COMBINING INTEGER PROGRAMMING AND TABLEAU-BASED REASONING: A HYBRID CALCULUS FOR THE DESCRIPTION LOGIC $\mathcal{SHQ}$

NASIM FARSINIAMARJ

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

NOVEMBER 2008

© NASIM FARSINIAMARJ, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-63309-0
Our file Notre référence
ISBN: 978-0-494-63309-0

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Nasim Farsiniamarj

Entitled: **Combining Integer Programming and Tableau-based Reasoning: A Hybrid Calculus for the Description Logic $\mathcal{SHQ}$**

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

Master of Computer Science

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

Signed by the final examining commitee:

____________________ Chair

Dr. Nematollaah Shiri

____________________ Examiner

Dr. Leila Kosseim

____________________ Examiner

Dr. René Witte

____________________ Supervisor

Dr. Volker Haarslev

Approved ____________________

Chair of Department or Graduate Program Director

________ 20 ____ ____________________

Dr. Robin Drew, Dean

Faculty of Engineering and Computer Science

Abstract

Combining Integer Programming and Tableau-based Reasoning: A Hybrid Calculus for the Description Logic $\mathcal{SHQ}$

Nasim Farsiniamarj

Qualified cardinality restrictions are expressive language constructs which extend the basic description logic $\mathcal{ALC}$ with the ability of expressing numerical constraints about relationships. However, the well-known standard tableau algorithms perform weakly when dealing with cardinality restrictions. Therefore, an arithmetically informed approach seems to be inevitable when dealing with these cardinality restrictions. This thesis presents a hybrid tableau calculus for the description logic $\mathcal{SHQ}$ which extends $\mathcal{ALC}$ by qualified cardinality restrictions, role hierarchies, and transitive roles. The hybrid calculus is based on the so-called atomic decomposition technique and combines arithmetic and logical reasoning. The most prominent feature of this hybrid calculus is that it reduces reasoning about qualified number restrictions to integer linear programming. Therefore, according to the nature of arithmetic reasoning, this calculus is not affected by the size of numbers occurring in cardinality restrictions. Furthermore, we give evidence on how this method of hybrid reasoning can improve the performance of reasoning by organizing the search space more competently. An empirical evaluation of our hybrid reasoner for a set of synthesized benchmarks featuring qualified number restrictions clearly demonstrates its superior performance. In comparison to other standard description logic reasoners, our approach demonstrates an overall runtime improvement of several orders of magnitude.

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Volker Haarslev, Dr. rer.-nat. habil., whose expertise, understanding, and patience added considerably to my graduate experience and his priceless remarks introduced me to the world of academia. His words and suggestions often boosted my courage and determination to write this thesis and helped polish my written English.

Very special thanks to my family for their support and love. I thank my father, Abbas Farsinia, for the discussions we had about the philosophy of mind which shaped my perspective about AI, and my mother, Dr. Ensieh Amin, whose guidance about academic work was and always be a great assist in my way. I would also thank my sister Negar for her boundless love.

I am also grateful to my colleagues Jiewen and Francis who helped me during the implementation and encouraged me while writing this report. And also I would like to thank my colleagues and friends Amir, Nasim, Jocelyne, Ahmed, and Jinzan.

Finally I would like to express my appreciation to Mary Shelly for '*Frankenstein*' and Arthur C. Clarke for '*The City and the Stars*' which developed my initial motivations to pursue my studies in Artificial Intelligence.

"Information is not knowledge,

Knowledge is not wisdom,

Wisdom is not truth,

Truth is not beauty,

Beauty is not love,

Love is not music;

and Music is THE BEST."

Frank Zappa

in "Joe's Garage: Act II & III" (Tower Records, 1979)

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Numbers in Description Logics 2

1.2 Research Objectives and Contributions 2

1.2.1 Objectives 2

1.2.2 Contributions 3

1.3 Thesis Organization 3

2 Preliminaries 5

2.1 Description Logics 5

2.1.1 Knowledge Base 6

2.1.2 Languages 7

2.1.3 Reasoning services 10

2.1.4 Tableau Reasoning 11

2.1.5 Correctness of an Algorithm 15

2.2 Complexity of Reasoning 17

2.2.1 Complexity of the Language: Theoretical Complexity 17

2.2.2 Complexity of the Algorithm: Practical Complexity 17

2.3 Atomic Decomposition 18
2.4 Integer Programming 19
2.4.1 Simplex 20
2.4.2 From Linear to Integer 20
2.5 Summary 22

3 Qualified Number Restrictions 23
3.1 Standard Tableau Algorithms 24
3.2 Complexity of Reasoning with $\mathcal{Q}$ 26
3.3 Optimizing Reasoning with $\mathcal{Q}$ 26
3.3.1 Signature Calculus 27
3.3.2 Algebraic Methods 28
3.3.3 A Recursive Algorithm for $\mathcal{SHQ}$ 28
3.3.4 Dependency-Directed Backtracking 29
3.4 Summary 30

4 A Hybrid Tableau Calculus for $\mathcal{SHQ}$ 31
4.1 Preprocessing 32
4.1.1 Converting $\mathcal{Q}$ to $\mathcal{N}$ 32
4.1.2 TBox Propagation 34
4.2 Atomic Decomposition 34
4.2.1 Example 35
4.3 Architecture 36
4.4 Tableau Rules for TBox Consistency 37
4.4.1 Completion Graph 37
4.4.2 Expansion Rules 38
4.4.3 Rule Descriptions 39

4.4.4 Example TBox 41
4.5 ABox Consistency 42
4.5.1 A Hybrid Algorithm for ABox Consistency 43
4.5.2 Tableau Rules for ABox Consistency 45
4.5.3 Example ABox 50
4.6 Proof of Correctness 52
4.6.1 Tableau 53
4.6.2 Termination 55
4.6.3 Soundness 57
4.6.4 Completeness 59

5 Practical reasoning 64
5.1 Complexity Analysis 65
5.1.1 Standard Tableaux 65
5.1.2 Hybrid Tableau 66
5.1.3 Hybrid vs. Standard 66
5.2 Optimization Techniques 68
5.2.1 Default Value for the Variable 69
5.2.2 Strategy of the *ch*-Rule 69
5.2.3 Variables Encoding 70
5.3 Dependency-Directed Backtracking or Backjumping 71
5.3.1 Backtracking in the Arithmetic Reasoner 72
5.3.2 Backjumping: Standard Algorithms vs. the Hybrid Algorithm 73

6 Reasoner Description 75
6.1 Architecture 75
6.2 Logical Module 77

6.2.1 Expansion Rules 78
6.3 Arithmetic Reasoner 83
6.3.1 Atomic Decomposition 84
6.3.2 Preprocessing 85
6.3.3 Branching 89
6.3.4 Integer Programming 90
6.4 Problems 91
6.5 Summary 92

7 Evaluation **93**

7.1 Benchmarking 93
7.2 Evaluation Results 94
7.2.1 Increasing Numbers 95
7.2.2 Backtracking 99
7.2.3 Satisfiable vs. Unsatisfiable Concepts 101
7.2.4 Number of Cardinality Restrictions 105
7.2.5 Number of At-least vs. Number of At-most 106
7.3 Conclusion 108

8 Conclusion and Future Work **109**

8.1 Discussion 110
8.1.1 Advantages 110
8.1.2 Disadvantages 111
8.1.3 Comparison with Related Work 112
8.2 Future Work 113
8.2.1 Extending and Refining the current research 113
8.2.2 Introducing New Structures 114

Bibliography 116

List of Figures

1	Grammar of the basic language $\mathcal{AL}$.	7
2	Example knowledge base	10
3	Sample tableau rules	13
4	Complexity of different DL languages	17
5	The rules handling $\mathcal{Q}$	24
6	Atomic Decomposition Example	36
7	Expansion rules for $\mathcal{SHQ}$ TBox	39
8	Expansion rules for $\mathcal{SHQ}$ ABox consistency	46
9	Initial completion graph. Dashed edges do not actually exist.	51
10	Completion graph after merging b and c	51
11	Final completion graph	53
12	Converting forest $\mathcal{F}$ to tableau T	58
13	Reasoner architecture	76
14	Architecture of the logical module	78
15	Illustration of the rules application when backtracking	83
16	Architecture of the arithmetic module	84
17	Comparing with standard algorithm: Effect of the value of numbers	97
18	Behavior of the hybrid reasoner.	98
19	Behavior of the hybrid reasoner: Linear growth of i	99
20	Behavior of the hybrid reasoner: Exponential growth of i	100

21 Effect of backtracking in different levels 101

22 Effect of backtracking in different levels: Number of clashes 102

23 The effect of satisfiability 103

24 The effect of satisfiability: The hybrid algorithm 104

25 The effect of number of qualified number restrictions 105

26 Ratio between number of at-least to the number of at-most restrictions 107

List of Tables

1 Semantics and symbols of different DL constructs. 8

2 Number of at-least restrictions - number of at-most restrictions 107

Chapter 1

Introduction

In the realm of artificial intelligence, developing a machine with the ability of understanding has always been a goal. The level of understanding grows from data to information, knowledge and perhaps it reaches wisdom. For this reason, logics where adopted as a medium for representing knowledge and to upgrade the level of understanding of the machine. Description Logic (DL) is a formal language to represent the knowledge about concepts, individuals, and their relations. It was intended to extend semantic networks with formal logical semantics. In the domain of the semantic web, the Web Ontology Language (OWL) is based on description logics. Moreover, it is widely used in various application domains such as configuration problems and medical informatics.

In order to be decidable and have more efficient decision procedures, description logic usually covers a limited subset of first order logic. However, it includes some expressive operations that are not in propositional logic. One of these constructors is the so-called numerical restriction which equips the language with the ability of expressing numerical restrictions on relationships.

1.1 Numbers in Description Logics

Qualified number restrictions are expressive constructs which express cardinality constraints on the relationships between individuals. For example, to model an engineering undergraduate program, the expression Student $\equiv$ ($\geq$ 141hasCredit) states that every student must have at least 141 credits. Similarly, the expression EngineeringStudent $\equiv$ ($\leq$ 10hasCredit.Arts) implies that every Engineering student must take at most 10 credits from the department of Arts.

There is always a trade-off between the expressiveness of the language and the efficiency of the algorithm implementing inference services. Likewise, adding numerical restrictions brings an extra complexity to the reasoning process. However, the standard tableau algorithms [HB91, BBH96, HST00, HS05] approach numerical expressions from a logical perspective. Therefore, standard reasoning approaches perform inefficiently when dealing with numerical restrictions.

1.2 Research Objectives and Contributions

Current standard approaches [BBH96, HST00, HS05] are known to be inefficient in dealing with numerical restrictions since they deal with numbers similar to logical expressions and ignore their arithmetic nature. Moreover, their trial-and-error approach to the problem of numerical constraints can become considerably expensive in the case of cardinality restrictions. Motivated by [OK99], we propose an algorithm which translates numerical semantics of DL into linear inequations.

1.2.1 Objectives

In this research we pursue the following objectives:

- Develop a formal hybrid calculus which combines the so-called tableau approach

with Linear Integer Programming in order to efficiently deal with numerical restrictions. Moreover, the calculus should be well-suited to be extended to more expressive logics.

- Examine the feasibility of implementing a reasoner based on the proposed calculus and evaluate the performance of such a hybrid reasoner for selected benchmark problems.

1.2.2 Contributions

We can summarize the contributions of this research as follows:

1. We present a hybrid tableau calculus which combines tableau reasoning with Integer Linear Programming for $\mathcal{SHQ}$ ABoxes. Moreover, we give a proof of termination, soundness, and completeness for the proposed tableau calculus.
2. We analyze the complexity of the proposed hybrid algorithm in comparison with the standard approaches.
3. We study practical aspects of implementing such a hybrid reasoner and accordingly propose a set of optimization techniques.
4. We report on the development of a hybrid prototype for the $\mathcal{ALCHQ}$ concept satisfiability test benefiting from the proposed optimization techniques.
5. We study the behavior of the hybrid reasoner and evaluate the effect of the techniques with respect to different parameters influencing hybrid reasoning.

1.3 Thesis Organization

This thesis consists of 8 chapters which are organized as in the following outline. Chapter 2 provides the required background about description logics and integer

programming. In Chapter 3, we briefly study current standard tableau algorithms dealing with cardinality restrictions as well as optimization techniques addressing the complexity of reasoning with numbers. In Chapter 4 which can be considered as the theory chapter, we formally present a hybrid calculus for $\mathcal{SHQ}$ taxonomy level which will later be extended to also handle assertional knowledge. At the end of this chapter we give a formal proof for the correctness of the proposed calculus.

In Chapter 5, we analyze the complexity of the proposed hybrid algorithm together with a comparison with the standard tableau algorithm. Moreover, in this chapter, we introduce a set of optimization techniques developed for hybrid reasoning. Afterward in Chapter 6, we describe the architecture of an implemented hybrid reasoner and report on its empirical evaluation in Chapter 7. In the last chapter, Chapter 8, we conclude this thesis with the lessons we learned from the theoretical analysis and empirical evaluation. Finally, we suggest several future work to pursue.

Chapter 2

Preliminaries

After having motivated the need to design a hybrid algorithm, we now present the formalisms required to discuss the related work in Chapter 3 and propose a new algorithm in Chapter 4. In the first section we formally define DL languages, knowledge bases, and reasoning services. Moreover, we define a tableau as a widely used approach for different DL reasoning services. In Section 2.2, we define and clarify the use of the term "complexity" which sometimes refers to the inherent complexity of the input language and sometimes refer to the complexity of the algorithm which decides it. Finally, we briefly define integer programming and possible alternatives to solve it.

2.1 Description Logics

Description Logics is a family of conceptual knowledge representation formalisms which is also in most cases a decidable fragment of first order logics. Since, it is aimed to be suitable for reasoning about the domain of discourse (the "world"), its expressiveness is restricted in comparison with first order logics. Moreover, unlike predicate logics where the structure of the input knowledge is often lost, description

logic is composed of three building blocks: concepts which are subsets of the domain, roles which define binary relations between elements of the domain, and individuals which are simply elements of the domain of reasoning. In the following sections we formally define $\mathcal{ALC}$ as the simplest propositionally complete subset of description logics and afterward define its different possible extensions.

In the following, we use A and B as atomic concept names which are subset of the domain, R and S as role names, C and D as concept expressions (possibly not atomic) which are built by means of different operators, and $a, b, c, \ldots$ as individuals.

Definition 1 (Interpretation). *In order to formally define different variations of description logics we define an interpretation, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ in which $\Delta^{\mathcal{I}}$ is a non-empty set referring to the domain and $\cdot^{\mathcal{I}}$ is the interpretation function that assigns a subset of the domain ($\Delta^{\mathcal{I}}$) to every concept name ($A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$) and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every role name ($R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$).*

2.1.1 Knowledge Base

A typical knowledge base presented in description logics is composed of two parts: the terminological knowledge (TBox) which describes the *vocabulary* and characteristics of the domain and the assertional knowledge (ABox) which contains the asserted knowledge regarding the actual individuals of the domain.

A *concept inclusion axiom* is a statement of the form $C \sqsubseteq D$. An interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ *iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and similarly satisfies $C \equiv D$ *iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and $D^{\mathcal{I}} \subseteq C^{\mathcal{I}}$.

Definition 2 (TBox). A TBox $\mathcal{T}$ is a finite set of axioms of the form $C \sqsubseteq D$ or $C \equiv D$ where C, D are concept expressions.

Definition 3 (ABox). An ABox $\mathcal{A}$ with respect to a TBox $\mathcal{T}$ is a finite set of

assertions of the form $a : C$, $(a, b) : R$, and $a \neq b$, where $a : C$ is satisfied if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, $(a, b) : R$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$, and $a \neq b$ if $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.

2.1.2 Languages

In this section we introduce the syntax and semantics of different constructs in description logics which may vary in different languages. The base DL language, $\mathcal{AL}$ (attributive language) introduced in [SSS91], is constructed as suggested in [Baa03] based on the grammar presented in Figure 1.

$C, D \rightarrow$	$\mid A$	atomic concept
	$\mid \top$	top or universal concept
	$\mid \bot$	bottom or empty concept
	$\mid \neg A$	atomic negation
	$\mid C \sqcap D$	conjunction
	$\mid \forall R.C$	universal restriction
	$\mid \exists R.\top$	unqualified existential restriction

Figure 1: Grammar of the basic language $\mathcal{AL}$.

The semantics of the constructs introduced in Figure 1 is defined as follows:

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \forall b, (a, b) \in R^{\mathcal{I}} \Rightarrow b \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \exists b, (a, b) \in R^{\mathcal{I}}\}.
\end{aligned}$$

One of the basic DL languages that is propositionally complete is $\mathcal{ALC}$. This language contains negation for general concept expressions, disjunction between concept expressions, and also qualified existential restrictions in addition to $\mathcal{AL}$. More expressive languages can have several extra features and constructs such as cardinality

restrictions, nominals, or inverse roles. In Table 1, we introduce syntax and semantics of basic DL languages as well as some well-known expressive constructs. In the following, let $R^{\#}(a, C)$ denote the cardinality of the set $\{b \in \Delta^{\mathcal{I}} | (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$.

Table 1: Semantics and symbols of different DL constructs.

