4) Students must earn at most 32 credits from the department of Science or from the department of Business.

(5) Students must earn at most 91 credits from the department of Engineering.

This simple problem is satisfiable. However, the way the standard tableau-based algorithm would solve it is rather inefficient. Most tableau-based algorithms [8, 30, 32] will test the satisfiability of a concept that is subsumed by the number restrictions by first satisfying all the *at-least* restrictions and then verifying whether none of the *at-most* restrictions has been violated. If one of the *at-most* restrictions has been violated, the algorithm will try to reduce the number of the created role fillers to satisfy the *at-most* restrictions.

In this case the algorithm will first create 260 *hasCredit* role successors so that 120 role successors will be instances of $Science \sqcup Engineering$ and 140 role successors will be the instances of $\top$. Then the nondeterministic *choose-rule* will assign to each of these 260 instances $Science \sqcup Business$ or $\neg(Science \sqcup Business)$ and $Engineering$ or $\neg Engineering$. In case an *at-most* restriction is violated, for example, if a student has more than 91 *hasCredit* role successors of $Engineering$, the nondeterministic *merge-rule* will try to reduce the number of these instances by merging pairs of non-disjoint instances until the upper bound specified by this *at-most* restriction is satisfied. The more there are interacting number restrictions and the higher are the numbers the more this approach becomes inefficient.

The authors of [22] propose a different way to solve this problem. Their approach is inspired by the methods for reasoning about sets described in [43]. Instead of creating role successors to satisfy the *at-least* number restrictions and then merging them to satisfy the *at-most* number restrictions they reduce the problem to a standard linear programming optimization problem. This approach requires certain preprocessing steps. The authors *un-qualify* the number restrictions by introducing a new role for each qualified *hasCredit* role. This transformation is necessary because the authors do not deal directly with qualified number restrictions. Instead they use universal restrictions to express them. Each newly introduced role must be a subrole of the original role resulting in the following new role hierarchy:

$$hasCredit_1 \sqsubseteq hasCredit$$

$$hasCredit_2 \sqsubseteq hasCredit$$

$$hasCredit_3 \sqsubseteq hasCredit$$

Then the transformed qualified number restrictions would look as follows:

$$\forall hasCredit.(Science \sqcup Engineering \sqcup Business) \tag{4.6}$$

$$\geq 140\, hasCredit.\top \tag{4.7}$$

$$\geq 120\, hasCredit_1 \sqcap \forall hasCredit_1.(Science \sqcup Engineering) \tag{4.8}$$

$$\begin{gathered} \leq 32\, hasCredit_2 \sqcap \forall hasCredit_2.(Science \sqcup Business) \\ \sqcap \forall (hasCredit \setminus hasCredit_2).\neg(Science \sqcup Business) \end{gathered} \tag{4.9}$$

$$\begin{gathered} \leq 91\, hasCredit_3 \sqcap \forall hasCredit_3 Engineering \\ \sqcap \forall (hasCredit \setminus hasCredit_3).\neg Engineering \end{gathered} \tag{4.10}$$

(4.8) is semantically equivalent to its qualified version $\geq 120\, hasCredit.(Science \sqcup Engineering)$ because the role fillers of $hasCredit_1$ in its un-qualified version must be $Science \sqcup Engineering$ due to the presence of the universal restriction [37].

(4.9) is semantically equivalent to $\leq 32\, hasCredit.(Science \sqcup Business)$ because in the un-qualified version the authors state that only $hasCredit_2$ role fillers can be $Science \sqcup Business$ [37].

Then new variables have to be introduced to represent the newly created roles. Let us assume that S stands for *Science*, B stands for *Business*, and E stands for *Engineering*. Thus,

hC_1 represents $hasCredit_1$ role fillers

hC_2 represents $hasCredit_2$ role fillers

hC_3 represents $hasCredit_3$ role fillers

hC_1hC_2 represents $hasCredit_1$ and $hasCredit_2$ role fillers

hC_1hC_3 represents $hasCredit_1$ and $hasCredit_3$ role fillers

hC_2hC_3 represents $hasCredit_2$ and $hasCredit_3$ role fillers

$hC_1hC_2hC_3$ represents $hasCredit_1$, $hasCredit_2$, and $hasCredit_3$ role fillers

Finally, the corresponding linear programming problem can be defined:

minimize

$$hC_1 + hC_2 + hC_3 + hC_1hC_2 + hC_1hC_3 + hC_2hC_3 + hC_1hC_2hC_3$$

subject to:

$$\begin{aligned} hC_1 + hC_2 + hC_3 + hC_1hC_2 + hC_1hC_3 + hC_2hC_3 + hC_1hC_2hC_3 &\geq 140 \\ hC_1 + hC_1hC_2 + hC_1hC_3 + hC_1hC_2hC_3 &\geq 120 \\ hC_2 + hC_1hC_2 + hC_2hC_3 + hC_1hC_2hC_3 &\leq 32 \\ hC_3 + hC_1hC_3 + hC_2hC_3 + hC_1hC_2hC_3 &\leq 91 \end{aligned}$$

and

$$hC_1 \geq 0$$
$$hC_2 \geq 0$$
$$hC_3 \geq 0$$
$$hC_1hC_2 \geq 0$$
$$hC_1hC_3 \geq 0$$
$$hC_2hC_3 \geq 0$$
$$hC_1hC_2hC_3 \geq 0$$

This system of linear inequalities will be passed to the linear program solver that will attempt to compute a non-negative integer solution. If there is no integer solution then a numeric clash will be produced. Otherwise the solver will provide us with the values of the variables, i.e., the number of role fillers. This approach is notably simpler and provides a better performance than the *choose-rule* of the standard tableau algorithm. However, it also has its limitations. Namely the number of generated variables might potentially be too large and as a result it might cause reasoners to run out of memory. This is one of the problems that we plan to address in our research.

4.2 Research Objectives

A significant amount of research has been done in Description Logic over the past two decades, however, we believe there is always room for pushing the boundaries of the state of the art in the field. We evaluated several potential ways to contribute

to this area and we decided to follow the path of [22] and [19] and focus on semantic reasoner optimization by means of efficiently classifying ontologies that contain large numbers of qualified number restrictions. We will limit this work to the description logic $\mathcal{ALCHQ}$.

Therefore our research objectives are the following:

- to design a novel calculus for the description logic $\mathcal{ALCHQ}$
- to apply the Branch and Price algorithm in order to efficiently reason on ontologies with entailments caused by qualified number restrictions
- to avoid backtracking that is used in Tableau-based reasoners and process disjunctions directly by means of resolution techniques
- to avoid creation of very large models as it is done in Tableau-based reasoners by using the saturation-based approach
- to implement a reasoner that will serve as proof of concept for the proposed calculus

Chapter 5

Description Logic Reasoner Avalanche

In this chapter we will present the Description Logic reasoner Avalanche. First we will give an overview of our work. After that we talk about the saturation graph that we use to store information discovered during the reasoning process. Then we introduce the normalization process that is a mandatory step that has to be taken to make an ontology compatible with our rules. Consequently, we will present the calculus and explain how our rules work. Finally, we will demonstrate how Avalanche reduces qualified number restrictions to linear programming problems and solves them with the help of the IBM CPLEX software package.

5.1 Saturation-based reasoner Avalanche

When we started working on Avalanche we set as one of our goals to design a novel calculus for the description logic $\mathcal{ALCHQ}$ that would employ the saturation-based approach. The reason why we chose to go this way is because of the potential advantages that come with it. We carefully studied related works in the area of DL to see what are the main challenges that we could address in our research. As it has been mentioned earlier in [42], when dealing with completion rules there are two

main sources of complexity - indeterminism caused by the presence of disjunctive clauses and creation of extremely large models caused by the presence of existential restrictions. The advantage of a saturation-based approach is that the size of the model produced by the algorithm is notably smaller than the size of the model that would be produced by a standard tableau algorithm that implements the completion rules. That is why developing a saturation-based calculus and implementing it in a reasoner appeared to be an interesting idea for our research.

Saturation was initially applied to reason on the description logic $\mathcal{EL}+$. One of the main features of saturation is that it allows to minimize the number of nodes in the completion graph by reusing existing nodes whenever it is possible. This approach is used in the implementation of the reasoner Konclude [56]. However, the authors of Konclude acknowledge the fact that the reasoner still has to rely on the original tableau rules to deal with the DL $\mathcal{SROIQ}$ and uses absorption only to optimize certain parts of the reasoning procedure.

We decided to start with expanding the existing calculus for the DL $\mathcal{EL}$ by adding more rules to cover the DL $\mathcal{ALCHQ}$ fragment. Most of our rules unfold axioms extracting explicitly stated subsumers and storing them in node labels of the constructed graph. The rest of the rules accumulate information or saturate labels until new subsumers can be inferred. Further, there is another fundamental difference between the approach described in [56] and the one presented in our work: Konclude does not make use of linear programming to reason on qualified number restrictions as opposed us which we consider the major strength of our work.

5.2 Saturation Graph

In this section we will present the saturation graph that is constructed by Avalanche during the reasoning process.

The saturation graph G when saturated represents a fully classified ontology where each node contains all its subsumers in its label. In the following subsections we will discuss different types of nodes in the graph and the purpose they serve.

5.2.1 Saturation Nodes

Each node in the saturation graph is uniquely identified by its representative concept. The representative concept can be a non-negated atomic concept extracted from an input ontology, an auxiliary concept created during the normalization process, an anonymous concept, or a concept that we generate to identify unfold and clone nodes. Each node has a label that contains its entailed subsumers. Some nodes also have labels with possible subsumers that are used when reasoning with qualified number restrictions. Each node will be discussed in detail further in this section.

When we start constructing the saturation graph we first create the node with the representative concept $\top$. $\top$ is a special concept that subsumes every other concept in the ontology. It can be considered as a root node of our graph. Another special concept is $\bot$. We do not have a dedicated node for this concept but we use it to mark nodes as unsatisfiable by adding $\bot$ to the subsumers label.

We never delete nodes but the nodes marked as unsatisfiable are excluded from the reasoning process. This is needed to avoid creating duplicate nodes. If we know that a node has already been proven to be unsatisfiable then we will reuse this knowledge during the reasoning process.

A node can have incoming and outgoing edges except for cloned nodes and unfold nodes that can have only outgoing edges. The edges are directed. They originate in the source node and end in the target node.

It is important to mention that Avalanche implements all the rules that will be presented later in this chapter but in some cases implementation differs from its theoretical representation. In the calculus we present two types of nodes - static nodes and anonymous nodes. In the implementation we also introduced auxiliary nodes, unfold nodes, and two types of *cloned nodes* - a clone to test for positive subsumption between two concepts and a clone to test for negative subsumption or disjointness between two concepts. This distinction is needed to facilitate implementation of the calculus and introduce certain optimizations.

5.2.2 Static Node

Static node is a super class of the identified node and of the auxiliary node. In the calculus we have no distinction between identified and auxiliary nodes but we have this distinction in the implementation. In the implementation it is more convenient to have two nodes as sometimes for the purpose of optimization we do not need to apply certain rules to the auxiliary nodes. For example, we do not need to know whether $A \sqsubseteq aux_1$. The concept aux_1 was not defined in the original ontology as it was introduced during the normalization process. Thus, testing whether there is subsumption between the two concepts would not add any valuable information that we could benefit from but it would increase the total reasoning time needed to classify an intology.

5.2.3 Identified Node

Both in the calculus and in the implementation identified nodes represent named concepts defined in the input ontology.

For example, if we ask Avalanche to classify a small ontology that does not need to be normalized and that contains the following axioms:

$$A \sqsubseteq B$$

$$A \sqsubseteq \geq 1\, R.C$$

$$C \sqcap D \sqsubseteq F$$

Avalanche will build a graph with five nodes with the following representative concepts: A, B, C, F, and $\top$. As a classification result the label of the node with the representative concept A will contain its subsumers: B and $\geq 1\, R.C$.

5.2.4 Auxiliary node

Auxiliary nodes possess the same properties as the identified nodes. They are represented by internal concepts that have been generated for an input ontology during the normalization phase. Avalanche is designed to work only with atomic concepts and axioms that conform to the DL $\mathcal{ALCHQ}$. Ontologies that contain language constructs outside $\mathcal{ALCHQ}$ will be rejected by the reasoner. The axioms that are composed of allowed language constructs but that contain non-atomic concepts must be normalized. Only after that saturation and completion rules can be applied.

In order to design the normalization rules we followed the approach presented in the technical report [3] for the DL $\mathcal{EL}$ and extended it further for $\mathcal{ALCHQ}$. The normalization rules will be presented further in this chapter.

We will demonstrate on an example how Avalanche normalizes axioms and introduces auxiliary concepts. Let us assume that in the input ontology we encounter the following axiom:

$$\exists R.C \sqcap D \sqsubseteq E$$

This is a valid $\mathcal{ALCHQ}$ axiom but it has to be normalized because the range of values of R is not expressed as an atomic concept. The result of the normalization process will look as follows:

$$C \sqcap D \sqsubseteq aux1$$
$$\exists R.aux1 \sqsubseteq E$$

As a result we introduced a new auxiliary concept aux_1.

5.2.5 Anonymous Node

Anonymous nodes are represented in the calculus and in the implementation. We need these nodes to create role successors and to discover unsatisfiability (or satisfiability) of concepts due to the presence of qualified number restrictions among their subsumers.

For example, let us take a node with the representative concept A that has subsumers $\geq 5R.C, \leq 2R.D, \geq 4R.E$. This example is trivially satisfiable at the moment. However, we do not know whether it will remain satisfiable. If we later discover that $C \sqsubseteq D$ then the concept A will become unsatisfiable. This is the reason why we need anonymous nodes.

Presence of an at least one *at-least* qualified number restriction in a set of node subsumers triggers construction of an edge to another node. If there are only *at-least* qualified number restrictions then we do not need to create anonymous nodes. In such a case we will connect the subsumer node with the nodes that are represented by the cardinalities of the qualified number restrictions. Furthermore, if we only have *at-most* qualified number restrictions we also do not need to create anonymous nodes. In fact in such a case nothing needs to be done. However, if we have both *at-most* and *at-least* qualified number restrictions we will need to call the ILP module that we also call QMediator in order to create a role successor. The role successor most likely will be an anonymous node. However, it is possible for the role successor to be an identified or an auxiliary node.

Going back to the example above, Avalanche will construct a new anonymous node and then once a new subsumption has been discovered the anonymous node will become unsatisfiable. As a result the node with the representative concept A will also become unsatisfiable.

5.2.6 Clone Node

If we want to test for subsumption or disjointness between two concepts we need to create a respective clone node. Such a node will carry relevant information from both the *subsumee* and the *subsumer* nodes. This information includes the qualified number restrictions of the subsumee node and the matching possible subsumers of the subsumer node. These nodes allow us to discover subsumptions caused by the presence qualified number restrictions in the ontology. If such a clone becomes unsatisfiable we know that there is a subsumption between two concepts.

We distinguish two types of clone nodes - a positive clone node or a subsumption clone and a negative clone node or a disjointness clone. If a positive clone fails the

subsumee concept will be subsumed by the subsumer concept and if a negative clone fails then the subsumee will be subsumed by the negation of the subsumer concept.

In general, we always compare pairs of concepts. In the worst case we would have to test all pairs of nodes in the ontology for subsumption and disjointness. We introduced a number of optimizations in order to avoid creating redundant nodes and also to avoid triggering unnecessary rule applications. These optimizations will be discussed later in detail. The general idea is to create only those clones that have higher chances of resulting in a subsumption or disjointness because not every clone is guaranteed to result in a subsumption.

5.2.7 Subsumption Clone

We create a subsumption clone to test for subsumption between two concepts. If the subsumption clone fails then we can conclude that the corresponding subsumption holds. As a result a new subsumer will be added to the subsumee node.

5.2.8 Disjointness Clone

We create a disjointness clone to test for disjointness between two concepts. If the disjointness clone fails then we can conclude that the two concepts are disjoint. As a result Avalanche will add a new negated subsumer to the subsumee node.

5.2.9 Unfold Node

Unfold or disjunction nodes had to be introduced to facilitate reasoning with disjunctions that unfold to qualified cardinality restrictions. They will be later discussed in the context of the corresponding rules. Briefly, sometimes we cannot immediately know if two concepts that are present in different disjunctions are disjoint due to being subsumed by qualified number restrictions. Then the unfold nodes will be created that will accumulate information until possibly the disjointness can be discovered.

5.2.10 Connecting Edge

An edge represents a connection between two nodes. All edges in the graph are directed. For example, if a node with the representative concept A is subsumed by $\exists R.C$ then Avalanche will create an edge between the node A and the node with the representative concept C. The node A will be the source node and the node C will be the target node. Further, the edges also connect source nodes to target nodes as a result of a call to the ILP module.

5.3 Normalization Process

5.3.1 Normal Form

Avalanche is able to classify $\mathcal{ALCHQ}$ ontologies of any size. However, in order to apply the saturation-based rules input ontologies have to be transformed into the normal form presented below. Here and further by the letters A and B we denote

atomic concepts, by the letters C, D... we denote atomic concepts and qualified number restrictions, and by any letter with a hat on top we denote complex concepts.

NF1 $A \sqsubseteq B$

NF2 $A_1 \sqcap A_2 \sqsubseteq B$

NF3 $A \sqsubseteq \bowtie n\, R.B$[1]

NF4 $\bowtie n\, R.A \sqsubseteq B$

NF5 $A \sqsubseteq B_1 \sqcup B_2$

NF6 $\top \sqsubseteq B_1 \sqcup ... \sqcup B_n$, where $2 \leq n \leq 3$

NF7 $\top \sqsubseteq \bowtie nR.B$

NF8 $A_1 \sqcap ... \sqcap A_n \sqsubseteq \bot$, where $2 \leq n \leq 3$

Before we can begin the normalization process we need to verify the input ontology and transform some axioms in order to prepare it for the normalization. We start with verifying that the ontology does not contain language constructs that are outside of the $\mathcal{ALCHQ}$ domain. If there is at least one axiom that is outside of $\mathcal{ALCHQ}$ the entire process will be stopped and an error message will be generated. If the ontology passes the verification step then we will continue with the transformations.

First, the input ontology must be converted into Negation Normal Form (NNF). Second, existential restrictions must be transformed into their corresponding *at-least* qualified number restrictions. Third, universal restrictions must be transformed into

[1] $\bowtie$ denotes $\geq$ or $\leq$

their corresponding *at-most* qualified number restrictions. Fourth, equivalence axioms must be transformed into pairs of subsumption axioms. Fifth, OWL declarations compatible with $\mathcal{ALCHQ}$ must be transformed into equivalent axioms: disjointness of concepts, and domain and range declarations. Finally, *exactly* number restrictions will be transformed into their equivalent qualified number restrictions. The *exactly* number restrictions that appear on the right-hand side of axioms will be transformed into pairs of *at-least* and *at-most* cardinality restrictions. For example, $B \sqsubseteq = n\, R.\hat{C}$ will be transformed into $\{B \sqsubseteq \geq n\, R.\hat{C},\ B \sqsubseteq \leq n\, R.\hat{C}\}$. The *exactly* cardinality restrictions that appear on the left-hand side of axioms will be transformed into pairs of *at-least* and *at-most* cardinality restrictions. For example, $= n\, R.\hat{C} \sqsubseteq \hat{B}$ will be transformed into $\geq n\, R.\hat{C} \sqcap \leq n\, R.\hat{C} \sqsubseteq \hat{B}$.

5.3.2 Left-hand Side Normalization Rules

At this stage we normalize only the left-hand sides of axioms. This process will terminate when no axiom can be transformed by any of the left-hand side normalization rules. During this process we introduce new *auxiliary* concepts.

The rules are described below and should be applied in the order they are presented :

NR1$_{lhs}$ $\geq n\, R.\hat{C} \sqsubseteq \hat{D}$ will be transformed to $\{\hat{C} \sqsubseteq aux_1, \geq n\, R.aux_1 \sqsubseteq \hat{D}\}$

According to our normal form the qualification of a cardinality restriction should be an atomic concept. For example, $\geq 1 R.(C \sqcap D) \sqsubseteq E$ will be transformed to $C \sqcap D \sqsubseteq aux_1, \geq 1\, R.aux_1 \sqsubseteq E$.

NR2$_{lhs}$ $\leq n\, R.\hat{C} \sqsubseteq \hat{D}$ will be transformed to $\{\hat{C} \sqsubseteq aux_1, \leq n\, R.aux_1 \sqsubseteq \hat{D}\}$

This rule applies the same strategy as rule **NR1**$_{lhs}$.

NR3$_{lhs}$ $\neg A \sqsubseteq \hat{C}$ will be transformed to $\{\top \sqsubseteq A \sqcup \hat{C}\}$

Negated concepts are not part of the normal form and should be normalized.

NR4$_{lhs}$ $A \sqcap \neg B \sqsubseteq \hat{C}$ will be transformed to $\{A \sqsubseteq B \sqcup \hat{C}\}$

This rule moves a negated concept from the left-hand side of an axiom to its right-hand side.

NR5$_{lhs}$ $\neg A \sqcap \neg B \sqsubseteq \hat{C}$ will be transformed to $\{\top \sqsubseteq A \sqcup B \sqcup \hat{C}\}$

The rule creates a global axiom. As a result every concept in the ontology will be subsumed by the new disjunction $\{A \sqcup B \sqcup C\}$.

NR6$_{lhs}$ $\geq nR.(\neg A) \sqsubseteq \hat{C}$ will be transformed to $\{\geq nR.aux_1 \sqsubseteq \hat{C}, \top \sqsubseteq A \sqcup aux_1\}$

Negated concepts that are role fillers of cardinality restrictions should also be normalized to conform to the normal form.

NR7$_{lhs}$ $\leq nR.(\neg A) \sqsubseteq \hat{C}$ will be transformed to $\{\leq nR.aux_1 \sqsubseteq \hat{C}, A \sqcap aux_1 \sqsubseteq \bot\}$

As a result of this transformation we introduce a new axiom that encodes that aux_1 is disjoint to A.

NR8$_{lhs}$ $\hat{A} \sqcup \hat{B} \sqsubseteq \hat{C}$ will be transformed to $\{\hat{A} \sqsubseteq \hat{C}, \hat{B} \sqsubseteq \hat{C}\}$

Disjunctions on the left-hand side of axioms should be normalized.

NR9$_{lhs}$ $\hat{C} \sqcap \hat{D} \sqsubseteq \hat{E}$ will be transformed to $\{\hat{C} \sqcap aux_1, \hat{D} \sqsubseteq aux_2,\ aux_1 \sqcap aux_2 \sqsubseteq \hat{E}\}$

A conjunction on the left-hand side of an axiom should be normalized.

5.3.3 Right-hand Side Normalization Rules

At this stage we normalize only the right-hand sides of axioms as the left-hand sides have already been normalized by the left-hand side normalization rules. This process will terminate when no axiom can be transformed by any of the right-hand side rule application. *Auxiliary* concepts will also be introduced during this process.

The rules are described below and should be applied in the order they are presented :

NR1$_{rhs}$ $\hat{C} \sqsubseteq \hat{D}$ will be transformed to $\{\hat{C} \sqsubseteq aux_1, aux_1 \sqsubseteq \hat{D}\}$

The right-hand side of any axiom should always be either an atomic concept or a disjunction or a cardinality restriction with an atomic concept as its qualification. For example, $C \sqsubseteq \leq 1\, R.(C \sqcap D)$ will be transformed into $C \sqsubseteq aux_1$ and $aux_1 \sqsubseteq \leq 1\, R.(C \sqcap D)$. The latter axiom will be eventually normalized by a corresponding rule.

NR2$_{rhs}$ $B \sqsubseteq \geq n\, R.\hat{C}$ will be transformed to $\{B \sqsubseteq \geq n\, R.aux_1, aux_1 \sqsubseteq \hat{C}\}$

The role filler of a restriction should always be an atomic concept.

For example, $A \sqsubseteq \geq 1R.(C \sqcap D)$ will be transformed to $aux_1 \sqsubseteq C \sqcap D$, and $A \sqsubseteq \geq 1\, R.aux_1$.

NR3$_{rhs}$ $B \sqsubseteq \leq n\, R.\hat{C}$ will be transformed to $\{B \sqsubseteq \leq n\, R.aux_1, \hat{C} \sqsubseteq aux_1\}$

This rule is applies the same strategy as the rule **NR2**$_{rhs}$.

NR4$_{rhs}$ $B \sqsubseteq C \sqcap D$ will be transformed to $\{B \sqsubseteq C, B \sqsubseteq D\}$

Conjunctions are not allowed on the right-hand side of an axiom.

NR5$_{rhs}$ $B \sqsubseteq \neg A$ will be transformed to $\{A \sqcap B \sqsubseteq \bot\}$

This is an alternative way to encode disjointness of concepts that conforms to our normal form.

NR6$_{rhs}$ $A \sqsubseteq B \sqcup \neg C$ will be transformed to $\{A \sqcap C \sqsubseteq B\}$

Negated concepts are not allowed by our normal form.

NR7$_{rhs}$ $A \sqsubseteq \neg B \sqcup \neg C$ will be transformed to $\{A \sqcap B \sqcap C \sqsubseteq \bot\}$

The result of this transformation is a new conjunction of disjoint concepts.

NR8$_{rhs}$ $A \sqsubseteq \geq nR.(\neg B)$ will be transformed to $\{A \sqsubseteq \geq nR.aux_1, B \sqcap aux_1 \sqsubseteq \bot\}$

Rule fillers of number restrictions cannot be negated concepts.

NR9$_{rhs}$ $A \sqsubseteq \leq nR.(\neg B)$ will be transformed to $\{A \sqsubseteq \leq nR.aux_1, \top \sqsubseteq B \sqcup aux_1\}$

Rule fillers of number restrictions cannot be negated concepts.

NR10$_{rhs}$ $A \sqsubseteq B \sqcup \hat{C}$ will be transformed to $\{A \sqsubseteq B \sqcup aux_1, aux_1 \sqsubseteq \hat{C}\}$

Only disjunctions of atomic concepts are allowed on the right-hand side of axioms.

NR11$_{rhs}$ $\bowtie n\, R.A \sqsubseteq \bot$ will be transformed to $\{\top \sqsubseteq \dot{\neg} \bowtie n\, R.A\}$

This rule transforms axioms where a cardinality restriction is subsumed by $\bot$.

NR12$_{rhs}$ $\bowtie n\, R.A \sqsubseteq \neg\hat{C}$ will be transformed to $\{\hat{C} \sqsubseteq \dot{\neg} \bowtie n\, R.A\}$

Negation is not allowed by our normal form.

5.3.4 Example of Normalization Rules Application

We created a small ontology to demonstrate how some of the normalization rules work.

$F \sqsubseteq G_1 \sqcap G_2 \sqcap G_3 \sqcap G_4 \sqcap G_5$

$E \sqsubseteq \geq 3R.(H \sqcap F)$

$E \sqsubseteq \exists R.A \sqcup \exists R.B \sqcup \exists R.C \sqcup \exists R.D$

$G1 \sqcup G2 \sqcup G3 \sqcup G4 \sqcup G5 \sqsubseteq H$

$A \sqcap B \sqcap C \sqsubseteq F$

During the preprocessing step $E \sqsubseteq \exists R.A \sqcup \exists R.B \sqcup \exists R.C \sqcup \exists R.D$ will be transformed into $E \sqsubseteq \geq 1R.A \sqcup \geq 1R.B \sqcup \geq 1R.C \sqcup \geq 1R.D$.

After that the algorithm will start with the normalization of the left-hand sides of the axioms: $G1 \sqcup G2 \sqcup G3 \sqcup G4 \sqcup G5 \sqsubseteq H$ will be transformed into $G_1 \sqsubseteq H, G_2 \sqsubseteq H, G_3 \sqsubseteq H, G_4 \sqsubseteq H, G_5 \sqsubseteq H$ by the rule $\mathbf{NR8}_{lhs}$.

Then, $A \sqcap B \sqcap C \sqsubseteq F$ will be transformed into $A \sqcap aux_1 \sqsubseteq F, B \sqcap C \sqsubseteq aux_1$ by applying the rule $\mathbf{NR9}_{lhs}$.

After that the algorithm will proceed with the normalization of the right-hand sides of the axioms. $F \sqsubseteq G_1 \sqcap G_2 \sqcap G_3 \sqcap G_4 \sqcap G_5$ will be normalized into $F \sqsubseteq G_1, F \sqsubseteq G_2, F \sqsubseteq G_3, F \sqsubseteq G_4, F \sqsubseteq G_5$ by the $\mathbf{NR4}_{rhs}$ rule.

Then $E \sqsubseteq\leq 3R.(H \sqcap F)$ will be normalized into $E \sqsubseteq\geq 3R.aux_2, aux_2 \sqsubseteq H, aux_2 \sqsubseteq F$ by the $\mathbf{NR3}_{rhs}$ rule.

Finally, $E \sqsubseteq \exists R.A \sqcup \exists R.B \sqcup \exists R.C \sqcup \exists R.D$ will be normalized into $E \sqsubseteq aux_3 \sqcup aux_4 \sqcup aux_5 \sqcup aux_6, aux_3 \sqsubseteq\geq 1R.D, aux_4 \sqsubseteq\geq 1R.C, aux_5 \sqsubseteq\geq 1R.B, aux_6 \sqsubseteq\geq 1R.A$ by the rule $\mathbf{NR10}_{rhs}$.

The resulting normalized ontology will look as follows:

$F \sqsubseteq G_1$

$F \sqsubseteq G_2$

$F \sqsubseteq G_3$

$F \sqsubseteq G_4$

$F \sqsubseteq G_5$

$G_1 \sqsubseteq H$

$G_2 \sqsubseteq H$

$G_3 \sqsubseteq H$

$G_4 \sqsubseteq H$

$G_5 \sqsubseteq H$

$A \sqcap aux_1 \sqsubseteq F$

$B \sqcap C \sqsubseteq aux_1$

$aux_2 \sqsubseteq H$

$aux_2 \sqsubseteq F$

$aux_3 \sqsubseteq\geq 1R.D$

$aux_4 \sqsubseteq\geq 1R.C$

$aux_5 \sqsubseteq\geq 1R.B$

$aux_6 \sqsubseteq\geq 1R.A$

$E \sqsubseteq \geq 3R.aux_2$ $E \sqsubseteq aux_3 \sqcup aux_4 \sqcup aux_5 \sqcup aux_6$

5.4 Calculus Presentation

In this section we will present the saturation-based calculus for the description logic $\mathcal{ALCHQ}$. The calculus consists of saturation and completion rules. The process of the application of these rules is called *reasoning*. This process is divided into two phases. First, during the unfolding phase we accumulate information by processing ontology axioms and extract information that is explicitly stated in the ontology. In this phase we apply each unfolding rule to each node in the graph only once. In the second phase that we call the saturation phase we apply the saturation-based rules until no new information can be added to the saturation graph. The result of this phase is a saturated graph where each node represents a concept of the input ontology containing all its entailed subsumers. It is important to mention that the second phase is divided into subphases as we try to minimize the application of the expensive rules.