Syntax	Semantics	Symbols
$(\neg C)^{\mathcal{I}}$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$	$\mathcal{C}$
$(C \sqcup D)^{\mathcal{I}}$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	$\mathcal{U}$
$(\exists R.C)^{\mathcal{I}}$	$\{a \in \Delta^{\mathcal{I}} \| \exists b, (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$	$\mathcal{E}$
$(\geq nR.C)^{\mathcal{I}}$	$\{a \in \Delta^{\mathcal{I}} \| R^{\#}(a, C) \geq n\}$	$\mathcal{Q}$
$(\leq mR.C)^{\mathcal{I}}$	$\{a \in \Delta^{\mathcal{I}} \| R^{\#}(a, C) \leq m\}$	$\mathcal{Q}$
$(\geq nR)^{\mathcal{I}}$	$(\geq nR.\top)^{\mathcal{I}}$	$\mathcal{N}$
$(\leq mR)^{\mathcal{I}}$	$(\leq mR.\top)^{\mathcal{I}}$	$\mathcal{N}$
R^-	$(a, b) \in R^{\mathcal{I}} \Leftrightarrow (b, a) \in (R^-)^{\mathcal{I}}$	$\mathcal{I}$
$R \sqsubseteq S$	$(a, b) \in R^{\mathcal{I}} \Rightarrow (a, b) \in (S)^{\mathcal{I}}$	$\mathcal{H}$
$\{o\}$	$\{o\}$ is a concept and $\|\{o\}^{\mathcal{I}}\| = 1$	$\mathcal{O}$
$Trans(R)$	$(a, b) \in R^{\mathcal{I}} \wedge (b, c) \in R^{\mathcal{I}} \Rightarrow (a, c) \in R^{\mathcal{I}}$	$\mathcal{S}$

If a and b are two individuals such that $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$, we call b an *R-successor* of a. Moreover, b is a role-filler (R-filler) for a. One of the expressive features that can equip the language with the ability of counting are cardinality restrictions. Unqualified number restrictions ($\mathcal{N}$), specify the least or most number of allowed successors for an individual. For example, the concept inclusion axiom Person $\sqsubseteq$ ($\leq$2 hasBiologicalParent) $\sqcap$($\geq$2 hasBiologicalParent) indicates that every individual which is a member of the concept Person has exactly two (distinct) hasBiologicalParent-successors.

In the case of qualified number restrictions ($\mathcal{Q}$), the cardinality restriction not only restricts the number of successors, but also specifies in which concept expression they must be. In fact, unqualified number restrictions, and existential restrictions ($\mathcal{E}$) are special cases of qualified number restrictions ($\exists R.C \equiv (\geq 1 R.C)$). For example, the assertion a : ($\leq$1hasParent.Male) states that a has at-most one hasParent-successor that is also a member of the concept Male. Notice, since we have the *open-world assumption* in DL reasoning, not having asserted that a successor of a is in the

concept Male does not imply that it is a member of ¬Male. Therefore, if one of the hasParent-successors of a is in Male, the rest must be explicitly in ¬Male.

Transitive roles are another expressiveness that is normally added to $\mathcal{ALC}$ which is referred by $\mathcal{S}$. Role inclusion axioms are also another type of axiom that are stored in the *role hierarchy* component of the KB (another component in addition to TBox and ABox). The language of interest in this research is $\mathcal{SHQ}$ which is basically $\mathcal{ALC}$ plus role hierarchies, transitive roles, and most significantly qualified cardinality restrictions.[1] Other expressive components such as nominals ($\mathcal{O}$) which introduce the use of individuals at the concept level or inverse roles ($\mathcal{I}$) are not considered in this thesis.

Example

By means of the example ontology in Figure 2, we will describe different terms, defined in this section. In this example, according to the TBox axiom (t.1), Employed and Unemployed are disjoint concepts. The second axiom (t.2) implies that the concept Lonely is subsumed by the unqualified number restriction ($\leq$1hasFriend) and Supported is a subconcept of the qualified number restriction ($\geq$2hasFriend.Considerate) according to (t.3). The last inclusion axiom (t.4), indicates that if an individual is Lonely or Unemployed, it is also a member of the concept Unlucky. Moreover, since every instance of Supported has at least two hasFriend-fillers while every instance of Lonley has at most one hasFriend-filler, the reasoner can infer that Lonely and Supported are two disjoint concepts.

In the assertional level, jack as a named individual, is a member of the concept Unemployed. Therefore, the reasoner can conclude that jack is not a member of the

[1] In this thesis, the terms number restriction and cardinality restriction are different terms referring to the same concept.

TBox		ABox	
Employed $\equiv \neg$Unemployed	(t.1)	jack : Unemployed	(a.1)
Lonely $\sqsubseteq \leq 1$hasFriend	(t.2)	(jack, jack) : hasFriend	(a.2)
Supported $\sqsubseteq \geq 2$hasFriend.Considerate	(t.3)	(jack, joe) : hasFriend	(a.3)
Lonely $\sqcup$ Unemployed $\sqsubseteq$ Unlucky	(t.4)	jack, joe : Considerate	(a.4)
		jack$\neq$joe	(a.5)

Figure 2: Example knowledge base

Employed concept. Moreover, jack has two distinct (according to (a.5)) hasFriend-successors[2] that are both Considerate (a.3 and a.4). Therefore, it can be concluded that jack is a member of the Supported concept. Having more than one hasFriend-successors, it cannot be a member of the concept Lonely. Although jack is a member of the concept ¬Lonely, the fact that it is in the concept Unemployed implies that it is a member of the concept expression Lonely $\sqcup$ Unemployed which concludes that jack is Unlucky.

2.1.3 Reasoning services

There are several reasoning services that a reasoner may provide. The *concept satisfiability test* which is one of the most basic services examines the satisfiability of a concept. In order to verify the satisfiability of a concept expression C, reasoners assert an unknown individual to this concept as $x : C$. If this assertion does not lead to a contradiction and the algorithm can find a model, the reasoner concludes that C is satisfiable.

Another reasoning service regarding TBoxes is the subsumption test which inquires a concept inclusion $C \sqsubseteq D$. Note that this subsumption is equivalent to the unsatisfiability of the concept expression $C \sqcap \neg D$. Moreover, unsatisfiability of the concept C is equivalent to the subsumption test $C \sqsubseteq \bot$. Therefore, these concept

[2]Notice that jack is a friend of himself.

satisfiability and subsumption tests can be reduced to each other. A basic ABox reasoning service is *instance checking* which questions if a is a member of C where a is a named individual in the ABox and the reasoner needs to consider the relevant ABox assertions as well as the TBox. *Instance retrieval* returns all the instances (named individuals) which are a member of a given concept and *realization* returns the most specific concept names that an individual is known to be a member of.

The TBox consistency test is referred to as the satisfiability test of all concept names, with respect to the inclusion axioms in a given TBox. Moreover, the *ABox consistency test* involves asserted knowledge about individuals of a given ABox.

2.1.4 Tableau Reasoning

One widely used approach to provide the reasoning services introduced in the previous section is *tableau* reasoning. Tableau reasoning is composed of a set of tableau rules which are fired by a tableau algorithm which constructs a tableau. A *tableau* is a data structure first introduced in [SSS91] for $\mathcal{ALC}$ which contains a finite description of a partial interpretation for an input KB. For the satisfiability test of a given concept C, the tableau algorithm tries to construct a model which contains an element x_0 for which $x_0 \in C^{\mathcal{I}}$ [BS00].

In the tableau algorithms, we assume for convenience that the concept expressions are in negation normal form, i.e., the negation sign only appears in front of concept names (atomic concepts). We compute the negation normal form of concept expressions according to the following equations.

$$\neg(C \sqcap D) \equiv \neg C \sqcup \neg D, \ \neg(C \sqcup D) \equiv \neg C \sqcap \neg D$$
$$\neg(\forall R.C) \equiv \exists R.(\neg C), \ \neg(\exists R.C) \equiv \forall R.(\neg C), \ \neg(\leq nR.C) \equiv \geq n+1R.C$$
$$\neg(\geq nR.C) \equiv \leq n-1R.C^{3}$$

[3]Note that $\leq 0R.C$ is always replaced with $\forall R.\neg C$

In the case of a concept satisfiability or TBox consistency test, tableau algorithms try to construct a representation of a model by constructing a *completion graph*. Later on, a model can be retrieved from a complete and clash-free graph. Nodes in the completion graph have a label which is a subset of possible concept expressions. For this reason we first define *clos* as the closure of a concept expression.

Definition 4 (Closure Function). The closure function $clos(E)$ is the smallest set of concepts such that: $E \in clos(E), (\neg D) \in clos(E) \Rightarrow D \in clos(E)$,
$(C \sqcup D) \in clos(E)$ or $(C \sqcap D) \in clos(E) \Rightarrow C \in clos(E)$ and $D \in clos(E)$,
$(\forall R.C) \in clos(E) \Rightarrow C \in clos(E)$, $(\exists R.C) \Rightarrow C \in clos(E)$,
$(\geq nR.C) \in clos(E)$ or $(\leq mR.C) \in clos(E) \Rightarrow C \in clos(E)$.

For a TBox $\mathcal{T}$ we define $clos(\mathcal{T})$ such that if $(C \sqsubseteq D) \in \mathcal{T}$ or $(C \equiv D) \in \mathcal{T}$ then $clos(C) \subseteq clos(\mathcal{T})$ and $clos(D) \subseteq \mathcal{T}$. Similarly for an ABox $\mathcal{A}$ we define *clos* function such that if $(a : C) \in \mathcal{A}$ then $clos(C) \subseteq clos(\mathcal{A})$.

Definition 5 (Completion Graph). G is a completion graph $G = (V, E, \mathcal{L})$ such that every node $x \in V$ is labeled by $\mathcal{L}(x) \subseteq clos(C)$ for a concept expression C and every edge $\langle x, y \rangle \in E$ is labeled by $\mathcal{L}(\langle x, y \rangle) \subseteq N_R$ where N_R is the set of role names.

A tableau algorithm to examine the satisfiability of a concept C starts with a root node x which is labeled $\mathcal{L}(x) = \{C\}$ and tries to expand the completion graph by means of the tableau rules. Tableau rules are built based on the semantics of the input DL language. For example, in order to impose the semantics of conjunction, a tableau algorithm may have a rule such as the **AND**-Rule in Figure 3.

Notice that rules are applied to one node at a time and may modify its label or add new nodes to the graph. The rules that create new nodes are called *generating rules*. Generating rules such as the **EXIST**-Rule in Figure 3 create new successors for the node on which they are fired. In fact, in case of languages such as $\mathcal{SHQ}$,

the completion graph will be tree-shaped. However, in the presence of expressive expressions such as nominals, we lose the tree-shaped model property.

Moreover, in presence of the inverse roles ($\mathcal{I}$), information can propagate back in the tree from lower levels to higher levels. For example, suppose y is an R-successor of x and for the label of x we have $\{\forall R^-.C\} \subseteq \mathcal{L}(y)$. In this case, the universal restriction in the label of y (which is in a deeper level in the tree than x) can modify the label of x and add C to $\mathcal{L}(x)$. In contrast, in the absence of inverse roles, once the label of a node cannot be modified by any expansion rule (tableau rule), we can assure that it will never be changed later after extending the completion graph.

AND-Rule	**if** $(C \sqcap D) \in \mathcal{L}(x)$ and $\{C, D\} \not\subseteq \mathcal{L}(x)$ **then** set $\mathcal{L}(x) := \mathcal{L}(x) \cup \{C, D\}$
OR-Rule	**if** $(C \sqcup D) \in \mathcal{L}(x)$ and $\{C, D\} \cap \mathcal{L}(x) = \emptyset$ **then** set $\mathcal{L}(x) := \mathcal{L}(x) \cup \{C\}$ or $\mathcal{L}'(x) := \mathcal{L}(x) \cup \{D\}$
EXIST-Rule	**if** $(\exists R.C) \in \mathcal{L}(x)$ and x has no R-successors in C **then** create a new node y and set $\mathcal{L}(y) := \{C\}$ and $\mathcal{L}(\langle x, y\rangle) = \{R\}$

Figure 3: Sample tableau rules

In case of an ABox consistency test, the given ABox may contain arbitrarily connected nodes which form not a necessarily tree-shaped graph. Moreover, they are not necessarily connected to build a single connected graph. Similar to [HST00] we define a completion forest in the following which is used for ABox tableau algorithms.

Definition 6 (Completion Forest). A completion forest $\mathcal{F} = (V, E, \mathcal{L})$, for an ABox $\mathcal{A}$ is composed of a set of arbitrarily connected nodes as the roots of the trees. Every node $x \in V$ is labeled by $\mathcal{L}(x) \subseteq clos(\mathcal{A})$ and each edge $\langle x, y\rangle \in E$ is labeled by the set $\mathcal{L}(\langle x, y\rangle) \subseteq N_R$ in which N_R is the set of roles occurring in $\mathcal{A}$ and $\mathcal{T}$. Finally, completion forests come with an explicit inequality relation $\dot{\neq}$ on nodes and an explicit

equality relation $\doteq$ which are implicitly assumed to be symmetric.[4]

Blocking

Reasoning in the presence of acyclic TBoxes is similar to concept satisfiability test. In other words, for acyclic TBoxes one can simply expand the definitions by adding D to the label of every individual $a : C$ for an inclusion axiom $C \sqsubseteq D$. However, in presence of general TBoxes which may contain cyclic inclusion axioms (where a concept name appears on both sides of the axiom), the algorithm needs to guarantee the termination by means of an extra mechanism called *blocking*.

For example, consider the case $x : A$ with respect to TBox $A \sqsubseteq \exists R.A$. The algorithm creates an R-successor y for x in A. Moreover, according to the inclusion axiom, the algorithm needs to create another R-successor for y and therefore the algorithm never terminates. Termination can be regained by trying to detect such cyclic computations, and then blocking the application of generating rules (such as the EXIST-Rule). Therefore, in such an example, the algorithm reuses y as an R-successor for itself. To avoid cyclic blocking (of x by y and y by x), we consider an order for the individual names, and define that an individual x may only be blocked by an individual y that occurs before x.

Nondeterminism

Due to the semantics of the language, some concept expressions have a nondeterministic nature. In other words, they can be interpreted in more than one way. For example, a disjunction $(C_1 \sqcup C_2 \ldots \sqcup C_n)$ is satisfied whenever C_1 or C_2 or $\ldots$ or C_n is satisfied. Therefore, it can be satisfied in at least n different ways. In order to reflect this non-determinism, a tableau rule opens different branches to proceed in

[4]According to the interpretation function $\mathcal{I}$, $a \dot{\neq} b$ if $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ and $a \doteq b$ if $a^{\mathcal{I}} = b^{\mathcal{I}}$.

the search space. For instance, the **OR-Rule** in Figure 3 is a nondeterministic rule which can have two consequences: $\mathcal{L}(x)$ or $\mathcal{L}'(x)$. In fact, it will result in the creation of two new completion graphs.

2.1.5 Correctness of an Algorithm

A tableau algorithm is *correct iff* it is sound, complete, and terminates for every finite KB as input. In order to prove correctness of tableau algorithms, a tableau needs to be formally defined (first used in [HST99]) as in Definition 7. An algorithm is sound *iff* it creates a tableau for every satisfiable (consistent) input and returns a clash for every inconsistent (unsatisfiable) input. Moreover, an algorithm is complete *iff* it can lead its rules such that it yields every given tableau.

Since the language of interest in this thesis is $\mathcal{SHQ}$, we define a tableau for a $\mathcal{SHQ}$ ABox algorithm based on [HST00]. Notice that a tableau is not necessarily a model and where one defines a new tableau, it must be proven that the defined tableau preserves the semantics of the input language (i.e., a model can be retrieved from every tableau). Such a proof is proposed for a $\mathcal{SHIQ}$ ABox tableau in [HST00].

Definition 7 ($\mathcal{SHQ}$ ABox Tableau). A tableau $T = (\mathbf{S}, \mathcal{L}, \mathcal{E}, \mathcal{J})$ is a tableau for a $\mathcal{SHQ}$ ABox $\mathcal{A}$ with respect to a role hierarchy $\mathcal{R}$[5] *iff*

- $\mathbf{S}$ is a non-empty set (elements in $\mathbf{S}$ represent individuals),
- $\mathcal{L} : \mathbf{S} \rightarrow 2^{clos(\mathcal{A})}$ maps each element in $\mathbf{S}$ to a set of concepts,
- $\mathcal{E} : N_R \rightarrow 2^{S \times S}$ maps each role to a set of pairs of elements in $\mathbf{S}$, and
- $\mathcal{J} : I_\mathcal{A} \rightarrow \mathbf{S}$ maps individuals occurring in $\mathcal{A}$ to elements in $\mathbf{S}$.

Furthermore, for all $s, t \in \mathbf{S}$, $C, C_1, C_2 \in clos(\mathcal{A})$, and $R, S \in N_R$, T satisfies:

[5]Let $\mathcal{R}$ be the set of role inclusion axioms such as $R \sqsubseteq S$ where $R, S \in N_R$.

P1 if $C \in \mathcal{L}(s)$, then $\neg C \notin \mathcal{L}(s)$,

P2 if $C_1 \sqcap C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ and $C_2 \in \mathcal{L}(s)$,

P3 if $C_1 \sqcup C_2 \in \mathcal{L}(s)$, then $C_1 \in \mathcal{L}(s)$ or $C_2 \in \mathcal{L}(s)$,

P4 if $\forall R.C \in \mathcal{L}(s)$ and $\langle s, t \rangle \in \mathcal{E}(R)$, then $C \in \mathcal{L}(t)$,

P5 if $\forall S.C \in \mathcal{L}(s)$ and $\langle s, t \rangle \in \mathcal{E}(R)$ for some $R \sqsubseteq S$, $Trans(R)$, then $\forall R.C \in L(t)$,

P6 if $\langle s, t \rangle \in \mathcal{E}(R)$ and $R \sqsubseteq S$, then $\langle s, t \rangle \in \mathcal{E}(S)$,

P7 if $\leq nR.C \in \mathcal{L}(s)$, then $|R^T(s, C)| \leq n$,

P8 if $\geq nR.C \in \mathcal{L}(s)$, then $|R^T(s, C)| \geq n$,

P9 if $\leq nR.C \in \mathcal{L}(s)$ and $|R^T(s, \top)| = m > n$,
then $|R^T(s, \neg C)| \geq m - n$

P10 if $a : C \in \mathcal{A}$ then $C \in \mathcal{L}(\mathcal{J}(a))$

P11 if $(a, b) : R \in \mathcal{A}$, then $\langle \mathcal{J}(a), \mathcal{J}(b) \rangle \in \mathcal{E}(R)$

P12 if $a \neq b \in \mathcal{A}$, then $\mathcal{J}(a) \neq \mathcal{J}(b)$

where $R^T(s, C) := \{t \in \mathbf{S} | \ \langle s, t \rangle \in \mathcal{E}(R) \wedge C \in \mathcal{L}(t)\}$.

Remark 1. Notice that in [HST99, HST00] if $(\leq nR.C) \in \mathcal{L}(s)$, $\langle s, t \rangle \in \mathcal{E}(R)$ then we have $\{C, \neg C\} \cap \mathcal{L}(t) \neq \emptyset$, which is not necessary but sufficient to satisfy and at-most restriction. A revised version such as condition **P9** guarantees that enough R-fillers are in $\neg C$ to preserve $(\leq nR.C)$.

2.2 Complexity of Reasoning

The complexity issue is of great importance when evaluating the feasibility of the use of description logic to represent knowledge. This term refers to two different concepts in the literature: (i) complexity of the language which also may be referred to as *theoretical complexity* and (ii) complexity of the algorithm which is also referred to as *practical complexity*.

2.2.1 Complexity of the Language: Theoretical Complexity

The complexity of the language, as its name suggests, is an inherent property of a language. The complexity of a decidable language is normally determined based on the size of its model and the time needed to construct the model. In fact, these are theoretical worst-case analysis. In Figure 4 derived from [Zol07] the complexity of some well-known languages is shown.

Language	Complexity
$\mathcal{ALC}$	PSpace-Complete
$\mathcal{ALCQ}$	PSpace-Complete
$\mathcal{ALC}$+ general TBox	ExpTime-Complete
$\mathcal{SHOIQ}$	NExpTime-Complete

Figure 4: Complexity of different DL languages

It is proven in [Tob01] that the satisfiability of $\mathcal{ALCQ}$ concepts is a PSpace-Complete problem, even if the numbers are represented using binary coding.

2.2.2 Complexity of the Algorithm: Practical Complexity

In a theoretical analysis, there is no need to introduce an effective procedure which solves an inference problem. In contrast, in a practical analysis, we determine the time and space complexity of a particular algorithm which is proposed to implement a

proposed inference problem. For example, as presented in Figure 4, the complexity of testing concept satisfiability for both $\mathcal{ALC}$ and $\mathcal{ALCQ}$ is PSpace-Complete. However, $\mathcal{ALCQ}$ is evidently a more expressive language. Therefore, the algorithm (possibly a tableau algorithm) which decides $\mathcal{ALCQ}$ must be more complicated and may consume more time and space.

In fact, the theoretical complexity of a language is independent from the algorithm which decides it and assumes that the algorithm *knows* which branch will survive in the case of non-determinism. Moreover, an algorithm that decides a language may have different behaviors for different inputs. Therefore, algorithms are mostly analyzed in average case or typical case scenario.

DL reasoning is known to be very complex and will not terminate in reasonable time if the algorithms are implemented exactly similar to tableau rules. Numerous optimization techniques have been proposed to overcome this complexity. A list of various recent optimizations that are used by the DL implementers are presented in [Bec06, Hor03, THPS07].

2.3 Atomic Decomposition

Atomic decomposition is a technique first proposed by Ohlbach and Koehler [OK99] for reasoning about sets. Later it was applied for concept languages such as in description logic for reasoning about role fillers. The idea behind the atomic decomposition is to consider all possible disjoint subsets of a role filler such that we have $|A \cup B| = |A| + |B|$ for two subsets (partitions) A and B. For example[6], assume we want to translate the following numerical restrictions presented as a DL concept into arithmetic inequations:

$$(\leq 3\ hasDaughter) \sqcap (\leq 4\ hasSon) \sqcap (\geq 5\ hasChild)$$

[6]This example is taken from [OK99].

Different partitions for this set of constraints are defined in the following:

$$
\begin{aligned}
c &= \text{children, not sons, not daughters.}\\
s &= \text{sons, not children, not daughters.}\\
d &= \text{daughters, not children, not sons.}\\
cs &= \text{children, which are sons, not daughters.}\\
cd &= \text{children, which are daughters, not sons.}\\
sd &= \text{sons, which are daughters, not children.}\\
csd &= \text{children, which are both sons and daughters.}
\end{aligned}
$$

Since it is an atomic decomposition and subsets are mutually disjoint, we can translate the three numerical restrictions into the following inequations:

$$
\begin{aligned}
|d| + |sd| + |cd| + |csd| &\leq 3\\
|s| + |sd| + |cs| + |csd| &\leq 4\\
|c| + |cd| + |cs| + |csd| &\geq 5
\end{aligned}
$$

Finding an integer solution for this system of inequations will result in a model for the initial set of cardinality restrictions. As there may exist more than one solution for this system, there may be a non-deterministic approach needed to handle it.

2.4 Integer Programming

A *Linear Programming* (LP) problem is the study of determining the maximum or minimum value of a linear function $f(x_1, x_2, \ldots, x_n)$ subject to a set of constraints. This set of constraints consists of linear inequations involving variables $x_1, x_2, \ldots, x_n$. We call f the objective (goal) function which must be either minimized or maximized. If all of the variables are required to have integer values, then the problem is called *Integer Programming* (IP) or Integer Linear Programming (ILP).

Definition 8 (Integer Programming). Integer Programming (IP) is the problem of

optimizing an objective (function) $f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \cdots + c_nx_n + d$ subject to a set of m linear constraints which can be formulated as:

$$\text{maximize (minimize) } C^T X$$
$$\text{subject to } AX \leq b$$
$$\text{and } x_i \text{ can only get integer values,}$$

where $X^T = [x_1\ x_2\ \ldots\ x_n]$, C is the matrix of coefficients in the goal function, $A_{m \times n}$ is the matrix of coefficients in the constraints, and $b^T = [b_1\ b_2\ \ldots\ b_m]$ contains the limit values in the inequations.

2.4.1 Simplex

It was proven by Leonid Khachiyan in 1979 that LP can be solved in polynomial time. However, the algorithm he introduced for this proof is impractical due to the high degree of the polynomial in its running time. The most widely used and shown to be practical algorithm is the *Simplex* method, proposed by George Dantzig in 1947.[7] The simplex method, constructs a polyhedron based on the constraints and objective function and then walks along the edges of the polyhedron to vertices with successively higher (or lower) values of the objective function until the optimum is reached [CLRS01]. Although LP is known to be solvable in polynomial time, the simplex method can behave exponentially for certain problems.

2.4.2 From Linear to Integer

Solving the linear programming problem may not yield an integer solution. Therefore, an additional method is required to guarantee the fact that the variables take integer values in the solution. There exists two general methods to achieve an integer solution:

[7] In fact, Leonid Kantorovich, a Russian mathematician used a similar technique in economics before Dantzig.

1. **Branch-and-bound:** Whenever a fractional value appears in the solution set, this method splits the search in two branches. For example, if $x_3 = 2.4$, the algorithm splits the current problem in two different problems such that in one of them the new constrain $x_3 \leq 2$ is added to A and in the other one $x_3 \geq 3$ is added to A. The optimized solution is therefore, the maximum (minimum) value of these two branches.

   Moreover, the algorithm prunes the fruitless branches. In other words, whenever a branch cannot obtain a value better than the optimum value yet found, the algorithm discards it.

2. **Branch-and-cut:** Whenever the optimum solution is not integer, the algorithm finds a linear constraint which does not violate the current set of constraints such that it eliminates the current non-integer solution from the feasible region (search space). This linear inequation which discards fractional region of the search space is called *cutting plane*.

   By adding cutting planes to A, the algorithm tries to yield an integer solution. However, the algorithm may reach a point where it cannot find a cutting plane. Therefore, in order to complete the search for an optimum integer solution it starts branch-and-bound.

In this research we decided to use branch-and-bound whenever simplex obtained a non-integer solution. Since the limits (matrix b) are integer values in our case, and the algorithm hardly ended up with a non-integer solution, we decided to avoid the high complexity of the branch-and-cut method.

2.5 Summary

In this chapter we explained required background and definitions to define our problem of interest in the next chapter. Starting from description logic formalisms we introduced qualified cardinality restrictions and the expressiveness they can give to the language. Furthermore, we introduced tableaux as well as a general framework in which one can prove correctness of an algorithm.

Moreover, we distinguished between two different types of complexity used in the literature: the inherent complexity of the input language vs. the complexity of the algorithm which decides it. In addition, we introduced atomic decomposition as a method invented by Ohlbach and Koehler to handle arithmetic aspects of concept languages. In Chapter 4 we demonstrate a hybrid algorithm which benefits from integer programming and atomic decomposition to address the high inefficiency of reasoning in description logics due to numerical restrictions. Before proposing the hybrid algorithm, in the next chapter we briefly survey the other present alternative solutions for this problem.

Chapter 3

Qualified Number Restrictions

In this chapter we will focus on qualified cardinality (number) restrictions as an expressive construct which equips the language with the ability of counting. Moreover, we describe the *standard* tableau approaches which handle qualified number restrictions and try to analyze their practical complexity. Afterwards, we present major optimization techniques proposed for the standard algorithms, regarding cardinality restrictions. Finally, we conclude this chapter by building the motivation for a more arithmetically informed approach.

By means of the cardinality restrictions, one can express numerical restrictions on the fillers of a role. For example, the expression $(\geq 1\mathsf{hasParent.Male}) \sqcap (\geq 1\mathsf{hasParent.Female})$ describes a restriction on an entity which has *at-least* one mother and *at-least* one father. There are two types of cardinality (numerical) restrictions: (1) unqualified restrictions ($\mathcal{N}$) which are expressed in the forms $(\geq nR)$ or $(\leq mR)$; (2) qualified number restrictions ($\mathcal{Q}$) shown by $(\geq nR.C)$ or $(\leq mR.C)$.[1] Since the unqualified restrictions can be expressed by qualified constructs ($\geq nR \equiv \geq nR.\top$ and $\leq mR \equiv \leq mR.\top$), we focus our attention on *qualified* cardinality restrictions.

[1] For a more formal definition see Section 2.1.2.

Adding cardinality restrictions to $\mathcal{ALC}$ can significantly increase the expressiveness of a language and consequently raise its practical complexity. Therefore, reasoning with qualified number restrictions can become a major source of inefficiency and requires special attention.

3.1 Standard Tableau Algorithms

Tableau algorithms normally try to examine the satisfiability of a concept or consistency of a TBox/ABox by constructing a model (*tableau*) for it. Similarly, in order to handle cardinality restrictions, they create the required number of role-fillers imposed by *at-least* restrictions $(\geq nR.C)$ and by modifying role-fillers, try to satisfy *at-most* restrictions $(\leq mR.C)$ whenever violated. Figure 5 presents a general version of the tableau rules, handling qualified number restrictions [HB91, BBH96]. The tableau calculi for more expressive languages also include similar rules to handle qualified cardinality restrictions [HST00, HS05].

≥-Rule	**if** $(\geq nR.C) \in \mathcal{L}(x)$ and there are no R-successors $y_1, y_2, \ldots, y_n$ for x such that $C \in \mathcal{L}(y_i)$ and $y_i \dot{\neq} y_j$ **then** create n new individuals $y_1, y_2, \ldots, y_n$ and set $\mathcal{L}(y_i) := \{C\}$, $\mathcal{L}(\langle x, y_i \rangle) := \{R\}$, and $y_i \dot{\neq} y_j$ for $1 \leq i \leq j \leq n$
***choose*-Rule**	**if** $(\leq mR.C) \in \mathcal{L}(x)$ and there exists an R-successor y of x such that $\{C, \neg C\} \cap \mathcal{L}(y) = \emptyset$, **then** set $\mathcal{L}(y) := \mathcal{L}(y) \cup \{C\}$ or $\mathcal{L}'(y) := \mathcal{L}(y) \cup \{\neg C\}$
≤-Rule	**if** 1.$(\leq nR.C) \in \mathcal{L}(x)$ and x has m R-successors such that $m > n$ and, 2. There exist R-successors y, z for x and not $y \dot{\neq} z$ **then** $Merge(y, z)$

Figure 5: The rules handling $\mathcal{Q}$

It can be observed in Figure 5 that the $\geq$-Rule tries to create n R-successors (R-fillers) in C for the individual x, in order to satisfy at-least restriction $(\geq nR.C)$.

Moreover, these individuals need to be distinguished to avoid that they are being merged later. Therefore, the $\geq$-Rule explicitly asserts this distinction by using the $\dot{\neq}$ operator.

Whenever we have an at-most restriction $(\leq nR.C)$ in the label of an individual, it is necessary to know how many R-successors of x are in C and how many are not in C (i.e., are in $\neg C$). Since we have the *open world assumption* in description logics, in order to remain sound and complete the algorithm nondeterministically decides, for each R-successor of x, whether it is in C or $\neg C$. This semantic branching can be achieved by means of a non-deterministic rule such as the **choose**-Rule in Figure 5 (R? in [HS05]). If there exists an at-most restriction $(\leq nR.C)$ in the label of a node x, this rule non-deterministically adds C or $\neg C$ to the label of every R-successor of x. Therefore, if there are m R-successors for x, the algorithm opens 2^m branches in the search space according to the at-most restriction $(\leq nR.C)$. In [Hor02] by explaining some sample ontologies derived from UML diagrams, it is demonstrated that this non-deterministic rule can be a major source of inefficiency in most DL-reasoners.

The $\leq$-Rule (a.k.a. *merge*-Rule) maintains the semantics of the at-most restrictions whenever violated, by merging extra successors. Whenever $(\leq nR.C)$ is in the label of x and x has m R-successors in C such that $m > n$, the algorithm needs to merge them into n nodes. If they are not mergable due to the assertions of the form $x\dot{\neq}y$, the algorithm returns a clash. Otherwise, it will nondeterministically try to merge m nodes into n nodes. Since there exists $m - n$ extra successors, there are $\binom{m}{2}\binom{m-1}{2}\ldots\binom{n+1}{2}/(m-n)!$ ways to merge them. This number can grow very fast when $m - n$ increases. Hence, the $\leq$-Rule can also be considered as a significant source of nondeterminism and consequently, inefficiency.

In these standard approaches, there exist two types of clashes:

1. A node x contains a clash if there exists a concept expression C such that

$\{C, \neg C\} \subseteq \mathcal{L}(x)$ or

2. If for a node x we have $(\leq nR.C) \in \mathcal{L}(x)$ and it has m R-successors $y_1, y_2, \ldots, y_m$ such that they are mutually distinct and $C \in \mathcal{L}(y_i)$, then x contains a clash.

The second type of clash occurs whenever the *merge*-Rule fails to merge extra nodes due to the fact that they are explicitly asserted to be distinct.

3.2 Complexity of Reasoning with $\mathcal{Q}$

From a theoretical point of view $\mathcal{ALCQ}$ concept satisfiability test is known to be PSpace-complete even with binary coding of the numbers [Tob01]. In fact this is also the upper-bound complexity for the languages $\mathcal{ALC}$ and $\mathcal{ALCQI}$, which are less and more expressive than $\mathcal{ALCQ}$. Moreover, the hardness of TBox consistency for $\mathcal{ALCQ}$ is EXPTime-complete [Tob01]. However, from a practical point of view, the complexity of reasoning for the expressive languages benefiting from qualified number restrictions is considerably high (see Section 3.1). In order to propose an algorithm that can work within a reasonable amount of time, one has to employ effective optimization techniques. In the following section some optimization techniques, regarding numerical restrictions are presented.

3.3 Optimizing Reasoning with $\mathcal{Q}$

There are many optimization techniques, employed in the reasoners handling qualified number restrictions. In the following, some optimization methods which are more specifically related to numerical restrictions will be described. The *signature* calculus tries to overcome the inefficiency caused by large numbers occurring in qualified number restrictions [HM01]. The algebraic methods try to optimize the algorithm when

creating the successors by choosing a branch which is arithmetically more informed [OK99, HTM01]. The *dependency-directed backtracking* technique, on the other hand, tries to discover the origins of a clash in order to avoid reproducing the same clash again [Hor02].

3.3.1 Signature Calculus

The complexity of the standard algorithms is evidently a function of the numbers occurring in qualified number restrictions; i.e., m and n in $(\leq mR.C)$ and $(\geq nR.C)$ (see Section 3.1). By increasing n in $(\geq nR.C)$, in fact we increase the number of R-successors of a node and then exponentially increase possible outcomes of the *choose*-Rule. On the other hand, the number of possible ways to merge n nodes into m nodes grows tremendously when increasing m and n. One way to handle large numbers of successors is by creating one "*proxy individual*" to represent more than one R-successor when all the successors share the same label.

The *signature* calculus presented in [HM01], in the same manner, creates proxy individuals as role fillers. More precisely, for every at-least restriction $(\geq nR.C)$ it creates one proxy individual in C which is a representative of n individuals. However, it will later *split* the proxy individual into more than one proxy individual, in order to satisfy the constraints imposed by the restrictions on the role-fillers. For example, if $(\leq mR.C)$ is in the label of a proxy individual x, where $m < n$ it nondeterministically tries to split x into more than one proxy individuals. In addition, it also requires a merge rule which nondeterministically merges extra proxy individuals that violate an at-most restriction.

3.3.2 Algebraic Methods

As stated in [OK99], "As soon as arithmetics comes into play, tableau approaches become very difficult to use". Even *simple*[2] arithmetic restrictions such as $(\geq nR.C)$ and $(\leq m\ R.C)$, can become a major source of inefficiency (see Section 3.1). An algebraic algorithm has been proposed in [OK99] to decide satisfiability of DL concept expressions. In fact, this non-tableau method approaches the problem from an arithmetic perspective and the whole reasoning is reduced to inequation solving. This method assumes that the *arithmetic system* is able to deal with disjunctions of inequations and translates disjunctions of logical expressions to "dis-equations". Moreover, it is not a tableau approach and does not try to build a model for the input concept expression. Consequently it only determines whether a concept expression is satisfiable or not and does not need the solution itself.

This method partitions the set of role-fillers into disjoint atoms by means of the *atomic decomposition* method. Furthermore, it translates all of the logical expressions into inequations. Thus, it returns "satisfiable" if there exists a solution for the corresponding set of inequations. Benefiting from an inequation solver, this method solves the problem in an arithmetically informed manner. Moreover, in [OK99] some new arithmetic constructs were introduced that can be handled easily by means of an inequation solver. However, this method which is not a calculus cannot be used to construct a model which is one of the major advantages of tableau-based algorithms.

3.3.3 A Recursive Algorithm for $\mathcal{SHQ}$

Combining algebraic methods introduced in [OK99] with tableau-based approaches, [HTM01] proposes a hybrid algorithm to decide consistency of general $\mathcal{SHQ}$ TBoxes.

[2] In [OK99] some more complicated structures are introduced that can be very useful in practical reasoning which are not even supported by the most expressive DL-reasoners (see Section 8.2.2).