5.4.1 Notation

Before presenting the semantics of the rules it is worth explaining our notation since some the symbols were introduced specifically for our calculus.

a) By A, B will denote atomic concepts and by C, D any concepts.

b) By ϕ we denote any subsumer that is allowed by $\mathcal{ALCHQ}$, except negated concepts, that is either a named concept, a qualified number restriction or a disjunction.

c) By $L(v)$ we denote a label of a node. By $L(v_A)$ we denote a label of a node with the representative concept A. The node label represents an intersection of all the subsumers $\bigsqcap_{i=1}^{n} C_i$ of the representative concept of the node. Further, $Q(v)$ is a sub-label of $L(v)$ that contains only qualified number restrictions of $L(v)$.

d) By $\dot{\neg}\phi$ we denote that the negated concept that we are about to add as a new subsumer must be in the negation normal form (NNF). This means that we have negated the concept before adding it as a new subsumer.

e) By τ we denote any concept including a negated one.

f) By $\langle r, q, n \rangle$ we denote a tuple returned by the ILP module, where r is a role, q is a qualification and n is a cardinality.

g) By q we denote an inequality.

h) By $\sigma(v)$ a function that extracts a cardinality, a role, and a role filler from an inequality that is a subsumer of the node v.

i) By V we denote all nodes in a graph.

j) By $\#v_q$ we denote a cardinality of a node v.

k) By $infeasible$ we mean that inequalities are infeasible and cannot be solved.

l) By $\bowtie n\, R.A$ we denote either $\leq nR.A$ or $\geq nR.A$.

m) By *is_new* we denote a boolean function that checks whether an inequality exists in a label.

n) By *add_to* we denote a function that adds an inequality to the label of possible subsumers of a given node.

o) By $L_P(v_B)$ we denote the label of possible subsumers of a node with the representative concept B.

p) By $L_P^{\neg}(v_B)$ we denote the label of non-possible subsumers of a node with the representative concept B.

q) By φ we denote a function that adds a new element to an existing set of preconditions.

r) By Q we denote qualified number restrictions.

s) By $clone(v, B)$ we denote a clone node that is used to test whether node v is subsumed by B.

t) By $\mathsf{C}_{\mathcal{T}}^{\mathsf{Q}}$ we denote a set of all qualified number restrictions and their negation that are present in the TBox.

u) By P we denote possible subsumers.

5.4.2 Avalanche Saturation-based Rules

We present the rules below. Their application strategies will be explained in the following subsection.

$R_{\sqsubseteq}$ **if** $A \sqsubseteq \phi \in \mathcal{T}, \phi \notin L(v_A)$ **then** add ϕ to $L(v_A)$

This rule adds the direct subsumers of a concept to a node.

For example, if we have an axiom $A \sqsubseteq B$ or $A \sqsubseteq \leq 1\, R.B$, or $A \sqsubseteq \geq 1\, R.B$, then we need to find a node with the representative concept A and if the corresponding subsumer is not yet present in the label of node A we will add it. Here, the subsumer is the right-hand side of the axiom and the subsumee is the left-hand side of the axiom.

$R_{\sqsubseteq\neq}$ **if** $A \in L(v), A \sqcap B \sqsubseteq \bot, \neg B \notin L(v)$ **then** add $\neg B$ to $L(v)$

This rule adds disjoint concepts of a given concept to a node.

For example, if we have an axiom $A \sqcap B \sqsubseteq \bot$ then all nodes that are subsumed by A will be also subsumed by $\neg B$.

$R_{\sqsubseteq *}$ **if** $A \in L(v), \tau \in L(v_A), \tau \notin L(v)$ **then** add τ to $L(v)$

This rule propagates subsumers. For each non-negated subsumer of a node, we look for the node where the representative concept is the non-negated subsumer. Then, we take all subsumers of the node and add them to the first node.

For example, if a node with the representative concept A is subsumed by B, then we will look for the node with the representative concept B. We will collect all its subsumers and add them to the subsumers of the node with representative concept A. For instance, the node B is subsumed by $\geq 1\, R.C$ among other subsumers and as a result of application of this rule the node A will also be subsumed by $\geq 1\, R.C$.

$R_{P_{\bowtie}}$ **if** $\bowtie n\, R.A \sqsubseteq B \in \mathcal{T}$, is_new$(\dot{\neg}(\bowtie n\, R.A), L_P(v_B))$ **then** add_to$(\dot{\neg}(\bowtie n\, R.A), L_P(v_B))$

In order to discover subsumption between concepts, we have introduced *possible subsumers*. Possible subsumers will be collected for each concept and stored in its node in a dedicated label. We collect all axioms where the left-hand side is a qualified number restriction and the right-hand side is an atomic concept. Then we add the negation of the qualified number restriction as a possible subsumer to the node with the representative concept equal to the right-hand side of the axiom.

For example, if we have an axiom $\geq 1\, R.C \sqsubseteq D$, we locate the node with the representative concept D and add a possible subsumer $\leq 0\, R.C$ to its label (as $\leq 0\, R.C \equiv \neg\geq 1\, R.C$).

A possible subsumer is stored as a tuple that consists of a precondition and a set of possible subsumers. In this case, the tuple will look as follows: $< \top, \leq 0\, R.C >$.

If we have more axioms that contain a qualified number restriction on the left-hand side and D on the right-hand side, we will perform a subsumption test in oder to avoid adding tuples that are subsumed by existing tuples. For example, if the node with the representative concept D already contains a possible subsumer tuple $< \top, \leq 0\, R.E >$ and we want to add a new tuple $< \top, \{\leq 0\, R.C >, \leq 0\, R.E\} >$ then the subsumption test will reject this tuple because both preconditions and possible subsumers of the existing tuple are subsumed by the new tuple.

There are other optimizations that we apply when adding new tuples. If a new tuple contains a concept in its preconditions that is also present as a negated concept in the label of the node then this tuple will be discarded. Tuples that contain the representative concept of the node will also not be added.

$R_{P_{\bowtie_{Dynamic}}}$ **if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, B \in L(v)$ **then** $add_to(P, L_P(v_B))$

This rule dynamically computes possible subsumers for a node and it has a number of prerequisites that must be satisfied before it can be executed:

- Both concepts A_i and A_j are auxiliary concepts
- At least one of them is subsumed by a qualified number restriction
- There are at least two axioms that satisfy all of the conditions above
- The axioms in question are subsumed by the same atomic concept

The example below demonstrates how this rule works.

We have the following three axioms and at least one conjunct in each conjunction is subsumed by a qualified number restriction:

$aux_1 \sqcap aux_2 \sqsubseteq x$

$aux_3 \sqcap aux_4 \sqsubseteq x$

$aux_5 \sqcap aux_6 \sqsubseteq x$

These axioms will be immediately transformed into possible subsumers and added to the corresponding label of the node with the representative concept x. The tuples generated by this rule will not be merged with any of the existing tuples.

The algorithm will take all conjuncts and create a cartesian product from them. Then each set from the resulting cartesian product will represent a new possible subsumer tuple that will need to be transformed. Each element of each set will either be unfolded into a qualified cardinality restriction and added to the possible subsumers set of the tuple or added to the set of preconditions of the tuple.

$R_{\sqsubseteq\neg}$ **if** $\phi \sqsubseteq A \in \mathcal{T}, \neg A \in L(v), \dot{\neg}\phi \notin L(v)$ **then** add $\dot{\neg}\phi$ to $L(v)$

For each negated subsumer of each node, this rule first finds all axioms where the negated subsumer is the non-negated right-hand side. Then, it adds the negated left-hand side as a subsumer to the node in question.

For example, a node with the representative concept A is subsumed by a concept $\neg B$. There are two axioms in the ontology where B is the subsumer: $C \sqsubseteq B$ and $\geq 1\, R.D \sqsubseteq B$. As a result, the rule will add to the node with the representative concept A two negated subsumers: $\leq 0\, R.D$ and $\neg C$.

$R_{\sqsubseteq\sqcap}$ **if** $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{T}, \{A_1, A_2\} \subseteq L(v), B \notin L(v)$ **then** add B to $L(v)$

This rule retrieves all the axioms that correspond to the template: $A \sqcap B \sqsubseteq C$. Then, if there is a node subsumed by both A and B, then the rule will add a new subsumer C to this node.

$R_{\sqsubseteq\bar{\sqcap}}$ **if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, \{\neg B, A_i\} \subseteq L(v), \neg A_j \notin L(v)$ **then** add $\neg A_j$ to $L(v)$

The rule retrieves all the axioms that correspond to the template: $A \sqcap B \sqsubseteq C$. Then, if there is a node subsumed by $\neg C$ and one of the conjuncts, for instance A, then the rule will add a new subsumer this this node: the negation of the second conjunct, in this case $\neg B$.

R_{fil} **if** $\langle r, q, n \rangle \in \sigma(v), \neg\exists v_q \in V : q \subseteq L(v_q), \#v_q \geq n$ **then** create $v_q \in V_A$ with $L(v_q) \leftarrow q$ and $\#v_q \leftarrow n$

This rule creates new edges between nodes. We create a new edge if we have at least one *at-least* qualified number restriction among the subsumers of a given node. We submit all the qualified number restrictions to the ILP module that we also call *QMediator* and based on its response we either create a new anonymous node, or reuse an existing anonymous or identified node. If the edge already exists then it will be reused.

For example, if a node contains only *at-least* qualified number restrictions we will create edges connecting this node with other nodes that are identified by the qualifications of the qualified number restrictions in question. Thus, if a node with the representative concept A is subsumed by $\geq 1\, R.C$ and $\geq 1 S.D$ we create an edge between the node A and the nodes with the representative concepts C and D. Then, if a node with representative concept A contains $\geq 1\, R.C, \geq 5\, R.E, \leq 4\, S.F$ and $\leq 1 S.D$ we will submit it along with some additional information to the ILP module. If these inequalities are feasible then the ILP module will return one or more triples consisting of a name of an edge,

a set of concepts, and a cardinality. For each triple we will create an edge with a corresponding name directed to the destination node. If the set of concepts contains only one element G then we will construct an edge connecting the source node to the target node with representative concept G. If the set contains more than one element, we will create an anonymous node that will represent the intersection of the members of the set. If the anonymous node already exists we will reuse it by connecting it to the source node.

$R_\perp$ **if** $\perp \notin L(v) \wedge (\text{infeasible}(L(v)) \vee \{A, \neg A\} \subseteq L(v))$ **then** add $\perp$ to $L(v)$

This rule marks a node as unsatisfiable by adding $\perp$ to its subsumers. No rules should be applied to the unsatisfiable node.

For example, assume a node contains a concept and its negation such as A and $\neg A$ for example. This node is therefore unsatisfiable and $\perp$ will be added to the its subsumers. If a node contains inequalities that are infeasible, for example $\geq 1\, R.C$ and $\leq 0\, R.C$ then the node is also unsatisfiable and $\perp$ will be added to its subsumers.

R_P **if** $B \in L(v_A), L_P(v_A) \not\subseteq L_P(v_B)$ **then** $\text{add_to}(L_P(v_A), L_P(v_B))$

This rule propagates possible subsumers to the subsumer nodes.

For example, if a concept A is subsumed by a concept B then we will add all possible subsumers of the concept A to the node with representative concept B.

$R_{P_\sqcap}$ **if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, \varphi(A_i, L_P(v_{A_j})) \not\subseteq L_P(v_B)$ **then** $\text{add_to}(\varphi(A_i, L_P(v_{A_j})), L_P(v_B))$

This rule handles propagation of possible subsumers due to the presence of axioms that contain binary subsumptions on the left-hand side.

For example, consider the axiom $A \sqcap B \sqsubseteq C$. In this case, we take all tuples in the possible subsumers of A, add the concept B to the precondition of each tuple, and then add these tuples to the possible subsumers of the node with representative concept C. If possible, we merge the tuples with the existing possible subsumer tuples in C. After that, we perform the same procedure for the node B. We we take all tuples in the possible subsumers of B, add the concept A to the precondition of each, and then add these tuples to the possible subsumers of the node with representative concept C.

$R_{\sqsubseteq \sqcup}^{\neg}$ **if** $B \sqsubseteq \bigsqcup_{i=1}^{n} A_i \in \mathcal{T}$

if $\bigcup_{i=1}^{n} \{\neg A_i\} \subseteq L(v_A), \neg B \notin L(v_A)$

then add $\neg B$ to $L(v_A)$

elsif $\neg A_j \in L(v_A), j \in 1..n, Q \sqsubseteq B \in \mathcal{T}, Q \in \mathsf{C}_{\mathcal{T}}^{\mathsf{Q}}$

then create a fresh node X with $L(X) = \{X, \dot{\neg} Q\}$ and

add $X \sqcup \bigsqcup_{i=1}^{n} A_i$ to $L(v_A)$

This rule contrapositively adds new subsumers from the axioms that contain disjunctions on the right-hand side. If there is such an axiom where the left-hand side is an atomic concept and the right-hand side is a disjunction and if the node in question is subsumed by all the negated disjuncts from the disjunction then the rule will add the negation of the left-hand side to the subsumers of the node. However, if the node does not contain all the negated disjuncts the rule will create a special unfold node and subsequently add the disjunction to that node.

For example, if there is an axiom $B \sqsubseteq C \sqcup D$ and if there is a node A that contains $\neg C$ and $\neg D$ then the rule will add $\neg B$ to the subsumers of the node A.

Further, if the node A contains $\neg C$ but does not contain $\neg D$ then the rule will check if there is an axiom with a qualified number restriction on the left-hand side and B on the right-hand side. Assuming that we have the following axiom: $\geq 1R.C \sqsubseteq B$ the rule will create a new node X and add $\leq 0R.C$ to its subsumees. After that the rule will add $B \sqsubseteq C$ to the subsumers of the node A.

$R_{\sqsubseteq\sqcap}$ **if** $\bigsqcup_{i=1}^{n} A_i \in L(v), \bigcap_{i=1}^{n} L(A_i) \not\subseteq L(v)$
then add $\bigcap_{i=1}^{n} L(A_i)$ to $L(v)$

If a node is subsumed by a disjunction but not subsumed by any of its disjuncts then this rule will add common subsumers of the disjuncts to the node.

For example, $A \sqsubseteq B \sqcup C, B \sqsubseteq D, C \sqsubseteq D$. The rule will add D to the node A.

$R_{\sqcup}$ **if** $\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\} \subseteq L(v)$
$res \leftarrow$ resolvent$(\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\}, L(v))$
if $res \neq \emptyset$
then add res to $L(v)$

This rule resolves disjunctions in a node. It will collect all its disjunction subsumers that do not contain subsumers of the node. Then it will try to collect all those axioms where the disjunct is the right-hand side and a qualified number restriction is the left-hand side. Then, the rule will call the ILP module to solve inequalities and if the inequalities cannot be solved the corresponding disjuncts can be resolved. If there are no clashing inequalities at the moment the rule will create a dedicated unfold node. If this node later becomes unsatisfiable it will be possible to resolve the associated disjunctions.

We introduced some optimizations to this rule that allow us to improve the overall performance of the system. First, from each disjunction we remove disjuncts that are present as negated subsumers of the node. Then we add a new subsumer to node that is either a derived unit or a new smaller disjunction. Second, we try resolve two disjunctions by means of removing disjoint concepts from them with the goal to derive a unit or a new smaller disjunction. Finally, when resolving disjunctions with the help of CPLEX we do it in two steps: first, we attempt to resolve pairs of disjunction subsumers and then we resolve all disjunction subsumers of the node.

$R_{\sqsubseteq_P}$ **if** $\{B, \neg B\} \cap L(v_A) = \emptyset$ **then**

let $x \in \{B, \neg B\}$

if $v' \in clone(v_A, x)$ **then**

if $\bot \in L(v')$

then add x to $L(v_A)$ **else**

if $Q(v_A, x) \nsubseteq L(v')$

then add_to$(Q(v_A, x), L(v'))$

if $Q(v_A) \nsubseteq L(v')$

then add_to$(Q(v_A), L(v'))$

elsif $Q(v_A, x) \neq \emptyset$ **then**

create $v' \in V_C, clone(v_A, x) \leftarrow \{v'\}$

$L(v') \leftarrow \{\top\} \cup Q(v_A, x) \cup Q(v_A)$

This rule tests for subsumption or disjointness between two concepts.

If we want to test whether a node with representative concept A is subsumed by a concept B we will need to create a dedicated clone node. Another clone node will need to be crated if we want to tests whether A is subsumed by $\neg B$.

If this clone eventually becomes unsatisfiable then we can conclude that the subsumption or disjointness under test holds.

As a part of the test we first check that the node A has not already been subsumed by B or $\neg B$ to avoid unnecessary operations. If the condition has been satisfied we can proceed with the construction of the clone node. We first collect qualified number restrictions that are subsumers of the node with the representative concept A. Then we add them to the subsumers of the newly created clone. After that we add to the subsumers of the clone node inequalities from all possible subsumer tuples of the node with representative concept B. Finally, if the clone becomes unsatisfiable due to a clash in the inequalities then we will add B to the subsumers of the node A. This can happen immediately or later during the reasoning process. Unfortunately not every clones results in subsumption.

$R_{\bowtie\perp}$ **if** $A \sqsubseteq \bowtie nR.C \in \mathcal{T}, \bowtie mR.D \in L(v), infeasible(Q(v) \cup \{\bowtie nR.C\}), \neg A \notin L(v)$
then $L(v) \rightarrow L(v) \cup \{\neg A\}$

The rule adds new subsumers based on a clash in inequalities. For each node subsumed by inequalities we can test whether it is subsumed by a concept $\neg A$ if we have an axiom of a form $A \sqsubseteq \bowtie nR.C$ and if there is a clash in inequalities in the node and $\bowtie nR.C$.

$R_{\bowtie\neg}$ **if** $A \sqsubseteq \bowtie nR.C \in \mathcal{T}, \dot{\neg}(\bowtie nR.C) \in L(v), \neg A \notin L(v)$
then $L(v) \rightarrow L(v) \cup \{\neg A\}$

If we have an axiom of the form $A \sqsubseteq \bowtie nR.C$ and there is a node subsumed by $\dot{\neg}(\bowtie nR.C)$ then this node should also be subsumed by $\neg A$.

$R_{\bowtie\sqsubseteq}$ **if** $\bowtie nR.A \sqsubseteq B \in \mathcal{T}, infeasible(Q(v) \cup \{\dot{\neg}(\bowtie nR.A)\}, B \notin L(v)$

then $L(v) \rightarrow L(v) \cup \{B\}$

This is another rule that adds new subsumers based on a clash in inequalities. For each node subsumed by inequalities we can test whether it is subsumed by a concept B if we have an axiom of a form $\bowtie nR.A \sqsubseteq B$ and if there is a clash in inequalities in the node and $\dot{\neg} \bowtie nR.A$.

5.4.3 Implementation Details

When we started implementing Avalanche our first priority was to implement the basic functionality of the system. Initially we implemented the rules presented above in such a way that they all were be applied until no new information could be added to the graph. One of the first optimization strategies that we came up with was to separate the rule application process into two main phases: the unfolding and the saturation phases.

During the unfolding phase each of the following rules $R_{\sqsubseteq}$, $R_{\sqsubseteq\neq}$, $R_{\sqsubseteq*}$, $R_{P_{\bowtie}}$, and $R_{P_{\bowtie_{Dynamic}}}$ is applied exactly once. The goal of this phase is to extract as much information as possible from the ontology. The rest of the rules are applied in the saturation phase that is itself divided into two sub-phases. We first apply rules $R_{\sqsubseteq*}$, $R_{P_{\bowtie_{Dynamic}}}$ - $R_{\sqsubseteq\sqcap\atop\sqcup}$ until no new information can be added to the graph. Then we apply $R_{\sqcup}$ and $R_{\sqsubseteq_P}$. After that we repeat the first step. The reason is that we try to delay clone generation as much as possible hoping to discover subsumptions or disjointness between concepts with other rules. Clones make our saturation graph larger and require calls to the ILP module QMediator to solve inequalities. The same holds for the$R_{\sqsubseteq_P}$ rule that creates unfold nodes that are also resolved with QMediator. In general we try to avoid unnecessary *QMediator* calls because each time the module

is called it needs to create a new model. Some models could be very complex and incur a significant computational overhead.

Another strategy that we came up with was to apply rules on demand. A node calls necessary rules based on its subsumers. For example, if a rule requires a new negated subsumer to be added to the subsumers of a node then this rule should only be executed if a new negated subsumer has been added to the node.

Further, we have introduced several types of nodes, such as anonymous and auxiliary nodes for example. Some of our rules are designed to skip these nodes during the reasoning process and therefore avoid unnecessary calculations that cannot give us new knowledge. For example, $R_{\sqsubseteq_P}$ will not create clones to test if two anonymous concepts or two clones subsume each other because this information will not add any value to the resulting graph.

Finally, we have always acknowledged that the ILP module QMediator despite being a great solution to deal with qualified number restrictions still creates computational overhead. That is why we came up with several heuristics that do not require calls to the external module. For example, we can detect obvious clashes between qualified number restrictions such as $\leq 0R.C$ and $\geq 1R.C$.

5.5 Reasoning with Qualified Number Restrictions

Avalanche is a complex rule-based system that implements a saturation-based reasoning algorithm. An early version of this algorithm is presented in [61]. The algorithm manages the application of rules on an input ontology by traversing the saturation graph. The ILP module QMediator is called when a rule needs to expand the underlying graph or when a clash has been detected in a node due to the presence of qualified number restrictions. With the help of the module we can reduce

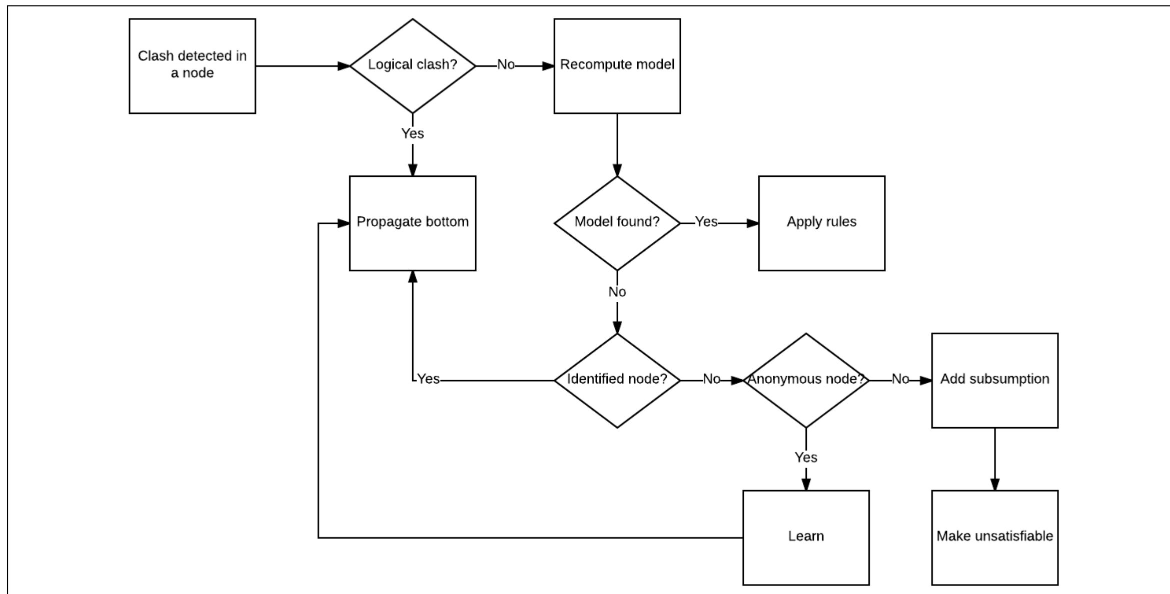

FIGURE 5.1: Clash Detection

the problem of deciding satisfiability of qualified number restrictions to the feasibility of inequalities. This gives us a clear advantage over the other existing systems when working with qualified number restrictions that result in subsumptions. The are several ready to use tools to efficiently solve systems of inequalities. One of them is IBM CPLEX that we have chosen for our implementation.

To avoid circular dependencies between the two systems QMediator cannot call or access any data from Avalanche. In order to communicate with the ILPModule Avalanche has to call the graph expansion rule, R_{fil}, on every node that is subsumed by qualified number restrictions. The rule in turn will call QMediator and pass the corresponding information: the qualified number restrictions, the subsumers of the qualifications and their unsatisfiable concept conjunctions. After that the QMediator will transform this information into a linear program and it will call CPLEX to solve it or in other words to find a model. The result of this call will be returned to the rule. Thus, the rule will have all the necessary information to expand the graph or to make the node unsatisfiable by adding $\bot$ (bottom) to its subsumers. As a result, the

expansion rule may create additional nodes in the graph - the anonymous nodes. An anonymous node represents a situation when a role filler is not a single concept (e.g., A) but rather an intersection of concepts ($A \sqcap B$).

In Figure 5.1 we show how the call to QMediator is integrated into the clash detection process for some of the nodes. If a node becomes unsatisfiable, then the cause of the unsatisfiability has to be identified. If there is a logical clash (e.g., A and $\neg A$ are present in the node) then the corresponding ancestors of the node will be made unsatisfiable. However, if the clash is due to the presence of qualified number restrictions then QMediator should be called and it should be asked to recompute a more constrained model. If a new model was computed, the rules can continue to be applied. If there is no model and the node is an identified node, then the corresponding ancestors of the node should be made unsatisfiable. If the node is an anonymous node this information will be recorded to avoid having the QMediator to recompute the same model. Otherwise the node in question must be a positive/negative cloned node. In this case it can be concluded that the subsumption/disjointness holds and the node will be marked as unsatisfiable.

Chapter 6

Avalanche Implementation Details

In this chapter we will present the design and implementation of Avalanche. We will start by presenting the underlying data structure - the saturation graph. We will then proceed with the description of the normalization and saturation-based rules. Following that we will discuss some of the optimization strategies that we have integrated into our system and talk about some of the implementation details. We will also explain how *Avalanche* communicates with the ILP module QMediator and how it interprets the information produced by the latter. The design and implementation of QMediator will be discussed in the following chapter. The overall goal of this chapter is to bridge the theoretical and the practical aspects of this work.

6.1 Overview of Avalanche

When we started this project and defined the scope of our research we decided to design calculus for the DL $\mathcal{ALCHQ}$ and implement it in a reasoner as a proof of concept. One of our research goals was to translate qualified number restrictions into linear programs and solve them as such. This step was planned to be delegated to an already existing external linear optimization module. As it often happens the

initial estimate proved to be overly optimistic. During the implementation of the reasoner we realized that the existing linear optimization module could not support growing needs of our evolving system. We had no choice but to take a detour from our original research plan and to implement our own linear optimization module that we named QMediator. The sole purpose of QMediator is to translate the information received from Avalanche into linear programs that will be solved by the IBM CPLEX linear solver. It acts as a *go-between*, hence the name. We chose to keep these two systems separate because Avalanche can function independently from QMediator as long as it does not need to solve *difficult* inequalities. For example, $\mathcal{EL}$ ontologies do not require QMediator. Even for more expressive ontologies we implemented a number of simple optimizations that can reason with qualified cardinality restrictions without calling the specialized module. However, these cases are rather rare. If we want to classify a real-world $\mathcal{ALCHQ}$ ontology we need to have a fully functioning Avalanche.

QMediator is a specialized system that focuses on efficiently solving inequalities received from Avalanche. Additionally, Avalanche has to precompute other important information that needs to be represented in QMediator, e.g. subsumption and disjointness of role fillers. The task of QMediator is to translate this information into a linear program and solve it by applying the *Branch-and-Price* method that had to be implemented from scratch. After having found the solution (or the absence of thereof), QMediator has to generate the return information that can be processed by Avalanche. Additionally, it could compute clash sources, in case the submitted system of inequalities has no solution. This information will be returned to *Avalanche* thus allowing it to continue its reasoning process. The interaction between the two systems was designed in such a way that QMediator can easily be replaced with any other potentially more efficient system or even be used by any other reasoner that

needs to solve inequalities by applying the *Branch-and-Price* method.

6.2 Communication between Avalanche and QMediator

In this section we will present the API of QMediator and explain how it communicates with Avalanche.

At some point of its execution Avalanche encounters inequalities that cannot be solved with its local optimizations. In such a situation it will call QMediator passing the following parameters:

Parameter	**Type**	**Description**
`inequalities`	`HashSet`	Inequalities that have to be solved
`subsumersOfRoleFillers`	`HashMap`	Mapping of role fillers of inequalities to their subsumers
`disjointsOfRoleFillers`	`HashMap`	Mapping of role fillers of inequalities to their disjoint concepts
`binarySubsumptions`	`HashMap`	Mapping of binary conjunctions to their subsumers
`disjointGroup`	`HashSet`	Groups of concepts that are unsatisfiable when appear together
`roleHierarchy`	`HashMap`	Mapping of roles to their subsumer roles
`roleHierarchyReversed`	`HashMap`	Mapping of roles to their subsumee roles
`isNodeTMP`	`Boolean`	True if the node that called QMediator is DisjunctionNode, false otherwise

weight	Integer	An integer value that will be used by QMediator to assign initial coefficients when constructing the Reduced Master Problem
applyBnP	Boolean	True if QMediator has to compute an integer solution if the initial solution is non-integer, false otherwise

TABLE 6.1: Input parameters for QMediator

Avalanche passes inequalities and additional information about roles and role fillers in the inequalities. It sets two boolean flags that allow to skip the expensive computation of clashing inequalities. `applyBnP` is set to false when we do not want to compute clash sources even if no solution has been found. In the next chapter we will explain in detail how QMediator uses these parameters to construct linear programs. QMediator reads this information, translates it into a linear program, solves it, and responds as described in the table 6.2.

Parameter	**type**	**Description**
solved	Boolean	True if there is a solution, false otherwise
edgeInformation	HashSet	Information about edges that *Avalanche* needs to continue its work
infeasibleSet	HashSet	Sets of conflicting inequalities that if removed will make the linear program feasible

TABLE 6.2: Input parameters for QMediator

The response information, in case a solution is present, will be represented as an object of the type `EdgeInformation`, which contains information about edges that have to be built in the completion graph. It will contain a name of an edge, role fillers, and a cardinality, e.g. {{ r }, { A, B }, 4}. Otherwise it will be empty. If there is

no solution but we want to know what the clash sources are, QMediator will return a set of sets of clashing inequalities.

Chapter 7

Linear Programming Engine QMediator

In this chapter we will present QMediator, the linear programming engine of Avalanche. We have already referred several times to QMediator when presenting Avalanche and its rules. Now we will explain in detail how this particular subsystem works.