Whenever potential R-successors of an individual (maximum number appearing in the at-least restrictions) exceeds the number of allowed R-successors with respect to the at-most restrictions, the *merge*-Rule is invoked by the algorithm. This rule collects all the numerical restrictions in the label of this node and by means of the atomic decomposition method, partitions the set if role-fillers and calls the in-equation solver. If there exists no non-negative integer solution for the set of inequations derived from the numerical restrictions, the algorithm returns a clash.

This algorithm assumes that each partition either must be empty or can be non-empty. In order to test whether a partition must be empty or not, it tests if the concept expression corresponding to that partition is satisfiable or not. In other words, each partition must be empty *iff* its corresponding concept expression is unsatisfiable. Hence, this algorithm will recursively call itself to examine the satisfiability of these concept expressions. Although it benefits from tableau rules, this algorithm cannot be considered as a calculus; i.e., its termination, soundness, and completeness were not proven.

3.3.4 Dependency-Directed Backtracking

The two previous techniques try to optimize the algorithm when creating successors. In fact, by partitioning the set of the role-fillers, they create successors in an informed manner, to avoid merging them later. Another way to optimize reasoning, in general, is dependency-directed backtracking (a.k.a. backjumping) [Hor02]. By means of the backjumping, an algorithm can detect the source of a clash and prune the search space to avoid facing the same clash again.

Notice that the only source of branching in the search space is due to the nondeterministic rules. Nondeterminism is exactly the reason that makes tableau algorithms inefficient. Therefore, backjumping can significantly improve the performance of the

highly nondeterministic calculi. The rules handling qualified number restrictions are a considerable source of non-determinism; i.e., the $\leq$-Rule and the *choose*-Rule. Therefore, dependency-directed backtracking can optimize these algorithms even in absence of large numbers [Hor02]. The technique described in [Hor02] in fact records the sources of a clash and *jumps over* choice points that are not related to the clash and tries to choose another branch at a nondeterministic point that is related to this clash.

3.4 Summary

In this chapter we briefly described the current solutions dealing with qualified number restrictions. By analyzing the practical complexity of the standard tableau calculi which follow the general pattern presented in Figure 5, we illustrated the high inefficiency of these calculi. Not observing the reasoning from an arithmetic perspective, these calculi act arithmetically blind and therefore inefficient.

Moreover, as long as DL-reasoners perform very weak when dealing with numbers, there is no room for proposing new arithmetic constructs. Therefore, an scalable calculus which deals with numbers more arithmetically informed rather than the current trial-and-error approaches may bring up this possibility.

In the next chapter we propose a hybrid calculus which in contrast with standard calculi [HST00, HS05, HST99] deals with qualified number restrictions in an arithmetically informed manner. Moreover, being a hybrid calculus (arithmetic and logical), in contrast with [OK99] it preserves the logical reasoning aspects and returns a tableau as a model for the input KB. On the other hand, it is proven to be a terminating, sound, and complete algorithm for $\mathcal{SHQ}$.

Chapter 4

A Hybrid Tableau Calculus for $\mathcal{SHQ}$

Extending $\mathcal{ALC}$ with qualified number restrictions provides the ability to express arithmetic restrictions on the role-fillers. This expressiveness can increase practical complexity of the reasoning when employing arithmetically uninformed algorithms (see Section 3.1). Therefore, a tableau calculus which benefits from arithmetic methods can improve the practical complexity of reasoning for qualified cardinality restrictions. In this chapter we propose a hybrid tableau algorithm which benefits from integer linear programming integer to properly handle numerical features of the language. At the beginning, the input ontology needs to be preprocessed by the algorithm so that it will be prepared as an input for the tableau rules. After the application of the tableau rules the algorithm either returns a clash or a complete and clash-free graph/forest. In Section 4.5 we extend the algorithm to work with arbitrary ABoxes as input.

In the following, three disjoint sets are defined; N_C is the set of concept names; $N_R = N_{RT} \cup N_{RS}$ is the set of all role names which consists of transitive (N_{RT}) and non-transitive (N_{RS}) roles; I is the set of all individuals, while I_A is the set of

named individuals occurring in the ABox $\mathcal{A}$. A role is called *simple* if it is neither transitive nor has any transitive sub-role. In order to remain decidable, qualified number restrictions are only allowed for simple roles. However, recent investigations in [YK07] show that this condition could be relaxed in the absence of inverse roles.

4.1 Preprocessing

Before applying the rules, the algorithm modifies the input ontology similar to the rewriting technique in [OK99]. Afterwards, the input is ready for the algorithm to compute the partitions and variables based on the atomic decomposition method.

4.1.1 Converting $\mathcal{Q}$ to $\mathcal{N}$

Let $\dot{\neg}C$ denote the standard *negation normal form* (NNF) of $\neg C$ such that $\dot{\neg}(\geq nR.C) \equiv \leq (n-1)R.C$, $\dot{\neg}(\leq nR.C) \equiv (\geq (n+1))R.C$, $\dot{\neg}(\forall R.C) \equiv (\geq 1 R.\dot{\neg}C)$. We define a recursive function *unQ* which rewrites a $\mathcal{SHQ}$ concept description such that qualified number restrictions are transformed into unqualified ones. It is important to note that this rewriting process always introduces a unique new role for each transformed qualified number restriction.

Definition 9 (*unQ*). This function transforms the input concept description into its NNF and replaces qualified number restrictions with unqualified ones.[1] In the following each R' is a new role in N_R with $\mathcal{R} := \mathcal{R} \cup \{R' \sqsubseteq R\}$:

$unQ(C) := C$ if $C \in N_C$

$unQ(\neg C) := \neg C$ if $C \in N_C$, $unQ(\dot{\neg}C)$ otherwise

$unQ(\forall R.C) := \forall R.unQ(C)$

$unQ(C \sqcap D) := unQ(C) \sqcap unQ(D)$

[1] This replacement method was first introduced in [OK99].

$$unQ(C \sqcup D) := unQ(C) \sqcup unQ(D)$$

$$unQ(\geq nR.C) := (\geq nR') \sqcap \forall R'.unQ(C) \quad (1)$$

$$unQ(\leq nR.C) := (\leq nR') \sqcap \forall (R \backslash R').unQ(\neg C) \quad (2)$$

Remark 2. According to [OK99] one can replace a qualified number restriction of the form $(\geq nR.C)$ by $\exists R' : (R' \sqsubseteq R) \in \mathcal{R} \wedge (\geq nR') \sqcap \forall R'.C$ and $(\leq nR.C)$ by $\exists R'$ such that $(R' \sqsubseteq R) \in \mathcal{R}, (\leq nR') \sqcap \forall R'.C \sqcap \forall R \backslash R'.(\neg C)$((1) and (2)). Therefore, the negation of $(\geq nR.C)$ which is equivalent to $(\leq (n-1)R.C)$ must be equal to the negation of the rewritten form. Accordingly, the negation of $(\geq nR.C)$ which is equivalent to $\leq (n-1)R.C$ will be $\forall R' \sqsubseteq R; (\leq (n-1)R') \sqcup \exists R'.(\neg C)$ which is unfortunately not a formula expressible in $\mathcal{SHQ}$. Similarly, $\neg(\leq mR.C) \equiv \geq (m+1)R.C$ will be equivalent to $\forall R' \sqsubseteq R; \geq (m+1)R' \sqcup \exists (R \backslash R').(\neg C)$ which is also inexpressible in $\mathcal{SHQ}$.

Hence, in order to avoid negating converted forms of qualified number restrictions, unQ must be applied initially to the negation normal form of the input TBox/ABox. Since (2) introduces a negation itself; this negated description needs to be converted to NNF before further applications of unQ. In other words, our language is not closed under negation w.r.t. the concept descriptions created by rule (1) or (2). However, our calculus ensures that these concept descriptions will never be negated at any time.

Since (2) is slightly different from what is proposed in [OK99], we prove this equivalence based on the semantics of the interpretation function $\mathcal{I}$.

Proposition 1. *$(\leq nR.C)$ is equisatisfiable with $(\forall (R \backslash R').\neg C \sqcap \leq nR')$ where $\exists R'$: $R' \sqsubseteq R$.*

Proof. The hypothesis can be translated to:

$(\leq nR.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \#\{y \mid \langle a, y \rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\} \Leftrightarrow$

$\exists \mathcal{I}' : \{a \in \Delta^{\mathcal{I}'} \mid \exists R' : R'^{\mathcal{I}'} \subseteq R^{\mathcal{I}'} \wedge \#\{y^{\mathcal{I}'} \mid \langle a, y \rangle \in R'^{\mathcal{I}'}\} \leq n \wedge (\forall b : \langle a, b \rangle \in R^{\mathcal{I}'} \wedge \langle a, b \rangle \notin R'^{\mathcal{I}'} \Rightarrow b \in \Delta^{\mathcal{I}'} \backslash C^{\mathcal{I}'})\}$.

($\Leftarrow$): If $a \in \Delta^{\mathcal{I}'}$, $\langle a, y\rangle \in R^{\mathcal{I}'}$, $y \in C^{\mathcal{I}'}$, and $\langle a, b\rangle \notin R'^{\mathcal{I}}$ we can conclude that $\langle a, y\rangle \in R'^{\mathcal{I}'}$ (because if $\langle a, y\rangle \notin R'^{\mathcal{I}'}$ then we had $y \in \Delta^{\mathcal{I}'}\backslash C^{\mathcal{I}'}$). Since we have $\#\{y \mid \langle a, y\rangle \in R'^{\mathcal{I}}\} \leq n$ we can define $\mathcal{I}$ such that it satisfies $(\leq nR.C)^{\mathcal{I}}$.
($\Rightarrow$): We can simply define $R^{\mathcal{I}'} = R'^{\mathcal{I}'} \cup R''^{\mathcal{I}'}$ such that for all $\langle a, b\rangle \in R^{\mathcal{I}}$ if $b \in C^{\mathcal{I}}$ then $\langle a, b\rangle \in R'^{\mathcal{I}'}$ and if $b \in \Delta^{\mathcal{I}}\backslash C^{\mathcal{I}}$ then $\langle a, b\rangle \in R''^{\mathcal{I}'}$. $\square$

4.1.2 TBox Propagation

Similar to [HST00], in order to propagate TBox axioms through all the individuals we define $C_{\mathcal{T}} := \prod_{C_i \sqsubseteq D_i \in \mathcal{T}} unQ(\neg C_i \sqcup D_i)$ and U as a new transitive role in the role hierarchy $\mathcal{R}$. A TBox $\mathcal{T}$ is consistent w.r.t. $\mathcal{R}$ *iff* the concept $C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}}$ is satisfiable w.r.t. $\mathcal{R}_U := \mathcal{R} \cup \{R \sqsubseteq U \mid R \in N_R\}$. By this means, we impose axioms in the TBox on all of the named and anonymous individuals.

4.2 Atomic Decomposition

For an individual a, based on the power set of $R_a = \{R \in N_R \mid (\leq nR) \in \mathcal{L}(a) \vee (\geq mR) \in \mathcal{L}(a)\}$, we partition the domain of role fillers for a. For every subset of R_a (except the empty set) we assign a partition p, representing fillers of the roles in that subset ($p^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$). Let P_a be the set of all the partitions for a, we define the function $\delta : P_a \rightarrow \mathcal{P}(R_a)$ to retrieve this subset: $\delta(p) := \{R \mid R \in R_a, p \text{ consists of R-fillers}\}$. Furthermore, we assign a variable v for each partition $p \in P_a$ ($v \hookrightarrow p$) and by means of the function α we denote all the roles related to a certain variable. In other words, $\alpha(v) = \delta(p)$ if $v \hookrightarrow p$.

Since we assume we have complete knowledge about the role hierarchy of the roles related to an individual, the absence of a role in $\delta(p)$ implicitly means the presence of its complement. Therefore partitions are mutually disjoint. However, the complement

of roles is not a part of $\mathcal{SHQ}$ and comes into play only when the $\forall(R\backslash R').C$ construct is used.

Since all the concept restrictions for role fillers in $\mathcal{SHQ}$ are propagated through universal restrictions on roles, we can conclude that all the individuals in a certain partition share the same restrictions and can be dealt with as a unit. We call this unit *proxy individual* which is a representative of possibly more than one individual.

Definition 10 (Individual Cardinality). We define $card : I \rightarrow \mathbb{N}$ to indicate cardinality of proxy individuals.

4.2.1 Example

Assume for the node x we have $\{\geq 3R', \leq 2S', \geq 1R'', \leq 1R''\} \subseteq \mathcal{L}(x)$ and these are the only cardinality restrictions in the label of x. Therefore, we will have $R_a(x) = \{R', S', R''\}$. Similar to what is described, this algorithm collects all the unqualified number restrictions[2] in the label of a node and computes all the partitions based on the power set of the set of roles. Therefore, for the set of roles $R_a(x)$, we have 8 different subsets. Since we never consider the empty set[3], we will have 7 different partitions such as in Figure 6.

Afterwards, the algorithm assigns a variable to each partition. Assuming a binary coding of the indices of variables, where the first digit from the right represents R', the second digit represents S', and the last digit represents the presence of R'', we will define the variables such that $v_i \hookrightarrow p_i$:

$$\alpha(v_{001}) = \{R'\}, \alpha(v_{010}) = \{S'\}, \alpha(v_{100}) = \{R''\}$$
$$\alpha(v_{011}) = \{R', S'\}, \alpha(v_{101}) = \{R', R''\}$$
$$\alpha(v_{110}) = \{S', R''\}, \alpha(v_{111}) = \{R', S', R''\}$$

[2]Notice that after the application of the unQ function, no qualified number restriction exists in the label of the nodes.

[3]If the label of an edge is the empty set in the completion graph, in fact it does not exist.

$$\delta(p_1) = \{R'\}, \delta(p_2) = \{S'\}, \delta(p_4) = \{R''\}$$
$$\delta(p_3) = \{R', S'\}, \delta(p_5) = \{R', R''\}$$
$$\delta(p_6) = \{S', R''\}, \delta(p_7) = \{R', S', R''\}$$

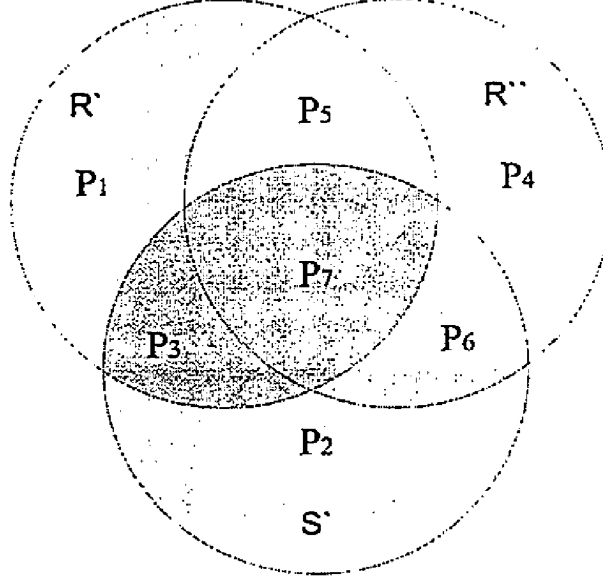

Figure 6: Atomic Decomposition Example

Hence, the unqualified number restrictions in $\mathcal{L}(x)$ can be translated to the following set of inequations where $=$ is the placeholder for $\leq$ and $\geq$:

$$\begin{aligned} v_{001} + v_{011} + v_{101} + v_{111} &\geq 3 \\ v_{010} + v_{011} + v_{110} + v_{111} &\leq 2 \\ v_{100} + v_{101} + v_{110} + v_{111} &= 1 \end{aligned}$$

4.3 Architecture

From an abstract point of view we can divide the reasoning module of this algorithm into two parts: The *logical module* and the *arithmetic module.* The arithmetic module is responsible to find an integer solution for a set of inequations or return a clash if no solution exists. The input of the arithmetic module is a set of inequations, provided by means of the function ξ which translates unqualified number restrictions

to inequations such that $\xi(R, \bowtie, n) := (\sum_{R \in \alpha(v_i)} v_i) \bowtie n$ where $\bowtie \in \{\leq, \geq\}$ and $n \geq 0$. For instance, in the example of Section 4.2.1 we had $\xi(R', \geq, 3) = v_{001} + v_{011} + v_{101} + v_{111} \geq 3$.

Furthermore, the arithmetic module returns solutions using the function $\sigma : \mathcal{V} \rightarrow \mathbb{N}$ which assigns a non-negative integer to each variable. Let V_x be the set of variables assigned to an individual x, we define a set of solutions Ω for x as $\Omega(x) := \{\sigma(v) = n \mid n \in \mathbb{N}, v \in V_x\}$. Notice that the goal function of the inequation-solver is to minimize the sum of the variables occurring in the input inequations.

4.4 Tableau Rules for TBox Consistency

In the following we present a tableau algorithm that accepts a general TBox $\mathcal{T}$ w.r.t. a role hierarchy $\mathcal{R}$ as input and either returns "inconsistent" if $\mathcal{T}$ is not consistent or otherwise "consistent" with a complete and clash-free completion graph. In order to examine the consistency of $\mathcal{T}$, the algorithm creates an assertion $x_0 : C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}}$ where $x_0 \in I$ is a new individual. Then by applying the expansion rules, it tries to construct a completion graph. Since the hybrid algorithm creates a graph which is slightly different from the standard completion graph (see Section 2.1.4), we will redefine it in the following.

4.4.1 Completion Graph

A completion graph $G = (V, E, \mathcal{L}, \mathcal{L}_E)$ for a general $\mathcal{SHQ}$ TBox $\mathcal{T}$ is a tree with x_0 as its root node. Every node $x \in V$ has a logical label $\mathcal{L}(x) \subseteq clos(\mathcal{T})$ and an arithmetic label $\mathcal{L}_E(x)$ as a set of inequations of the form $\xi(R, \bowtie, n)$ with $\bowtie \in \{\leq, \geq\}$; each edge $\langle x, y \rangle \in E$ is labeled by the set $\mathcal{L}(\langle x, y \rangle) \subseteq N_R$.

Blocking A node x in a graph G is blocked by a node y *iff* y is an ancestor of x such

that $\mathcal{L}(x) \subseteq \mathcal{L}(y)$.[4] Since we have the tree-shaped model property for $\mathcal{SHQ}$ TBoxes, we can conclude that all of the successors of a blocked node are also blocked.

Definition 11 (Clash triggers). A node x contains a clash *iff* there exists a concept name $A \in N_C$ such that $\{A, \neg A\} \subseteq \mathcal{L}(x)$ or $\mathcal{L}_E(x)$ has no non-negative integer solution. A completion graph is clash-free *iff* none of its nodes contains a clash, and is complete if no expansion rule is applicable to any of its nodes.

4.4.2 Expansion Rules

The algorithm starts with the graph G with $x_0 : C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}}$ as its root node. Moreover, for the root node x_0 we set $card(x_0) = 1$. After the application of the rules in the Figure 7, the algorithm returns 'consistent' if it yields a complete and clash-free graph or otherwise 'inconsistent'.

The algorithm considers the following priorities for the expansion rules:

1. All the rules except the $\leq$-Rule, the $\geq$-Rule, and the fil-Rule have the highest priority.

2. The $\leq$-Rule and the $\geq$-Rule have the second highest priority.

3. The generating rule which is the fil-Rule has the lowest priority.

Moreover, there are three limitations on the expansion of the rules:

- priority of the rules,

- rules are only applicable to nodes that are not blocked,

- in order to preserve role hierarchies for every $R \sqsubseteq S \in \mathcal{R}$: if for a node $x \in V$ we have $R \in \alpha(v)$ but $S \notin \alpha(v)$, this variable needs to be always zero and therefore we set $\mathcal{L}_E(x) := \mathcal{L}_E(x) \cup \{v \leq 0\}$.

[4] Notice that whenever we deal with ABoxes, we must take into consideration that ABox individuals can never be blocked.

$\sqcap$-Rule	**if** $(C_1 \sqcap C_2) \in \mathcal{L}(x)$ and not both $C_1 \in \mathcal{L}(x)$ and $C_2 \in \mathcal{L}(x)$ **then** set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-Rule	**if** $(C_1 \sqcup C_2) \in \mathcal{L}(x)$ **and** $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ **then** set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1\}$ **or** $\mathcal{L}'(x) = \mathcal{L}(x) \cup \{C_2\}$
$\forall$-Rule	**if** $\forall R.C \in \mathcal{L}(x)$ and there exists a y and R' such that $R' \in \mathcal{L}(\langle x, y\rangle)$ and $C \notin \mathcal{L}(y)$ and $R' \sqsubseteq R$ **then** set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\forall_\backslash$-Rule	**if** $\forall R \backslash S.C \in \mathcal{L}(x)$ and there exists a y and R' for which $R' \in \mathcal{L}(\langle x, y\rangle)$, $R' \sqsubseteq R$, $R' \not\sqsubseteq S$, and $C \notin \mathcal{L}(y)$ **then** set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\forall_+$-Rule	**if** $\forall R.C \in \mathcal{L}(x)$ and there exists a y and S for which $S \in \mathcal{L}(\langle x, y\rangle)$, $S \sqsubseteq R$, $S \in N_{RT}$, and $\forall S.C \notin \mathcal{L}(y)$ **then** set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{\forall S.C\}$
$\geq$-Rule	**If** $(\geq nR) \in \mathcal{L}(x)$ and $\xi(R, \geq, n) \notin \mathcal{L}_E(x)$ **then** set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{\xi(R, \geq, n)\}$
$\leq$-Rule	**If** $(\leq nR) \in \mathcal{L}(x)$ and $\xi(R, \leq, n) \notin \mathcal{L}_E(x)$ **then** set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{\xi(R, \leq, n)\}$
ch-Rule	**If** there exists v occurring in $\mathcal{L}_E(x)$ with $\{v \geq 1, v \leq 0\} \cap \mathcal{L}_E(x) = \emptyset$ **then** set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{v \geq 1\}$ **or** set $\mathcal{L}'_E(x) = \mathcal{L}_E(x) \cup \{v \leq 0\}$
fil-Rule	**If** there exists v occurring in $\mathcal{L}_E(x)$ such that $\sigma(v) = n$ with $n > 0$ and if x is not blocked and has no such successor **then** create a new node y and set $\mathcal{L}(\langle x, y\rangle) = \alpha(v)$ and $card(y) = n$

Figure 7: Expansion rules for $\mathcal{SHQ}$ TBox

Similar to [FFHM08b, FFHM08a] which propose an algorithm for $\mathcal{ALCQ}$ concept satisfiability, the rules in Figure 7 handle the $\mathcal{SHQ}$ TBox consistency test. In addition to [FFHM08b, FFHM08a], this algorithm benefits from *proxy individuals*.

4.4.3 Rule Descriptions

The function of the $\sqcap$-Rule, $\sqcup$-Rule, $\forall$-Rule, and the $\forall_+$-Rule is similar to standard tableau algorithms (see Section 2.1.4). The $\forall_+$-Rule preserves the semantics of the transitive roles. The $\forall_\backslash$-Rule handles the new universal restriction expression introduced by the transformation function unQ. All these rules which have the highest

priority among the expansion rules, extend $\mathcal{L}(x)$ with new logical expressions. After the application of these rules the logical label of the node x cannot be expanded anymore. This fact is a consequence of the characteristics of $\mathcal{SHQ}$ TBox consistency which yields a tree-shaped graph. In other words, the labels of a node cannot be later affected by its successors in the graph.

$\leq$-Rule, $\geq$-Rule:

Since all of the logical constraints on a node are collected by the rules with the highest priority, after their application the algorithm, in fact, has collected all the numerical restrictions for a node. Therefore, it is possible to compute the final partitioning with respect to these restrictions. The $\leq$-Rule and the $\geq$-Rule translate the numerical restrictions, based on the atomic decomposition technique, into inequations. Consequently, they will add these inequations to $\mathcal{L}_E(x)$ for a node x.

***ch*-Rule:**

The intuition behind the *ch*-Rule is due to the partitioning consequences. When we partition the domain of all the role-fillers for an individual, we actually consider all the possible cases for the role-fillers. If a partition p is logically unsatisfiable, the corresponding variable $v \hookrightarrow p$ should be zero. But if it is logically satisfiable, nothing but the current set of inequations can restrict the number of individuals being members of this partition. Hence, from a logical point of view there are two cases: an empty partition or a non-empty partition. On the other hand, the arithmetic reasoner is unaware of the satisfiability of a concept representing a partition. Therefore, in order to organize the search space with respect to this semantic branching, the algorithm distinguishes between these two cases: $v \geq 1$ or $v \leq 0$.

***fil*-Rule:**

The *fil*-Rule with the lowest priority is the only generating rule. It always creates successors for a node x based on the non-negative integer solution provided by the

arithmetic reasoner. Hence, it will never create successors for a node that might violate numerical restrictions of this node. Therefore, there is no need for a mechanism of merging nodes created by this rule. In order to avoid infinite loops, the *fil*-Rule does not create successors for a node which is blocked by another node.

4.4.4 Example TBox

Consider the TBox $\mathcal{T} = \{C \sqsubseteq (\geq 2R.D) \sqcap (\leq 1S.(C \sqcup D))\}$ and the role hierarchy $\mathcal{R} = \{R \sqsubseteq S\}$. To test the satisfiability of the concept C, we start the algorithm by $\mathcal{L}(x_0) = \{C\}$ and adding $\neg C \sqcup ((\geq 2R.D) \sqcap (\leq 1S.(C \sqcup D)))$ to the label of x_0. We abbreviate this concept expression by $C_\mathcal{T}$ which will later be propagated to other nodes of the tree by means of the universal transitive role U.

Preprocessing: The algorithm converts the qualified number restrictions by means of the *unQ* function such that:
$unQ(C_\mathcal{T}) = \neg C \sqcup ((\geq 2R' \sqcap \forall R'.D) \sqcap (\leq 1S' \sqcap \forall S \backslash S'.(\neg C \sqcap \neg D)))$ where R' and S' are new roles and the role hierarchy will be extended as $\mathcal{R} = \mathcal{R} \cup \{S' \sqsubseteq S, R' \sqsubseteq R\}$.

After applying all the rules with the highest priority, since C is asserted to the label of x_0, we will have:

$$\mathcal{L}(x_0) = \{C, \geq 2R', \forall R'.D, \leq 1S', \forall S \backslash S'.(\neg C \sqcap \neg D), C_\mathcal{T}, \forall U.C_\mathcal{T}\}$$

Consequently the $\geq$-Rule and $\leq$-Rule become applicable and the partitions will be computed by the algorithm. Since there are two numerical restrictions in $\mathcal{L}(x_0)$, there will be $2^2 - 1 = 3$ partitions and therefore three variables to construct the inequations.

If $\alpha(v_{01}) = \{R'\}$, $\alpha(v_{10}) = \{S'\}$, and $\alpha(v_{11}) = \{R', S'\}$ we will have:

$$\mathcal{L}_E(x_0) = \begin{cases} v_{01} + v_{11} & \geq & 2. \\ v_{10} + v_{11} & \leq & 1 \end{cases}$$

The goal function in the inequation solver is to minimize $v_{01} + v_{10} + v_{11}$. If the *ch*-Rule sets $v_{11} \geq 1$ but the other variable must be less or equal zero, an arithmetic clash will occur. Therefore, one possible solution can be when $v_{01} \geq 1$ and $v_{11} \geq 1$ which is $\sigma(v_{01}) = 1$ and $\sigma(v_{11}) = 1$. Now that we have a solution, the *fil*-Rule becomes applicable and generates two successors x_1 and x_2 for the node x_0 such that $card(x_1) = 1$, $\mathcal{L}(\langle x_0, x_1 \rangle) = \{R'\}$ and $card(x_2) = 1$, $\mathcal{L}(\langle x_0, x_2 \rangle) = \{R', S'\}$.

Since R' is in the label of the edges $\langle x_0, x_1 \rangle$ and $\langle x_0, x_2 \rangle$ the $\forall$-Rule will be invoked for $\forall R'.D \in \mathcal{L}(x_0)$ and D will be added to the logical label of x_1 and x_2. In addition, since $R' \sqsubseteq S$ but $R' \not\sqsubseteq S'$ the $\forall_\backslash$-Rule will be fired for the node x_1 and $\neg C \sqcap \neg D$ will be added to $\mathcal{L}(x_1)$.

In the current graph we have $\{D, \neg D\} \subseteq \mathcal{L}(x_1)$ which is a clash and the algorithm needs to explore another branch in the search space. Another solution can be where $v_{11} \leq 0$, $v_{10} \leq 0$, and $v_{01} \geq 1$ which is $\sigma(v_{01}) = 2$ for which the *fil*-Rule will create x_3 such that $card(x_3) = 2$, $\mathcal{L}(\langle x_0, x_3 \rangle) = \{R'\}$. Similar to x_1, this node will contain a clash. Since no other branch is left to explore, the algorithm returns only clashes which means the concept C is unsatisfiable.

4.5 ABox Consistency

Handling assertional knowledge can be more difficult than TBox consistency for the hybrid algorithm due to the following reasons:

Extra asserted successors: The hybrid algorithm always creates the role-fillers according to the solution provided by the inequation solver. Therefore, it will never create extra successors that may violate an at-most restriction and need to be merged. In the assertional knowledge, in contrast, an individual can have arbitrarily many successors which may be possibly more than what is allowed. Therefore, a hybrid algorithm, to handle ABoxes needs a mechanism to detect if extra successors exist and also a mechanism to merge them.

Back propagation: One of the characteristics of the $\mathcal{SHQ}$ TBox consistency that makes it efficient for the hybrid algorithm is its tree-shaped model. In other words, when no more number restrictions can be added by the successors of a node, the algorithm can compute the partitions just once and locally for a node. Moreover, the set of inequations for a node will be unaffected by the successors of a node. This feature can make the algorithm practically efficient to use. However, in the case of ABox consistency, the given ABox can contain a loop for which it is not possible to define an ordering. Hence, an algorithm for general ABoxes in $\mathcal{SHQ}$ must be able to handle incremental knowledge for the asserted individuals.

4.5.1 A Hybrid Algorithm for ABox Consistency

In the following we extend our hybrid algorithm to ABox consistency for $\mathcal{SHQ}$ which addresses the requirements of such an algorithm.

Definition 12. *If $x, y \in I_{\mathcal{A}}$ then R_{xy} is the role name which is only used to represent that y is an R-filler of x and $R_{xy} \sqsubseteq R$. In other words whenever $(x, z) : R_{xy}$ we have $y^{\mathcal{I}} = z^{\mathcal{I}}$.*

Re-writing ABox assertions: We replace the assertion $(b, c) : R$ by $b : (\geq 1 R_{bc}) \sqcap (\leq$

$1R_{bc}$). The reason that we translate ABox role assertions into number restrictions is due to the fact that they actually impose a numerical restriction on a node. For example, the assertion $(b, c) : R$ means there is one and only one R_{bc}-filler for b which is c. Since the hybrid algorithm needs to consider *all* the numerical restrictions before creating an arithmetic solution and generating the successors for a node, it is necessary to consider this restriction as well.

Definition 13 (Completion Forest). Similar to a completion graph we define a completion forest $\mathcal{F} = (V, E, \mathcal{L}, \mathcal{L}_E)$ for a $\mathcal{SHQ}$ ABox $\mathcal{A}$ that is composed of a set of arbitrarily connected nodes as the roots of the trees. Every node $x \in V$ is labeled by $\mathcal{L}(x) \subseteq clos(\mathcal{A})$ and $\mathcal{L}_E(x)$ as a set of inequations of the form $\xi(R, \bowtie, n)$ with $\bowtie \in \{\leq, \geq\}$; each edge $\langle x, y\rangle \in E$ is labeled by the set $\mathcal{L}(\langle x, y\rangle) \subseteq N_R$. We maintain the distinction between nodes of the forest by the relation $\dot{\neq}$.

The algorithm starts with the forest $\mathcal{F}_{\mathcal{A}}$, composed of the named individuals as root nodes. For each $a_i \in I_{\mathcal{A}}$ a root node x_i will be created and $\mathcal{L}(x_i) = \{C \mid (a_i : C) \in \mathcal{A}\}$. Notice that we *do not* set the label of the edges as $\mathcal{L}(\langle x_i, x_j\rangle) := \{R \mid (a_i, b_i) : R \in \mathcal{A}\}$ since they are encoded as cardinality restrictions of the form $(\leq 1R_{a_i b_i}) \sqcap (\geq 1R_{a_i b_i})$ and added to $\mathcal{L}(x_i)$. Additionally, for every root node x we set $card(x) = 1$ and the root nodes cannot be blocked.

The following restrictions are imposed on variables due to the semantics of the logic $\mathcal{SHQ}$:

1. In order to preserve ABox assertions of the form $a \dot{\neq} b$ if $\{R, S\} \subseteq \alpha(v)$ and $R \sqsubseteq R_{xa}$ and $S \sqsubseteq R_{xb}$ we will set $\mathcal{L}_E(x) := \mathcal{L}_E(x) \cup \{v \leq 0\}$.

2. Also since fillers of R_{xy} and S_{xy} have to be equivalent, if we have $R_{xy} \in \alpha(v)$ but $S_{xy} \notin \alpha(v)$ we will set $\mathcal{L}_E(x) := \mathcal{L}_E(x) \cup \{v \leq 0\}$.

4.5.2 Tableau Rules for ABox Consistency

In the following we present an algorithm which extends the algorithm introduced in section 4.4 to handle ABox consistency. We extend the tableau rules presented in Figure 7 with the *merge*-Rule and the *reset*-Rule which have the highest priority and adapt the *fil*-Rule for the ABox consistency algorithm in Figure 8. Furthermore, an adaptation procedure is added to the arithmetic reasoner to maintain existing solutions according to the current partitioning.

The hybrid algorithm converts ABox assertions of the form $(a, b) : R$ into numerical restrictions. Therefore, this rewriting will reduce the problem of detecting extra successors to inequation solving. In other words, the role assertions will be converted to cardinality restrictions which will be later processed by the arithmetic reasoner. Moreover, a new rule needs to be added to the tableau rules which merges extra asserted successors according to the solution provided by the arithmetic reasoner.

Merging example

Consider three nodes x, y, and z as root nodes of a forest $\mathcal{F}$ and the following labels:

$$\mathcal{L}(x) = \{\leq 1 R\},\ R \in \mathcal{L}(\langle x, y\rangle),\ R \in \mathcal{L}(\langle x, z\rangle)$$

The algorithm converts role assertions to numerical restrictions and we will have $\mathcal{L}(x) = \{\leq 1R, \leq 1R_{xy}, \geq 1R_{xy}, \leq 1R_{xz}, \geq 1R_{xz}\}$ where $\{R_{xy} \sqsubseteq R, R_{xz} \sqsubseteq R\} \subseteq \mathcal{R}$. Consider the variables such that $\alpha(v_{001}) = \{R\}$, $\alpha(v_{010}) = \{R_{xy}\}$, $\alpha(v_{100}) = \{R_{xz}\} \ldots$, $\alpha(v_{111}) = \{R, R_{xy}, R_{xz}\}$. According to the role hierarchy $\mathcal{R}$ we will have $v_{010} \leq 0$, $v_{100} \leq 0$, and $v_{110} \leq 0$.