7.1 Interaction with Avalanche

QMediator is a system that accepts input from Avalanche, processes it, and then translates it to a linear program. The linear program itself will be solved by another external system, IBM CPLEX. Thus, QMediator can be a considered as a bridge or a mediator between Avalanche and CPLEX.

QMediator is called by several saturation rules in Avalanche to create and solve systems of linear inequalities that are represented in Avalanche as qualified cardinality restrictions. As a result, QMediator either returns a solution or identifies on

demand the inequalities that make the input system of linear inequalities in question infeasible. We call these inequalities the clash set or the source of clash. The logic for handling those is in the rules themselves.

Before we started working on QMediator we had already known that a naively implemented system would not scale. That is why we implemented a powerful and quite complicated optimization technique called *Branch-and-Price* that has some control over CPLEX and its approach to search of the solution process. Nevertheless even with this optimization each CPLEX call can potentially become extremely time-consuming. The computation of a clash set is a particularly expensive operation. For this reason we had to limit the overall interaction of Avalanche with QMediator and to compute the clash set on only demand. In most cases when inequalities are infeasible we do not need to know the clash set as it is sufficient to be aware of the fact that the given inequalities are infeasible.

7.2 Input Presentation

The input inequalities that are passed to QMediator can either be an *at-least* or an *at-most* qualified cardinality restrictions. Each inequality is composed of the following four parts:

- a sign that indicates if it is an *at-least* or an *at-most* inequality
- a cardinality that indicates a lower or an upper bound
- a role that is also called a property
- a role filler that is also called a qualification

For example, $\leq 3\, R.C$ is an *at-most* inequality with a cardinality equal to 3, a role R, and a role filler C.

QMediator also internally works with equalities that are be preceded by the *equals* sign. For example, $= 3\, R.C$.

In addition to the inequalities QMediator accepts as input certain information about their roles and role fillers. This includes information about subsumers, disjoint concepts, binary subsumption axioms, role hierarchy, sets of concepts that are infeasible together, type of a node that has called QMediator (e.g. static, anonymous etc.), and a coefficient. All this information will be used later to create the linear program.

7.3 Branch and Price Approach

QMediator leverages the *Branch-and-Price* optimization method that includes *Delayed Column Generation* and *Branch-and-Bound* algorithms. Column Generation is the algorithm that drives the solution process. Keep in mind that if Column Generation is unable to find an integer solution, this is not a guarantee that it does not exist. In such cases, Branch-and-Price would be invoked and by means of branching and pruning it would either determine what the integer solution is or prove that such a solution does not exist.

7.3.1 Column Generation

The main idea and a great advantage of Column Generation is that the underlying algorithm creates a very limited number of internal variables that are used to solve system of linear inequalities. The algorithm will create only those variables that will lead to a solution. This can be achieved by splitting the original linear program into two new programs: *Reduced Master Problem* known as RMP and *Pricing Problem*

known as PP. Thus Column Generation is just a series of RMP-PP calls. The result of each call adds new information to the next linear program.

This approach significantly reduces the time needed to solve a system of linear inequalities and thus makes it possible to solve very large linear programs in a reasonable amount of time. In this section we will give an overview of the linear program construction. In the following sections we will present some examples that will help to understand the entire process.

In order to start executing the Column Generation algorithm we first need to preprocess the input inequalities. During the normalization phase we have transformed qualified equality constraints into *at-least* and *at-most* constraints to facilitate the application of other rules but in the context of QMediator we prefer to represent them again as equality constraints. For example, $\geq 3\, R.C$ and $\leq 3\, R.C$ would be transformed to $= 3\, R.C$.

Then we will need to extract the *at-least* and *at-most* role fillers and store them in their corresponding data structures. Role fillers from *equals* constraints will appear in both data structures. We will also need to keep track of the roles of the equalities and inequalities. If the submitted inequalities contain at least two different roles we will have to compute a role hierarchy that will later be represented in PP.

After that we will need to check if the concept $\top$ is present in the set of the previously extracted role fillers. This concept will be represented in PP as well.

Finally, we will do some other minor processing steps and then we will be ready to proceed with the construction of the first RMP and the first PP.

The RMP is a minimization problem. Therefore, the value of each objective function at each iteration will be decreasing but we do not expect it to become equal to 0. In order to implement the first RMP we will need to create artificial variables. We call them h variables. These variables are created only once and later are used by

every subsequent RMP model. An artificial variable is created for each submitted qualified cardinality restriction. For example, $\leq 3\, R.C$ would be represented by h_1. The first objective function of the first RMP will be composed entirely of artificial variables. These variables are multiplied by a coefficient that has been precomputed from the original set of qualified cardinality restrictions. In order to compute the coefficient we take the sum of all *at-least* and *equals* cardinalities and multiply it by 10. When we reach the last PP we expect that current RMP does not contain any artificial variables, i.e. the values of all artificial variables should be equal to 0. If there is at least one non-zero artificial variable it means that the linear program cannot be solved. We assign a very large coefficient to the artificial variables as at each iteration we will be finding better variables with smaller coefficients that will eventually remove all of the artificial variables from the solution.

As a result, the RMP returns a partition variable x and a coefficient that will be used to create the next PP. All PPs will share the same role variables r, role hierarchy rh variables and b variables. Therefore we create them upfront and then pass them as parameters. Similarly to h variables, r variables represent inequalities but in the context of PP. b variables represent role fillers of inequalities.

Subsequently PP adds a new variable to the next RMP. The new variable is created from non-zero r and b variables. When PP cannot create a new variable we assume that the execution of the linear optimization has come to an end.

The value of the objective function of PP will be first equal to a very small negative value and with each iteration this value will be increasing until it reaches 0. At this point we know that this is the final PP and we can check if the problem has a solution.

We distinguish two cases in QMediator. One case is applicable to all types of nodes and the other is only applicable to unsatisfiable unfold nodes. In the latter

case it is necessary to compute a clash set. We need to introduce some modifications to Column Generation to accommodate small differences in the processing of unsatisfiable unfold nodes. When we create artificial variables for all other nodes we exclude *at-most* 0 qualified cardinality restrictions. However, we must include these inequalities when we create artificial variables for unsatisfiable unfold nodes. After that we will proceed as usual with the generation of r and b variables. Then we will proceed with the creation of PP constraints.

7.3.2 Generation and Interpretation of Constraints

Before presenting examples it is worth explaining how the different constraints are created and what they represent.

If inequities that have to be solved by QMediator come from an unsatisfiable disjunction node we have to create constraints for *at-most* inequalities and *exactly* 0 equalities. The constraints will be composed of r and b variables. The format of these constraints is the following:

$$r - b = 0 \tag{7.1}$$

which means that if a given r variable has been enabled then its corresponding b variable must also b enabled.
The r and b variables here must both refer to the same role filler.

Then we add role hierarchy constraints. We create additional role hierarchy variables - the rh variables. We first need to match r variables with rh variables and b variables. Next we need to map r variables to rh variables. After that if we are not dealing with an unsatisfiable unfold node we encode *at-most* the 0 inequalities and the *exactly* 0 equalities by means of r variables and b variables. Subsequently we

add constraints that map *rh* variables to other *rh* variables. The formats of these constraints are the following:

$$rh - r + b \leq 1 \tag{7.2}$$

This type of constraint represents a mapping from a role of an inequality to its super role and a corresponding role filler. It means that if we enable a given *rh* variable we must also enable its corresponding *r* variable

$$r - rh \leq 0 \tag{7.3}$$

This type of constraint represents a mapping from a role of a qualified number restriction to its super role. The constraint stipulates that if we enable the *r* variable then we also must enable a corresponding *rh* variable

$$rh - b \leq 0 \tag{7.4}$$

This type of constraint represents a mapping of a role hierarchy variable to a corresponding *b* variable. The constraint stipulates that if we enable the *rh* variable then we also should enable its corresponding *b* variable

$$rh_1 - rh_2 \leq 0 \tag{7.5}$$

This type of constraint represents a mapping of a role hierarchy variable to another role hierarchy variable. The constraint stipulates that if a subrole variable is enabled then its super role variable must also be enabled

Next if a node is not an unsatisfiable unfold node we define disjointness of concepts by means of b variables. The format of this constraint is the following:

$$b_1 + b_2 \leq 1 \tag{7.6}$$

Here b_1 and b_2 represent disjoint concepts. The constraint stipulates that that only one of these variables can be enabled.

For all other nodes in the presence of role hierarchy we propagate universal restrictions down to their subsumees. The format of the first constraint is the following:

$$r - b \leq 0 \tag{7.7}$$

The constraint stipulates that if a role variable is switched on then its corresponding role filler b variable must also be switched on. If there is no role hierarchy we still need to represent universal restrictions. If we are not processing an unsatisfiable tmp node we map r variables with r variables of at-most 0 qualified cardinality restrictions if they share the same role. The format of the second constraint is the following:

$$r_1 - r_2 \leq 0 \tag{7.8}$$

The constraint stipulates that if it is not an unsatisfiable unfold node we match b variables with r variables of *at-most* 0 inequalities. The format of this constraint is the following:

$$r - b \leq 0 \tag{7.9}$$

If it is an unfold node and role hierarchy is present we will match b variables to r variables. The format of this constraint is the following:

$$b - r = 0 \tag{7.10}$$

If it is an unfold node but there is no role hierarchy then we will also have to match b and r variables. The format of this constraint is the following:

$$r - b = 0 \tag{7.11}$$

Further, we match r variables of *at-least* or *exactly* cardinality restrictions with corresponding b variables. The format of this constraint is the following:

$$r - b \leq 1 \tag{7.12}$$

where role filler of the inequality linked to a r variable will correspond to a b variable.

We represent subsumptions for all of the nodes by means of b variables. The format of this constraint is the following:

$$b_1 - b_2 \leq 0 \tag{7.13}$$

where b_1 is a subsumee and b_2 is a subsumer The constraint stipulates that if the variable that represents a given subsumee is enabled then the variable that represents the subsumer must be enabled as well.

We also represent binary subsumptions. The format of this constraint is the following:

$$b_1 + b_2 - b_3 \leq 1 \tag{7.14}$$

where b_1 and b_2 are subsumees and b_3 is a subsumer

Finally, we represent the presence of Thing. The format of this constraint is the following:

$$b_1 - b_{Thing} \leq 0 \tag{7.15}$$

which means that if switch on b_1 then we also must switch on b_{Thing}. In such a way we ensure that the concept Thing subsumes every other concept.

7.3.3 Branch-and-Bound

When the Column generation algorithm terminates it is possible that a solution exists but it is non-integer, i.e. at least one variable has a non-integer value. In such a situation we have to call the Branch and Bound algorithm that will attempt to assign new values to the existing variables by creating additional partition variables. The main idea remains the same - we execute another series of RMP-PP calls. However, we need to modify the RMP and the PP programs accordingly. The PP will have only minor modifications. We will create additional branch variables that will be added to the objective function and we will create new constraints that will match the new branch variables with existing b variables. In RMP we will add new branching variables to the existing constraints and we also create new constraints that will assign values to the branch variables. The algorithm itself is very simple. Every non-integer partition variable can have one of two possible values: these are obtained by rounding the existing non-integer value up or down to the nearest integer. Then we will need to check which of the values could lead us to an integer solution. There

is the same termination condition - when the value of the objective function of PP is equal to 0 we know that the algorithm has finished its execution and no other variables can be created.

7.3.4 Clash Set Detection

In cases when a non-integer variable is still present even after applying the Branch and Bound algorithm we assume that there is no integer solution to the problem. In our case this means that the given inequalities are infeasible. In this case, if requested, we will invoke the dedicated algorithm to compute the clash set, i.e. the inequalities that clash together but if removed from the original set of inequalities the remaining inequalities will become feasible. For example, $\leq 2\, R.C$ and $\geq 1\, R.C$ always constitute a clash set regardless of the other inequalities.

In this example, if required to compute the clash set, CPLEX would return only one of the inequalities $\leq 2\, R.C$ or $\geq 1\, R.C$ as removing any one of them would make the linear program feasible. This does not exactly match our needs: while it does return the inequalities that render the input model unsolvable, CPLEX does not tell us which of the remaining inequalities do they clash with. Therefore, in order to obtain this additional information, it was necessary to implement our own algorithm for computing clash sets on top of the existing CPLEX functionality. This algorithm is very simple to implement but it is very computationally expensive and that is why we were careful not to execute it unless absolutely necessary. The algorithm is based on removing and reinstating inequalities in the original input and executing another series of RMP-PP calls. The output is the minimum clash set of the original set of inequalities. A detailed description of the algorithm is presented below.

Let's assume that we have the following three infeasible constraints:
$\{max, min_1, min_2\}$

We temporarily remove *max* from the set and test $\{min_1, min_2\}$. If it is feasible we know that *max* constitutes the clash set. Therefore we reinstate *max*.
Then we remove min_1 and test $\{max, min_2\}$. It is infeasible. min_1 will be deleted because it does not change anything.
Next we remove min_2 and test $\{max\}$. If it's feasible we reinstate min_2. The result $\{max, min_2\}$ is therefore deemed to be a minimum clash set.

7.4 Examples and Result Interpretation

In this section we present two examples of previously described concepts. First, we will illustrate how we create RMP and PP linear programs. We would also demonstrate how we create and add new variables to the corresponding constraints. Secondly, we will show how the Branch-and-Bound algorithm works when the initial solution contains a non-integer partition variable.

7.4.1 Simple Example

Let us examine a simple system of inequalities that only requires two iterations of RMP - PP calls to resolve. We assume that the *fil*-rule passes the following inequalities when calling QMediator:

$$\geq 20\, R.Thing$$
$$\leq 10\, R.A'$$
$$\leq 10\, R.B$$
$$\leq 10\, R.A$$

In order to solve these we first need to create artificial variables:

Artificial variable h_2 for $\leq 10\, R.B$

Artificial variable h_0 for $\geq 20\, R.Thing$

Artificial variable h_3 for $\leq 10\, R.A$

Artificial variable h_1 for $\leq 10\, R.A'$

Then our first RMP model will look as follows:

minimize $200.0 * h_2 + 200.0 * h_0 + 200.0 * h_3 + 200.0 * h_1$

subject to:

$1.0 * h_2 \leq 10$

$1.0 * h_0 \geq 20$

$1.0 * h_3 \leq 10$

$1.0 * h_1 \leq 10$

The linear program will be solved by CPLEX. As a result we determine the value of the objective function to be equal to 4000. Keep in mind that the actual value is of little interest to us: we just need to know that there exists an integer solution to the linear program.

Finally, as a part of the RMP result we also extract dual values of artificial variables and use them as coefficients in the next PP.

The current artificial variables have the following dual values:

Artificial variable h_2, dual value = 0

Artificial variable h_0, dual value = 200

Artificial variable h_3, dual value = 0

Artificial variable h_1, dual value= 0

Now we have enough information to build our first PP model.

These are the b variables that will be used by all future PP models:

b variable b_3, role filler = B

b variable b_2, role filler = A

b variable b_0, role filler = A′

b variable b_1, role filler = Thing

Below are the r variables that will also be used by all future PP models:

r variable r_0, inequality=$\leq$ 10 $R.B$, current coefficient = 0

r variable r_1, inequality=$\geq$ 20 $R.Thing$, current coefficient = 200

r variable r_3, inequality=$\leq$ 10 $R.A'$, current coefficient = 0

r variable r_2, inequality=$\leq$ 10 $R.A$, current coefficient = 0

The dual values computed by the previous RMP will be used as coefficients for r variables. They will be updated after each RMP execution.

Now we can build our first PP model:

minimize $1 * b_3 + 1.0 * b_2 + 1.0 * b_0 + 1.0 * b_1 - 0.0 * r_0 - 200.0 * r_1 - 0.0 * r_3 - 0.0 * r_2$

subject to:

$1 * b_3 - 1 * r_0 = 0$

$-1 * b_1 + 1 * r_1 \leq 0$

$1 * b_0 - 1 * r_3 = 0$

$1 * b_2 - 1 * r_2 = 0$

$1 * b_0 - 1 * b_1 \leq 0$

$1 * b_2 - 1 * b_1 \leq 0$

$1 * b_3 - 1 * b_1 \leq 0$

There is only one non-zero r variable: r_1. There is also only one non-zero b variable: b_1. Next, we will create a new partition variable x_{Thing} with a coefficient 1. The coefficient is the size of the set of non-zero b variables. The value of the objective function is -199. We then proceed with another RMP - PP iteration, because the value of the objective function is not yet 0.

We will reuse the existing artificial variables and we will add a new partition variable to the corresponding constraints. The new RMP model will look as follows:

minimize $1 * x_{Thing} + 200 * h_2 + 200 * h_0 + 200 * h_3 + 200 * h_1$

subject to:

$1 * h_2 \leq 10$

$1 * x_{Thing} + 1.0 * h_0 \geq 20$

$1 * h_3 \leq 10$

$1 * h_1 \leq 10$

The value of the objective function of the RMP is equal to 20. There are no non-zero artificial variables. Dual values of the artificial variables are the following:

Artificial variable h_2, dual value equal to 0

Artificial variable h_0, dual value equal to 1

Artificial variable h_3, dual value equal to 0

Artificial variable h_1, dual value equal to 0

Now we can proceed with the creation of a new PP model. The variables r and b will be reused. The coefficients of r variables will be updated based on the result of the last RMP model.

PP model:

minimize $1 * b_3 + 1 * b_2 + 1 * b_0 + 1 * b_1 - 0 * r_0 - 1 * r_1 - 0 * r_3 - 0 * r_2$

subject to:

$1 * b_3 - 1 * r_0 = 0$

$-1 * b_1 + 1 * r_1 \leq 0$

$1 * b_0 - 1 * r_3 = 0$

$1 * b_2 - 1 * r_2 = 0$

$1 * b_0 - 1 * b_1 \leq 0$

$1 * b_2 - 1 * b_1 \leq 0$

$1 * b_3 - 1 * b_1 \leq 0$

The value of the objective function is equal to 0. Therefore there would be no further RMP-PP iterations. Since the last RMP model did not contain any non-zero artificial variables we know that CPLEX has successfully terminated and therefore a solution exists for the input system of inequalities. Furthermore, all partition variables have integer values. This indicates that not only a solution exists for the original system, but it is also an integer solution. This solution would be as follows:

Edges: {R}, Fillers: $\{\neg A'\}$, Cardinality: 0

Edges: {R}, Fillers: {Thing}, Cardinality: 10

Edges: {R}, Fillers: $\{\neg A, \neg B, \neg A'\}$, Cardinality: 10

Edges: {R}, Fillers: $\{\neg B, \neg A'\}$, Cardinality: 0

The solution determines the role successors that we must create. Fillers represent the representative concepts and cardinality is the number of successors. We annotate edges with the cardinalities and will use non-literal fillers to build anonymous nodes. In case there is only one filler we will not need to build an anonymous node as we can use one of the identified nodes.

7.4.2 Branch and Bound Example

This example will illustrate how we apply the Branch and Bound algorithm. We first try to solve the submitted inequalities in the usual way by starting a series of RMP-PP iterations. The final RMP result will contain partition variables with non-integer values. We will show what we do in a such a situation.

QMediator receives as an input the following inequalities:

$\leq 3\, R.Thing$

$\geq 3\, R.Thing$

$\leq 1\, R.A$

$\geq 1\, R.A$

$\leq 1\, R.B$

$\leq 1\, R.C$

$\leq 1\, R.D$

$\leq 1\, R.F$

We will create our first RMP model with the following artificial variables that will be later used by all subsequent RMP models:

Artificial variable h_5 for the inequality $\leq 1\, R.F$

Artificial variable h_2 for the inequality $\leq 1\, R.B$

Artificial variable h_1 for the inequality $= 1\, R.A$

Artificial variable h_0 for the inequality $= 3\, R.Thing$

Artificial variable h_4 for the inequality $\leq 1\, R.D$

Artificial variable h_3 for the inequality $\leq 1\, R.C$

The RMP model will look as follows:

minimize $80.0 * h_5 + 80.0 * h_2 + 80.0 * h_1 + 80.0 * h_0 + 80.0 * h_4 + 80.0 * h_3$

subject to:

$1 * h_5 \geq 1$

$1 * h_2 \geq 1$

$1 * h_1 = 1$

$1 * h_0 = 3$

$1 * h_4 \geq 1$

$1 * h_3 \geq 1$

RMP objective value: 640

RMP returns the following dual values as a result:

Artificial variable h_5, dual value = 80

Artificial variable h_2, dual value = 80

Artificial variable h_1, dual value = 80

Artificial variable h_0, dual value = 80

Artificial variable h_4, dual value = 80

Artificial variable h_3, dual value = 80

Now we can proceed with the construction of the PP model. We first create b variables:

b variable b_3, role filler is F

b variable b_4, role filler is D

b variable b_5, role filler is C

b variable b_0, role filler is B

b variable b_1, role filler is A

b variable b_2, role filler is Thing

Then we create r variables with updated coefficients:

r variable r_2, inequality is $= 1\ R.A$, current coefficient = 80

r variable r_1, inequality is $\leq 1\ R.B$, current coefficient = 80

r variable r_5, inequality is $\geq 1\ R.C$, coefficient = 80

r variable r_4, inequality is $\leq 1\ R.D$, current coefficient = 80

r variable r_0, inequality is $\leq 1\ R.F$, current coefficient = 80

r variable r_3, inequality is $= 3\ R.Thing$, current coefficient = 80

The PP model will look as follows:

minimize $1.0 * b_3 + 1.0 * b_4 + 1.0 * b_5 + 1.0 * b_0 + 1.0 * b_1 + 1.0 * b_2 - 80.0 * r_2 - 80.0 * r_1 - 80.0 * r_5 - 80.0 * r_4 - 80.0 * r_0 - 80.0 * r_3$

subject to:

$1 * b_1 - 1 * r_2 = 0$

$-1 * b_1 + 1 * r_2 \leq 0$

$-1 * b_0 + 1 * r_1 \leq 0$

$-1 * b_5 + 1 * r_5 \leq 0$

$-1 * b_4 + 1 * r_4 \leq 0$

$-1 * b_3 + 1 * r_0 \leq 0$

$1 * b_2 - 1 * r_3 = 0$

$-1 * b_2 + 1 * r_3 \leq 0$

$1 * b_4 + 1 * b_0 \leq 1$

$1 * b_0 - 1 * b_2 \leq 0$

$1 * b_1 - 1 * b_2 \leq 0$

$1 * b_3 - 1 * b_2 \leq 0$

$1 * b_4 - 1 * b_2 \leq 0$

$1 * b_5 - 1 * b_2 \leq 0$

The value of the objective function of the PP is -395.

non-zero r variables:

r variable r_2

r variable r_5

r variable r_4

r variable r_0

r variable r_3

non-zero b variables:

b variable b_3

b variableb_4

b variable b_5

b variableb_1

b variable b_2

With the information above we can create a new partition variable $x_{FDCAThing}$. Its coefficient will be equal to 5 which is the number of non-zero b variables. Next we will need to add the new partition variable to the subsequent RMP. Artificial variables will be reused from the original RMP. The new partition variable will be added to the RMP constraints where artificial variables represent the constraints that constitute the partition variable.

The RMP model will look as follows:

minimize: $5 * x_{FDCAThing} + 80 * h_5 + 80 * h_2 + 80 * h_1 + 80 * h_0 + 80 * h_4 + 80 * h_3$

subject to:

$1 * x_{FDCAThing} + 1 * h_5 \geq 1$

$1 * h_2 \geq 1$

$1 * x_{FDCAThing} + 1 * h_1 = 1$

$1 * x_{FDCAThing} + 1 * h_0 = 3$

$1 * x_{FDCAThing} + 1 * h_4 \geq 1$

$1 * x_{FDCAThing} + 1 * h_3 \geq 1$

This system of inequalities will eventually be solved and the new objective value will be 245. New coefficients will be assigned to the artificial variables that will be used in the next PP.

New dual values of the artificial variables:

Artificial variable h_5, dual value = 80

Artificial variable h_2, dual value = 80

Artificial variable h_1, dual value = -315

Artificial variable h_0, dual value = 80

Artificial variable h_4, dual value = 80

Artificial variable h_3, dual value = 80

After that we can create our next PP model. The r and b variables are the same as before but their coefficients have changed.

minimize $1 * b_3 + 1 * b_4 + 1 * b_5 + 1 * b_0 + 1 * b_1 + 1 * b_2 + 315 * r_2 - 80 * r_1 - 80 * r_5 - 80 * r_4 - 80 * r_0 - 80 * r_3$

subject to:

the constraints of the PP are identical to the constraints in the previous PP model.

The value of the objective function is equal to -316.

non-zero r variables

r variable r_5

r variable r_4

r variable r_0

r variable r_3

non-zero b variables

b variable b_3

b variable b_4

b variable b_5

b variable b_2

As a result we will create another partition variable $x_{FDCThing}$ with its coefficient equal to 4. Next we will add the newly created variable to a new RMP model:

minimize $4 * x_{FDCThing} + 5 * x_{FDCAThing} + 80 * h_5 + 80 * h_2 + 80 * h_1 + 80 * h_0 + 80 *$

$h_4 + 80 * h_3$

subject to:

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_5 \geq 1$

$1 * h_2 \geq 1$

$1 * x_{FDCAThing} + 1 * h_1 = 1$

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_0 = 3$

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_4 \geq 1$

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_3 \geq 1$

The value of the objective function is 93. The dual values of the artificial variables are:

h_5, dual value = 0

h_2, dual value = 80

h_1, dual value = 1

h_0, dual value = 4

h_4, dual value = 0

h_3, dual value = 0

We will skip the next iterations and will show what happens when we reach the PP with the objective value equal to 0. We first show the RMP model that precedes the PP model in question. We will add a new partition variable x_{CThing}. Its coefficient will be 2. The RMP model will be as follows:

minimize $2 * x_{AThing} + 2 * x_{DThing} + 2 * x_{BThing} + 2 * x_{CThing} + 4 * x_{FDCThing} + 5 * x_{FDCAThing} + 2 * x_{FThing} + 80 * h_5 + 80 * h_2 + 80 * h_1 + 80 * h_0 + 80 * h_4 + 80 * h_3$

subject to:

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * x_{FThing} + 1 * h_5 \geq 1$

$1 * x_{BThing} + 1 * h_2 \geq 1$

$1 * x_{AThing} + 1 * x_{FDCAThing} + 1 * h_1 = 1$

$1 * x_{AThing} + 1 * x_{DThing} + 1 * x_{BThing} + 1 * x_{CThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * x_{FThing} + 1 * h_0 = 3$

$1 * x_{DThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_4 \geq 1$

$1 * x_{CThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_3 \geq 1$

The objective value of this RMP model is 8.

Dual values of artificial variables are:

h_5, dual value = 1

h_2, dual value = 1

h_1, dual value = 1

h_0, dual value = 1

h_4, dual value = 1

h_3, dual value = 1

non-zero partition variables are:

Partition variable x_{AThing}, value = 0.33333333333333337

Partition variable x_{DThing}, value = 0.33333333333333337

Partition variable x_{BThing}, value = 1

Partition variable x_{CThing}, value = 0.33333333333333337

Partition variable $x_{FDCAThing}$, value = 0.6666666666666666

Partition variable x_{FThing}, value = 0.33333333333333337

All h variables have value 0 but there are many non-integer partition variables. Since the objective value of the previous PP model that we had skipped was not equal to 0 yet we have to continue and create another PP model:

minimize $1 * b_3 + 1 * b_4 + 1 * b_5 + 1 * b_0 + 1 * b_1 + 1 * b_2 - 1 * r_2 - 1 * r_1 - 1 * r_5 - 1 * r_4 - 1 * r_0 - 1 * r_3$

subject to:

$1 * b_1 - 1 * r_2 = 0$

$-1 * b_1 + 1 * r_2 \leq 0$

$-1 * b_0 + 1 * r_1 \leq 0$

$-1 * b_5 + 1 * r_5 \leq 0$

$-1 * b_4 + 1 * r_4 \leq 0$

$-1 * b_3 + 1 * r_0 \leq 0$

$1 * b_2 - 1 * r_3 = 0$

$-1 * b_2 + 1 * r_3 \leq 0$

$1 * b_4 + 1 * b_0 \leq 1$

$1 * b_0 - 1 * b_2 \leq 0$

$1 * b_1 - 1 * b_2 \leq 0$

$1 * b_3 - 1 * b_2 \leq 0$

$1 * b_4 - 1 * b_2 \leq 0$

$1 * b_5 - 1 * b_2 \leq 0$

$1 * b_1 - 1 * r_2 = 0$

The objective value of this PP is equal to 0 which means that CPLEX has terminated and we have a solution. However, the previous RMP model contained non-integer partition variables. This means that we have to invoke the Branch and Bound algorithm in order to assign new values to the existing partition variables or possibly create additional partition variables.

We pick one variable from the set of all non-integer partition variables and we assign it two possible values. Let us assume that x_{AThing} was picked, its value being 0.33333333333333337. We can have two possible values: $x_{AThing} \leq 1$ or $x_{AThing} \geq 1$. We try the latter first.

The RMP model with a newly created branch variable will look as follows:

minimize $2 * x_{AThing} + 2 * x_{DThing} + 2 * x_{BThing} + 2 * x_{CThing} + 4 * x_{FDCThing} + 5 * x_{FDCAThing} + 2 * x_{FThing} + 80 * h_5 + 80 * h_2 + 80 * h_1 + 80 * h_0 + 80 * h_4 + 80 * h_3$

subject to:

$1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * x_{FThing} + 1 * h_5 \geq 1$

$1 * x_{BThing} + 1 * h_2 \geq 1$

$1 * x_{AThing} + 1 * x_{FDCAThing} + 1 * h_1 = 1$

$1 * x_{AThing} + 1 * x_{DThing} + 1 * x_{BThing} + 1 * x_{CThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} +$
$1 * x_{FThing} + 1 * h_0 = 3$

$1 * x_{DThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_4 \geq 1$

$1 * x_{CThing} + 1 * x_{FDCThing} + 1 * x_{FDCAThing} + 1 * h_3 \geq 1$

$1 * x_{AThing} \geq 1$

The last inequality assigns one of two possible values to the branch variable.

RMP will terminate with its objective value equal to 8. The dual values of the artificial variables are the following:

Artificial variable h_5, dual value = 1

Artificial variable h_2, dual value = 1

Artificial variable h_1, dual value = 0

Artificial variable h_0, dual value = 1

Artificial variable h_4, dual value = 1

Artificial variable h_3, dual value = 1

The dual value of the branch variable x_{AThing} is 1.

non-zero partition variables are:

x_{AThing}, value = 1

x_{BThing}, value = 1

$x_{FDCThing}$, value = 1

The dual value of of the branch variable is 1.

Now we construct a new PP model. The PP model will include the new branch variable that we had created earlier. It will be added to the objective function and to

the constraints.

minimize $1.0 * b_3 + 1.0 * b_4 + 1.0 * b_5 + 1.0 * b_0 + 1.0 * b_1 + 1.0 * b_2 + -0.0 * r_2 - 1.0 * r_1 - 1.0 * r_5 - 1.0 * r_4 - 1.0 * r_0 - 1.0 * r_3 - 1.0 * branch_1$

subject to:

$-1.0 * b_2 + 1.0 * branch_1 \leq 0$

$-1.0 * b_1 + 1.0 * branch_1 \leq 0$

$1.0 * b_1 + 1.0 * b_2 - 1.0 * branch_1 \leq 1$

$1.0 * b_1 - 1.0 * r_2 = 0.0$

$-1.0 * b_1 + 1.0 * r_2 \leq 0$

$-1.0 * b_0 + 1.0 * r_1 \leq 0$

$-1.0 * b_5 + 1.0 * r_5 \leq 0$

$-1.0 * b_4 + 1.0 * r_4 \leq 0$

$-1.0 * b_3 + 1.0 * r_0 \leq 0$

$1.0 * b_2 - 1.0 * r_3 = 0$

$-1.0 * b_2 + 1.0 * r_3 \leq 0$

$1.0 * b_4 + 1.0 * b_0 \leq 1$

$1.0 * b_0 - 1.0 * b_2 \leq 0$

$1.0 * b_1 - 1.0 * b_2 \leq 0$

$1.0 * b_3 - 1.0 * b_2 \leq 0$

$1.0 * b_4 - 1.0 * b_2 \leq 0$

$1.0 * b_5 - 1.0 * b_2 \leq 0$

This PP model will have an objective value equal to 0. Given the fact that the previous RMP model did not contain non-integer partition variables we conclude that we have successfully found a solution to the submitted inequalities. Now we can construct and return the result to the *fil*-rule that will be able to construct node successors:

We will have three successors:

First successor:

Edges: {R}

Fillers: {F, D, C, $\neg A$}

Cardinality: 1

Second successor:

Edges: {R}

Fillers: {A}

Cardinality: 1

Third successor:

Edges: {R}

Fillers: {B, $\neg A$}

Cardinality: 1

Chapter 8

Proofs

In this chapter we will present proofs of Soundness, Completeness, and Termination of the proposed algorithm that we presented in Chapter 5 .

8.1 Termination

Theorem (Termination) The proposed algorithm terminates when applied to an $\mathcal{ALCHQ}$ ontology.

Algorithm 1 Reasoner Avalanche

1: **Input**: Ontology,
2: Normalization rules, Saturation rules, and Completion rules
3: **Output**: Saturated graph
4: $G \leftarrow CreateGraph(Ontology)$
5: $O_{Norm} \leftarrow NormalizeOntology(Ontology)$
6: $G \leftarrow InitializeGraph(G, Ontology)$
7: $G \leftarrow UnfoldOntology(O_{Norm}, G)$
8: **while** G is not saturated **do**
9:     **while** there are saturation rules to apply **do**
10:         $G \leftarrow ApplySaturationRules(SaturationRules, O_{Norm}, G)$
11:     **end while**
12:     $G \leftarrow ApplySubsumptionRules(SubsumptionRules, O_{Norm}, G)$
13:     $G \leftarrow ApplyDisjunctionRules(DisjunctionRules, O_{Norm}, G)$
14: **end while**

Proof

The algorithm 1 constructs a directed graph. The graph is composed of unique interconnected nodes. Each edge between two nodes is also unique. In **line 4** the algorithm starts with creation of the graph. During this step the algorithm calls the *CreateGraph* function to create the node **Top** or **Thing**. Then in **line 5** the algorithm normalizes an input ontology by calling the *NormalizeOntology* function. The normalization process is described in 5.3. In the following step in **line 6** *InitializeGraph* initializes the graph by creating nodes for all atomic concepts. At this step we make every node a subsumee of **Top**. In **line 7** *UnfoldOntology* populates the graph with simple knowledge that can be sourced directly from the axioms. The rules that are applied at this step are listed in 5.4.3 and presented in 5.4.2. Then in **line 10** the algorithm starts the reasoning process and its main procedure *ApplySaturatoinRules* - application of the saturation rules. These rules are also listed in 5.4.3 and presented in 5.4.2. **Line 12** and **line 13** are the supporting procedures *ApplySubsumptionRules* and *ApplyDisjunctionRules* - application of the subsumption rules and application of the disjunction rules respectively. These rules will be applied repeatedly until the graph is saturated (**line 8**). The saturation rules will be applied as long as they detect a change in the state of the graph (**line 9**). The graph is considered saturated when no changes have been detected.

The rules do not remove nodes from the graph, nor do they remove concepts from node or edge labels. A rule can modify the graph as follows: (i) create a new node and connect it to its source node with an edge, (ii) connect existing two nodes with an edge, (iii) add new concepts to node labels or roles to edge labels . If a rule modifies the graph it will set a corresponding boolean variable `graphModified` to **true**. Thus the algorithm will know that the graph has not been saturated yet and it needs to do another iteration of rule applications. The algorithm will stop once

this variable will be equal to **false**. Further, a rule will not be applied to a node if its consequence is already in a label of the node thus ensuring termination of the algorithm.

The algorithm works in two stages - first it unfolds the graph, i.e. the unfolding phase, and then it applies the saturation, subsumption, and disjunction rules, i.e. the reasoning phase. During the unfolding phase the rules are applied only once to each axiom in the ontology. During the reasoning phase each node maintains a queue of rules to be applied to this node. The queue doesn't allow duplicates. Once all the nodes executed all their pending rules from the queue the subsumption rules following by the disjunction rules will be applied to all matching nodes. If there was any change to the graph then the rule queue will be checked again.

There are three rules that can create new nodes: R_{fil} that is one of the saturation-based rules, the $R_{\sqsubseteq_P}$ that is one of the rules that computes subsumptions, and $R_{\sqcup}$ that is one of the rules that deals with disjunctions. The R_{fil} rule either connects a node to an already existing node or creates a new node. The rule will be executed when a new qualified number restriction has been added to a node. The rule will not be executed on the same qualified number restriction more than once. Furthermore, the rule can only create a finite number of nodes because the number of qualified number restrictions is bounded by the inequalities in the ontology. The rule $R_{\sqsubseteq_P}$ either immediately detects a subsumption between two concepts or creates a new clone node that will accumulate information that might possibly lead to a subsumption later. The rule will create only one clone node to test for subsumption between the concepts. In the worst case we will need to create a clone node for each pair of concepts. The rule $R_{\sqcup}$ creates new unfold nodes. This rule will called when a new disjunction has been added to the node. Thus it will not resolve the

same problem more than once. All three of the rules mentioned above are implemented using CPLEX. CPLEX will always return either a solution or the absence of thereof. Further, since the rules are designed to reuse existing nodes we do not need to implement *blocking* - the technique used in reasoner development in order to avoid construction of infinite models [42].

Thus the design of the rules and their execution process ensure that the algorithm terminates.

8.2 Soundness

In Table 8.1, Table 8.2, and Table 8.3 we summarize the inference rules in order to facilitate the reading of the proofs presented below. Each rule is composed of two parts - the **if** part or preconditions that must be satisfied for the rule to be applicable and the **then** part or the consequence of the rule application. More details on the rules can be found in Section 5.4.2. Furthermore, the symbols that are used in the rules are defined in Section 5.4.2 and summarized in Appendix 1.

Corollary 8.2.1 (Infeasibility) *Infeasibility of qualified number restrictions is tested using the IBM CPLEX software package. Qualified number restrictions in a node label are translated into a linear problem that is delegated to CPLEX. If the problem can be solved CPLEX will find a solution. CPLEX implements the SIMPLEX algorithm that has been proven to be sound, complete, and terminating [15]. More details on CPLEX and construction of linear programs from qualified number restrictions can be found in Chapter 7.*

Definition 8.2.1 (Subsumption) *Subsumption between two concepts A and B ($A \sqsubseteq B$) is represented by the presence of the concept B in the label $L(v_A)$.*

Definition 8.2.2 (Concept Satisfiability) *A concept A is satisfiable if the label of a node with the representative concept A is not subsumed by Bottom: $\bot \notin L(v_A)$.*

Definition 8.2.3 (Label) *By $\mathcal{L}(v)$ we denote the label $(L(v))^{\mathcal{I}}$ and by $\mathcal{Q}(v)$ we denote the label $(Q(v))^{\mathcal{I}}$. A label contains an intersection of all its elements: $\mathcal{L}(v) = \cap_{\phi \in L(v)} \phi^{\mathcal{I}}$. As a part of the initialization process for each node a corresponding label $L(v)$ has been created that contains the concept $\top$ as well as the representative concept of an identified node v. See Section 5.4.1 Definition c) for more details.*

Definition 8.2.4 (Label Satisfiability) *A node label $L(v)$ is satisfiable if it is not subsumed by bottom ($\mathcal{L}(v) \neq \emptyset$). Each rule application extends $L(v)$ thus creating $L'(v)$. According to Section 5.4.1 Definition c) if $L'(v)$ is satisfiable then $L(v)$ is also satisfiable since $L(v) \subseteq L'(v)$.*

Definition 8.2.5 (Graph Satisfiability) *A graph G is satisfiable if $\mathcal{L}(v_\top) \neq \emptyset$. Each rule application modifies a node in the graph G thus producing a new graph G'. If G' is satisfiable then G is also satisfiable since G is a subset of G'. As a result all entailed subsumptions will be computed. For instance, a label $L(v_A)$ will contain all subsumers of A.*

Definition 8.2.6 (TBox Satisfiability) *TBox is satisfiable if it satisfies all axioms. Satisfiability check of TBox is performed in order to verify whether $\top \sqsubseteq \bot$.*

Definition 8.2.7 (Graph Completeness) *A graph G is complete if no rule can be applied.*

Definition 8.2.8 (Model) *If a graph G is complete and satisfiable then a model can be derived from G.*

Theorem 8.2.1 (Soundness) *Let $\mathcal{T}$ be a satisfiable TBox and G a satisfiable graph for $\mathcal{T}$. Let G' be the graph after applying one of the rules. Then G' is also a satisfiable graph for $\mathcal{T}$.*

Proof

In the following we verify that each rule conclusion is indeed an entailment.

For the sake of simplicity when presenting proofs we assume that the rule is applicable.

Rule	Precondition	Consequence
$R_{\sqsubseteq}$	**if** $A \sqsubseteq \phi \in \mathcal{T}, \phi \notin L(v_A)$	**then** add ϕ to $L(v_A)$
$R_{\sqsubseteq\neq}$	**if** $A \in L(v), A \sqcap B \sqsubseteq \bot, \neg B \notin L(v)$	**then** add $\neg B$ to $L(v)$
$R_{\sqsubseteq *}$	**if** A $\in L(v), \tau \in L(v_A), \tau \notin L(v)$	**then** add τ to $L(v)$
$R_{\bowtie\bot}$	**if** $A \sqsubseteq \bowtie nR.C \in \mathcal{T}, \bowtie mR.D \in L(v)$, $infeasible(Q(v) \cup \{\bowtie nR.C\}), \neg A \notin L(v)$	**then** $L(v) \rightarrow L(v) \cup \{\neg A\}$
$R_{\bowtie\neg}$	**if** $A \sqsubseteq \bowtie nR.C \in \mathcal{T}, \dot{\neg}(\bowtie nR.C) \in L(v), \neg A \notin L(v)$	**then** $L(v) \rightarrow L(v) \cup \{\neg A\}$
$R_{\bowtie\sqsubseteq}$	**if** $\bowtie nR.A \sqsubseteq B \in \mathcal{T}$, $infeasible(Q(v) \cup \{\dot{\neg}(\bowtie nR.A)\}, B \notin L(v)$	**then** $L(v) \rightarrow L(v) \cup \{B\}$
$R_{\sqsubseteq\neg}$	**if** $\phi \sqsubseteq A \in \mathcal{T}, \neg A \in L(v), \dot{\neg}\phi \notin L(v)$	**then** add $\dot{\neg}\phi$ to L(v)
$R_{\sqsubseteq\sqcap}$	**if** $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{T}, \{A_1, A_2\} \subseteq L(v), B \notin L(v)$	**then** add B to $L(v)$
$R_{\sqsubseteq\bar{\sqcap}}$	**if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, \{\neg B, A_i\} \subseteq L(v), \neg A_j \notin L(v)$	**then** add $\neg A_j$ to $L(v)$
$R_{\bot}$	**if** $\bot \notin L(v) \wedge$ (infeasible$(L(v)) \vee \{A, \neg A\} \subseteq L(v))$	**then** add $\bot$ to (v)
R_{fil}	**if** $\langle r, q, n\rangle \in \sigma(v), \neg\exists v_q \in V : q \subseteq L(v_q), \#v_q \geq n$	**then** create $v_q \in V_A$ with $L(v_q) \leftarrow q$ and $\#v_q \leftarrow n$

TABLE 8.1: Summary of the Saturation-based Rules (ϕ = any subsumer except for a negated one, τ = any subsumer, r = role, q = qualified number restriction, n = cardinality)

$R_{\sqsubseteq}$: **if** $R_{\sqsubseteq}$ is applicable to the axiom $A \sqsubseteq \phi \in \mathcal{T}$, **then** ϕ can be added to $L(v_A)$ and $\phi^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$. According to our definition of a node label (see Section 5.4.1 Definition c)), we add a new subsumer to the intersection of all the subsumers in the label, thus restricting the label and creating a new superset $L'(v)$. If the $L'(v)$ is satisfiable then its subset $L(v)$ is also satisfiable.

$R_{\sqsubseteq\neq}$: **if** $R_{\sqsubseteq\neq}$ is applicable to the axiom $A \sqcap B \sqsubseteq \bot \in \mathcal{T}$ and $A \in L(v_A)$ **then** we can safely add $\neg B$ to $L(v_A)$. This statement holds because the input axiom entails that A and B are disjoint concepts. Since the above axiom is equivalent to $A \sqsubseteq \neg B$, it holds that $\neg B^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$.

$R_{\sqsubseteq_*}$: **if** $R_{\sqsubseteq_*}$ is applicable to the node v, $A^{\mathcal{I}} \subseteq \mathcal{L}(v)$ **and** $\tau^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$ **then** $\tau^{\mathcal{I}} \subseteq \mathcal{L}(v)$. This statement holds due to the Transitive property of subsumption that preserves satisfiability of $\mathcal{T}$ and allows us to derive transitive subsumers.

$R_{P_{\bowtie\perp}}$: **if** $R_{P_{\bowtie\perp}}$ is applicable to the axiom $A \sqsubseteq\bowtie nR.C \in \mathcal{T}$, $\bowtie mR.D \in L(v)$ **and** $infeasible(Q(v) \cup \{\bowtie nR.C\})$ **then** $\neg A^{\mathcal{I}} \subseteq \mathcal{L}(v)$. This statement holds because the axiom $A \sqsubseteq\bowtie nR.C$ is equivalent to $\top \sqsubseteq \neg A \sqcup \bowtie nR.C$ making this axiom applicable to every label. **If** $\bowtie nR.C$ clashes with any inequalities contained in the label $Q(v)$ **then** a new subsumer can be derived: $\neg A^{\mathcal{I}} \subseteq \mathcal{L}(v)$. The reason is that if one of the disjuncts clashes with elements of the label, then the other disjunct must be selected.

$R_{\bowtie\neg}$: **if** $R_{\bowtie\neg}$ is applicable to the axiom $A \sqsubseteq\bowtie nR.C \in \mathcal{T}$, $\dot\neg(\bowtie nR.C) \in L(v)$ **then** $\neg A^{\mathcal{I}} \subseteq \mathcal{L}(v)$. This statement holds due to the contrapositive reading of the above axiom. The rule is a special case of the rule $R_{P_{\bowtie\perp}}$ that facilitates the implementation. However, unlike $R_{P_{\bowtie\perp}}$, $R_{\bowtie\neg}$ does not rely on CPLEX, rather it searches the node label for qualified number restrictions and their negations.

$R_{\bowtie\sqsubseteq}$: **if** $R_{\bowtie\sqsubseteq}$ is applicable to the axiom $\bowtie nR.A \sqsubseteq B$ **and** *infeasible*$(Q(v) \cup \{\dot\neg(\bowtie nR.A)\})$ **then** B must be added to $L(v)$. This statement holds because the above axiom can be transformed into a general axiom $\top \sqsubseteq \dot\neg(\bowtie nR.A) \sqcup B$, making it applicable to every node. Since $\dot\neg(\bowtie nR.A)$ causes a clash in $L(v)$, it follows that B is entailed.

$R_{\sqsubseteq\neg}$: **if** $R_{\sqsubseteq\neg}$ is applicable to the axiom $\phi \sqsubseteq \neg A \in \mathcal{T}$ **and** $A \subseteq L(v)$ **then** $\neg\phi^{\mathcal{I}} \subseteq \mathcal{L}(v)$. This statement holds due to the contrapositive reading of the input axiom: $\phi \sqsubseteq \neg A$ is equivalent to $A \sqsubseteq \neg\phi$.

$R_{\sqsubseteq\sqcap}$: **if** $R_{\sqsubseteq\sqcap}$ is applicable to the axiom $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}$ **and** $L(v_{A_i}) \cup L(v_{A_j}) \subseteq L(v)$ **then** $B^{\mathcal{I}} \subseteq \mathcal{L}(v)$. This statement holds because a label represents an

intersection of all its elements. This allows us to derive new subsumptions based on the presence of binary subsumption axioms.

$R_{\sqsubseteq_\sqcap^\neg}$: **if** $R_{\sqsubseteq_\sqcap^\neg}$ is applicable to the axiom $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}$ **and** $\{\neg B, A_i\} \subseteq L(v_A)$ **then** $\neg A_j$ can be added to $L(v_A)$ and $\neg A_j^\mathcal{I} \subseteq \mathcal{L}(v_A)$. This statement holds because the above axiom is equivalent to $\top \sqsubseteq \neg A_i \sqcup \neg A_i \sqcup B$. **If** $\neg B \in L(v_A)$ and $A_i \in L(v_A)$ then since a disjunction represents a choice, the subsumptions $\{\neg B, A_i\} \subseteq L(v_A)$ will result in a clash in node v_A. Therefore the only viable choice is $\neg A_j \in L(v_A)$.

$R_\bot$: **if** $A, \neg A \subseteq L(v)$ **or** $infeasible(Q(v))$ **then** $\mathcal{L}(v) \subseteq \emptyset$. Thus a node can become unsatisfiable if it is subsumed by a concept and its complement or due to a clash in qualified number restrictions.

R_{fil}: **if** CPLEX returns a solution $< r, q, n > \in \sigma(v)$ **then** the rule R_{fil} is applicable to the node v. A new successor will be created that will connect the node v to a new node v_q via the edge r if it has not been created already. The edge will store the cardinality of the successor v_q. We rely on CPLEX to identify v_q and determine what edge should connect v to v_q. This is possible because CPLEX is sound, complete, and terminating [15]. This rule always creates a successor.

$R_{P_\bowtie}$: **if** $R_{P_\bowtie}$ is applicable to the axiom $\bowtie nR.C \sqsubseteq B$ **then** a *possible subsumer* can be created for the node v_B. The possible subsumer is needed in order to index information that will be used by the rule $R_{\sqsubseteq_P}$ to discover subsumptions between concepts.

The rule $R_{P_\bowtie}$ transforms axioms into global axioms and encodes them as *tuples* that consist of a set of *preconditions* and a set of inequalities. The default element in the set of preconditions is $\top$. The axiom above is equivalent to $\top \sqsubseteq \dot\neg(\bowtie nR.C) \sqcup B$. It will be encoded as a possible subsumer as

Rule	**Precondition**	**Consequence**
$R_{P_{\bowtie}}$	**if** $\bowtie n\, R.A \sqsubseteq B \in \mathcal{T}$, is_new($\dot{\neg}(\bowtie n\, R.A)$,$L_P(v_B)$)	**then** add_to($\dot{\neg}(\bowtie n\, R.A), L_P(v_B)$)
R_{P_*}	**if** $B \in L(v_A), L_P(v_A) \not\subseteq L_P(v_B)$	**then** add_to($L_P(v_A), L_P(v_B)$)
$R_{P_{\sqcap}}$	**if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, \varphi(A_i, L_P(v_{A_j})) \not\subseteq L_P(v_B)$	**then** add_to($\varphi(A_i, L_P(v_{A_j})), L_P(v_B)$)
$R_{P_{\bowtie Dyn}}$	**if** $A_i \sqcap A_j \sqsubseteq B \in \mathcal{T}, B \in L(v)$	**then** *add_to*($P, L_P(v_B)$)
$R_{\sqsubseteq_P}$	**if** $\{B, \neg B\} \cap L(v_A) = \emptyset$	**then** let $x \in \{B, \neg B\}$ **if** $v' \in clone(v_A, x)$ **then** **if** $\bot \in L(v')$ **then** add x to $L(v_A)$ **else** **if** $Q(v_A, x) \not\subseteq L(v')$ **then** add_to($Q(v_A, x), L(v')$) **if** $Q(v_A) \not\subseteq L(v')$ **then** add_to($Q(v_A), L(v')$) **elsif** $Q(v_A, x) \neq \emptyset$ **then** create $v' \in V_C, clone(v_A, x) \leftarrow \{v'\}$ $L(v') \leftarrow \{\top\} \cup Q(v_A, x) \cup Q(v_A)$

TABLE 8.2: Summary of the Possible Subsumers Rules (*is_new* = does not exist in the label)

$< \top, \{\dot{\neg}(\bowtie nR.C)\} >$ and added to the dedicated label $L_p(v_B)$ of the node B. A special function *add_to* has been introduced to manage addition of possible subsumers.

R_{P_*}: **if** $B \in L(v_A)$ and $L_P(v_A) \not\subseteq L_P(v_B)$ **then** possible subsumers in the label $L_P(v_A)$ should be added to the label $L_P(v_B)$. The rule propagates possible subsumers up the subsumption hierarchy. This statement holds due to the transitive property of subsumption: **if** $\bowtie R.C \sqsubseteq A$ and $A \sqsubseteq B$ **then** it implies that $\bowtie R.C \sqsubseteq B$. Therefore possible subsumers that were created for v_A should be also be created for or propagated to v_B.

$R_{P_{\sqcap}}$: **if** the rule $R_{P_{\sqcap}}$ is applicable to the axiom $A_i \sqcap A_j \sqsubseteq B$ **then** $L_P(v_{A_i})$ and $L_P(v_{A_j})$ can be added to the label L_P of the node v_B. This rule complements the rules $R_{P_{\bowtie}}$ and R_P by covering subsumptions caused by the presence of the binary subsumption axioms.

The rule $R_{P_\sqcap}$ ensures that in order for a node v to be subsumed by B via possible subsumers propagated from A_i, the node v has to be already subsumed by A_j. The function φ constructs possible subsumers to reflect this situation: A_i will be added to the preconditions of the possible subsumers of the node v_{A_j} and A_j to the preconditions of the possible subsumers of the node v_{A_i}. The newly constructed possible subsumers will then be added to $L_P(v_B)$.

$R_{P_{\bowtie Dyn}}$**: if** the rule $R_{P_{\bowtie Dyn}}$ is applicable to the axiom $A_i \sqcap A_j \sqsubseteq B$ then a *possible subsumer* can be created for the node v_B. This rule is an extension of the rule $R_{P_\sqcap}$ that also extracts information from binary subsumption axioms. The difference between the two rules is that $R_{P_\sqcap}$ processes one axiom at a time, while $R_{P_{\bowtie Dyn}}$ considers a group of axioms that result from the normalization process. More on that can be found in Section 5.4

For example, we have the following axioms where the conjuncts are auxiliary concepts generated during the normalization process:
$aux_1 \sqcap aux_2 \sqsubseteq A$
$aux_3 \sqcap aux_4 \sqsubseteq A$
Each auxiliary concept in the axioms above is a subsumer of a qualified cardinality restriction:
$q_1 \sqsubseteq aux_1$
$q_2 \sqsubseteq aux_2$
$q_3 \sqsubseteq aux_3$
$q_4 \sqsubseteq aux_4$

To represent different possible combinations of possible subsumers we create tuples from a cartesian product of $\{\{aux_1, aux_2\}, \{aux_3, aux_4\}\}$:
$< T, \{q_1, q_3\} >$

$< T, \{q_1, q_4\} >$

$< T, \{q_2, q_3\} >$

$< T, \{q_2, q_4\} >$

These tuples will be added to $L_P(v_B)$.

Next, let us slightly modify the example above by making only one conjunct to be an auxiliary concept:

$X_1 \sqcap aux_1 \sqsubseteq A$

$X_2 \sqcap aux_2 \sqsubseteq A$

Provided that the conjuncts aux_1 and aux_2 are still subsumers of qualified number restrictions:

$q_1 \sqsubseteq aux_1$

$q_2 \sqsubseteq aux_2$

This will then result in the following possible subsumer tuple:

$< T, X_1, X_2, \{q_1, q_2\} >$.

Those concepts that were not unfolded to qualified number restrictions became part of the preconditions.

$R_{\sqsubseteq_P}$: **if** the rule $R_{\sqsubseteq_P}$ is applicable to $L(v_A)$ and a concept B **then** we have to test whether A is subsumed by B or is disjoint to B.

Scenario $A \sqsubseteq B$: When we test for subsumption we take possible subsumers of $L_P(v_B)$ and $Q(A)$. If there is a clash in $\{L_P(v_B) \cup Q(A)\}$ then we can conclude that $A \sqsubseteq B$. If the subsumption cannot be concluded immediately then a dedicated *clone* node will be constructed that will hold this information until

more information will be accumulated that can possibly lead to a subsumption.

We test for subsumption between two concepts by transforming subsumption axioms into global axioms as it was described in the rule $R_{P_{\bowtie}}$. For example, take the pair of axioms: $A \sqsubseteq \geq 3R.C$ and $\geq 2R.C \sqsubseteq B$. We would like to test whether $A \sqsubseteq B$. We transform the second axiom into a global axiom: $\top \sqsubseteq \leq 1R.C \sqcup B$. Then we need to test whether $\leq 1R.C$ clashes with $\geq 3R.C$. The inequalities do indeed clash, which allows us to conclude that $A \sqsubseteq B$.

Scenario $A \not\sqsubseteq B$: Disjointness is a result of a clash between $Q(v_A)$ and $Q(v_B)$. If the inequalities in these labels clash then we can conclude that $A \sqcap B \sqsubseteq \bot$ or $\neg A^{\mathcal{I}} \subseteq \mathcal{L}(v_B)$ and $\neg B^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$. If we cannot immediately conclude disjointness due to the absence of a clash in qualified number restrictions, then we will create a dedicated *clone* node that will continue to accumulate information until disjointness can be concluded or until the reasoning process stops because the graph is saturated.

Rule	Precondition	Consequence
$R_{\sqsubseteq\neg}$	**if** $B \sqsubseteq \bigsqcup_{i=1}^{n} A_i \in \mathcal{T}$	**if** $\bigcup_{i=1}^{n}\{\neg A_i\} \subseteq L(v_A), \neg B \notin L(v_A)$ **then** add $\neg B$ to $L(v_A)$ **elsif** $\{\neg A_j, \bigsqcup_{i=1}^{n} A_i\} \subseteq L(v), j \in 1..n, Q \sqsubseteq B \in \mathcal{T}, Q \in \mathsf{C}_{\mathcal{T}}^{\mathsf{Q}}$, no auxiliary node v_U exists with $L(v_U) = \{U, \dot{\neg} Q\}$ **then** create $v_U \in V_X$, $L(v_U) = \{U, \dot{\neg} Q\}$ with U fresh in $\mathcal{T}$, and add $U \sqcup \bigsqcup_{i=1.i \neq j}^{n} A_i$ to $L(v)$
$R_{\sqsubseteq\sqcap}$	**if** $\bigsqcup_{i=1}^{n} A_i \in L(v), \bigcap_{i=1}^{n} L(A_i) \not\subseteq L(v)$	**then** add $\bigcap_{i=1}^{n} L(A_i)$ to $L(v)$
$R_{\sqcup}$	**if** $\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\} \subseteq L(v)$	*res* $\leftarrow$ resolvent$(\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\}, L(v))$ **if** *res* $\neq \emptyset$ **then** add *res* to $L(v)$

TABLE 8.3: Summary of the Disjunction Rules (*res* = the result of the resolution, resolvent = function that resolves disjunctions)

$R_{\sqsubseteq\neg}$: **if** $B \sqsubseteq \bigsqcup_{i=1}^{n} A_i \in \mathcal{T}$ is applicable to $v(A)$ then there are two possible scenarios:

Scenario 1: **if** $\bigcup_{i=1}^{n}\{\neg A_i\} \subseteq L(v_A)$ **then** $(\neg B)^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$. This statement holds due to the contrapositive reading of the above axiom. If a node label is subsumed by all the elements of the disjunction in Negated Normal Form then we can conclude that $L(v)$ also must be subsumed by the negated antecedent of the axiom.

Scenario 2:

else if $\{\neg A_j, \bigsqcup_{i=1}^{n} A_i\} \subseteq L(v), j \in 1..n, Q \sqsubseteq B \in \mathcal{T}, Q \in \mathsf{C}_{\mathcal{T}}^{\mathsf{Q}}$, no auxiliary node v_U exists with $L(v_U) = \{U, \dot{\neg} Q\}$ **then** create $v_U \in V_X$, $L(v_U) = \{U, \dot{\neg} Q\}$ with U fresh in $\mathcal{T}$, and add $U \sqcup \bigsqcup_{i=1.i\neq j}^{n} A_i$ to $L(v)$

If a node label is not subsumed by all the elements of the disjunction in Negated Normal Form then a dedicated node should be created that will hold information until a subsumption can be derived. The node serves a similar purpose as the clone node presented in the rule $R_{\sqsubseteq_P}$. This statement holds because **Scenario 2** works like **Scenario 1** but when the information needed to derive the antecedent is not present in $L(v)$ when the rule is applied.

$R_{\sqsubseteq^{\sqcap}_{\sqcup}}$: **given that** $R_{\sqsubseteq^{\sqcap}_{\sqcup}}$ is applicable to the axiom $\bigsqcup_{i=1}^{n} A_i \in L(v)$ **then** add $\bigcap_{i=1}^{n} L(A_i)$ to $L(v)$. This statement holds because this rule adds to $L(v)$ a non-empty intersection of the subsumers of the all the disjuncts of $\bigsqcup_{i=1}^{n} A_i$.