Hence, after removing all the variables that must be zero, the following system of

$\sqcap$-Rule	if $(C_1 \sqcap C_2) \in \mathcal{L}(x)$ and not both $C_1 \in \mathcal{L}(x)$ and $C_2 \in \mathcal{L}(x)$ then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-Rule	if $(C_1 \sqcup C_2) \in \mathcal{L}(x)$ and $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1\}$ or $\mathcal{L}'(x) = \mathcal{L}(x) \cup \{C_2\}$
$\forall$-Rule	if $\forall R.C \in \mathcal{L}(x)$ and there exists a y and R' such that $R' \in \mathcal{L}(\langle x, y\rangle)$ and $C \notin \mathcal{L}(y)$ and $R' \sqsubseteq R$ then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\forall_{\backslash}$-Rule	if $\forall R \backslash S.C \in \mathcal{L}(x)$ and there exists a y and R' for which $R' \in \mathcal{L}(\langle x, y\rangle)$, $R' \sqsubseteq R$, $R' \not\sqsubseteq S$, and $C \notin \mathcal{L}(y)$ then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\forall_{+}$-Rule	if $\forall R.C \in \mathcal{L}(x)$ and there exists a y and R', S for which $R' \in \mathcal{L}(\langle x, y\rangle)$, $R' \sqsubseteq S$, $S \sqsubseteq R$, $S \in N_{RT}$, and $\forall S.C \notin \mathcal{L}(y)$ then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{\forall S.C\}$
$\geq$-Rule	If $(\geq nR) \in \mathcal{L}(x)$ and $\xi(R, \geq, n) \notin \mathcal{L}_E(x)$ then set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{\xi(R, \geq, n)\}$
$\leq$-Rule	If $(\leq nR) \in \mathcal{L}(x)$ and $\xi(R, \leq, n) \notin \mathcal{L}_E(x)$ then set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{\xi(R, \leq, n)\}$
ch-Rule	If there exists v occurring in $\mathcal{L}_E(x)$ with $\{v \geq 1,\ v \leq 0\} \cap \mathcal{L}_E(x) = \emptyset$ then set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{v \geq 1\}$ or set $\mathcal{L}_E(x) = \mathcal{L}'_E(x) \cup \{v \leq 0\}$
reset-Rule	if $(\leq nR) \in \mathcal{L}(x)$ or $(\geq nR) \in \mathcal{L}(x)$, and for all $v \in V_x$ we have $R \notin \alpha(v)$ then set $\mathcal{L}_E(x) := \emptyset$ and for every successor y of x set $\mathcal{L}(\langle x, y\rangle) := \emptyset$.
merge-Rule	if there exists $x, a, b \in I_{\mathcal{A}}$ such that $R_{xa} \in \mathcal{L}(\langle x, b\rangle)$ then merge the nodes a, b and replace every occurrence of a by b.
fil-Rule	If there exists v occurring in $\mathcal{L}_E(x)$ such that $\sigma(v) = n$ with $n > 0$: (i) if $n = 1$ and there exists a $R_{xb} \in \alpha(v)$ then if $\mathcal{L}(\langle x, b\rangle) = \emptyset$ set $\mathcal{L}(\langle x, b\rangle) := \alpha(v)$ (ii) elsif x is not blocked and has no such successor then create a new node y and set $\mathcal{L}(\langle x, y\rangle) := \alpha(v)$ and $card(y) = n$

Figure 8: Expansion rules for $\mathcal{SHQ}$ ABox consistency

inequations in $\mathcal{L}_E(x)$ needs to be solved:

$$(*)\begin{cases} v_{001} + v_{011} + v_{101} + v_{111} & \leq & 1, \\ v_{011} + v_{111} & = & 1 \\ v_{101} + v_{111} & = & 1 \end{cases}$$

The only non-negative solution for $(*)$ is achieved when it is decided by the *ch*-Rule that $(v_{111} \geq 1)$ and all other variables are equal zero. This solution which is $\sigma(v_{111}) = 1$ will invoke the *fil*-Rule in Figure 8 which makes y and z the successors of x such that $\mathcal{L}(\langle x, y\rangle) = \{R, R_{xy}, R_{xz}\}$ and $\mathcal{L}(\langle x, z\rangle) = \{R, R_{xy}, R_{xz}\}$. Consequently, since $R_{xy} \in \mathcal{L}(\langle x, z\rangle)$, the *merge*-Rule in Figure 8 becomes applicable for the nodes y, z and merges them.

When a new numerical restriction is added to $\mathcal{L}(x)$, the algorithm needs to refine the partitioning assigned to x and consequently V_x. However, the current state of the forest is a result of existing solutions based on the previous partitioning. In fact, the newly added numerical restriction has been added to $\mathcal{L}(x)$ after an application of the *fil*-Rule. Therefore, we can conclude that the newly added numerical restriction is a consequence of the solution for the previous set of inequations. Hence, if the algorithm does not maintain the existing solutions, in fact it may remove the cause of the current partitioning which would result in unsoundness.

There exist at least two approaches to handle this characteristics of arbitrary ABoxes, *back propagation*:

Global decomposition: One can treat an individual in an arbitrary ABox, as a nominal. Because of the global effect of nominals, the algorithm must consider a global partitioning for *all* the roles occurring in the ABox $\mathcal{A}$ [FHM08]. Hence, all possible partitions and therefore variables will be computed before starting the application of the rules. Consequently, whenever a numerical restriction is added

to the label of a node, there is no need to recompute the partitioning to construct the new inequation.. Although global partitioning enables the algorithm to deal with back propagation, according to the large number of partitions/variables, it can make the algorithm highly inefficient in practice. Moreover, since our algorithm introduces a new role for each ABox assertion $(a, b) : R$, the number of role names and consequently partitions will be enormously large for ABoxes with many assertions.

Incremental local partitioning: In contrast with nominals which increase the expressiveness of the language and have a global effect on the nodes, individuals in ABoxes have a local effect and can be handled locally. Moreover, an input ABox is assumed to contain a large number of individuals, whereas relatively smaller number of nominals. Therefore, it is more reasonable to deal with back propagations locally by means of incremental partitioning as proposed in the following.

In arbitrary ABoxes, similar to the effect of inverse roles, a root node can be influenced by its successors and suddenly a new atomic decomposition needs to be computed. On the other hand, since the current state of the forest is based on the previous solution, the algorithm needs to maintain it and also adapt it to the new partitioning. In order to fulfill these requirements, the hybrid algorithm performs the following tasks only for the root nodes.

Whenever a concept expression of the form $(\leq nR)$ or $(\geq mR)$ is added to $\mathcal{L}(x)$[5] (which means it did not already exist in $\mathcal{L}(x)$):

1. The *reset*-Rule becomes applicable for x which sets $\mathcal{L}_E(x) := \emptyset$ and clears the arithmetic label of the outgoing edges of x.

[5]This case can occur whenever (i) we have a cycle composed of root nodes and (ii) after merging two nodes.

2. Now that $\mathcal{L}_E(x)$ is empty, the $\leq$-Rule and $\geq$-Rule will be invoked to recompute the partitions and variables. Afterwards they will add the set of inequations based on the new partitioning.

3. If $(\sigma(v_i) = n) \in \Omega(x)$, where $v_i \in V_x$ corresponds to the previous partitioning, then set

$$\mathcal{L}_E(x) := \mathcal{L}_E(x) \cup \{ \sum_{v'_j \in V^i_x} v'_j \geq n, \sum_{v'_j \in V^i_x} v'_j \leq n\}$$

where $V^i_x := \{v'_j \in V'_x \mid \alpha(v_i) \subseteq \alpha(v'_j)\}$ and $v'_j \in V'_x$ are based on the new partitioning.

The third task in fact maintains the previous solutions in x and records them by means of inequalities in $\mathcal{L}_E(x)$. Therefore, the solution based on the new partitioning will be recreated by the arithmetic reasoner. To observe the functioning of these tasks in more detail, we refer the reader to Section 4.5.3.

Remark 3. When the algorithm records the solution $\sigma(v_i) = n$ by means of inequations, we can consider two cases for the successors that had been generated by the *fil*-Rule, based on the previous solution:

1. The successor is a root node y. Therefore, the corresponding solution must be of the form $\sigma(v_i) = 1$ where $R_{xy} \in \alpha(v_i)$. This solution will be translated to $\sum v'_j = 1$ for all j such that that $\alpha(v_i) \subseteq \alpha(v'_j)$. The solution for this equality will be also of the form $\sigma(v'_s) = 1$ for some $v_s \in V^i_x$. Since $R_{xy} \in \alpha(v_i)$ and $\alpha(v_i) \subseteq \alpha(v'_s)$, we can conclude that $R_{xy} \in \alpha(v'_s)$. Hence, the new solution will enhance the edge between x and y and possibly extend[6] its label.

2. The successor is not a root node and represents an anonymous individual x_i and $card(x_i) = n$. In this case n in the corresponding solution can be greater or

[6]Since $\alpha(v_i) \subseteq \alpha(v'_s)$, the new solution will not remove any role name from the label of this edge.

equal 1 and later the algorithm can create a solution for which p new nodes will be created, where $1 \leq p \leq n$. In other words, the node x_i will be removed from the forest and will be replaced by p new nodes. Since there exists no edge from non-root nodes to root nodes they never propagate back any information in the forest. Therefore, removing x_i from the forest does not violate restrictions on the root nodes.

4.5.3 Example ABox

Consider the ABox $\mathcal{A}$ with respect to an empty TBox and empty role hierarchy: $\mathcal{A} = \{a :\leq 1R, b : \forall S.(\forall R.(\geq 3S.C)), (a,b) : R, (a,c) : R, (b,d) : R, (c,d) : S, (d,c) : R\}$. Assume the algorithm generates a forest with root nodes a, b, c, and d.

Preprocessing:

1. Applying the function unQ:
   $\mathcal{R} = \mathcal{R} \cup \{S' \sqsubseteq S\}$ and $(\geq 3S.C) \rightarrow \geq 3S' \sqcap \forall S'.C$

2. Converting ABox role assertions: $\mathcal{R} = \mathcal{R} \cup \{R_{ab} \sqsubseteq R, R_{ac} \sqsubseteq R, R_{bd} \sqsubseteq R, S_{cd} \sqsubseteq S, R_{dc} \sqsubseteq R\}$ and $\mathcal{L}(a) = \mathcal{L}(a) \cup \{\leq 1R_{ab}, \geq 1R_{ab}, \leq 1R_{ac}, \geq 1R_{ac}\}$, $\mathcal{L}(b) = \mathcal{L}(b) \cup \{\leq 1R_{bd}, \geq 1R_{bd}\}$, $\mathcal{L}(c) = \mathcal{L}(c) \cup \{\leq 1S_{cd}, \geq 1S_{cd}\}$, $\mathcal{L}(d) = \mathcal{L}(d) \cup \{\leq 1R_{dc}, \geq 1R_{dc}\}$

Applying the rules for ABox

The $\leq$-Rule and $\geq$-Rule are applicable for all of the nodes and translate the numerical restrictions into inequations (Figure 9). Node a is similar to node x in the *merging example* in Section 4.5.2 and invokes the *merge*-Rule for c and b.

Assume the *merge*-Rule replaces every occurrence of c by b, we will have $\mathcal{L}(b) = \{\leq 1R_{bd}, \geq 1R_{bd}, \leq 1S_{bd}, \geq 1S_{bd}, \forall S.(\forall R.(\geq 3S' \sqcap \forall S'.C))\}$ and for d we will have $\mathcal{L}(d) = \{\leq 1R_{db}, \geq 1R_{db}\}$ (Figure 10). We have four unqualified number restrictions in $\mathcal{L}(b)$ (equivalent to two equality restrictions) which will be transformed into inequations

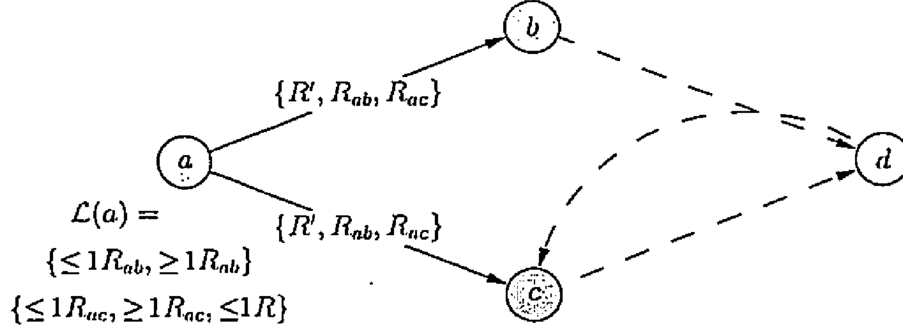

Figure 9: Initial completion graph. Dashed edges do not actually exist.

by the $\leq$-Rule and the $\geq$-Rule. Assuming $\alpha(v_{01}) = \{R_{bd}\}$ and $\alpha(v_{10}) = \{S_{bd}\}$, we will have $v_{01} + v_{11} = 1$ and $v_{10} + v_{11} = 1$. According to the second limitation on variables based on the ABox semantics, $v_{01} \leq 0$ and $v_{10} \leq 0$. Accordingly, there is only one solution for $\mathcal{L}_E(b)$ which is $\sigma(v_{11}) = 1$ and makes the *fil*-Rule set $\mathcal{L}(\langle b, d\rangle) = \{S_{bd}, R_{bd}\}$.

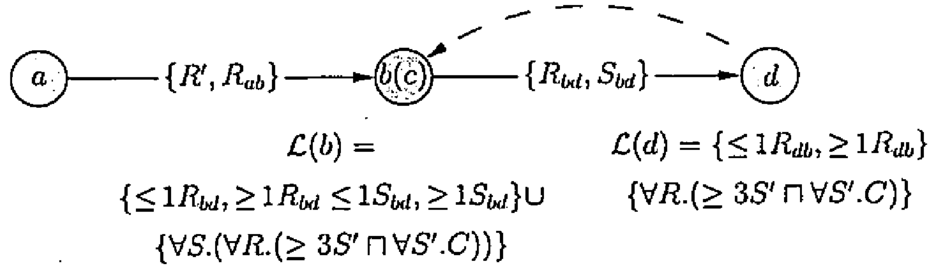

Figure 10: Completion graph after merging b and c

Afterwards, the $\forall$-Rule becomes applicable for the node b and adds $\forall R.(\geq 3S' \sqcap \forall S'.C)$ to $\mathcal{L}(d)$. There is only one inequation for d which results in setting $\mathcal{L}(\langle d, b\rangle) = \{R_{db}\}$. After this setting, as $R_{db} \sqsubseteq R$ the $\forall$-Rule adds $(\geq 3S' \sqcap \forall S'.C)$ to $\mathcal{L}(b)$. Since

$\geq 3S'$ did not exist in $\mathcal{L}(b)$, the algorithm performs *reset*(b). The $\leq$-Rule and the $\geq$-Rule will be fired again to recompute the partitions, considering the new numerical restriction ($\geq 3S'$). Let $\alpha(v'_{001}) = \{R_{bd}\}$, $\alpha(v'_{010}) = \{S_{bd}\}$, and $\alpha(v'_{100}) = \{S'\}$, the solution $\sigma(v_{11}) = 1$ for b must be expanded according to the new partitioning. Considering $\alpha(v_{11}) = \{R_{bd}, S_{bd}\}$ which is a subset of $\alpha(v'_{011})$ and $\alpha(v'_{111})$, the equation $v'_{011} + v'_{111} = 1$ will be added to $\mathcal{L}_E(b)$ as a placeholder of $\sigma(v_{11}) = 1$ and we will have:

$$(**)\begin{cases} \underline{v'_{001}} + v'_{011} + \underline{v'_{101}} + v'_{111} & = & 1 \\ \underline{v'_{010}} + v'_{011} + \underline{v'_{110}} + v'_{111} & = & 1 \\ v'_{100} + \underline{v'_{101}} + \underline{v'_{110}} + v'_{111} & \geq & 3 \\ v'_{011} + v'_{111} & = & 1 \end{cases}$$

According to the second limitation on variables according to ABox semantics, the variables v'_{001}, v'_{010}, v'_{101}, and v'_{110} must be less than zero. One of the solutions for (**) can be achieved when $v'_{111} \geq 1$, $v'_{011} \leq 0$, and $v'_{100} \geq 1$ are decided by the *ch*-Rule which is $\sigma(v'_{111}) = 1$ and $\sigma(v'100) = 2$. Subsequently, the *fil*-Rule will be fired for these solutions which adds S' to $\mathcal{L}(\langle b, d\rangle)$ and creates a new non-root node b' for which $\mathcal{L}(\langle b, b'\rangle) := \{S'\}$ and $card(b') = 2$. Finally, the $\forall$-Rule becomes applicable for $\forall S'.C$ in b and adds C to $\mathcal{L}(b)$ and $\mathcal{L}(b')$ (Figure 11).

4.6 Proof of Correctness

In this section we prove the correctness of the proposed hybrid algorithm for the Abox consistency test. Weaker problems such as the TBox consistency or concept satisfiability test can be reduced to the consistency test for ABoxes with respect to a given TBox and role hierarchy. In order to prove the correctness of an algorithm, one has to prove its termination, soundness, and completeness.

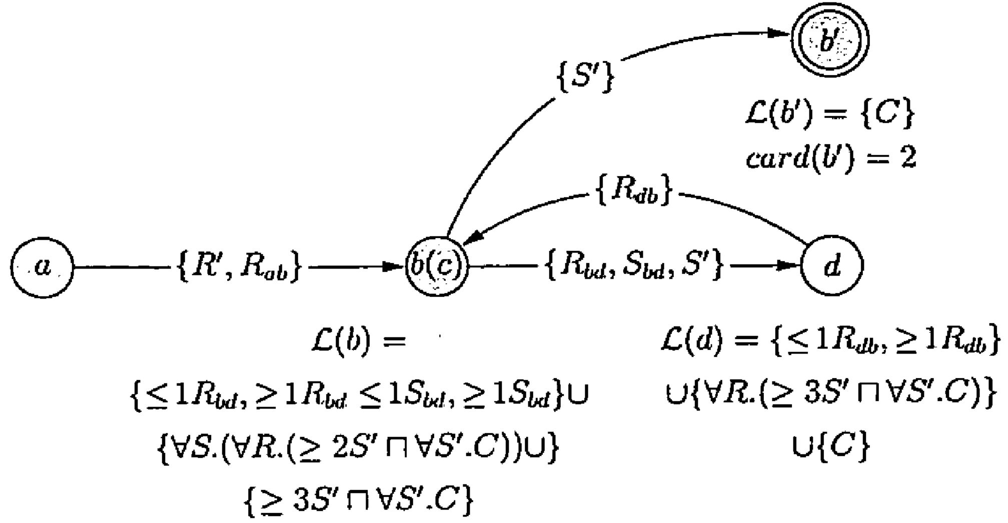

Figure 11: Final completion graph

4.6.1 Tableau

When proving the soundness and completeness of an algorithm, a *tableau* is defined as an abstraction of a model to facilitate comparing the output of the algorithm with a model. In fact, it is proven that a model can be constructed based on the information in a tableau and for every model there exists a tableau [HS07]. The similarity between tableaux and completion graphs, which are the output of the tableau algorithms, makes it easier to prove the correctness of the algorithm.

Due to the fact that the input of the hybrid tableau algorithm is in $\mathcal{SHN}^{\backslash}$[7], we define a tableau for $\mathcal{SHN}^{\backslash}$ ABoxes with respect to a role hierarchy $\mathcal{R}$. The following definition is similar to [HST00].