For example, $A \sqsubseteq C \sqcup D \in \mathcal{T}$, **if** $B \in L(v_C)$ **and** $B \in L(v_D)$ **then** $B^{\mathcal{I}} \subseteq \mathcal{L}(v_A)$.

$R_{\sqcup}$: **if** $\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\} \subseteq L(v)$ **then** *resolvent*$(\bigcup_{j=1}^{m}\{\bigsqcup_{i=1}^{n_j} A_{j_i}\})$ with $L(v)$. **If** *res* $\neq \emptyset$ **then** add it to $L(v)$. *resolvent* is the function that we use to resolve disjunctions with negated subsumers in node labels. For each disjunction in $L(v)$ we will remove from it all the disjuncts that clash with other concepts in $L(v)$. This way we end up with a smaller disjunction or even just a concept that will

be added as new subsumer to $L(v)$. This statement holds because the rule implements the Resolution rule [38].

For example, **if** $C \sqcup D \in L(v)$ **and** $\neg C \in L(v)$ **then if** we apply the rule $R_\sqcup$ to $L(v)$ **then** $D^\mathcal{I} \subseteq \mathcal{L}(v)$ where D is the *res*, as $C \sqcap \neg C \sqsubseteq \bot$.

8.3 Completeness

Theorem 8.3.1 (Completeness) *Let $\mathcal{T}$ be a satisfiable TBox and G a complete, satisfiable graph for $\mathcal{T}$. Then all entailments of $\mathcal{T}$ are represented in G.*

Proof Let us assume that we have a complete graph G and there is an entailed subsumption $A \sqsubseteq \tau$ that is not present in the graph, i.e. $\tau \notin L(v_A)$. Below we will prove by contradiction that it cannot happen because all the rules will be triggered and derive every subsumption that is entailed by the axioms in the TBox $\mathcal{T}$.

According to our normal form presented in Section 5.3 τ must occur in at least one axiom of $\mathcal{T}$ on its left-hand side or right-hand side and it can be one of the following four instances: an atomic concept, a negated atomic concept, a disjunction or a qualified number restriction. We will prove for each instance of τ that our rules are complete.

Missing subsumption: τ is an atomic concept

Let us assume that τ is an atomic concept and $\tau \notin L(v_A)$. This cannot happen because one of the following rules will discover the missing subsumption:

$R_\sqsubseteq$: let us assume that there is an axiom $A \sqsubseteq \tau$. Then the rule $R_\sqsubseteq$ will fire and add τ to $L(v_A)$ thus contradicting the statement that $\tau \notin L(v_A)$.

$R_{\sqsubseteq_*}$: let us assume that there are node labels A_1 and A_2 such that $A_1 \in L(v_{A_2})$ and $\tau \in L(v_{A_1})$. Then the rule $R_{\sqsubseteq_*}$ will fire and add the transitive subsumer τ to $L(v_{A_2})$.

$R_{\sqsubseteq_\sqcap}$: let us assume that there is an axioms $A_1 \sqcap A_2 \sqsubseteq \tau$ and both A_1 and A_2 can be found in $L(v_A)$. Then the rule $R_{\sqsubseteq_\sqcap}$ will fire and add τ to $L(v_A)$.

$R_{\sqsubseteq_P}$: let us assume that there are $L(v_A)$ and $L(v_\tau)$ and it is entailed by the TBox T that A is subsumed by τ due to the presence of number restrictions in $L(v_A)$ and axioms of the form of $q \sqsubseteq \tau$ where q is a qualified number restriction. Then the rules $R_{P_\bowtie}$, $R_{P_\sqcap}$, or $R_{P_{\sqcap_{Dynamic}}}$ will fire and create the necessary *possible subsumers* and as a result the rule $R_{\sqsubseteq_P}$ will add τ to $L(v_A)$. The subsumption will be discovered because the rule $R_{\sqsubseteq_P}$ will create a special node $clone_{A \sqsubseteq \tau}$ that will contain all number restrictions from $L(v_A)$ as well as all possible subsumers generated for v_τ. The ILP module will be subsequently called and since the subsumption is entailed the module will detect that the inequalities in the clone node are infeasible. As a result τ will be added to $L(v_A)$.

Consider the following three examples that demonstrate the interaction between the rules $R_{P_\bowtie}$, $R_{P_\sqcap}$, $R_{P_{\sqcap_{Dynamic}}}$ and the rule $R_{\sqsubseteq_P}$. These rules are crucial for completeness of the calculus as they permit us to stay complete without the need of backtracking.

Example 1. This example demonstrates how the rules $R_{P_\bowtie}$ and $R_{\sqsubseteq_P}$ are applied to discover subsumptions between concepts. Let us assume that $\geq 3R.C \in L(v_A)$ and that there is the axiom $\geq 2R.C \sqsubseteq B$ in the TBox $\mathcal{T}$. It is entailed by $\mathcal{T}$ that A is subsumed by B. If $B \notin L(v_A)$ then the rules $R_{P_\bowtie}$ and $R_{\sqsubseteq_P}$ will derive it. The rule $R_{P_\bowtie}$ will create a possible subsumer for B that will be r $< \top, \leq 1R.C >$. Then the rule $R_{\sqsubseteq_P}$ will create a clone node $clone_{A \sqsubseteq B}$. The

clone node will contain two number restrictions: $\geq 3R.C$ and $\leq 1R.C$. The ILP module will be called and it will detect that these inequalities are infeasible. As a result B will be added to $L(v_A)$.

Example 2. This example demonstrates how the rules $R_{P_\sqcap}$ and $R_{\sqsubseteq_P}$ are applied to discover subsumptions between concepts. Let us assume that there are the axioms $B_1 \sqcap B_2 \sqsubseteq B$ and $\geq 2R.C \sqsubseteq B_1$ in the TBox $\mathcal{T}$. Further, $B_2 \in L(v_A)$ and $\geq 3R.C \in L(v_A)$. It is entailed by $\mathcal{T}$ that A is subsumed by B. In order to derive it the rules will work as follows. The rule $R_{P_\bowtie}$ will add the possible subsumer $< \top, \leq 1R.C >$ to $L_P(B_1)$. Then the rule $R_{P_\sqcap}$ will add a possible subsumer $< \{\top, B_2\}, \leq 1R.C >$ to B to represent the binary subsumption axiom. Finally, the rule R_P will create a clone node $clone_{A \sqsubseteq B}$ that will contain $\leq 1R.C$ and $\geq 3R.C$. The ILP module will be called on the clone node and it will detect that its inequalities are infeasible and add B to $L(v_A)$.

Example 3. This example demonstrates how the rules $R_{P_{\sqcap_{Dynamic}}}$ and and $R_{\sqsubseteq_P}$ are applied to discover subsumptions between concepts. Let us assume that there are the axioms $aux_1 \sqcap aux_2 \sqsubseteq B_1$, $aux_3 \sqcap aux_4 \sqsubseteq B_2$. Furthermore, $\geq 3R.C \in L(v_A)$, $\geq 1R.C \in aux_1$, $\geq 1R.C \in aux_3$. It is entailed by the TBox $\mathcal{T}$ that $B \in L(v_A)$. This inference will be derived as follows: first, the rule $R_{P_\bowtie}$ will create possible subsumers for the applicable concepts. Then the rule $R_{P_{\sqcap_{Dynamic}}}$ will create possible subsumers for B_1 and B_2. This rule compared to $R_{P_\sqcap}$ considers more than one binary subsumption axiom. Finally, the rule R_P will create a clone node $clone_{A \sqsubseteq B}$. The ILP module will be called and it will detect a clash in the number restrictions in the clone node. As a result B will be added to $L(v_A)$.

$R_{\sqsubseteq^{\sqcap}_{\sqcup}}$**:** let us assume that there is $L(v_A)$ and $C \sqcup D$ in $L(v_A)$ with every disjunct being subsumed by B, i.e. $B \in L(v_C)$ and $B \in L(v_D)$. Then the rule $R_{\sqsubseteq^{\sqcap}_{\sqcup}}$ will fire and

add τ to $L(v_A)$.

$R_{\sqcup}$: let us assume that there is $L(v_A)$ so that $\tau \sqcup B \in L(v_A)$ and $\neg B \in L(v_A)$. Then the rule $R_{\sqcup}$ will fire and add τ to $L(v_A)$.

$R_{\perp}$: let us assume that there is $L(v_A)$ so that $L(v_A)$ contains either a concept and its complement or clashing cardinality restrictions then the rule $R_{\perp}$. Then the rule $R_{\perp}$ will fire and add $\perp$ to the label $L(v_A)$.

R_{fil}: let us assume that there is $L(v_A)$ so that $L(v_A)$ contains at least two number restrictions where one is an *at-least* number restriction and another one is an *at-most* number restriction. If the rule R_{fil} cannot immediately detect clash between the inequalities it will create an *anonymous* node that will store these inequalites. If eventually this node becomes unsatisfiable due to infeasibility of the inequalities then the rule R_{fil} will detect it and mark v_A and the anonymous node as unsatisfiable by adding $\perp$ to their subsumers. The infeasibility will be determined by leveraging the ILP module. Furthermore, the system will also learn that the given combination of number restrictions is infeasible and if there is another node that triggers the rule R_{fil} with the same number restrictions it will also become unsatisfiable by referring to the unsatisfiable anonymous node.

Missing subsumption: $\neg\tau$ is a negation of an atomic concept τ

Let us assume that $\neg\tau$ is a negated atomic concept and $\neg\tau \notin L(v_A)$. This cannot happen because one of the following rules will discover the missing subsumption:

$R_{\sqsubseteq}$: if the rule $R_{\sqsubseteq}$ has been applied to the axiom $A \sqsubseteq \neg\tau$ then the rule will add $\neg\tau$ to $L(v_A)$ thus contradicting the statement that $\neg\tau \notin L(v_A)$.

$R_{\sqsubseteq\neg}$: if the rule $R_{\sqsubseteq\neg}$ has been applied to the axiom $A \sqcap \tau \sqsubseteq \bot$ then the rule will add $\neg\tau$ to $L(v_A)$.

$R_{\sqsubseteq P}$: if the rule $R_{\sqsubseteq P}$ has been applied to $L(v_A)$ and $L(v_\tau)$ provided that $q_1 \in L(v_A)$ and $q_2 \in L(v_\tau)$ where q_1 and q_2 are clashing cardinality restrictions then the rule will add $\neg\tau$ to $L(v_A)$.

$R_{\sqsubseteq *}$: if the rule $R_{\sqsubseteq *}$ has been applied to $L(v_{A_1})$ and $L(v_{A_2})$ provided that $A_1 \in L(v_{A_2})$ and $\neg\tau \in L(v_{A_1})$ then the rule will add the transitive subsumer $\neg\tau$ to $L(v_{A_2})$.

$R_{\sqsubseteq^{\cap}_{\sqcup}}$: if the rule $R_{\sqsubseteq^{\cap}_{\sqcup}}$ has been applied to $L(v_A)$ provided that $C \sqcup D \in L(v_A)$, $\neg\tau \in L(v_C)$ and $\neg\tau \in L(v_D)$ then the rule will add $\neg\tau$ to $L(v_A)$.

$R_{\sqsubseteq^{\neg}_{\sqcup}}$: if the rule $R_{\sqsubseteq^{\neg}_{\sqcup}}$ has been applied to the axiom $A \sqcap \tau \sqsubseteq B$ provided that $A \in L(v_A)$ and $\neg B \in L(v_A)$ then the rule will add $\neg\tau$ to $L(v_A)$.

$R_{\bowtie\bot}$: if the rule $R_{\bowtie\bot}$ has been applied the axiom $\tau \sqsubseteq q$ where q is a cardinality restriction and q clashes with cardinality restrictions in $L(v_A)$ then the rule will add $\neg\tau$ to $L(v_A)$.

Missing subsumption: τ is a disjunction

Let us assume that τ is a disjunction and $\tau \notin L(v_A)$. This cannot happen because one of the following rules will discover the missing subsumption:

$R_{\sqsubseteq}$: if the rule $R_{\sqsubseteq}$ has been applied to the axiom $A \sqsubseteq \tau$ then the rule will add τ to the label $L(v_A)$ thus contradicting the statement that $\tau \notin L(v_A)$.

$R_{\sqsubseteq *}$: if the rule $R_{\sqsubseteq *}$ has been applied to the labels A_1 and A_2 so that $A_1 \in L(v_{A_2})$ and $\tau \in L(v_{A_1})$ then the rule will add τ to the label $L(v_{A_2})$.

$R_{\sqcup}$: if the rule $R_{\sqcup}$ has been applied to the label $L(v_A)$ that contains two disjunctions that can be resolved into a smaller disjunction τ then the rule will add τ to the label $L(v_A)$.

Missing subsumption: τ is a cardinality restriction

Let us assume that τ is a cardinality restriction and $\tau \notin L(v_A)$. This cannot happen because one of the following rules will discover the missing subsumption:

$R_{\sqsubseteq}$: if the rule $R_{\sqsubseteq}$ has been applied to the axiom $A \sqsubseteq \tau$ then the rule will add τ to $L(v_A)$ thus contradicting the statement that $\tau \notin L(v_A)$.

$R_{\sqsubseteq *}$: if the rule $R_{\sqsubseteq *}$ has been applied to the labels $L(v_A)$ and $L(v_B)$ so that $B \in L(v_A)$ and $\tau \in L(v_B)$ then the rule will add τ to $L(v_A)$.

Chapter 9

Complexity Analysis

In this chapter we will discuss the complexity of the proposed calculus that has been implemented in the reasoner Avalanche.

It has already been proven that the computational complexity of the description logic $\mathcal{ALCHQ}$ for which we created our calculus is **ExpTime-Complete** [29].

The time complexity of the Simplex algorithm implemented in CPLEX is exponential in the worst case scenario[34]. As it has been shown by Klee and Minty, the Simplex algorithm may need to use an exponential number of pivot rules [34] in order to find the solution of an integer linear program.

The time complexity of the proposed calculus, as implemented in Avalanche, is double exponential in the worst case as the algorithm needs to apply rules to each node that are stored in the node's rules queue. Each rule execution adds more rules to the queue. Some rules need to call CPLEX that is worst case exponential. Thus the time complexity becomes double exponential in the worst case.

During the reasoning process the algorithm has to create a certain number of internal nodes. Different types of nodes serve different purposes and are needed to ensure the algorithm is sound and complete.

In order to test for subsumption between two named concepts the algorithm will

need to create clone nodes. In the worst case it will create at most $n^2_{pos_clone}$ positive clone nodes and also at most $n^2_{neg_clone}$ negative clone nodes. As the positive and negative clones are needed to test for subsumption and disjointness between concepts in the worst case the algorithm will have to test all possible pairs of concepts.

Furthermore, in the worst case the algorithm will create one anonymous node per *at-least* cardinality restriction in a node. Thus in the worst case we will have at most n_{anonym} anonymous nodes where n_{anonym} is the number of *at-least* cardinality restrictions in the ontology.

The algorithm will also create at most n^2_{unfold} unfold nodes for each node that contains at least two disjunctions in the attempt to resolve two disjunctions. In this case n_{unfold} is a number of disjunctions in a node.

Thus it implies that the worst-case complexity of the proposed calculus is double exponential as it is defined below. However, by leveraging CPLEX we are able to process ontologies with qualified number restrictions in the best-case polynomial time [62].

$$2^{2^{n_{rules}}} + 2^{n_{nodes}} + n^2_{pos_clone} + n^2_{neg_clone} + n_{anonym} + n^2_{unfold}$$

Chapter 10

Performance Evaluation

In this chapter we will discuss performance evaluation of Avalanche compared to other prominent description logic reasoners such as Fact++, Hermit, JFact, Konclude, and Racer.

In the sections below we compare performance of the reasoners on six sets of benchmarks and use graphs to visualize the results. Each of the six graphs represents the CPU time needed for each reasoner to classify the benchmark ontologies. The y-axis represents the CPU time in seconds and the x-axis represents the benchmark ontologies. We use the logarithmic scale for the y-axis and set classification timeout for each ontology at 1000 seconds.

When we started to implement our reasoner we needed test ontologies that would contain interacting qualified number restrictions that would lead to subsumptions in order to not only evaluate the reasoner's runtime as well as to ensure that it is sound, complete, and terminating. Unfortunately due to our specific needs we could not find the exact ontologies that would meet our criteria. For this reason we had to modify existing ontologies and to adapt them to our needs. In particular we adapted benchmarks from [18]. We also designed some benchmarks ourselves. Therefore the performance of Avalanche will be compared to the performance of

other reasoners on the benchmarks that were either designed or modified by us as we compare them based on their ability to classify ontologies that focus on entailments based on qualified number restrictions.

Briefly, the results indicate that Avalanche performs better than other reasoners when classifying ontologies with interacting qualified number restriction that result in entailments thus achieving the main goal of our research. When other reasoners time out Avalanche is the only one that can classify all the ontologies. We believe that we managed to accomplish this due to the adoption of the Branch-and-Price algorithm.

The complete results that include exact runtimes that each reasoner needed to classify benchmark ontologies can be found in Appendix C.

We used the following reasoner versions: Fact++ 1.6.3, Hermit 1.3.8.1099, JFact 1.2.2, Konclude 0.6.2-544, and Racer 3.0.

We ran the benchmarks on the HP DL580 Scientific Linux SMP server. The server has four 15-core processors (Gen8 Intel Xeon E7-4890v2 2.8 GHz). Each processor has 256GB of shared RAM. The total RAM is 1TB. The cores support hyperthreading.

10.1 Canadian Parliament Benchmarks

The Canadian Parliament benchmarks represent a real-life situation. They model provinces, their residents, and factions. These benchmark ontologies contain definitions of what it is to be a resident of a province and how many members can have factions of different size. These ontologies were derived and adapted for our needs from [18].

Both $\mathcal{ALCQ}$ and $\mathcal{ELQ}$ benchmarks model the Canadian Parliament. $\mathcal{ELQ}$ is equivalent to $\mathcal{ALCQ}$ but the syntax of $\mathcal{ELQ}$ is restricted. $\mathcal{ALCQ}$ allows qualified restrictions, existential and universal restrictions, conjunctions, disjunctions, and negations of concepts and expressions but $\mathcal{ELQ}$ allows only qualified restrictions, conjunctions, and existential restrictions.

10.1.1 Performance Evaluation for $\mathcal{ALCQ}$ Ontologies

This is an extract from one of the Canadian Parliament benchmarks - the canadian-parliament-ALCQ ontology. The ontology models all the provinces of Canada and classifies them into tiny, small, medium, or big factions based on the number of members. Below we will show how it is modelled for the province of Alberta:

First we define that Alberta (AB) is a Canadian province.

$$AB \sqsubseteq CanadianProvince$$

The Alberta faction (ABfaction) should have 28 members that are residents of Alberta (ABres).

$$\forall hasMember.ABres \sqcap \geq 21\, hasMember.ABres \sqcap = 28\, hasMember.ABres \sqsubseteq ABfaction$$

A faction is of medium size if it has between 16 and 40 members.

$$\forall hasMember.CanRes \sqcap \geq 16\, hasMember.CanRes \sqcap \leq 40\, hasMember.CanRes \sqsubseteq mediumProvinceFaction$$

Thus, it is entailed that the Alberta faction is a medium size faction.

The other Canadian Parliament benchmarks contain different variations of this ontology that contain exactly qualified cardinality restrictions, existential cardinality restrictions, and disjunctions.

Given that Canada has only 10 provinces these ontologies are rather small and also appear to be not very computationally difficult. The ontology we used in the example above contains only 35 axioms. However, not every reasoner can classify these ontologies in an acceptable amount of time.

$\mathcal{ALCQ}$ benchmarks contain 8 ontologies. In the Table 10.1 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor of Avalanche. The speedup factor of Avalanche compared to another reasoner is defined by dividing the total runtime of the reasoner by the total runtime of Avalanche. According to the speedup factor Avalanche is one order of magnitude faster than any other reasoner except for Racer that is a highly optimized system.

	CPU time	Speedup factor of Avalanche
Avalanche	539.94	1.0
Fact++	7952.78	14.7
Hermit	8001.86	14.8
JFact	8406.36	15.5
Konclude	5947.59	11.0
Racer	84.19	0.1

TABLE 10.1: Total CPU time and speedup factor for $\mathcal{ALCQ}$ benchmarks

As it can be observed in Figure 10.1, the performance of Avalanche is very stable. Konclude starts rather well but quickly times out like Hermit, JFact, and Fact++. Racer initially is very fast but eventually it plateaus.

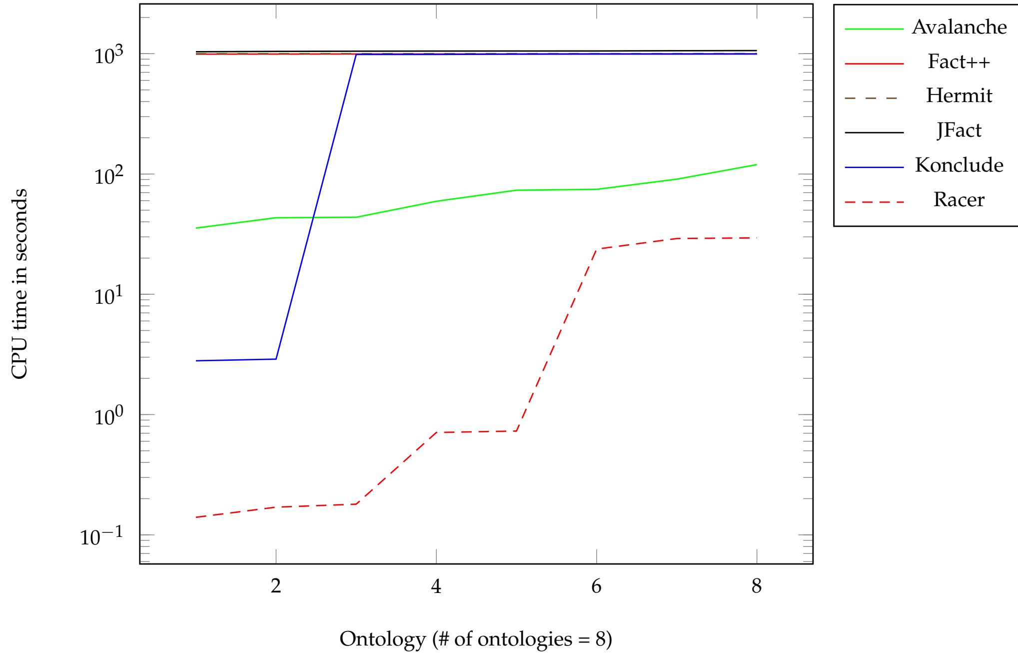

FIGURE 10.1: $\mathcal{ALCQ}$ benchmark runtimes in seconds

10.1.2 Performance Evaluation for $\mathcal{ELQ}$ Ontologies

The $\mathcal{ELQ}$ ontologies contain variations of the $\mathcal{ALCQ}$ ontologies expressed in $\mathcal{ELQ}$ syntax. For example, universal restrictions have to be replaced with equivalent qualified number restrictions. Negations and disjunctions should also be expressed by means of qualified number restrictions. Below we model that all members of the medium size faction should be Canadian residents or that there can be no more than 0 non-Canadian residents.

$$\geq 16\, hasMember.CanRes \sqcap \leq 0\, hasMember.nonCanRes \sqcap \leq 40\, hasMember.CanRes \sqsubseteq mediumProvinceFaction$$

Thus it could be expected that the $\mathcal{ELQ}$ benchmarks should not be more difficult to classify than the $\mathcal{ALCQ}$ benchmarks but in reality the syntax changes negatively affect performance of all the reasoners. These benchmarks contain 22 ontologies. In

the Table 10.2 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor. As Avalanche was designed to deal with qualified number restrictions for these ontologies it is not only the fastest reasoner but also the only reasoner that does not time out.

	CPU time	Speedup factor of Avalanche
Avalanche	1.402	1.0
Fact++	22.005	15.6
Hermit	22.007	15.6
JFact	23.779	16.9
Konclude	20.989	14.9
Racer	6.024	4.2

TABLE 10.2: Total CPU time and speedup factor for $\mathcal{ELQ}$ benchmarks

As it can be seen in Figure 10.2 Hermit cannot classify any of the $\mathcal{ELQ}$ ontologies.

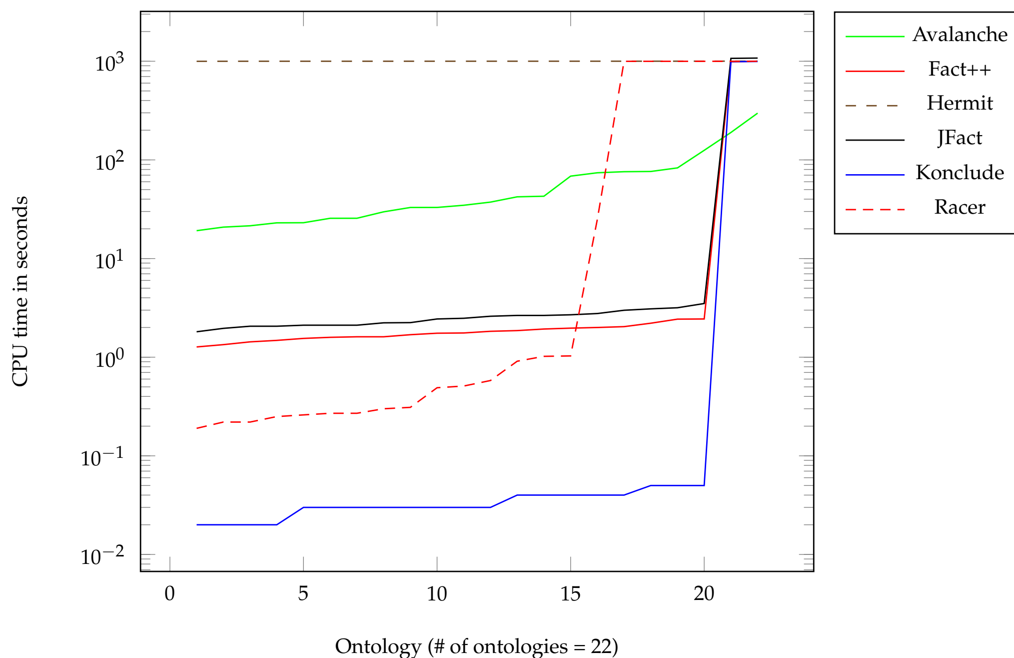

FIGURE 10.2: $\mathcal{ELQ}$ benchmark runtimes in seconds

10.2 Satisfiable and Unsatisfiable $\mathcal{ALCHQ}$ Benchmarks

These ontologies were derived and adapted for our needs from the benchmarks used in [18]. In addition to these ontologies also contain role hierarchies ($\mathcal{H}$).

Both satisfiable and unsatisfiable $\mathcal{ALCHQ}$ ontologies are small ontologies that model different entailments due to interactions between qualified restrictions. The example below is a typical benchmark ontology.

$$\leq 9\, r.\neg(A) \sqcup \leq 10\, r.\neg(B) \sqcap \geq 20\, r.(A \sqcup B) \sqcap \leq 10\, r.A \sqcap \leq 10\, r.B \sqsubseteq C$$

The other benchmark ontologies model different interactions between qualified number restrictions, or contain bigger cardinalities, e.g. 1000 instead of 10, or contain different combinations of negations and disjunctions. Still, most reasoners find it hard to classify these ontologies. Avalanche is the only reasoner that does not time out. We believe that these benchmarks show the true potential of applying Linear Programming together with the Branch-and-Price algorithm to semantic reasoning because this is the reason why Avalanche is superior to other reasoners for these benchmarks.

10.2.1 Performance Evaluation for Satisfiable $\mathcal{ALCHQ}$ Ontologies

Satisfiable $\mathcal{ALCHQ}$ benchmark contains 147 ontologies. They are called satisfiable because we are testing satisfiability of the concept Thing that should be satisfiable.

In the Table 10.3 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor. Avalanche is the fastest reasoner and one order of magnitude faster than Hermit.

	CPU time	Speedup factor of Avalanche
Avalanche	5160.30	1.0
Fact++	21240.69	4.1
Hermit	95003.48	18.4
JFact	25238.94	4.8
Konclude	27047.94	5.2
Racer	14854.21	2.8

TABLE 10.3: Total CPU time and speedup factor for Sat-$\mathcal{ALCHQ}$ benchmarks

Figure 10.3 illustrates that Avalanche is the only reasoner that is capable of classifying all the benchmark ontologies. All the other tested reasoners time out.

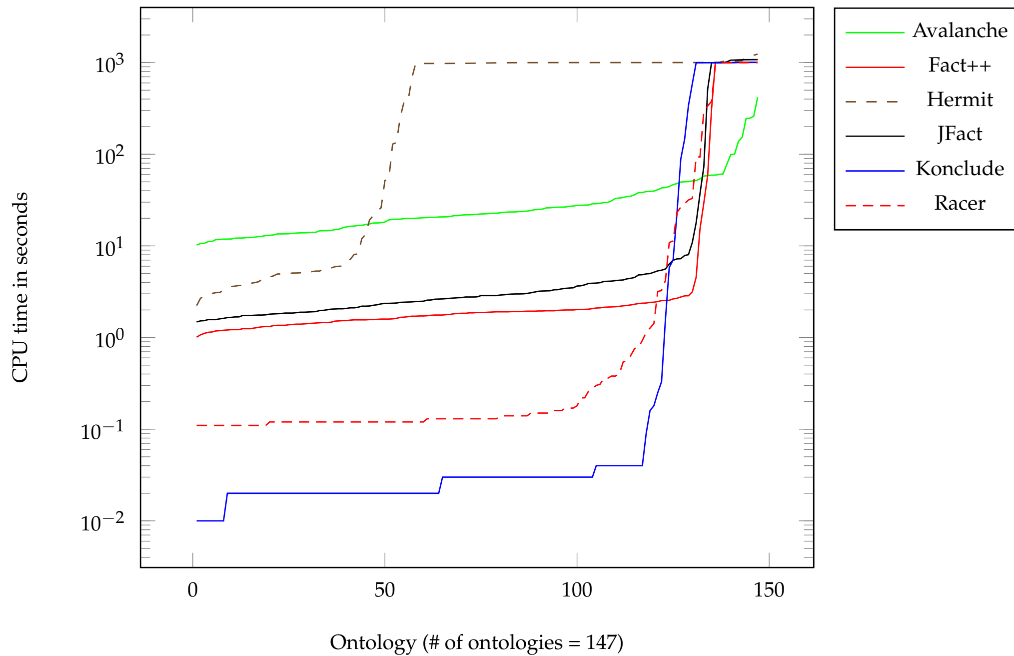

FIGURE 10.3: Sat-$\mathcal{ALCHQ}$ benchmark runtimes in seconds

10.2.2 Performance Evaluation for Unsatisfiable $\mathcal{ALCHQ}$ Ontologies

Unsatisfiable $\mathcal{ALCHQ}$ benchmark contains 99 unsatisfiable ontologies. The difference between the satisfiable and the unsatisfiable benchmarks is that the concept Thing should be unsatisfiable if all the entailments have been discovered. In the Table 10.4 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor. Avalanche is the fastest reasoner and two orders of magnitude faster than Hermit, one order of magnitude faster than Konclude, JFact, and Fact++.