Definition 14 (Tableau for $\mathcal{SHN}^{\backslash}$). Let $R_{\mathcal{A}}$ be the set of role names and $I_{\mathcal{A}}$ the set of individuals in ABox $\mathcal{A}$, $T = (\mathbf{S}, \mathcal{L}^T, \mathcal{E}, \mathcal{J})$ is a tableau for $\mathcal{A}$ with respect to role hierarchy $\mathcal{R}$, where:

[7] By $\mathcal{SHN}^{\backslash}$ we mean the language $\mathcal{SHN}$ with subtraction on the roles and since we removed all of the qualified number restrictions from the language, without existential restrictions ($\exists R.C$).

- S is a non-empty set of elements (representing individuals),
- $\mathcal{L}^T : \mathsf{S} \to 2^{clos(\mathcal{A})}$ maps elements of S to a set of concepts,
- $\mathcal{E} : R_{\mathcal{A}} \to 2^{\mathsf{S}\times\mathsf{S}}$ maps each role to a set of pairs of elements in S,
- $\mathcal{J} : I_{\mathcal{A}} \to \mathsf{S}$ maps individuals occurring in $\mathcal{A}$ to elements of S.

Moreover, for every $s, t \in \mathsf{S}$, $A \in N_C$, $C_1, C_2, C \in clos(\mathcal{A})$, $R, S \in N_R$ the following properties hold for T, where $R^T(s) := \{t \in \mathsf{S} | \langle s, t\rangle \in \mathcal{E}(R)\}$:

P1 If $A \in \mathcal{L}^T(s)$, then $\neg A \notin \mathcal{L}^T(s)$.

P2 If $C_1 \sqcap C_2 \in \mathcal{L}^T(s)$, then $C_1 \in \mathcal{L}^T(s)$ and $C_2 \in \mathcal{L}^T(s)$.

P3 If $C_1 \sqcup C_2 \in \mathcal{L}^T(s)$, then $C_1 \in \mathcal{L}^T(s)$ or $C_2 \in \mathcal{L}^T(s)$.

P4 If $\forall R.C \in \mathcal{L}^T(s)$ and $\langle s, t\rangle \in \mathcal{E}(R)$, then $C \in \mathcal{L}^T(t)$.

P5 If $\forall R.C \in \mathcal{L}^T(s)$, for some $S \sqsubseteq R$ we have S is transitive, and $\langle s, t\rangle \in \mathcal{E}(S)$, then $\forall S.C \in \mathcal{L}^T(t)$.

P6 If $\forall R \backslash R'.C \in \mathcal{L}^T(s)$, $\langle s, t\rangle \in \mathcal{E}(R)$, but $\langle s, t\rangle \notin \mathcal{E}(R')$, then $C \in \mathcal{L}^T(t)$.

P7 If $\langle s, t\rangle \in \mathcal{E}(R)$, and $R \sqsubseteq S$, then $\langle s, t\rangle \in \mathcal{E}(S)$.

P8 If $\geq nR \in \mathcal{L}^T(s)$, then $\#R^T(s) \geq n$

P9 If $\leq mR \in \mathcal{L}^T(s)$, then $\#R^T(s) \leq m$

P10 If $(a : C) \in \mathcal{A}$ then $C \in \mathcal{L}^T(\mathcal{J}(a))$

P11 If $(a, b) : R \in \mathcal{A}$, then $\langle \mathcal{J}(a), \mathcal{J}(b)\rangle \in \mathcal{E}(R)$

P12 If $a \neq b \in \mathcal{A}$, then $\mathcal{J}(a) \neq \mathcal{J}(b)$

Lemma 1. *A $\mathcal{SHQ}$ ABox $\mathcal{A}$ has a tableau iff $unQ(\mathcal{A})$ has a $\mathcal{SHN}^{\backslash}$ tableau, where $unQ(\mathcal{A})$ indicates $\mathcal{A}$ after applying unQ to every concept expression occurring in $\mathcal{A}$.*

Lemma 1 is a straightforward consequence of the equisatisfiability of C and $unQ(C)$ for every concept expression C in $\mathcal{SHQ}$ (see Section 4.1 and [OK99]).

4.6.2 Termination

In order to prove termination of the hybrid algorithm, we prove that it constructs a finite forest. Since the given ABox has always a finite number of individuals (i.e., root nodes), it is sufficient to prove that the hybrid algorithm creates finite trees in which the root nodes represent ABox individuals. On the other hand, due to the fact that we include non-deterministic rules, the $\sqcup$-Rule and the *ch*-Rule, we must also prove that the algorithm creates finitely many forests due to non-determinism.

Lemma 2 (Termination). *The hybrid algorithm terminates for a given ABox $\mathcal{A}$ with respect to a role hierarchy $\mathcal{R}$*[8].

Proof. Let $m = |clos(\mathcal{A})|$ and k be the number of different numerical restrictions after the preprocessing step. Therefore, m is an upper bound on the length of a concept expression in the label of a node and k is the maximum number of roles participating in the atomic decomposition of a node. The algorithm creates a forest that consists of arbitrarily connected root nodes and their non-root node successors which appear in trees. The termination of the algorithm is a consequence of the following facts:

1. There are only two non-deterministic rules: the $\sqcup$-Rule and the *ch*-Rule. The $\sqcup$-Rule can be fired at most m times for a node x, which is the maximum length of $\mathcal{L}(x)$. On the other hand, the *ch*-Rule can be fired at most 2^{V_x} times and

[8] Since TBox axioms are propagated through the universal transitive role, we do not mention the TBox as an input of the algorithm (see Section 4.1)

V_x is bounded by 2^k. Accordingly, we can conclude that the non-deterministic rules can be fired finitely for a node and therefore the algorithm creates finitely many forests.

2. The only rule that removes a node from the forest is the *merge*-Rule which removes a root-node each time. Since there are $|I_{\mathcal{A}}|$ root nodes in the forest, this rule can be fired at most $|I_{\mathcal{A}}|$ times for a node. Moreover, according to the fact that the algorithm never creates a root node, it cannot fall in a loop of removing and creating the same node.

3. The only generating node is the *fil*-Rule which can create at most $|V_x|$ successors for a node x. Therefore, the out degree of the forest is bounded by $|V_x| \leq 2^k$.

4. According to the conditions of blocking, there exist no two nodes with the same logical label in a path in the forest, starting from a root node. In other words, the length of a path starting from a root node is bounded by the number of different logical labels (i.e., m).

5. The arithmetic reasoner always terminates for a finite set of inequations as the input.

According to (3) the out-degree of the forests is finite and due to (4) the depth of the trees is finite. Therefore the size of each forest created by the hybrid algorithm is bounded. On the other hand, according to (1), the algorithm can create finitely many forests. Considering (2) and (5), we can conclude the termination of the hybrid algorithm. □

4.6.3 Soundness

To prove the soundness of an algorithm, we must prove that the model, constructed based on a complete[9] and clash-free completion forest does not violate the semantics of the input language.[10] According to Lemma 1 and the fact that a model can be obtained from a $\mathcal{SHQ}$ tableau (proven in [HST00]), it is sufficient to prove that a $\mathcal{SHN}^{\backslash}$ tableau can be obtained from a complete and clash-free forest.

Lemma 3 (ABox semantics). *The hybrid algorithm preserves the semantics of ABox assertions; i.e., assertions of the form $(a,b) : R$ and $a \dot{\neq} b$.*

Proof. The algorithm replaces assertions of the form $(a,b) : R$ with $a : (\leq 1R_{ab} \sqcap \geq 1R_{ab})$. Consider x_a is the node corresponding to the individual $a \in I_{\mathcal{A}}$ in the forest $\mathcal{F}$ and likewise x_b for $b \in I_{\mathcal{A}}$. According to the definition of R_{ab}, cardinality restrictions. and assuming the fact that the algorithm correctly handles unqualified number restrictions, one can conclude that for some $v \in V_{x_a}$, $\sigma(v) = 1$ such that $R_{ab} \in \alpha(v)$. Therefore according to the condition (i) of the *fil*-Rule in Figure 8, R_{ab} will be added to $\mathcal{L}(\langle x_a, x_b \rangle)$. Since the algorithm preserves the role hierarchy and $R_{ab} \sqsubseteq R$ the assertion $(a,b) : R$ is satisfied.

On the other hand, due to the restriction $\leq 1R_{ab}$, for every $v' \neq v$ if $R_{ab} \in \alpha(v')$ then $\sigma(v') = 0$. Hence, a set of solutions $\mathcal{L}(\langle x_a, x_b \rangle)$ cannot be modified more than once by the *fil*-Rule for more than one variable and consequently, the label of $\langle x_a, x_b \rangle$ does not depend on the order of variables for which the *fil*-Rule applies.

Moreover, whenever we have an assertion of the form $a \dot{\neq} b$, the algorithms sets $v \leq 0$ for all the variables v such that $\exists R \in N_R; R_{xa} \in \alpha(v) \wedge R_{xb} \in \alpha(v)$. By means of this variable constraint, the algorithm makes it impossible for the arithmetic reasoner to create a solution which requires the algorithm to merge a and b. □

[9] A completion forest is complete if no expansion rule is applicable to any of its nodes.
[10] The semantics of $\mathcal{SHQ}$ ABox are demonstrated in Table 1.

S	:=	$\{x_1, x_2, \ldots, x_m \mid$ for every node x in $\mathcal{F}$, with $card(x) = m\}$
$\mathcal{L}^T(x_i)$	:=	$\mathcal{L}(x)$ for $1 \leq i \leq m$ if $card(x) = m$
$\mathcal{E}(R)$	:=	$\{\langle x_i, y_j \rangle \mid R' \in \mathcal{L}(\langle x, y \rangle) \wedge R' \sqsubseteq R\}$
$\mathcal{J}(a)$	:=	x^a if x^a is a root node in $\mathcal{F}$ representing individual $a \in I_{\mathcal{A}}$. If $x^b \in V$ is merged by x^a such that every occurrence of x^b is replaced by x^a, then $\mathcal{J}(b) = x^a$.

Figure 12: Converting forest $\mathcal{F}$ to tableau T

Lemma 4 (Soundness). *If the expansion rules can be applied to a $\mathcal{SHQ}$-ABox $\mathcal{A}$ and a role hierarchy $\mathcal{R}$ such that they yield a complete and clash-free completion forest, then $\mathcal{A}$ has a tableau w.r.t. $\mathcal{R}$.*

Proof. A $\mathcal{SHN}^\backslash$ tableau T can be obtained from a complete and clash-free completion forest $\mathcal{F} = (V, E, \mathcal{L}, \mathcal{L}_E)$ by mapping nodes in $\mathcal{F}$ to elements in T which can be defined from $\mathcal{F}$ as $T := (\mathbf{S}, \mathcal{L}^T, \mathcal{E}, \mathcal{J})$ such as in Figure 12.

In the following we prove the properties of a $\mathcal{SHN}^\backslash$ tableau for T:

- Since $\mathcal{F}$ is clash-free, **P1** holds for T.

- If $C_1 \sqcap C_2 \in \mathcal{L}^T(x_i)$, it means that $(C_1 \sqcap C_2) \in \mathcal{L}(x)$ in the forest $\mathcal{F}$. Therefore the $\sqcap$-Rule is applicable to the node x which adds C_1 and C_2 to $\mathcal{L}(x)$. Hence C_1 and C_2 must be in $\mathcal{L}^T(x_i)$ and we can conclude **P2** and likewise **P3** for T. Similarly, properties **P4**, **P5**, and **P6** are respectively guaranteed by the $\forall$-Rule, $\forall_+$-Rule, and $\forall_\backslash$-Rule.

- Assume $R \sqsubseteq S$, if $\langle x_i, y_j \rangle \in \mathcal{E}(R)$, we can conclude that $R \in \mathcal{L}(\langle x, y \rangle)$ in $\mathcal{F}$. The solutions created by the arithmetic reasoner maintain the role hierarchy by setting $v \leq 0$ if $R \in \alpha(v)$ but $S \notin \alpha(v)$. Moreover, since every R-successor is an S-successor, the role hierarchy is considered and properly handled by the $\forall$-Rule, $\forall_+$-Rule, and $\forall_\backslash$-Rule. Therefore, we will have $S \in \mathcal{L}(\langle x, y \rangle)$ and accordingly $\langle x_i, y_j \rangle \in \mathcal{E}(S)$. Hence, the property **P7** holds for T.

- The $\leq$-Rule and the $\geq$-Rule are invoked after the logical label, $\mathcal{L}(x)$, cannot be extended anymore. In other words, they create a correct partitioning based on all the numerical restrictions for a node. Therefore, the solution created by the arithmetic reasoner satisfies all the inequations in $\mathcal{L}_E(x)$. If $(\leq mR) \in \mathcal{L}^T(x_i)$, we had $(\leq mR) \in \mathcal{L}(x)$ for the corresponding node in $\mathcal{F}$. According to the atomic decomposition for x, the $\leq$-Rule will add $\Sigma v_i \leq m$ to $\mathcal{L}_E(x)$. Therefore, the solution $\Omega_j(x)$ for $\mathcal{L}_E(x)$ will satisfy this inequation and if $R \in \alpha(v_i^j) \wedge \sigma(v_i^j) \geq 1$ for $1 \leq i \leq k$ then $(\sigma(v_1^j) + \sigma(v_2^j) + \ldots + \sigma(v_k^j)) \leq m$. For every $\sigma(v_i^j) = m_i$ the fil-Rule creates an R-successor y_i with cardinality m_i for x. This node will be mapped to m_i elements in the tableau T which are R-successors of $x_i \in$ S. Therefore, x_i will have at most m R-successors in T and we can conclude that **P9** hold for T and **P8** is satisfied similarly.

- The hybrid algorithm sets $card(x_a) = 1$ and $\mathcal{L}(x_a) := \{C \mid (a : C) \in \mathcal{A}\}$ for every node x_a in $\mathcal{F}$ which represent an individual $a \in I_{\mathcal{A}}$. Therefore, an ABox individual will be represented by one and only one node and **P10** is satisfied. **P11** and **P12** are due to Lemma 3.

□

4.6.4 Completeness

In order to be complete, an algorithm needs to ensure that it explores all possible solutions. In other words, if a tableau T exists for an input ABox, the algorithm can apply its expansion rules in such a way that yields a forest $\mathcal{F}$ from which we can obtain T, applying the procedure in Figure 12.

Lemma 5. *In a complete and clash-free forest, for a node $x \in V$ and its successors $y, z \in V$, if $\mathcal{L}(\langle x, y\rangle) = \mathcal{L}(\langle x, z\rangle)$ then $\mathcal{L}(y) = \mathcal{L}(z)$.*

Proof. The only task that extends the logical label of a node is through the $\sqcup$-Rule, $\sqcap$-Rule, $\forall$-Rule, $\forall_+$-Rule, and the $\forall_\backslash$-Rule. Since $\mathcal{L}(\langle x, y\rangle) = \mathcal{L}(\langle x, z\rangle)$, the $\forall$-Rule, $\forall_+$-Rule, and the $\forall_\backslash$-Rule will have the same effect and extend $\mathcal{L}(y)$ and $\mathcal{L}(z)$ similarly. We can consider two cases:

(i) If y and z are non-root nodes, we have $\mathcal{L}(y) = \mathcal{L}(z) = \emptyset$ before starting the application of the expansion rules. Therefore, when extended similarly, they will remain identical after the application of the tableau rules.

(ii) If y is a root node, then there exists a role name $R \in N_R$ such that $R_{xy} \in \mathcal{L}(\langle x, y\rangle)$. Therefore, if $\mathcal{L}(\langle x, y\rangle) = \mathcal{L}(\langle x, z\rangle)$ then $R_{xy} \in \mathcal{L}(\langle x, z\rangle)$ which results in merging y and z in a single node by the *merge*-Rule. Therefore, we can still conclude that $\mathcal{L}(y) = \mathcal{L}(z)$. □

Corollary 1. According to the mapping from a forest $\mathcal{F}$ to a $\mathcal{SHN}^\backslash$ tableau T (Figure 12), every $\mathcal{L}^T(t)$ in T is equal to $\mathcal{L}(x)$ in $\mathcal{F}$ if x is mapped to t. Moreover, every $R \in \mathcal{L}(\langle x, y\rangle)$ is mapped to $\langle t, s\rangle \in \mathcal{E}(R)$. Therefore $\mathcal{L}(\langle x, y\rangle) = \mathcal{L}(\langle x, z\rangle)$ is equivalent to $\{R \in N_R | \langle s, t\rangle \in \mathcal{E}(R)\} = \{R \in N_R | \langle s, t'\rangle \in \mathcal{E}(R)\}$ where x is mapped to s, y to t, and z to t'. Furthermore, $\mathcal{L}(y) = \mathcal{L}(z)$ is equivalent to $\mathcal{L}^T(t) = \mathcal{L}^T(t')$. Thus, according to the Lemma 5, we can conclude:

$$\{R \in N_R | \langle s, t\rangle \in \mathcal{E}(R)\} = \{R \in N_R | \langle s, t'\rangle \in \mathcal{E}(R)\} \Rightarrow \mathcal{L}^T(t) = \mathcal{L}^T(t')$$

Lemma 6. *If a set of non-negative integer solutions $\Omega(x)$ based on the set of inequations $\mathcal{L}_E(x)$ causes a clash, all other non-negative integer solutions for $\mathcal{L}_E(x)$ will also trigger a clash.*

Proof. Assume we have a solution $\Omega(x) = \{\sigma(v_1) = m_1, \sigma(v_2) = m_2, \ldots, \sigma(v_n) = m_n\}$, which can only occur when $v_i \geq 1$ for $1 \leq i \leq n$ is decided by the *ch*-Rule and all other variables in V_x are equal to zero. Suppose we have a different solution $\Omega'(x)$

for the set of inequations in $\mathcal{L}_E(x)$ including variable constraints decided by the *ch*-Rule such that $\Omega'(x) = \{\sigma'(v_1) = p_1, \sigma'(v_2) = p_2, \ldots, \sigma'(v_n) = p_n\}$. The label of the edges, created by the *fil*-Rule only depends on the variables. Therefore, considering Lemma 5 the forest generated based on $\Omega'(x)$ will contain the same nodes as $\Omega(x)$, however with different cardinalities. Since clashes do not depend on the cardinality of the nodes, we can conclude that $\Omega'(x)$ will result in the same clash as $\Omega(x)$. $\square$

Corollary 2. According to Lemma 6, all of the solutions for $\mathcal{L}_E(x)$ will end up with the same result; either all of them yield a complete and clash-free forest or return a clash.