	CPU time	Speedup factor of Avalanche
Avalanche	1899.91	1.0
Fact++	18770.67	9.8
Hermit	65292.20	34.3
JFact	21692.30	11.4
Konclude	21066.03	11.0
Racer	8046.46	4.2

TABLE 10.4: Total CPU time and speedup factor for Unsat-$\mathcal{ALCHQ}$ benchmarks

The results presented in Figure 10.4 illustrate that Avalanche is the only reasoner that can classify all the benchmark ontologies.

10.3 Performance Benchmarks

Performance benchmarks were developed by us to validate classification results produced by Avalanche. These are small ontologies that test for subsumptions due to different interactions between concepts. This test suite grew from a dozen to more than 150 ontologies over the course of our work. Therefore we decided to include them into this thesis as they also give interesting insights on performance of other

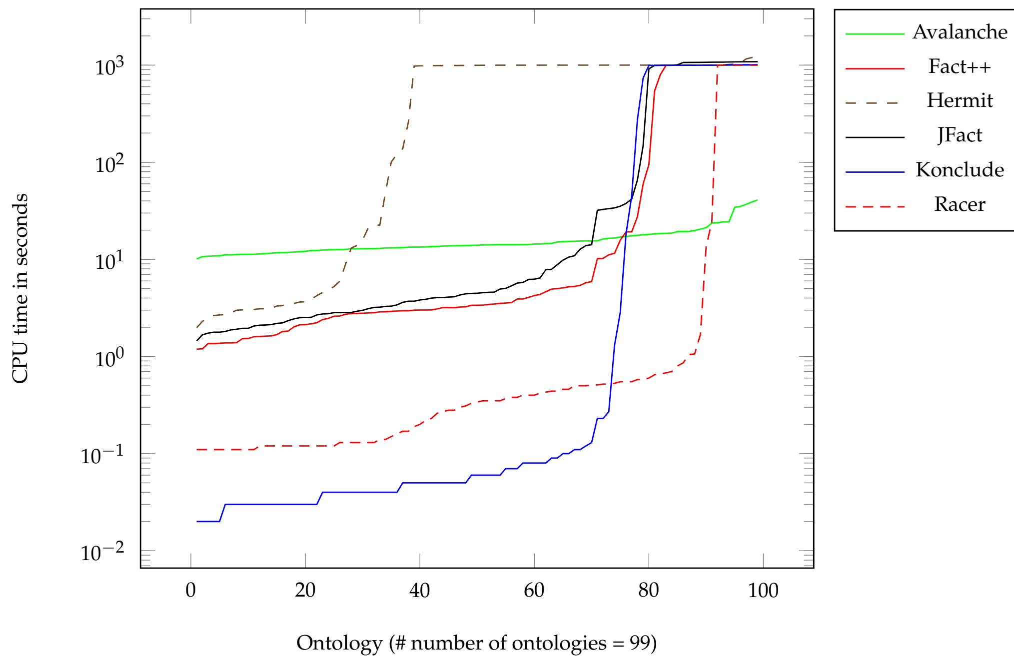

FIGURE 10.4: Unsat-$\mathcal{ALCHQ}$ benchmark runtimes in seconds

reasoners as we did not expect that our simple examples could not be classified in an acceptable amount of time by other reasoners.

Each ontology is designed to test for subsumption between two concepts. However, in order to discover the subsumption other subsumptions should be discovered first. We have two benchmark sets - first set contains ontologies where Thing is satisfiable and second set contains ontologies where Thing should become unsatisfiable if all entailments have been discovered.

10.3.1 Performance Evaluation for Satisfiable Performance Benchmarks

Satisfiable Performance benchmark contains 179 ontologies. Below we will present one of the typical benchmark ontologies to give a general idea of the benchmark

collection. In the example it is entailed that $C \sqsubseteq D$.

$$A \sqcap B \sqsubseteq E$$

$$C \sqsubseteq \geq 1\, R.A \sqcap \geq 1\, R.B \sqcap \leq 1\, R.Thing$$

$$\geq 1\, R.F \sqsubseteq D$$

$$\geq 1\, R.F \sqsubseteq D$$

As the example above does not have complicated entailments due to qualified number restrictions all the reasoners are able to classify it without any issues. Below we will present another benchmark ontology that does not appear to be very difficult but it can only be classified by Avalanche and Racer due to interacting qualified number restrictions. As it can seen an ontology should not contain many axioms or have big cardinality values in order to be too difficult to be classified within a reasonable amount of time. The examples like this prove that the idea behind Avalanche has a great potential for reasoner development.

$$AB \sqcap (\geq 1\, s1.(XA \sqcap XB)) \sqsubseteq A$$

$$\neg A \sqsubseteq notA$$

$$(\geq 1\, s1.XA) \sqcap (\geq 1\, s1.XB)$$

$$AB \sqcap (\geq 1\, s1.(XA \sqcap XB)) \sqsubseteq A$$

$$AB \sqcap (\geq 1\, s1.Thing) \sqsubseteq B$$

$$\neg B \sqsubseteq notB$$

$$(\geq 20\, r.AB) \sqcap (\leq 10\, r.A) \sqcap (\leq 10\, r.B)$$

$$(\geq 1\, s2.XA) \sqcap (\geq 1\, s2.XB)$$

$$C \sqcap (\geq 1\, s2.(XA \sqcap XB)) \sqsubseteq X2$$

$$C \sqcap (\geq 2\, s2.Thing) \sqsubseteq X1$$

$$\neg A \sqsubseteq notA$$

$$\neg B \sqsubseteq notB$$

$$\leq 9\, r.notA$$

$$C \sqcap (\geq 2\, s2.Thing) \sqsubseteq X1$$

$$\leq 10\, r.notB$$

$$C \sqcap (\geq 1\, s2.(XA \sqcap XB)) \sqsubseteq X2$$

$$AB \sqcap (\geq 1\, s1.(XA \sqcap XB)) \sqsubseteq A$$

$$C \sqcap (\geq 1\, s2.(XA \sqcap XB)) \sqsubseteq X2$$

$$AB \sqcap (\geq 1\, s1.(XA \sqcap XB)) \sqsubseteq A$$

$$C \sqcap (\geq 1\, s2.(XA \sqcap XB)) \sqsubseteq X2$$

In the Table 10.5 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor. Avalanche is the second fastest reasoner after Racer.

The results Figure 10.5 illustrate that only Avalanche and Racer can classify all the ontologies and four other reasoners time out.

	CPU time	Speedup factor of Avalanche
Avalanche	4.703	1.0
Fact++	12.376	2.6
Hermit	18.540	3.9
JFact	14.590	3.1
Konclude	13.923	2.9
Racer	729	0.16

TABLE 10.5: Total CPU time and speedup factor for Sat-$\mathcal{ALCHQ}$ benchmarks

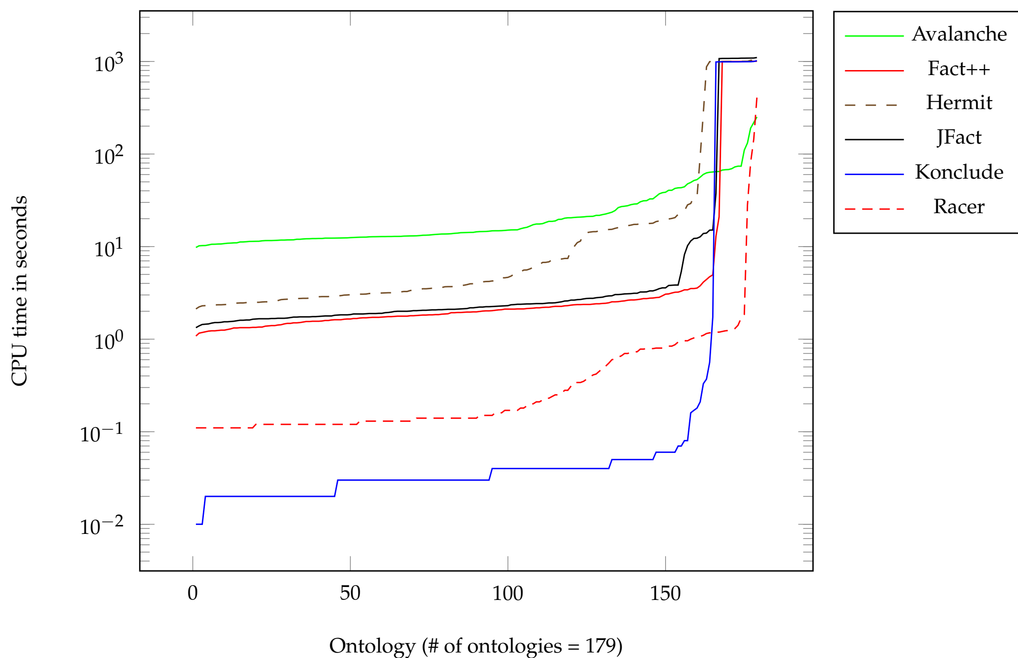

FIGURE 10.5: P-Sat benchmark runtimes in seconds

10.3.2 Performance Evaluation for Unsatisfiable Performance Ontologies

Unsatisfiable Performance benchmark contains 155 ontologies. In the Table 10.6 we present for each reasoner the total CPU time in seconds needed to classify all the benchmarks and the speedup factor. Although Racer has a better CPU time than Avalanche it cannot classify all the ontologies and times out for some of them.

	CPU time	Speedup factor of Avalanche
Avalanche	2.698	1.0
Fact++	16.309	6.0
Hermit	22.159	8.2
JFact	17.607	6.5
Konclude	16.562	6.1
Racer	2.195	0.8

TABLE 10.6: Total CPU time and speedup factor for Unsat-$\mathcal{ALCHQ}$ benchmarks

The results presented in Figure 10.6 illustrate that Avalanche is the only reasoner that does not time out.

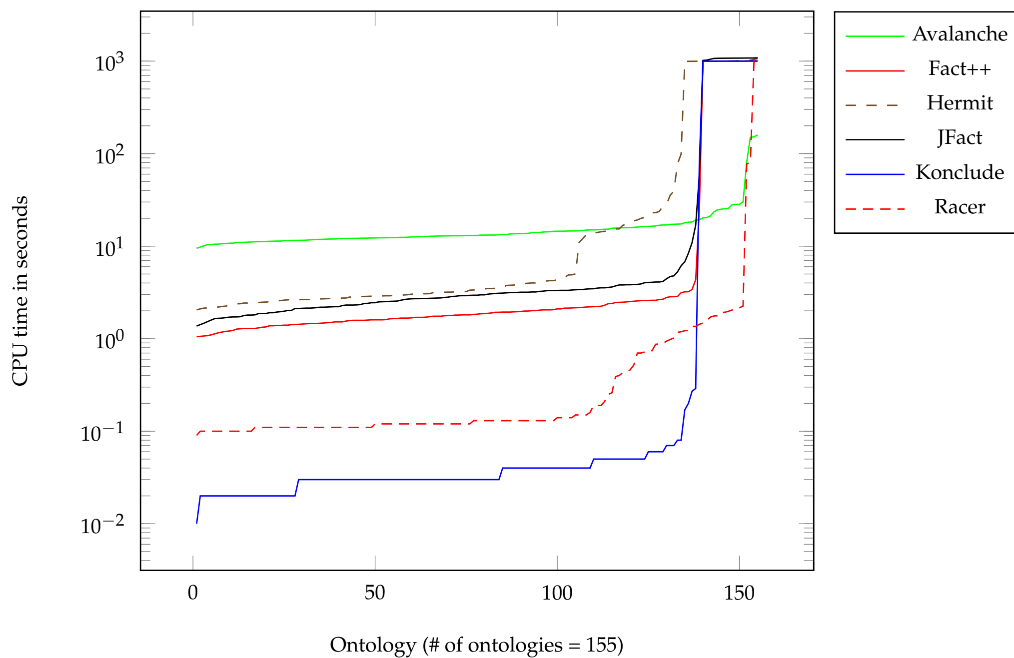

FIGURE 10.6: P-Unsat benchmark runtimes in seconds

Chapter 11

Conclusions and Future Work

11.1 Conclusions

In this thesis we have presented a novel calculus for the description logic $\mathcal{ALCHQ}$. The calculus has been implemented in a system called Avalanche that serves as a proof of concept. Avalanche implements saturation-based rules and constructs a graph that represents a classified ontology by applying these rules. The information that is accumulated in the graph will never be removed. Once no more rules can be applied we know that the graph is saturated and a fully classified taxonomy can be derived from it as each graph node represents a named concept and contains all its entailed subsumers.

We have designed Avalanche as a saturation-based reasoner by gradually extending the existing rules starting with description logic $\mathcal{EL}$. Avalanche reads axioms and derives information from them. Being a saturation-based reasoner also means that Avalanche can avoid creating excessive models as it is done in Tableau based reasoners that need to construct models.

The main advantage of Avalanche compared to other reasoners is an ability to

classify ontologies that contain qualified number restrictions entailing subsumptions. The process of reasoning with number restrictions is delegated to a special module that implements Integer Linear Programming and the column generation algorithm Branch-and-Price algorithm. To the best of our knowledge, none of the existing reasoners has leveraged Branch-and-Price (see Chapter 3). Both HARD [20] and Racer have implemented only ILP without the column generation. Based on our results it can be seen that Avalanche has a clear advantage over the existing reasoners when trying to solve problems that contain entailments based on interactions of qualified number restrictions.

Another distinguishing characteristic of Avalanche is that during the reasoning process it accumulates information and never needs to backtrack as Tableau reasoners do. In order to make it possible different types of internal nodes have been introduced that serve different purposes. Anonymous nodes test unsatisfiability of nodes due the presence of number restrictions as their subsumers. Clone nodes collect information that might eventually result in a subsumption between concepts also due to number restrictions. Once these nodes have accumulated enough information to derive new subsumptions this information will be recorded in the graph. Unlike Tableau based reasoners Avalanche does not construct models that might eventually fail due to the presence of disjunctions in ontologies. It uses resolution techniques to reason on disjunctions.

To summarize, we have addressed and solved our objectives by designing a calculus for the description logic $\mathcal{ALCHQ}$ and implemented it in a saturation-based reasoner Avalanche. The reasoner being saturation-based is able to create smaller models and does not need to backtrack as it employs the resolution techniques to handle disjunctions. Lastly, it implements the Branch-and-Price algorithm to reduce qualified number restrictions to linear programs.

11.2 Future Work

Clearly, we understand that our implementation has room for improvement, since our focus so far has been on reasoning with large numbers of qualified number restrictions and this goal has been achieved. However, since our work so far has not been primarily concerned with general performance, we believe that we would need to further optimize the implementation of Avalanche in order for it to be able to successfully compete with other prominent reasoners on all kinds of ontologies.

Further, a major refactoring of Avalanche has to be planned in order to introduce a new strategy for the application of the saturation-based rules to the input ontology. This could potentially have a positive effect on the reasoner's runtime. In order to achieve this the reasoner's performance has to be re-evaluated on case by case basis so that it would be possible to come up with more efficient heuristics for rule applications.

Our ultimate goal would be to extend our calculus by introducing $\mathcal{I}$ - the inverse roles and $\mathcal{O} - nominals$ that would require us to produce a calculus for the description logic $\mathcal{ALCHOIQ}$ or $\mathcal{SHOIQ}$.

Finally, Avalanche could also become a goal oriented reasoner instead of just classifying ontologies. It could test satisfiability of a concept or focus on subsumption between two concepts.

In conclusion, we believe that our work greatly contributed to the DL research field and hopefully it will be of use.

Bibliography

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications., second ed.* Cambridge University Press, New York, NY, USA, 2007.

[2] Franz Baader, Sebastian Brand, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. In *In Proc. of International Joint Conference on Artificial Intelligence (IJCAI2005)*, pages 364–369. Morgan-Kaufmann Publishers, 2005.

[3] Franz Baader, Sebastian Brand, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope. Technical report, Technische Universität Dresden, 2005.

[4] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ Envelope Further. In Kendall Clark and Peter F. Patel-Schneider, editors, *In Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions*, 2008.

[5] Franz Baader, Jan Hladik, Carsten Lutz, Frank Wolter, and Liverpool L Zf. From tableaux to automata for description logics. *Fundamenta Informaticae*, 57, 2003.

[6] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, and Enrico Franconi. An Empirical Analysis of Optimization Techniques for Terminological Representation Systems - or: Making KRIS get a move on, 1992.

[7] Franz Baader and Ralf Küsters. Matching in Description Logics with Existential Restrictions. In *In Proc. of International Conference on Knowledge Representation and Reasoning (KR2000)*, pages 261–272. Morgan Kaufmann Publishers, 2000.

[8] Franz Baader and Ulrike Sattler. An Overview of Tableau Algorithms for Description Logics. *Studia Logica*, 69(1):5–40, 2001.

[9] A. Bate, B. Motik, B. Cuenca Grau, F. Simančík, and I. Horrocks. Extending consequence-based reasoning to $\mathcal{SRIQ}$. In *Proceedings of KR*, pages 187–196, 2016.

[10] Andrew Bate, Boris Motik, Bernardo Cuenca Grau, Frantisek Simancik, and Ian Horrocks. Extending consequence-based reasoning to SHIQ. In *Proceedings of the 28th International Workshop on Description Logics, Athens, Greece, June 7-10*, 2015.

[11] Andrew Bate, Boris Motik, Bernardo Cuenca Grau, David Tena Cucala, František Simančík, and Ian Horrocks. Consequence-Based Reasoning for Description Logics with Disjunctions and Number Restrictions. J. of Artificial Intelligence Research, 63:625-690, 2018.

[12] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper Tableaux. In *Proceedings of the European Workshop on Logics in Artificial Intelligence*, JELIA '96, pages 1–17, London, UK, 1996. Springer-Verlag.

[13] Martin Buchheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[14] Canadian Parliament. https://en.wikipedia.org/wiki/House_of_Commons_of_Canada.

[15] Vaclav Chvatal. Linear Programming, Freeman, 1983.

[16] CPLEX Optimizer. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/. Accessed on: 2015.21.01.

[17] Donini, F.M., Massacci, F. EXPTIME tableaux for ALC. J. of Artificial Intelligence 124(1), 87–138, 2000.

[18] Jocelyne Faddoul, Reasoning Algebraically with Description Logics. PhD thesis, Concordia University. 2011.

[19] Jocelyne Faddoul and Volker Haarslev. Algebraic tableau reasoning for the description logic SHOQ, Journal of Applied Logic 8(4): 334-355 (2010), Special Issue on Hybrid Logics

[20] Jocelyne Faddoul and Volker Haarslev Optimizing Algebraic Tableau Reasoning for SHOQ: First Experimental Results Proceedings of the 2010 International Workshop on Description Logics (DL-2010), pp. 161-172 Waterloo, Canada, May 4-7, 2010.

[21] FaCT++. http://owl.cs.manchester.ac.uk/tools/fact/.

[22] Nasim Farsiniamarj and Volker Haarslev. Practical Reasoning with Qualified Number Restrictions: A Hybrid Abox Calculus for the Description Logic $\mathcal{SHQ}$. *AI Commun.*, 23(2-3):205–240, April 2010.

[23] Birte Glimm, Ian Horrocks, and Boris Motik. Optimized Description Logic Reasoning via Core Blocking. In Jurgen Giesl and Reiner Hahnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 457–471. Springer Berlin Heidelberg, 2010.

[24] Birte Glimm, Ian Horrocks, Boris Motik, Rob Shearer, and Giorgos Stoilos. A Novel Approach to Ontology Classification. *Journal of Web Semantics*, 14:84–101, July 2012.

[25] Birte Glimm and Yevgeny Kazakov. Role Conjunctions in Expressive Description Logics. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 391–405. Springer Berlin Heidelberg, 2008.

[26] V. Haarslev, K. Hidde, R. Möller, and M. Wessel. The RacerPro knowledge representation and reasoning system. *Semantic Web*, 3(3):267–277, 2012.

[27] V. Haarslev and R. Möller. RACER system description. In *Proceedings of International Joint Conference on Automated Reasoning*, pages 701–705, 2001.

[28] HermiT. http://www.hermit-reasoner.com/download.html.

[29] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph. Foundations of Semantic Web Technologies 2009 Chapman & Hall/CRC

[30] Bernhard Hollunder and Franz Baader. Qualifying Number Restrictions in Concept Languages. Technical report, DFKI, 1991.

[31] Ian Horrocks and Ulrike Sattler. Optimised Reasoning for $\mathcal{SHIQ}$. In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002.

[32] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In *Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning*, LPAR '99, pages 161–180, London, UK, 1999. Springer-Verlag.

[33] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with Individuals for the Description Logic $\mathcal{SHIQ}$. In *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, pages 482–496, London, UK, 2000. Springer-Verlag.

[34] V. Klee and G.J. Minty. How Good is the Simplex Algorithm? In O. Shisha, editor, Inequalities III, pp. 159-175. Academic Press, New York, NY, 1972.

[35] Konclude. http://www.derivo.de/en/produkte/konclude/.

[36] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The Incredible ELK. *J. Autom. Reason.*, 53(1):1–61, June 2014.

[37] Markus Krötzsch, František Simančík, Ian Horrocks. A Description Logic Primer, Technical Report, arXiv.org, volume CoRR abs/1201.4089 January 2012

[38] George F. Luger. Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 2008. 0321545893, Addison-Wesley Publishing Company, USA, 6th edition.

[39] Carsten Lutz. Complexity of Terminological Reasoning Revisited. In *Lecture Notes in Artificial Intelligence*, pages 181–200. Springer-Verlag, 1999.

[40] Boris Motik, Rob Shearer, and Ian Horrocks. A Hypertableau Calculus for $\mathcal{SHIQ}$. In Diego Calvanese, Enriso Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Sergio Tessaris, and Anny-Yasmin Turhan, editors, *Proc. of the 20th Int. Workshop on Description Logics (DL 2007)*, pages 419–426, Brixen/Bressanone, Italy, June 8–10 2007. Bozen/Bolzano University Press.

[41] Boris Motik, Rob Shearer, and Ian Horrocks. Optimized reasoning in description logics using hypertableaux. In Frank Pfenning, editor, *Automated Deduction - CADE21*, volume 4603 of *Lecture Notes in Computer Science*, pages 67–83. Springer Berlin Heidelberg, 2007.

[42] Boris Motik, Rob Shearer, and Ian Horrocks. Hypertableau Reasoning for Description Logics. *Journal of Artificial Intelligence Research*, 36:165–228, 2009.

[43] Hans Jürgen Ohlbach and Jana Koehler. Modal Logics, Description Logics and Arithmetic Reasoning. *Artificial Intelligence*, 109:1–31, 1999.

[44] Racer. https://www.ifis.uni-luebeck.de/index.php?id=385.

[45] A. Robinson. Automatic Deduction with Hyper-Resolution. *Int. Journal of Computer Mathematics*, pages 227–234, 1965.

[46] Matthias Samwald. Genomic CDS: an example of a complex ontology for pharmacogenetics and clinical decision support. In *2nd OWL Reasoner Evaluation Workshop*, pages 128–133. CEUR, 2013.

[47] Matthias Samwald. An update on genomic CDS, a complex ontology for pharmacogenomics and clinical decision support. In *3rd OWL Reasoner Evaluation Workshop*, pages 58–63. CEUR, 2014.

[48] The Semantic Web Stack. http://en.wikipedia.org/wiki/Semantic_Web_Stack. Accessed on: 2015.01.05.

[49] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *Proc. of the 5th Int. Workshop on OWL Experiences and Directions (OWLED 2008)*, volume 432 of *CEUR (http://ceur-ws.org/)*, 2008.

[50] F. Simančík, B. Motik, and I. Horrocks. Consequence-based and fixed-parameter tractable reasoning in description logics. *Artificial Intelligence*, 209:29–77, 2014.

[51] František Simančík, Yevgeny Kazakov, and Ian Horrocks. Consequence-Based Reasoning beyond Horn Ontologies. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1093–1098. AAAI Press/IJCAI, 2011.

[52] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007.

[53] SNOMED Clinical Terms (SNOMED CT). http://ihtsdo.org/snomed-ct/. Accessed on: 22.06.2016.

[54] Andreas Steigmiller, Birte Glimm, and Thorsten Liebig. Coupling Tableau Algorithms for Expressive Description Logics with Completion-Based Saturation Procedures. In Stephane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 449–463. Springer International Publishing, 2014.

[55] Andreas Steigmiller, Birte Glimm, and Thorsten Liebig. Optimised Absorption for Expressive Description Logics. In Riccardo Rosati Meghyn Bienvenu, Magdalena Ortiz and Mantas Simkus, editors, *Proceedings of the 27th International Workshop on Description Logics (DL 2014)*, volume 1193 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014.

[56] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: System Description. *Web Semantics: Science, Services and Agents on the World Wide Web*, Volume 27, Issue C, August 2014, pp 78-85.

[57] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Extended Caching, Backjumping and Merging for Expressive Description Logics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 514–529. Springer Berlin Heidelberg, 2012.

[58] David Tena Cucala, Bernardo Cuenca Grau, and Ian Horrocks. Sequoia: A Consequence Based Reasoner for SROIQ. In *Proceedings of the Description Logic Workshop (DL 2019)*, volume 2373 of CEUR Workshop Proceedings. 2019.

[59] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proceedings of the Third International Joint Conference on Automated Reasoning, (IJCAR2006)*, pages 292–297, Berlin, Heidelberg, 2006. Springer-Verlag.

[60] Dmitry Tsarkov, Ian Horrocks, and Peter F. Patel-Schneider. Optimizing Terminological Reasoning for Expressive Description Logics. *Journal of Automated Reasoning*, 39(3):277–316, 2007.

[61] Jelena Vlasenko, Maryam Daryalal, Volker Haarslev, and Brigitte Jaumard. A Saturation-based Algebraic Reasoner for ELQ. In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, pages 110-124, July 2016, Coimbra, Portugal.

[62] Jelena Vlasenko, Volker Haarslev, Brigitte Jaumard. Pushing the Boundaries of Reasoning About Qualified Cardinality Restrictions. In *Proceedings of the*

11th International Symposium on Frontiers of Combining Systems (FroCoS 2017), Brasilia, Brazil, LNAI 10483, pp. 95-112.

Appendix A

Publications

A Saturation-based Algebraic Reasoner for ELQ.

Jelena Vlasenko, Maryam Daryalal, Volker Haarslev, Brigitte Jaumard,

In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning (PAAR 2016)*, Coimbra, Portugal, CEUR, pp. 110-124.

Pushing the Boundaries of Reasoning About Qualified Cardinality Restrictions.

Jelena Vlasenko, Volker Haarslev, Brigitte Jaumard,

In *Proceedings of the 11th International Symposium on Frontiers of Combining Systems (FroCoS 2017)*, Brasilia, Brazil, LNAI 10483, pp. 95-112.

Appendix B

Example

B.1 Example of Rule Applications

In this example we will demonstrate how some of the normalization and the saturation rules work. In particular, we will show how and why anonymous and clone nodes are created. We will also explain what happens when these nodes become unsatisfiable.

Consider the following TBox T:

1. $C \sqsubseteq E$
2. $E \sqsubseteq \geq 3\, R.(\leq 2\, S.E)$
3. $\geq 2\, R.(\leq 2\, S.C) \sqsubseteq D$

This TBox entails only one subsumption: $E \sqsubseteq D$. Below we will show how the saturation-based rules are applied in order to discover this subsumption. First, the system will start with the application of the normalization rules. In this example the second and the third axioms should be normalized. The first axiom is already in the normal form. After the transformation the TBox T will look as follows:

1. $C \sqsubseteq E$

2. $E \sqsubseteq \geq 3\, R.aux_2$

3. $aux_2 \sqsubseteq \leq 2\, S.E$

4. $\geq 2\, R.aux_1 \sqsubseteq D$

5. $\leq 2\, S.C \sqsubseteq aux_1$

The axiom $E \sqsubseteq \geq 3\, R.(\leq 2\, S.E)$ will be transformed into $E \sqsubseteq \geq 3\, R.aux_2$ and $aux_2 \sqsubseteq \leq 2\, S.E$ by the normalization rule $NR2_{rhs}$. $\geq 2\, R.(\leq 2\, S.C) \sqsubseteq D$ will be transformed into $\geq 2\, R.aux_1 \sqsubseteq D$ and $\leq 2\, S.C \sqsubseteq aux_1$ by the normalization rule $NR3_{lhs}$.

As the algorithm builds a graph during the saturation-based phase it will start with the creation of a node for every named concept in the TBox T. As a result, we will have the following nodes:

- Node for the representative concept $\top$. The algorithm always creates a node for the concept $\top$ even if it is not explicitly mentioned in the TBox T since $\top$ is a subsumer of every concept in an ontology.
- Node for the representative concept C.
- Node for the representative concept E.
- Node for the representative concept D.
- Node for the representative concept aux_1.
- Node for the representative concept aux_2.

Then the algorithm adds two initial subsumers to the newly created nodes: the representative concept of the node that is the concept for which the node was created and $\top$ (see Figure B.1).

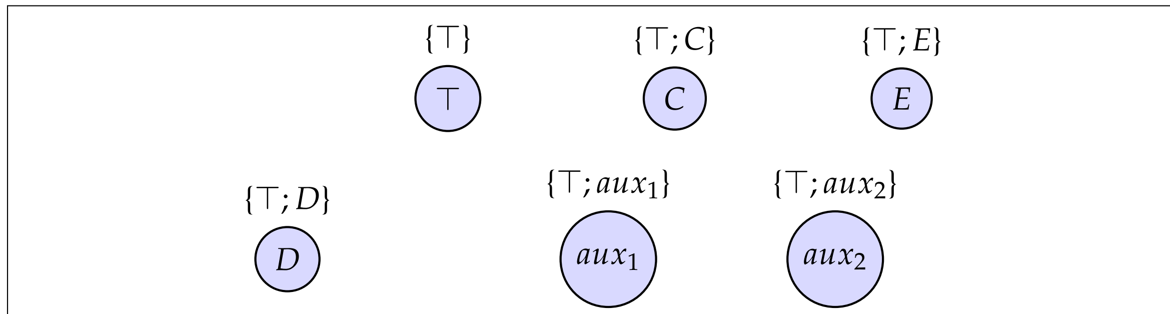

FIGURE B.1: Initialization

After that the saturation-based rules will be applied to the normalized TBox T. First the unfolding rules that are a part of the saturation-based rules will be applied. In this example only two unfolding rules are applicable. The first one is the rule $R_{\sqsubseteq}$. It will add E to the subsumers of the node with the representative concept C. The rule will also add $\leq 2\, S.E$ to the subsumers of the node aux_2, and $\geq 3\, R.aux_2$ to the subsumers of the node E (see Figure B.2).

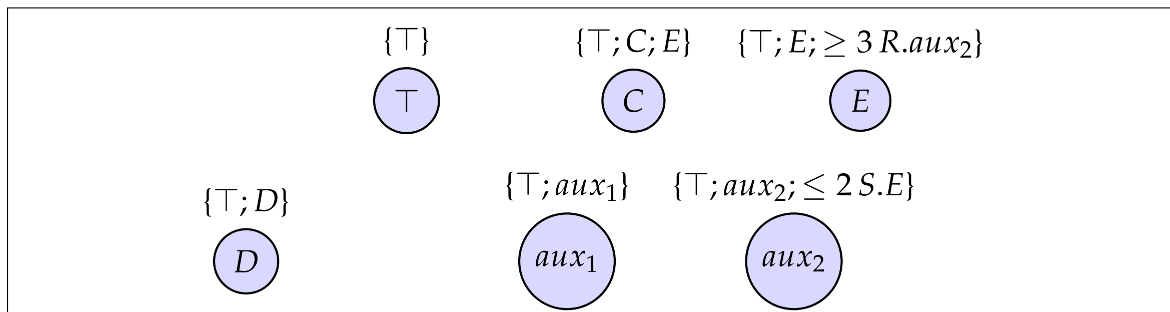

FIGURE B.2: Application of the Rule $R_{\sqsubseteq}$

The second rule that will be applied is $R_{P_{\bowtie}}$. It will add a possible subsumer $\leq 1\, R.aux_1$ to the node D.

At this point we will take a shortcut and demonstrate application only of those rules that will lead us to the entailed subsumption. Please note that *Avalanche* will not be able to make this guess and will apply all the rules until they stop adding new information to the graph. The system will compute possible subsumers for all

the nodes and create clones to test subsumption and disjointness between all pairs of named concepts.

Now we will proceed with the application of the rest of the saturation-based rules. We apply $R_{\sqsubseteq_P}$ to create a clone node in order to test whether $E \sqsubseteq D$. The new node will contain the following subsumers: $\top, \geq 3\, R.aux_2, \leq 1\, R.aux_1$ (see Figure B.3).

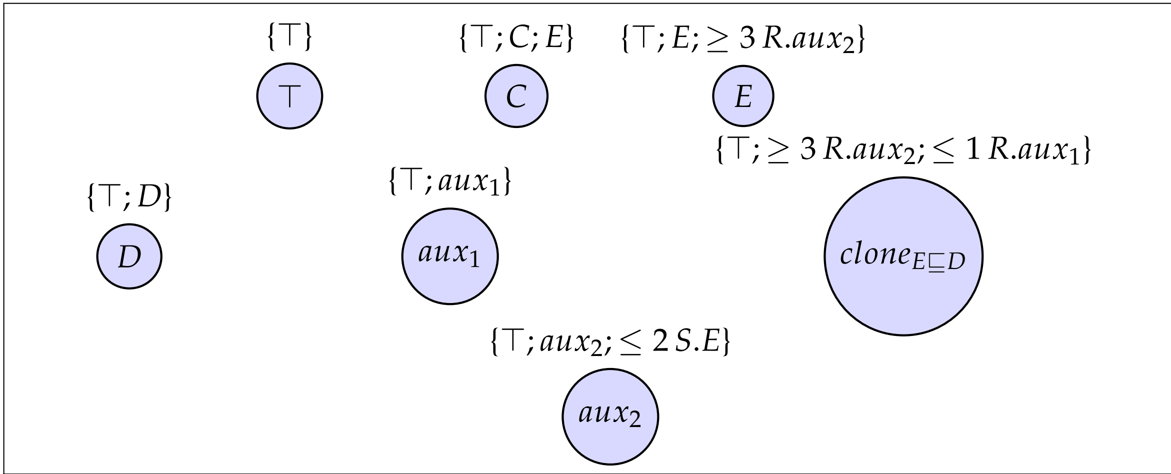

FIGURE B.3: Creation of a Cloned Node

Then the R_{fil} will be applied to the clone node $clone_{E \sqsubseteq D}$. The ILP module will be called and it will return two tuples: $(< R >, < aux_2 >, < 1 >)$ and $(< R >, < \neg aux_1, aux_2 >, < 2 >)$. Based on the returned tuples the rule will create two edges. First edge will connect the clone node $clone_{E \sqsubseteq D}$ with the existing node aux_2. Second edge will connect $clone_{E \sqsubseteq D}$ with the newly created anonymous node $anonym$. The anonymous node will contain the following subsumers: $\top$, $\neg aux_1$, and aux_2. We need to create the anonymous node because of the concepts $< \neg aux_1, aux_2 >$ that were returned by the second tuple. This can be interpreted as follows: the role filler for the role R is an intersection $\neg aux_1 \sqcap aux_2$. Since there is no node in our graph with such a role filler we need create an anonymous node. This is demonstrated in Figure B.4.

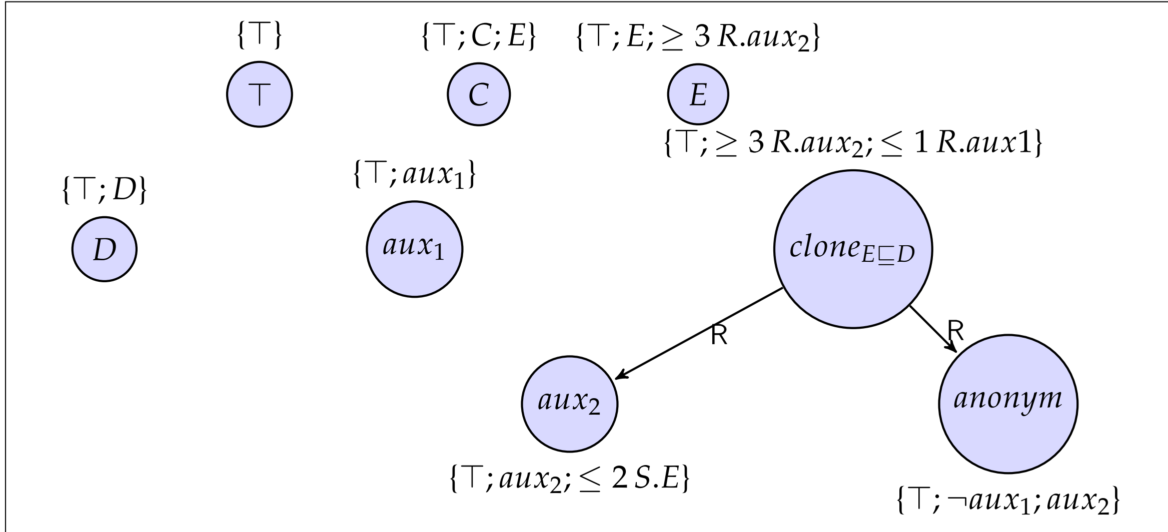

FIGURE B.4: Creation of an Anonymous Node

Then the rule $R_{P_{\bowtie}}$ will propagate $\leq 2\, S.E$ to the subsumers of the node *anonym*. After that the rule $R_{\sqsubseteq\neq}$ will fire and add $\geq 3\, S.C$ to the subsumers of *anonym* (see Figure B.5). These two qualified cardinality restrictions are infeasible and as a result the node will become unsatisfiable. $\perp$ will be added to the subsumers of the node. As a result since the anonymous node is unsatisfiable we learn that aux_2 and $\neg aux_1$ are disjoint concepts. Finally the algorithm will propagate $\perp$ to the source of the edge that is $clone_{E\sqsubseteq D}$. It will try to construct another model. However, since there is no other model the node $clone_{E\sqsubseteq D}$ will also become unsatisfiable. As a result D will be added to the subsumers of E.

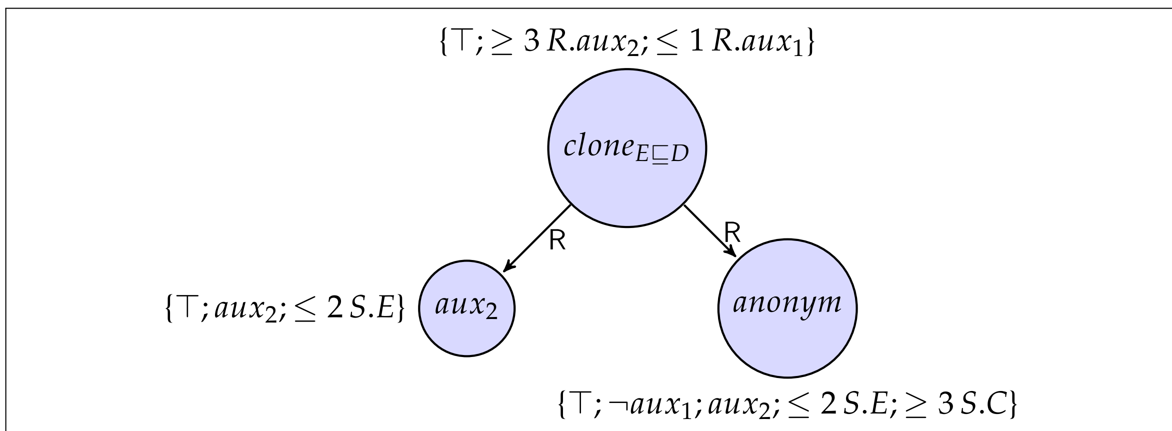

FIGURE B.5: Propagation of Subsumers to the Anonymous Node

Appendix C

Evaluation

C.1 Performance Evaluation of $\mathcal{ALCQ}$ Ontologies

The $\mathcal{ALCQ}$ ontologies used for the following benchmarks model the Canadian Parliament thus representing a real-life situation. These ontologies contain a large number of qualified restrictions in addition to conjunctions, disjunctions, existential restrictions, and negations of concepts and expressions. We compared the performance of Avalanche, Fact++, Hermit, JFact, Konclude, and Racer. The results are presented in Table C.1.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
canadian-parliament-ALCQ	59.33	991.04	999.77	1058.1	2.8	0.17
canadian-parliament-ALCQ-full	74.55	995.64	1000.25	1044.59	985.21	29.07
canadian-parliament-ALCQ-full-inc	119.66	992.36	1002.33	1049.04	993.02	0.73
canadian-parliament-ALCQ-inc	90.57	995.84	1000.95	1052.4	993.17	0.71
canadian-parliament-factions-10	43.69	993.66	999.9	1052.64	2.89	0.18
canadian-parliament-full-factions-1	35.52	995.96	1000.05	1060.91	994.28	0.14
canadian-parliament-full-factions-10	73.33	993.95	1000.62	1051.18	990.73	29.45
canadian-parliament-full-factions-2	43.29	994.33	997.99	1037.5	985.49	23.74

TABLE C.1: Benchmarks using $\mathcal{ALCQ}$ Ontologies

C.2 Performance Evaluation of $\mathcal{ELQ}$ Ontologies

The $\mathcal{ELQ}$ ontologies used also model the Canadian Parliament but they are less expressive than $\mathcal{ALCQ}$. These ontologies also contain a large number of qualified restrictions as well as conjunctions and existential restrictions. We compared performance of Avalanche, Fact++, Hermit, JFact, Konclude, and Racer. The results are presented in Table C.2.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
canadian-parliament-ELQ	82.96	1.55	1000.16	2.77	0.03	0.51
canadian-parliament-ELQ-full	125.02	2.04	1000.95	3.09	0.05	999.11
canadian-parliament-ELQ-full-inc	189.64	989.09	1000.56	1076.91	990.29	0.91
canadian-parliament-ELQ-inc	297.74	999.47	1001.12	1069.38	994.11	1.03
canadian-parliament-factions-1	19.16	1.83	1001.49	2.06	0.04	0.19
canadian-parliament-factions-10	68.46	2.21	1000.23	2.44	0.05	0.49
canadian-parliament-factions-10-nr	75.92	1.97	1000.06	2.24	0.03	0.58
canadian-parliament-factions-2	20.82	1.27	1000.6	3.5	0.02	0.22
canadian-parliament-factions-2-nr	25.6	1.59	1000.98	2.65	0.02	0.22
canadian-parliament-factions-3	23.01	1.86	1000.48	2.69	0.02	0.26
canadian-parliament-factions-3-nr	25.59	1.75	1001.68	3.17	0.04	0.25
canadian-parliament-factions-4	32.96	2	1000.73	2.06	0.04	0.27
canadian-parliament-factions-4-nr	29.75	1.43	1001.91	2.6	0.03	0.27
canadian-parliament-factions-5	37.33	2.43	1000.2	2.11	0.03	0.3
canadian-parliament-factions-5-nr	42.23	1.93	1000.31	2.11	0.03	0.31
canadian-parliament-full-factions-1	21.48	1.34	1002.09	2.99	0.03	1.02
canadian-parliament-full-factions-10	76.31	1.61	1000.78	1.81	0.04	998.77
canadian-parliament-full-factions-10-nr	74.04	1.48	1000.06	2.48	0.03	998.86
canadian-parliament-full-factions-2	23.07	1.69	1000.89	2.23	0.03	24.97
canadian-parliament-full-factions-3	32.93	1.61	1000.96	2.65	0.02	998.35
canadian-parliament-full-factions-4	34.69	1.76	999.1	1.96	0.05	998.46
canadian-parliament-full-factions-5	42.82	2.44	998.26	2.11	0.04	998.4

TABLE C.2: Benchmarks using $\mathcal{ELQ}$ Ontologies

C.3 Performance Evaluation of Satisfiable $\mathcal{SHQ}$ Ontologies

Below we present the results of the performance evaluation on satisfiable $\mathcal{SHQ}$ ontologies. The difference between $\mathcal{ALCQ}$ and $\mathcal{SHQ}$ is the presence of $\mathcal{H}$ - role hierarchies. The results are presented in Table C.3.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
C-SAT-exp-ALCQ-1	17.37	2.85	995.65	7.25	996.68	0.13
C-SAT-exp-ALCQ-2	22.12	999.41	1000.06	1076.87	996.71	0.15
C-SAT-exp-ALCQ-3	16.47	998.56	23.87	1073.43	995.52	0.38
C-SAT-exp-ALCQ-4	17.79	1000.78	1145.1	1019.96	1006.13	3.18
C-SAT-exp-ALCQ-5	17.92	1003.74	1236.11	994.57	1006.25	31.69
C-SAT-exp-ALCQ-6	20.02	1003.8	990.84	995.71	1007.36	335.65
C-SAT-exp-ELQ-1	29.78	1.39	5.7	2.99	0.04	0.12
C-SAT-exp-ELQ-2	23.72	1.38	12.01	2.43	0.03	0.12
C-SAT-exp-ELQ-3	21.85	1.86	1001.71	2.77	0.03	0.13
C-SAT-exp-ELQ-3-normalized	21.38	1.89	998.66	1.76	0.03	0.12
C-SAT-exp-ELQ-4	29.73	2.14	998.76	2.7	0.02	0.12
C-SAT-exp-ELQ-5	27.87	1.22	999.75	2.05	0.02	0.12
C-SAT-lin-ALCHQ-1	20.31	1.41	3.03	2.01	0.18	0.12
C-SAT-lin-ALCHQ-10	24.97	1002.19	997.15	1062.89	995.96	0.12
C-SAT-lin-ALCHQ-2	20.66	1.25	4.3	2.11	0.09	0.11
C-SAT-lin-ALCHQ-3	18.21	1.95	5.74	4.53	0.16	0.13
C-SAT-lin-ALCHQ-4	19.53	1.91	128.94	7.85	0.33	0.11
C-SAT-lin-ALCHQ-5	29.15	4.54	1029.56	17.86	1.56	0.15
C-SAT-lin-ALCHQ-6	26.06	29.47	1000.9	73.14	5.81	0.12
C-SAT-lin-ALCHQ-7	23.45	307.57	985.9	513.73	22.34	0.11
C-SAT-lin-ALCHQ-8	27.93	1001.41	990.27	1065.58	88.56	0.13
C-SAT-lin-ALCHQ-9	22.69	1003.15	994.86	1070.01	340.18	0.12
C-SAT-lin-ALCQ-1	17.76	2.22	5.06	2.14	0.04	0.12
C-SAT-lin-ALCQ-10	14.66	2.68	1008.74	10.9	995.29	0.12
C-SAT-lin-ALCQ-2	26.44	1.82	5.93	2.08	0.04	0.12
C-SAT-lin-ALCQ-3	23.4	1.76	8.17	2.63	0.25	0.12
C-SAT-lin-ALCQ-4	21.68	1.22	254.66	3.24	7.01	0.12
C-SAT-lin-ALCQ-5	20.75	1.56	1037.98	3.44	145.93	0.11
C-SAT-lin-ALCQ-6	20.47	2.47	996.31	3.89	586.75	0.12
C-SAT-lin-ALCQ-7	16.72	1.25	984.03	5.43	998.04	0.12
C-SAT-lin-ALCQ-8	17.13	1.95	992.52	7.21	998.82	0.14
C-SAT-lin-ALCQ-9	22.38	2.11	993.8	8.03	998.77	0.14

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
C-sat-unsat-30-sol-12	12.32	1.71	998.22	4.18	0.04	0.14
C-sat-unsat-30-sol-14	10.66	1.59	997.2	4.13	0.03	0.11
C-sat-unsat-30-sol-16	13.89	2.64	999.38	3.97	0.02	0.12
C-sat-unsat-30-sol-18	10.62	1.59	999.14	4.99	0.01	0.12
C-sat-unsat-30-sol-20	10.22	1.36	999.97	4.43	0.02	0.11
C-sat-unsat-30-sol-22	12.4	1.57	1000.27	4.37	0.02	0.11
C-sat-unsat-30-sol-24	13.71	1.88	1000.16	4.85	0.03	0.17
C-UnSAT-exp-ELQ-1	27.2	2.43	6.49	1.53	0.02	0.13
C-UnSAT-exp-ELQ-2	22.7	1.94	1043.01	1.62	0.04	0.12
C-UnSAT-exp-ELQ-3	22.3	1.84	1006.01	1.65	0.03	0.12
C-UnSAT-exp-ELQ-3-normalized	11.15	1.71	3.64	1.89	0.03	0.13
C-UnSAT-exp-ELQ-4	24.85	1.96	1000.12	2.36	0.03	0.12
C-UnSAT-exp-ELQ-5	26.57	1.19	1001.07	2.2	0.02	0.12
rest-ratio-1-0	14.73	1.15	3.9	2.73	0.01	0.13
rest-ratio-1-1	19.96	1.27	4.94	2.25	0.02	0.14
rest-ratio-1-10	27.72	1.72	3.11	1.91	0.02	0.14
rest-ratio-1-2	23.23	1.18	5.99	2.57	0.02	0.14
rest-ratio-1-3	26.47	1.99	5.31	2.42	0.01	0.14
rest-ratio-1-4	24.34	1.55	5.24	1.74	0.02	0.14
rest-ratio-1-5	33.28	1.4	4.59	1.81	0.02	0.16
rest-ratio-1-6	31.08	1.36	4.98	1.74	0.02	0.17
rest-ratio-1-7	21.87	2.1	5.12	2.36	0.02	0.18
rest-ratio-1-8	43.29	1.76	3.87	6.97	0.03	0.31
rest-ratio-1-9	32.95	2.02	4.05	2.87	0.03	0.15
restr-num-1-1-1	16.14	1.56	3.26	2.11	0.02	0.13
restr-num-1-1-2	12.32	1.07	2.92	2.28	0.02	0.13
restr-num-1-1-OR	13.17	2.38	2.68	1.82	0.02	0.12
restr-num-1-1-OR-atmost	11.73	2	3.7	1.52	0.02	0.12
restr-num-1-10-1	58.42	1.76	1000.48	4.95	0.02	295.62
restr-num-1-10-2	50.13	1.9	26.04	1.74	0.03	945.45
restr-num-1-10-3	44.38	2.86	13.4	2.38	0.02	998.16
restr-num-1-11	49.74	1.2	53.1	2.77	0.02	998.57
restr-num-1-12	60.99	1.83	51.68	2.44	0.02	997.45
restr-num-1-13	99.05	2.17	376.7	1.92	0.02	997.37
restr-num-1-14	100.26	1.42	978.65	2.56	0.04	997.18
restr-num-1-15	153.96	1.66	675.67	2.45	0.03	996.76
restr-num-1-16	138.11	3.15	979.41	3.14	0.03	997.85
restr-num-1-17	260.51	1.98	979.09	2.78	0.03	998.11
restr-num-1-18	245.14	1.6	1051.99	3.23	0.02	998.77
restr-num-1-19	418.7	1.91	1048.2	2.68	0.03	998.78
restr-num-1-2-1	14.08	2.24	3.46	1.57	0.01	0.11
restr-num-1-2-2	14.6	1.96	3.97	2.06	0.02	0.12

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
restr-num-1-2-OR	15.05	1.52	3.09	1.66	0.04	0.12
restr-num-1-2-OR-atmost	16.77	1.91	3.7	2.66	0.02	0.12
restr-num-1-20	246.77	2.16	1036.96	2.87	0.03	999.1
restr-num-1-3-1	12.97	1.47	4.71	1.89	0.03	0.12
restr-num-1-3-2	16.36	1.72	5.31	1.87	0.02	0.12
restr-num-1-3-OR	19.19	1.56	2.83	2.06	0.03	0.12
restr-num-1-3-OR-atmost	20.15	1.35	978.53	1.57	0.04	0.11
restr-num-1-4-1	20.47	1.32	5.06	2.99	0.02	0.13
restr-num-1-4-2	20.36	1.86	4.97	2.48	0.04	0.12
restr-num-1-4-OR	22.52	1.29	2.23	1.48	0.03	0.13
restr-num-1-4-OR-atmost	27.04	1.53	980.51	3.2	0.03	0.13
restr-num-1-5-1	19.58	1.46	5.15	3.32	0.03	0.15
restr-num-1-5-2	27.68	1.67	4.28	2.62	0.01	0.17
restr-num-1-5-OR	39.15	1.73	3.6	1.75	0.02	0.13
restr-num-1-5-OR-atmost	36.18	2.04	977.43	2.72	0.03	0.13
restr-num-1-6-1	28.85	1.32	4.94	1.6	0.02	0.28
restr-num-1-6-2	23.97	1.88	5.09	1.85	0.03	0.16
restr-num-1-6-OR-atmost	40.98	1.93	998.7	1.85	0.03	0.12
restr-num-1-7	39.34	1.11	6.02	1.83	0.02	0.36
restr-num-1-8-1	34.83	1.46	8.06	2.2	0.02	0.78
restr-num-1-8-2	33.69	1.25	5.59	1.77	0.03	0.54
restr-num-1-9-1	34.66	1.45	19.21	1.9	0.03	1.42
restr-num-1-9-2	38.13	1.59	7.15	2.04	0.02	1.14
restr-num-10-1	11.18	2.07	1000.03	6.35	0.02	0.11
restr-num-10-2	13.51	1.92	1000.4	4.09	0.03	0.13
restr-num-10-3	19.9	1.57	1000.98	2.77	0.03	0.13
restr-num-10-4	23.7	1.43	1000	2.87	0.02	0.26
restr-num-10-5	23.49	1.69	1000.27	3.49	0.02	0.93
restr-num-10-6	25.98	1.96	1000.27	3.91	0.02	23.05
restr-num-10-7	47.88	2.53	999.85	3.47	0.03	28.98
restr-num-10-8	59.22	1.62	999.87	3.4	0.02	93.46
restr-num-10-9	59.14	2.37	999.81	3.76	0.03	93.14
restr-num-10-var-1	11.85	1.21	999.96	2.64	0.02	0.12
restr-num-10-var-10	78.07	1.92	1000.1	2.91	0.02	0.76
restr-num-10-var-2	14.65	2.27	999.98	1.97	0.03	0.12
restr-num-10-var-3	19.48	2.35	1000.14	1.68	0.02	0.15
restr-num-10-var-4	21.96	1.44	1000.3	2.49	0.03	0.22
restr-num-10-var-5	28.86	2.03	1001.11	2.47	0.02	0.22
restr-num-10-var-6	42.95	1.51	1000.3	3.03	0.04	0.36
restr-num-10-var-7	46.54	2	1000.82	2.93	0.01	0.4
restr-num-10-var-8	60.32	2.77	1000.44	2.87	0.02	0.56
restr-num-10-var-9	57.83	1.88	977.78	3.33	0.02	0.64

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
restr-num-5-1	11.89	2.15	1000.22	1.57	0.03	0.11
restr-num-5-10	51.66	1.22	980.18	3	0.03	26.51
restr-num-5-2	13.8	1.4	975.68	1.68	0.03	0.13
restr-num-5-3	13.54	2.54	976.86	2.87	0.03	0.13
restr-num-5-4	53.86	1.56	978.2	5.6	0.04	0.16
restr-num-5-5	26.31	2.31	982.68	1.76	0.04	0.3
restr-num-5-6	38.22	1.94	999.07	2.98	0.03	1.27
restr-num-5-7	39.6	2	990.15	2.37	0.02	4.19
restr-num-5-8	50.24	2.19	977.95	3.65	0.03	10.86
restr-num-5-9	51.13	1.53	979.62	2.96	0.02	11.35
restr-num-ELQ-1-10	15.21	1.78	134.29	1.98	0.03	0.13
sat-unsat-3-sol-12	14.06	1.79	377.7	1.85	0.02	0.11
sat-unsat-3-sol-14	11.71	1.75	1000.54	2.21	0.02	0.11
sat-unsat-3-sol-16	12.14	1.94	1001.21	2.33	0.01	0.11
sat-unsat-3-sol-18	13.57	1.92	1000.35	1.8	0.04	0.12
sat-unsat-3-sol-20	13.88	2.02	1000.12	5.16	0.02	0.12
sat-unsat-3-sol-22	12.72	1.14	1000.25	3.08	0.02	0.11
sat-unsat-3-sol-24	12.2	2.42	1000.42	3.22	0.02	0.11
sat-unsat-30-sol-12	13.96	1.84	1000.03	4.25	0.01	0.11
sat-unsat-30-sol-14	13.5	1.58	1000.74	5.34	0.02	0.12
sat-unsat-30-sol-16	11.86	1.36	1001.65	3.93	0.03	0.11
sat-unsat-30-sol-18	15.76	1.01	1000.03	4.11	0.02	0.11
sat-unsat-30-sol-20	12.65	1.91	1000.75	4.82	0.02	0.12
sat-unsat-30-sol-22	12.18	2.55	1000.55	4.22	0.02	0.12
sat-unsat-30-sol-24	13.03	1.31	1000.6	3.67	0.02	0.12
SHQ-CSAT-exp-1	21.39	15.33	1002.55	37.12	998.99	0.12
SHQ-CSAT-exp-2	20.72	998.7	1000.88	1075.49	999.41	0.15
SHQ-CSAT-exp-3	25.18	995.99	21.49	1077.43	1000.27	0.38
SHQ-CSAT-exp-4	20.95	996.77	1152.87	1076.19	1005.78	3.26
SHQ-CSAT-exp-5	23	58.81	1204.02	995.16	1006.95	32.98
SHQ-CSAT-exp-6	19.87	995.96	978.47	995.46	1007.62	378.18

TABLE C.3: Benchmarks for $\mathcal{ALCHQ}$-SAT Ontologies

C.4 Performance Evaluation of Unsatisfiable $\mathcal{SHQ}$ Ontologies

Below we present the results of the performance evaluation on unsatisfiable $\mathcal{SHQ}$ ontologies. The difference between satisfiable and unsatisfiable $\mathcal{SHQ}$ ontologies is that the concept Thing is unsatisfiable in unsatisfiable ontologies. The results are presented in Table C.4.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
Backtracking1-1	13.85	3.02	1.98	4.54	0.03	0.55
Backtracking1-10	10.69	2.96	22.57	4.04	0.04	0.4
Backtracking1-11	14.23	3.01	121.6	3.61	0.05	0.38
Backtracking1-12	12.02	3.23	261	4.58	0.07	0.66
Backtracking1-13	15.11	2.92	1000.15	6.25	0.03	0.58
Backtracking1-14	14.24	2.72	1001.43	6.22	0.04	0.7
Backtracking1-15	14.39	3.19	1000.46	3.11	0.04	0.8
Backtracking1-2	14.23	3.53	3.34	4.3	0.05	0.55
Backtracking1-3	11.15	4.94	3.64	7.9	0.27	0.44
Backtracking1-4	12.62	3.55	3.5	4.42	0.06	0.5
Backtracking1-5	14.1	2.23	3.12	5.34	0.06	0.28
Backtracking1-6	12.89	2.89	3.87	4.46	0.05	0.35
Backtracking1-7	12.87	3.91	3.02	4.48	0.13	0.38
Backtracking1-8	13.71	5.01	4.86	8.81	0.11	0.35
Backtracking1-9	11.77	2.61	5.95	4.95	0.05	0.52
Backtracking2-1	11.26	3	3.1	4.09	0.07	0.27
Backtracking2-10	13.39	2.17	1000.9	3.3	0.04	0.35
Backtracking2-2	15.56	3.47	2.68	5.7	0.04	0.46
Backtracking2-3	12.1	5.27	2.29	5.79	0.04	0.42
Backtracking2-4	11.28	5.9	3.1	3.22	0.04	0.46
Backtracking2-5	13.19	3.91	8.85	3.71	0.1	0.4
Backtracking2-6	11.48	4.07	137.86	2.99	0.06	0.58
Backtracking2-7	11.89	3.25	1000.93	3.82	0.09	0.51
Backtracking2-8	11.27	5.09	999.89	4.6	0.04	0.6
Backtracking2-9	13.38	5.75	999.63	2.92	0.05	0.55
Backtracking3	10.78	1.83	2.82	2.22	0.03	0.17
Backtracking3-1	10.11	2.79	2.54	2.74	0.05	0.51
Backtracking3-10	13.37	4.24	1000.48	3.88	0.05	0.68
Backtracking3-11	12.34	3.44	999.03	4.12	0.05	0.43
Backtracking3-12	11.12	3.6	998.73	2.83	0.23	0.44

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
Backtracking3-13	12.87	2.94	999.23	5.03	0.09	0.5
Backtracking3-14	13.1	4.65	999.51	4.04	0.08	0.52
Backtracking3-15	14.39	5.22	999.15	3.28	0.07	0.53
Backtracking3-16	16.51	4.36	1000.38	2.76	0.08	0.65
Backtracking3-17	14.24	2.83	998.19	2.53	0.11	1.06
Backtracking3-18	13.14	3.37	998.77	2.84	0.08	1.05
Backtracking3-19	14.64	5.4	995.86	3.4	0.08	1.69
Backtracking3-2	13.96	2.96	3.03	1.95	0.03	0.23
Backtracking3-20	15.49	3.01	995.75	2.44	0.1	0.86
Backtracking3-3	12.94	2.88	4.56	2.1	0.03	0.16
Backtracking3-4	13.58	2.6	100.51	1.78	0.02	0.31
Backtracking3-5	11.39	2.4	1001.32	2.06	0.05	0.22
Backtracking3-6	15.45	1.53	999.97	1.78	0.03	0.33
Backtracking3-7	15.32	2.02	999.99	2.34	0.03	0.34
Backtracking3-8	12.67	2.78	1001.68	2.13	0.05	0.37
Backtracking3-9	12.55	3.19	1000.5	2.19	0.06	0.4
C-sat-unsat-30-nosol-1	11.82	1.6	1000.12	3.2	0.04	0.3
C-sat-unsat-30-sol-1	13.24	11.15	1000.35	35.28	0.05	0.2
C-sat-unsat-30-sol-10	13.57	15.75	999.95	37.63	0.08	0.35
C-sat-unsat-30-sol-2	15.23	2.12	1000.75	1.88	0.03	0.19
C-sat-unsat-30-sol-4	11.57	2.47	1000.25	10.55	0.06	0.5
C-sat-unsat-30-sol-6	12.48	19.16	1000.19	32.71	0.04	0.26
C-sat-unsat-30-sol-8	12.99	3.37	1001.65	9.82	0.06	0.17
C-UnSAT-exp-ALCQ-1	16.22	1001.97	999.89	1077.87	997.25	0.28
C-UnSAT-exp-ALCQ-2	17.2	1000.1	1000.8	1083.88	998.88	24.54
C-UnSAT-exp-ALCQ-3	15.48	1001.66	21.23	1083.94	999.43	999.15
C-UnSAT-exp-ALCQ-4	14.22	1003.01	1197.52	1008.34	1008.57	999.35
C-UnSAT-exp-ALCQ-5	13.78	1003.17	1256.28	995.65	1008.03	999.27
C-UnSAT-exp-ALCQ-6	19.38	1004.31	984.11	996.26	1008.33	999.47
C-UnSAT-lin-ALCHQ-1	14.18	1.63	3.31	2.83	0.03	0.12
C-UnSAT-lin-ALCHQ-10	23.66	1003.7	999.51	1073.94	998.68	0.13
C-UnSAT-lin-ALCHQ-2	24.35	1.2	5.25	2.51	0.03	0.13
C-UnSAT-lin-ALCHQ-3	20.53	2.13	13.93	6.44	0.12	0.15
C-UnSAT-lin-ALCHQ-4	17.97	3.09	987.86	13.86	1.3	0.12
C-UnSAT-lin-ALCHQ-5	18.33	27.57	1015.55	65.57	18.02	0.12
C-UnSAT-lin-ALCHQ-6	18.66	543.78	1001.41	915.89	274.77	0.13
C-UnSAT-lin-ALCHQ-7	21.23	1002.68	989.29	1073.46	996.92	0.12
C-UnSAT-lin-ALCHQ-8	18.51	1002.02	990.31	1072.29	996.09	0.13
C-UnSAT-lin-ALCHQ-9	19.34	1000.65	996.65	1065.15	997.35	0.12
C-UnSAT-lin-ALCQ-1	14.12	1.53	3	1.81	0.04	0.13
C-UnSAT-lin-ALCQ-10	14.23	999.46	990.5	1082.35	997.52	0.14
C-UnSAT-lin-ALCQ-2	17.7	1.36	3.52	1.91	0.03	0.11

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
C-UnSAT-lin-ALCQ-3	16.98	1.39	15.88	3.99	0.23	0.12
C-UnSAT-lin-ALCQ-4	17.49	1.19	988.51	7.82	2.84	0.12
C-UnSAT-lin-ALCQ-5	19.84	2.77	1025.58	14.15	45.14	0.11
C-UnSAT-lin-ALCQ-6	19.4	11.55	989.3	33.29	736.43	0.13
C-UnSAT-lin-ALCQ-7	12.64	94.68	1008.07	147.78	996.46	0.14
C-UnSAT-lin-ALCQ-8	18.55	797.69	991.93	1084.79	994.89	0.13
C-UnSAT-lin-ALCQ-9	15.37	1000.95	993.12	1067.61	996.58	0.11
sat-unsat-3	12.91	1.62	3.05	1.74	0.02	0.11
sat-unsat-3-sol-1	12.35	1.37	2.7	1.67	0.02	0.11
sat-unsat-3-sol-10	13.86	1.68	49.36	1.95	0.03	0.12
sat-unsat-3-sol-2	13.99	1.61	2.64	2.83	0.02	0.12
sat-unsat-3-sol-4	14.59	1.81	3.66	2.68	0.03	0.12
sat-unsat-3-sol-6	23.85	1.38	4.24	1.45	0.03	0.11
sat-unsat-3-sol-8	18.08	1.38	13.04	2.11	0.03	0.12
sat-unsat-30-nosol-ALCHQ-1	10.92	2.81	1000.33	2.52	0.03	0.12
sat-unsat-30-sol-1	11.29	10.25	1000.96	33.89	0.03	0.11
sat-unsat-30-sol-10	13.89	10.16	1000.27	32	0.05	0.12
sat-unsat-30-sol-2	10.84	1.36	1001.55	3.71	0.02	0.11
sat-unsat-30-sol-4	11.74	3.38	1000.66	12.72	0.04	0.11
sat-unsat-30-sol-6	13.44	19.22	1001.94	41.71	0.04	0.11
sat-unsat-30-sol-8	16.59	3.19	1000.55	10.9	0.04	0.11
SHQ-CUnSAT-exp-1	24.39	1002.13	998.71	1066.92	994.79	0.12
SHQ-CUnSAT-exp-2	34.41	998.67	1000.52	1065.94	998.55	13.43
SHQ-CUnSAT-exp-3	36.94	999.79	22.34	1080.24	999.35	999.49
SHQ-CUnSAT-exp-4	35.16	999.37	1060.38	1070.06	1005.13	999.52
SHQ-CUnSAT-exp-5	40.99	59.98	1158.56	993.13	1012.7	999.43
SHQ-CUnSAT-exp-6	39.01	996.51	977.84	995.54	1008.02	998.43

TABLE C.4: Benchmarks for $\mathcal{SHQ}$-UNSAT Ontologies

C.5 Performance Evaluation of Satisfiable P Ontologies

Below we present the performance evaluation results on satisfiable **P** ontologies. These ontologies were initially created to validate classification results produced by Avalanche. This test suite grew from a dozen to more than 150 ontologies. Therefore we decided to keep them as they also give interesting insights on performance of other reasoners. The results are presented in Table C.5.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_1	11.69	2.11	2.29	2.01	0.02	0.11
p_10	12.08	2.63	2.56	3.05	0.03	0.12
p_10a	10.77	1.42	2.35	2.76	0.02	0.11
p_11	11.21	1.39	2.7	3.32	0.02	0.16
p_12	15.15	2.37	2.75	2.08	0.02	0.12
p_12a	11.99	1.94	2.94	1.94	0.02	0.12
p_12b	11.65	1.7	2.52	2.38	0.02	0.11
p_13	10.59	3.08	2.3	2.28	0.03	0.13
p_13a	13.07	1.32	3.14	1.89	0.02	0.11
p_14	10.42	1.23	3.97	3.51	0.02	0.14
p_14a	10.96	2.12	2.88	1.77	0.02	0.14
p_14b	11.75	1.35	4.19	2.84	0.02	0.14
p_15	14.82	1.48	2.36	2.11	0.02	0.13
p_15a	12.53	2.39	2.12	1.73	0.03	0.14
p_16	11.41	1.34	2.77	2.42	0.02	0.13
p_16a	13.97	2.12	2.69	3.82	0.02	0.13
p_17	10.86	1.25	3.5	1.74	0.03	0.14
p_18	12.97	2.11	3.99	2.82	0.03	0.11
p_19	16.14	1.96	3.07	2.18	0.03	0.12
p_2	12.88	1.24	2.93	2.03	0.01	0.11
p_20	15.17	1.7	2.49	1.6	0.02	0.19
p_21	43.27	3.15	2.23	1.93	0.02	0.12
p_22	47.57	2.02	2.74	1.64	0.02	0.13
p_22a	52.92	2.16	2.34	3.09	0.02	0.15
p_23	13.7	1.96	2.61	1.52	0.02	0.12
p_24	12.2	1.5	3.52	1.87	0.03	0.11
p_25	12.49	2.02	3.55	2.22	0.03	0.15
p_26	12.14	1.59	4.16	1.68	0.02	0.12
p_27	17.69	1.64	2.5	2.1	0.02	0.12
p_28	21.23	3.77	2.31	1.54	0.02	0.12
p_29	12.82	1.56	2.71	2.93	0.03	0.13
p_3	40.57	1.77	3.55	1.88	0.03	0.12
p_30	19.66	1.88	4.25	2.36	0.04	0.14
p_30a	16.66	1.73	3.02	3.32	0.04	0.14
p_30b	13.32	1.34	2.67	3.84	0.04	0.14
p_30c	14.8	1.29	3.17	2.76	0.03	0.14
p_30d	10.9	1.4	2.46	2.46	0.03	0.11
p_30small	17.52	1.33	3.96	3.08	0.04	0.13
p_31	10.6	1.16	2.76	2.41	0.02	0.11
p_32	21.18	2.11	277.98	2.06	0.03	0.15
p_32a	23.52	3.41	998.41	3.58	0.07	0.14
p_32b	14.94	1.72	16.04	1.78	0.03	0.13

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_32cdf	15.57	2.11	94.12	2.04	0.02	0.14
p_33	9.78	1.65	2.37	2.08	0.02	0.11
p_34	12.74	2.78	3	1.67	0.02	0.11
p_35	11.4	2.65	4.26	2.12	0.03	0.11
p_35a	12.8	2.7	2.43	1.78	0.02	0.11
p_35b	12.89	2.43	2.8	3.23	0.03	0.12
p_36a	10.98	2.37	2.84	1.58	0.03	0.12
p_36b	11.83	1.75	2.48	1.89	0.02	0.12
p_36c	12.64	1.56	2.5	2.3	0.02	0.16
p_37a	12.84	2.18	2.99	2.47	0.01	0.12
p_37b	10.67	2.26	2.31	1.89	0.02	0.11
p_38a	10.24	2.36	4.6	2.35	0.03	0.12
p_38b	14.82	2.57	2.5	1.49	0.03	0.13
p_39a	12.34	1.46	3.06	2.75	0.02	0.12
p_39b	17.23	1.27	2.88	2.1	0.03	0.12
p_3a	17.53	2.74	4.11	1.61	0.02	0.12
p_3b	18.23	1.82	3.69	3.58	0.03	0.13
p_3c	28.78	2.41	3.11	2.4	0.03	0.14
p_3d	18.92	1.55	2.95	1.87	0.03	0.14
p_3e	27.63	1.67	3.06	2.46	0.02	0.13
p_4	73.89	1.32	3.23	3.84	0.02	0.13
p_40	31.22	2.12	3.69	3.13	0.04	0.12
p_41a	27.29	2.81	2.88	1.99	0.03	0.11
p_41b	30.44	2.94	2.54	2.42	0.02	0.11
p_42a	26.36	1.75	3.05	1.9	0.03	0.11
p_42b	31.34	1.33	2.53	2.8	0.02	0.12
p_43	32.76	1.71	2.44	2.18	0.02	0.11
p_44	43.1	1.2	3.27	1.44	0.02	0.12
p_45	38.2	1.72	3.13	1.66	0.02	0.12
p_46	40.55	2.53	3.17	3.13	0.02	0.12
p_47	38.64	1.38	2.53	2.24	0.04	0.17
p_47a	32.7	3.06	3.82	2.38	0.01	0.12
p_48	42.49	2.35	3.16	3.03	0.03	0.2
p_49	56.25	1.87	3.7	1.84	0.02	0.2
p_4a	67.03	1.74	3.12	1.78	0.03	0.26
p_5	51.69	2.2	17.04	2.67	0.05	0.79
p_5-nnf	44.23	1.48	17.38	3.4	0.05	1.41
p_5-norm	190.41	2.29	17.88	3.24	0.04	1
p_50	64.7	1.23	3.73	2.59	0.02	0.12
p_51	67.94	2.67	3.61	2	0.03	0.13
p_52	218.5	1000.65	997.08	1088.79	999.13	0.15
p_53a	251.41	999.28	996.58	1077.15	992.5	0.14

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_53b	131.47	995.96	997.04	1073.61	993.71	0.15
p_53c	110.33	998.56	998.76	1077.35	993.74	0.14
p_53d	73.92	998.17	997.99	1080.3	991.61	0.14
p_53e	37.11	998.63	999.32	1101.75	990.94	0.13
p_53f	34.82	998.66	1003.21	1080	994.02	0.14
p_54	15.9	4.13	3.22	2.44	0.05	0.12
p_54a	20.82	2.15	3.24	2.53	0.04	0.13
p_54b	20.75	1.81	2.46	2.03	0.04	0.13
p_54c	14.53	1.08	2.87	1.82	0.06	0.12
p_54d	13.67	2.59	3.15	2.09	0.03	0.12
p_55	22.14	2.26	2.76	2.68	0.07	0.12
p_55a	12.23	1.94	2.35	1.65	0.04	0.11
p_56	18.72	999.42	1000.87	1086.36	989.87	1.19
p_56a	20.93	1005.36	1069.4	1087.33	986.96	80.84
p_56aa	20.26	12.88	874.31	37.63	1.73	0.31
p_56ab	20.46	4.94	7.18	12.31	0.18	0.18
p_56ab-norm	28.34	3.3	6.74	11.49	0.56	0.23
p_56ab-small	19.74	1.56	4.34	5.37	0.05	0.16
p_56ac	21.73	3.41	6.75	12.24	0.21	0.17
p_56ad	22.98	3.56	5.59	13.84	0.37	0.21
p_56ae	18.72	3.52	6.26	15.1	0.16	0.23
p_56af	21.11	4.41	6.4	15.04	0.33	0.17
p_56af-extended	24.5	4.72	5.96	13.98	1017.95	0.22
p_56ag	20.47	2.65	4.65	12.75	0.17	0.17
p_56ah	15.05	1.53	3.39	3.03	0.04	0.14
p_56ah-bug-1	13.59	1.84	3.46	1.9	0.03	0.12
p_56ah-bug-2	12.84	2.44	3.5	1.83	0.04	0.12
p_56ah-bug-mod-1	12.58	1.69	2.71	2.1	0.06	0.12
p_56ah-bug-mod-2	11.61	2.08	3.68	1.56	0.04	0.13
p_56ah-vh-1	14.43	2.14	3.88	2.64	0.06	0.14
p_56ah-vh-2	14.22	1.22	2.45	3.8	0.02	0.14
p_56ah-vh-small	12.4	2.54	3.27	2.64	0.05	0.13
p_56ai	12.78	1.77	3.02	2.27	0.05	0.12
p_56b	13.43	1.95	2.94	2.9	0.04	0.18
p_56c	20.63	1004.15	1026.9	1084.81	991.41	144.67
p_56d	21.73	1004.37	1035.97	1085.07	990.91	410.91
p_57a	12.65	20.98	1001.14	1077.53	990.29	0.7
p_57b	14.15	998.8	1004.14	1079.81	990.59	0.84
p_58a	12.77	2.01	997.75	10.23	0.04	0.94
p_58b	12.33	1.84	995.11	8.19	0.05	1.1
p_59a	13.03	3.52	17.74	2.5	0.03	0.55
p_59b	12.33	1.77	11.45	1.51	0.04	1.07

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_5a	62.5	2.07	14.33	1.96	0.04	1.17
p_5b	60.2	1.8	18.91	2.94	0.08	0.73
p_5b-norm	63.94	1.97	14.84	2.24	0.03	1.69
p_5b-norm-test	72.52	1.82	17.7	1.76	0.04	1.26
p_5ba	64.5	2.36	16.47	2.72	0.04	1.3
p_5ba-norm	67.68	1.35	20.55	1.74	0.04	1.85
p_5ca	69.36	1.97	15.28	1.74	0.03	0.96
p_5cb	63.51	1.51	15.74	2.58	0.05	0.73
p_5d	49.29	1.4	16.81	2.42	0.03	0.7
p_6	15.13	2.26	9.99	2.25	0.04	1.06
p_60a	11.81	1.55	14.48	1.74	0.06	0.61
p_60b	12.82	3.21	14.79	2.07	0.05	0.39
p_61a	14.24	1.67	18.62	1.45	0.04	0.45
p_61b	14.56	1.62	24.36	1.71	0.03	0.48
p_62a	11.99	1.57	29.21	1.51	0.03	0.42
p_62b	12.15	2.47	22.26	1.68	0.06	0.34
p_62c	13.77	2.21	19.86	2.22	0.03	0.51
p_63	10.65	2.41	22.69	2	0.04	0.8
p_64	10.26	1.33	28.25	1.75	0.03	0.28
p_6a	14.65	2.03	34.28	2.29	0.04	0.34
p_6b	13.14	2.77	34.56	3.1	0.05	0.64
p_6c	12.36	2.74	20.13	2.26	0.04	0.78
p_6d	13.9	2.21	14.14	1.68	0.04	0.72
p_6e	11.75	2.18	13.8	1.61	0.05	1.23
p_6f	12.2	1.83	19.82	1.87	0.03	0.96
p_6g	13	2.83	19.34	1.55	0.05	1.15
p_6h	12.23	1.5	15.18	2.14	0.04	1.19
p_7	13.66	1.79	3.42	1.46	0.05	0.35
P_70a	13.52	1.93	7.34	1.6	0.04	1.21
p_70b	11.28	2.24	9.11	2.4	0.03	1.24
p_71a	11.54	1.83	14.5	1.73	0.03	1.03
P_71b	11.22	1.62	17.66	1.83	0.04	0.8
p_72	14.18	1.18	17.36	1.39	0.08	0.78
p_73a	11.38	1.59	11.03	2.39	0.03	0.82
p_73b	13.27	2.33	16.32	1.81	0.03	0.79
p_74	13.04	1.99	15.36	2.05	0.06	0.87
p_75a	28.94	1.65	7.44	2	0.03	0.79
p_75aa	22.49	1.93	17.77	2.13	0.04	0.6
p_75b	27.03	2.05	7.28	1.67	0.04	1.17
p_76a	11.61	1.25	5.74	1.83	0.04	0.8
p_76b	14.51	1.86	7.31	2.2	0.04	0.84
p_76c	10.19	1.62	5.05	2	0.03	0.66

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_76d	11.8	1.77	4.2	1.75	0.04	0.93
p_77	12.42	1.33	5.33	1.69	0.04	28.13
p_78a	12.88	1.79	6.84	2.05	0.03	0.17
p_78b	14.12	1.72	7.45	1.66	0.03	0.34
p_78c	14.86	1.61	5.57	1.87	0.03	0.21
p_78d	12.33	2.53	4.6	1.66	0.04	0.25
p_78e	12.62	3.22	4.8	3.23	0.05	0.25
p_79	13.25	2.37	3.81	1.33	0.04	0.24
p_8	12.38	1.43	3.35	1.66	0.04	0.41
p_80	11.6	2.3	5.24	1.54	0.06	0.37
p_9	11.36	1.76	3.56	2.07	0.03	0.28

TABLE C.5: Benchmarks for Performance Ontologies

C.6 Performance Evaluation of Unsatisfiable P Ontologies

Below we present performance evaluation on unsatisfiable **P** ontologies. The difference between satisfiable and unsatisfiable **P** ontologies is that the concept Thing is unsatisfiable in unsatisfiable ontologies. The results are presented in Table C.6.

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_01	11.22	2.5	2.2	3.32	0.03	0.11
p_02	11.1	3.22	2.69	3.85	0.03	0.11
p_03	13.71	2.23	2.43	2.73	0.04	0.11
p_03a	12.32	4.35	2.85	2.87	0.04	0.12
p_03b	14.02	2.85	2.84	3.24	0.03	0.13
p_03c	16.43	1.46	3.5	3.81	0.03	0.12
p_03d	12.27	1.7	3.03	3.16	0.03	0.12
p_03e	17.31	1.38	3.07	3.2	0.03	0.12
p_04	13.33	2.48	2.64	3.36	0.03	0.11
p_04a	11.76	1.57	2.64	3.47	0.02	0.12
p_05	15.76	1.49	3.45	3.72	0.03	0.13
p_05-nnf	13.75	1.21	2.61	3.1	0.03	0.13
p_05-norm	21.18	2.4	3.15	4.05	0.05	0.15
p_05a	15.06	1.93	3.19	2.45	0.02	0.12
p_05b	15.86	1.42	2.69	2.72	0.03	0.12

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_05b-norm	14.66	1.65	3.19	3.33	0.02	0.13
p_05b-norm-test	16.94	2.21	2.19	3.4	0.02	0.12
p_05ba	16.98	2.1	2.94	2.34	0.04	0.11
p_05ba-norm	17.25	1.29	2.89	2.2	0.02	0.12
p_05ca	16.54	2.05	2.54	3.11	0.04	0.13
p_05cb	14.2	1.81	3.04	3.26	0.02	0.13
p_05d	16.25	2.14	2.7	3.44	0.03	0.13
p_06	13.19	1.94	2.16	4.1	0.02	0.11
p_06a	18.29	1.27	3.49	2.6	0.03	0.13
p_06b	12.61	1.88	2.45	2.75	0.02	0.12
p_06c	11.04	1.95	3.36	2.22	0.04	0.16
p_06d	12.11	2.2	3.51	3.62	0.03	0.13
p_06e	11.16	1.45	2.5	3.38	0.04	0.13
p_06f	10.93	1.8	2.86	2.92	0.03	0.12
p_06g	14.98	1.6	3.76	3.13	0.03	0.13
p_06h	11.56	1.9	5.02	2.65	0.03	0.12
p_07	10.51	1.73	2.71	2.74	0.04	0.11
p_08	13.2	2.16	2.75	3.81	0.02	0.1
p_11	12.59	1.99	2.69	2.95	0.02	0.1
p_12	15.84	1.38	2.1	3.3	0.03	0.11
p_12a	12.91	1.29	2.74	3.55	0.03	0.11
p_12b	11.96	1.78	2.15	2.71	0.01	0.11
p_13	19.18	1.28	2.91	3.33	0.02	0.11
p_14	14.6	1.13	2.65	3.03	0.02	0.12
p_14b	20.24	1.77	2.32	3.52	0.02	0.13
p_15	10.99	1.4	2.93	3.83	0.02	0.11
p_15a	13.16	1.44	3.23	3.16	0.04	0.11
p_16	14.48	2.05	2.25	2.76	0.04	0.11
p_16a	12.73	1.93	3.35	2.92	0.02	0.11
p_17	11.59	1.58	3.18	2.79	0.03	0.11
p_18	12.96	1.06	3.95	2.91	0.03	0.11
p_19	11.53	1.86	2.43	4.16	0.03	0.12
p_20	14.5	1.43	2.05	2.72	0.02	0.15
p_21	12.79	2.6	2.57	3.91	0.03	0.12
p_22	17.19	1.97	3.17	2.4	0.02	0.12
p_22a	19.27	1.67	2.99	3.46	0.03	0.13
p_23	11.95	1.42	2.63	2.98	0.03	0.11
p_24	12.23	2.24	3.06	4.44	0.03	0.11
p_25	11.7	1.71	2.45	2.53	0.02	0.11
p_26	12.09	2.79	2.14	3.4	0.02	0.12
p_27	12.92	1.58	2.82	3.18	0.02	0.11
p_29	13.17	1.39	2.9	2.32	0.03	0.11

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_30	13.01	1.96	3.5	3.15	0.02	0.13
p_30a	12.39	2.54	2.75	3.86	0.02	0.12
p_30b	13.55	2.58	2.48	3.58	0.03	0.14
p_30c	12.97	1.39	2.94	3.2	0.02	0.13
p_30d	12.45	1.6	2.58	3.32	0.03	0.11
p_30small	12.34	2.59	3.83	3.33	0.03	0.19
p_31	10.51	2.3	2.65	2.68	0.03	0.12
p_32	16.07	1.07	77.18	3.32	0.03	0.14
p_32a	25.08	3.14	1009.81	3.83	0.03	0.13
p_32b	14.04	2.14	10.99	3.18	0.06	0.13
p_32cdf	12.32	1.61	99.18	3.55	0.06	0.14
p_33	13.19	1.83	2.39	3.18	0.05	0.1
p_34	11.31	1.23	3.06	4.07	0.04	0.14
p_35	12.36	1.1	2.73	2.19	0.03	0.1
p_35a	11.81	2.01	2.64	2.17	0.05	0.11
p_35b	9.49	2.6	4.2	2.7	0.04	0.11
p_36a	16.32	1.18	2.89	2.31	0.04	0.1
p_36b	15.98	1.7	3.05	2.02	0.05	0.11
p_36c	17.38	1.29	2.29	1.69	0.04	0.11
p_37a	16.83	1.84	4.24	1.66	0.05	0.1
p_37b	16.34	2.06	2.91	2.13	0.04	0.1
p_38a	12.99	1.68	3.78	3.08	0.04	0.11
p_38b	15.82	2.56	4.49	2.12	0.04	0.11
p_39a	12.66	1.75	2.46	1.87	0.03	0.12
p_39b	11.21	1.75	2.82	2.86	0.17	0.11
p_40	11.94	1.94	3.49	2.41	0.29	0.12
p_41a	11.48	1.08	2.38	1.78	0.05	0.1
p_41b	10.67	1.81	3.21	2.51	0.06	0.1
p_42a	10.42	2.18	2.23	1.47	0.08	0.1
p_42b	10.13	2.15	3.21	1.65	0.05	0.1
p_43	12.15	2.59	3.44	1.91	0.03	0.09
p_44	12.2	1.59	3.22	1.37	0.2	0.1
p_45	12.4	1.58	2.97	1.59	0.03	0.12
p_46	12.66	1.8	4.17	2.15	0.03	0.1
p_47	9.78	1.64	2.49	2.81	0.05	0.1
p_47a	12.5	1.94	3.98	1.72	0.05	0.1
p_48	12.15	1.16	3.16	2.14	0.07	0.11
p_49	12.46	2.22	3.8	2.94	0.04	0.12
p_50	14.33	1.78	2.88	2.98	0.03	0.15
p_51	17.92	1.4	4.24	1.72	0.07	0.15
p_52	80.72	1001.47	997.11	1080.12	998.87	0.19
p_53a	158.96	1002.12	996.83	1069.08	999.51	0.11

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_53b	149.68	1001.64	995.86	1072.41	999.39	0.25
p_53c	152.05	1001.15	997.85	1075.64	999.58	0.21
p_53d	30.12	1002.22	995.7	1077.39	999.58	0.39
p_53e	23.46	1001.24	1000.48	1078.44	999.48	0.46
p_53ea	25.45	1001.07	994.65	1087.03	999.5	0.7
p_53ea-small	28.24	1006.27	1016.16	1074.68	999.46	0.19
p_53eb	25.61	1001.46	994.26	1080.28	999.52	0.4
p_53ec	28.04	1001.47	995.44	1076.19	999.47	0.43
p_53f	24.75	1001	1000.95	1074.14	999.49	0.44
p_54	12.88	1.2	2.73	2.95	0.03	0.14
p_55	14.61	2.39	3.38	1.94	0.04	0.13
p_55a	12.13	2.52	2.65	1.8	0.03	0.11
p_56	28.28	1000.82	1002.31	1080.39	999.51	80.75
p_56a	13.3	1006.11	1063.6	1016.95	999.33	994.66
p_56aa	13.7	3.4	1005.45	16.88	0.27	0.26
p_56ab	15.26	2.46	4	5.27	0.05	0.12
p_56ac	15.23	2.03	4.88	4.03	0.05	0.13
p_56ad	14.59	2.86	4.01	6.14	0.08	0.12
p_56ae	15.63	2.01	4.11	4.7	0.05	0.12
p_56af	20.35	2.1	4.34	4.1	0.07	0.13
p_56af-extended	18.15	1.34	4.9	10.91	0.06	0.13
p_56ag	15.12	2.67	4.58	4.78	0.06	0.12
p_56ah	10.75	1.52	2.87	1.8	0.04	0.13
p_56ai	13.99	1.46	3.93	1.96	0.03	0.13
p_56c	14.52	1007.71	1013.21	1047.12	999.51	1.71
p_56d	15.67	1007.95	1045.26	1025.08	999.5	984.93
p_57a	14.93	24.5	1022.43	58.18	79.29	1.17
p_57b	14.77	1001.39	1000.11	1077.13	999.5	77.23
p_58a	13.05	1.6	997.95	8.4	0.05	2.16
p_58b	11	3.21	996.34	6.77	0.05	2.25
p_59a	10.39	1.58	18.76	1.73	0.03	1.77
p_59b	11.19	2.05	14.31	2.45	0.03	1.93
p_60a	13	1.6	14.48	1.87	0.02	1.56
p_60b	12.27	1.46	17.74	1.53	0.04	1.98
p_61a	13.05	1.52	15.38	2.71	0.03	2.16
p_61b	11.35	1.67	14.06	2.02	0.04	2.04
p_62a	12.2	1.48	13.37	2.14	0.04	1.93
p_62b	14.29	1.87	21.09	2.31	0.05	1.76
p_62c	13.6	1.47	13.74	2.12	0.03	1.43
p_63	13.33	2.84	13.82	2.55	0.04	1.22
p_64	11.35	1.52	15.31	1.81	0.02	1.36
p_70a	12.12	1.33	14.93	3.06	0.03	1.23

Ontology	Avalanche	Fact++	Hermit	JFact	Konclude	Racer
p_70b	11.48	1.81	12.09	1.87	0.03	0.87
p_72	12.81	1.75	16.82	1.98	0.03	0.72
p_73a	11.29	1.55	22.97	1.42	0.04	0.94
p_73b	10.64	1.5	23.02	2.5	0.03	0.52
p_74	11.54	2.17	26.98	2.52	0.03	1.19
p_75a	14.52	1.67	20	2.22	0.04	0.88
p_75aa	13.66	1.65	20.41	1.71	0.03	0.98
p_75b	15	2.65	23.73	2.28	0.03	0.73
p_76	11.57	1.31	28.27	2.19	0.02	0.74
p_78a	12.97	1.99	35.53	2.21	0.03	0.89
p_78b	12.89	1.05	39.41	2.54	0.03	0.7
p_78c	10.78	1.29	17.25	2.31	0.03	1.36
p_78d	11.84	2.47	19.22	1.67	0.03	1.48
p_79	11.87	1.7	14.58	1.91	0.02	1.02

TABLE C.6: Benchmarks for Unsatisfiable Performance Ontologies