Lemma 7 (Completeness). *Let $\mathcal{A}$ be a $\mathcal{SHQ}$-ABox and $\mathcal{R}$ a role hierarchy. If $\mathcal{A}$ has a tableau w.r.t. $\mathcal{R}$, then the expansion rules can be applied to $\mathcal{A}$ such that they yield a complete and clash-free completion forest.*

Proof. We assume we have a $\mathcal{SHN}^{\backslash}$ tableau $T = (\mathbf{S}, \mathcal{L}^T, \mathcal{E}, \mathcal{J})$ for $\mathcal{A}$ and we claim that the hybrid algorithm can create a forest $\mathcal{F} = (V, E, \mathcal{L}, \mathcal{L}_E)$ from which T can be obtained. The procedure of retrieving T from $\mathcal{F}$ is presented in Figure 12. We prove this by induction on the set of nodes in V.

Consider a node x in $\mathcal{F}$ and the expansion rules in Figure 8 and let $s \in \mathbf{S}$ in T be the element that is mapped from x. We actually want to prove that with guiding the application of the non-deterministic rules on x we can extend $\mathcal{F}$ such that it can still be mapped to T.

- The $\sqcup$-Rule: If $(C_1 \sqcup C_2) \in \mathcal{L}(x)$ then $(C_1 \sqcup C_2) \in \mathcal{L}^T(s)$. The $\sqcup$-Rule adds C_1 or C_2 to $\mathcal{L}(x)$ which is in accordance with the property **P2** of the tableau where for some concept $E \in \{C_1, C_2\}$ we have $E \in \mathcal{L}^T(s)$. The $\sqcap$-Rule, $\forall$-Rule, $\forall_+$-Rule, and the $\forall_\backslash$-Rule, which are deterministic rules, are similar to the $\sqcup$-Rule. In fact these rules are built exactly based on their relevant tableau property.

- The ch-Rule: Consider in T for s we have $t_1, t_2, \ldots, t_n$ as the successors of s, i.e., $\exists R \in N_R, \langle s, t_i \rangle \in \mathcal{E}(R)$. Intuitively, we cluster these successors in groups of elements with the same label $\mathcal{L}^T$. For example if $t_k, \ldots, t_l$ have the same label, according to Corollary 1, $N_{kl} := \{R \in N_R | \langle s, t_j \rangle \in \mathcal{E}(R)\}$ will be identical for t_j, $k \leq j \leq l$. We define a variable v_{kl} for such set of role names such that $\alpha(v_{kl}) = N_{kl}$. In order to have T as the mapping of $\mathcal{F}$, the ch-Rule must impose $v_{kl} \geq 1$.

  According to properties **P8** and **P9** of the tableau, $\leq nR$ and $\geq mR$ are satisfied in T for s. Therefore, the inequations based on these variables will have a non-negative integer solution. Notice that the set of variable constraints created based on T may result in a different solution. For example in T, the element s may have t_1 and t_2 as successors with the label $\mathcal{L}_1^T$ which sets $v \geq 1$ and t'_1, t'_2, and t'_3 as successors with the label $\mathcal{L}_2^T$ which sets $v' \geq 1$. However, in the solution based on these variable constraints we may have three successors with the label $\mathcal{L}_1^T$ and two successors with the label $\mathcal{L}_2^T$. Nevertheless, according to the Lemma 6 this fact does not violate the completeness of the algorithm.

- The $reset$-Rule is a deterministic rule which is only applicable to root nodes. Clearing the label of the outgoing edges and also $\mathcal{L}_E(x)$ does not violate properties of the tableau mapped from $\mathcal{F}$. This is due to the fact that the label of the outgoing edges from x will later be set by the fil-Rule which has a lower priority.

- The $merge$-Rule is also only applicable to root nodes. Assume individuals $a, b, c \in I_A$ are such that b and c are successors of a that must be merged according to an at-most restriction $a : \leq nR$. Since T is a tableau the restriction $\leq nR \in \mathcal{L}^T(\mathcal{J}(a))$ imposes that $\mathcal{J}(b) = \mathcal{J}(c)$. On the other hand, if x_a, x_b, and x_c are root nodes representing a, b, and c, the $merge$-Rule will merge x_b and

x_c according to the solution for $\mathcal{L}_E(x_a)$. In the mapping from $\mathcal{F}$ to T, x_b and x_c will be mapped to the same element that implies $\mathcal{J}(b) = \mathcal{J}(c)$ which follows the structure of T.

- The $\leq$-Rule and the $\geq$-Rule only modify $\mathcal{L}_E(x)$. Therefore, they will not affect the mapping of T from $\mathcal{F}$.
- The fil-Rule, with the lowest priority, generates successors for x according to the solution provided by the arithmetic reasoner for $\mathcal{L}_E(x)$. Since $\mathcal{L}_E(x)$ conforms to the at-most and at-least restrictions in the label of x and according to the variable constraints decided by the ch-Rule, the solution will be consistent with T. Notice that every node x in $\mathcal{F}$ for which $card(x) = m > 1$ will be mapped to m elements in S.

The resulting forest $\mathcal{F}$ is clash free and complete due to the following properties:

1. $\mathcal{F}$ cannot contain a node x such that $\{A, \neg A\} \subseteq \mathcal{L}(x)$ since $\mathcal{L}(x) = \mathcal{L}^T(s)$ and property **P1** of the definition of a tableau would be violated.
2. $\mathcal{F}$ cannot contain a node x such that $\mathcal{L}_E(x)$ is unsolvable. If $\mathcal{L}_E(x)$ is unsolvable, this means that there exists a restriction of the form $(\geq nR)$ or $(\leq mR)$ in $\mathcal{L}(x)$ and therefore $\mathcal{L}^T(s)$ that cannot be satisfied which violates property **P8** and/or **P9** of a tableau.

□

Chapter 5

Practical reasoning

There is always a conflict between the expressiveness of a DL language and the difficulty of reasoning. Increasing the expressiveness of a reasoner with qualified number restrictions can become very expensive in terms of efficiency. As shown in Chapter 3, a standard algorithm to deal with qualified number restrictions must extend its tableau rules with at least two non-deterministic rules; i.e., the *choose*-Rule and the $\leq$-Rule. In order to achieve an acceptable performance, a tableau algorithm needs to employ effective optimization techniques. As stated in [Hor03], the performance of the tableau algorithms even for simple logics is a problematic issue.

In this chapter we briefly analyze the complexity of both standard and hybrid algorithms. Based on the complexity analysis, we address the sources of inefficiency in DL reasoning. Moreover, we propose some optimization techniques for the hybrid algorithm to overcome its high *practical complexity*. In the last section we give special attention to dependency-directed backtracking as a major optimization technique and compare its effect on both the standard and the hybrid algorithm.

5.1 Complexity Analysis

In this section we analyze the complexity of the hybrid algorithm as well as the typical standard algorithm presented in Figure 5 in Chapter 3. To analyze the complexity of concept satisfiability test with respect to qualified number restrictions, we count the number of branches that the algorithm creates in the search space[1].

In the following we assume a node $x \in V$ in the completion graph/forest that contains p at-least restrictions and q at-most restrictions in its label:

$$\{\geq n_1 R_1.C_1, \geq n_2 R_2.C_2, \ldots, \geq n_p R_p.C_p\} \subseteq \mathcal{L}(x)$$

$$\{\leq m_1 R'_1.C'_1, \leq m_2 R'_2.C'_2, \ldots, \leq m_q R'_q.C'_q\} \subseteq \mathcal{L}(x)$$

such that $R_i, R'_j \in N_R$ and $C_i, C'_j \in clos(\mathcal{T})$.

5.1.1 Standard Tableaux

A standard tableau algorithm as was shown in Figure 5 creates n R-successors in C for each at-least restriction of the form $\geq nR.C$. Moreover, in order to avoid that they are being merged, it sets them as mutually distinct individuals. Assuming that no C_i is subsumed by a C_j, there will be $N := n_1 + n_2 + \ldots + n_p$ successors for x which are composed of p sets of successors, such that successors in each set are mutually distinct.

Moreover, according to every at-most restriction $\leq m_i R'_i.C'_i$ the *choose*-Rule will create two branches in the search space for each successor. Therefore, based on the q at-most restrictions in $\mathcal{L}(x)$, there will be 2^q cases for each successor of x. Since x has N successors, there will be totally $(2^q)^N$ cases to be examined by the algorithm. Notice

[1]Notice that every non-deterministic rule that can have k outcomes, opens k new branches in the search space.

that the creation of these 2^{qN} branches is independent from any clash occurrence and the algorithm will always invoke the *choose*-Rule $N \times p$ times.

Suppose the algorithm triggers a clash according to the restriction $\leq m_i R'_i.C'_i$. If there exist M R'_i-successors in C'_i such that $M > m_i$, the algorithm opens $f(M, m_i)$ $:= \binom{M}{2}\binom{M-1}{2}\dots\binom{m_i+1}{2}/(M - m_i)!$ new branches in the search space which is the number of possible ways to merge M individuals into m_i individuals. In the worst case, if $m := min\{m_i; 1 \leq i \leq q\}$ there will be $f(N, m)$ ways to merge all the successors of x. Therefore, in the worst-case one must explore $(2^q)^N \times f(N, m)$ branches.

5.1.2 Hybrid Tableau

During the preprocessing step, the hybrid algorithm converts all the qualified number restrictions into unqualified ones which introduces $p + q$ new role names. According to the atomic decomposition presented in Section 4.2, the hybrid algorithm defines $2^{p+q} - 1$ partitions and consequently variables for x; i.e. $|V_x| = 2^{p+q} - 1$. The *ch*-Rule opens two branches for each variable in V_x. Therefore, there will be totally 2^{V_x} cases to be examined by the arithmetic reasoner. Hence, the *ch*-Rule will always be invoked $|V_x| = 2^{p+q} - 1$ times and creates $2^{2^{p+q}}$ branches in the search space. Hence, the complexity of the algorithm seems to be characterized by a double-exponential function of $p+q$; moreover, considering the NP-completeness of *Integer Programming* one can conclude that the worst case complexity of such an algorithm is dramatically high.

5.1.3 Hybrid vs. Standard

Comparing the standard algorithm complexity with the complexity of the hybrid algorithm, we can conclude:

- The complexity of the standard algorithm is a function of N and therefore the

numbers occurring in the most restrictions can affect the standard algorithm exponentially. Whereas in the hybrid algorithm, the complexity is independent from N due to its arithmetic approach to the problem.

- Let *initial complexity* refer to the complexity of the tasks that the algorithm needs to perform independently from the occurrence of a clash. That is to say, the tasks that need to be done in all the cases (whether worst-case or the best-case). Particularly, the *initial complexity* of the standard algorithm is due to the *choose*-Rule ($(2^q)^N$) and the *initial complexity* of the hybrid algorithm is due to the *ch*-Rule ($2^{2^{p+q}}$). Therefore, whenever $N \times q < 2^{p+q}$, the time spent for initializing the algorithm is greater for the hybrid algorithm in comparison with the standard algorithms.

- The major source of complexity in the standard algorithm is due to the *merge*-Rule. Being highly nondeterministic, this rule can be a major source of inefficiency. Therefore, in the case of hardly satisfiable concept expressions, the standard algorithm can become very inefficient. In contrast, the hybrid algorithm generates and merges the successors of an individual deterministically and based on an arithmetically correct solution for a set of inequations.[2]

- Whenever a clash occurs, the algorithm needs to backtrack to a choice point to choose a new branch. The sources of nondeterminism due to numerical restrictions in the standard algorithm are more than one: the *choose*-Rule and the *merge*-Rule, whereas in the hybrid algorithm we have only the *ch*-Rule. Therefore, in the hybrid algorithm it is easier to track the sources of a clash.

[2]Note that the hybrid algorithm never merges anonymous (non-root) nodes.

5.2 Optimization Techniques

As there are several reasoning services, different optimization techniques have been developed to address them. For example, *absorption* [HT00a, HT00b] or *lazy unfolding* [BHN+94] are some optimization techniques for TBox services, such as classification or subsumption. These optimization techniques normally facilitate subsumption testing and by avoiding unnecessary steps in the TBox reasoning improve the performance of the reasoner. The hybrid algorithm is meant to address the performance issues regarding reasoning with qualified number restrictions independently from the reasoning service. In other words, by means of the hybrid reasoning, we want to improve reasoning at the concept satisfiability level which definitely affects TBox and ABox reasoning.

At the concept satisfiability level, the major source of inefficiency is due to the high nondeterminism. In fact, nondeterministic rules such as the $\sqcup$-Rule in Figure 7 or the *choose*-Rule in Figure 5 create several branches in the search space. In order to be complete, an algorithm needs to explore all of these branches in the search space. Optimization techniques mainly try to reduce the size of the search space by pruning some of these branches. Moreover, some heuristics can help the algorithm to guess which branches to explore first. In fact, the more knowledge the algorithm uses to guide the exploration, the less probable its decision will fail later.

Although it seems that the hybrid algorithm is double-exponential and the large number of variables seems to be hopelessly inefficient, there are some heuristics and optimizations techniques which make it feasible to use. In the following we briefly explain three heuristics which can significantly improve the performance of the algorithm in the typical case.

5.2.1 Default Value for the Variable

In the semantic branching based on the concept *choose*-Rule, in one branch we have C and in the other branch we have $\neg C$ in the label of the nodes. However, due to the *ch*-Rule (for variables) in one branch we have $v \geq 1$ whereas in the other branch $v \leq 0$. In contrast to concept branching according to the *choose*-Rule,[3] in variable branching we can ignore the existence of the variables that are less or equal zero. In other words, the arithmetic reasoner only considers the variables that are greater or equal one.

Therefore, by setting the default value of $v \leq 0$ for every variable, the algorithm does not need to invoke the *ch*-Rule $|V_x|$ times before starting to find a solution for the inequations. More precisely, the algorithm starts with the default value of $v \leq 0$ for all of the variables in $|V_x|$. Obviously, the solution for this set of inequations, which is $\forall v_i \in V_x; \sigma(v_i) = 0$, cannot satisfy any at-least restriction. Therefore, the algorithm must choose some variables in V_x to make them greater or equal one. Although in the worst case the algorithm still needs to try $2^{|V_x|}$ cases, by setting this default value it does not need to invoke the *ch*-Rule when it is not necessary. In other words, by benefiting from this heuristics, the initial complexity of the hybrid algorithm is no longer 2^{p+q}.

5.2.2 Strategy of the *ch*-Rule

As explained in the previous section, in a more optimized manner, the algorithm starts with the default value of zero for all the variables. Afterwards, it must decide to set some variables greater than zero in order to find an arithmetic solution. The order in which the algorithm chooses these variables can help the arithmetic reasoner

[3]The *choose*-Rule opens two branches in the search space according to the at-most restriction $\leq mR.C$ such that in one of them C is in the label of the individual and in the other $\neg C$.

find the solution faster.

We define *don't care* variables as the set of variables that have appeared in an at-least restriction but in no at-most restriction. Therefore, these variables have no restrictions other than logical restrictions which later on will be processed by the algorithm. Therefore, according to arithmetic limitations, any non-negative integer value can be assigned to these variables and we can let them exist in all of the inequations unless they trigger a logical clash.

Moreover, we define the *satisfying variables* as the set of variables which occur in an at-least restriction and are not *don't care* variables. Since these are the variables that occur in an at-least restriction, by assigning them to be greater or equal to one, the algorithm can lead the arithmetic reasoner to a solution. Whenever a node that is created based on v causes a clash, by means of the dependency-directed backtracking we will set $v \leq 0$ and therefore remove v from the *satisfying variables* set. When the *satisfying variables* set becomes empty the algorithm can conclude that the set of qualified number restrictions in $\mathcal{L}(x)$ is unsatisfiable.

Notice that the number of variables that can be decided to be greater than zero in an inequation is bounded by the number occurring in its corresponding numerical restriction. For example, in the inequation $v_1 + v_2 + \ldots + v_{100} \geq 5$, although we have 100 variables in the inequation, not more than five v_i can be greater or equal than one at the same time.

5.2.3 Variables Encoding

One of the interesting characteristics of the variables is that we can encode their indices in binary format to easily retrieve the role names related to them. On the other hand, we do not need to assign any memory space for them unless they have a value greater than zero based on an arithmetic solution.

5.3 Dependency-Directed Backtracking or Backjumping

As introduced in [Hor02], dependency-directed backtracking or backjumping is a backtracking method which detects the sources of a clash and tries to bypass branching points that are not related to the sources of the clash. By means of this method, an algorithm can prune branches that will end up with the same sort of clash. As demonstrated in [Hor02], this method improved the performance of the FaCT system to deal much more effectively with qualified number restrictions.

Similarly in the hybrid algorithm, whenever we encounter a logical clash for a successor y of x, we can conclude that the corresponding variable v_y for the partition in which y resides must be zero. Therefore, we can prune all branches for which $v_y \geq 1 \in \mathcal{L}_E(x)$. This simple method of backtracking can exponentially decrease the size of the search space by pruning half of the branches each time the algorithm detects a clash. For example, in the general case of $\mathcal{L}(x)$, by pruning all the branches where $v_y \geq 1$, we will in fact prune $2^{|V_x|-1} = 2^{2^{p+q}-1}$ branches w.r.t. the *ch*-Rule which is half of the branches.

We can improve this by a more complex dependency-directed backtracking in which we prune all the branches that have the same reason for the clash of v_y. For instance, assume the node y that is created based on $\sigma(v_y) = k$ where $k \geq 1$ ends up with a clash. Since we have only one type of clash other than the arithmetic clash, assume the clash is because of $\{A, \neg A\} \subseteq \mathcal{L}(y)$ for some $A \in N_C$. Moreover, assume we know that A is caused by a $\forall R_i.A$ restriction in its predecessor x and $\neg A$ by $\forall S \backslash T_j.(\neg A) \in \mathcal{L}(x)$. It is possible to conclude that all the variables v for which $R_i \in \alpha(v) \wedge T_j \notin \alpha(v)$ will end up with the same clash.[4]

[4] Notice that in the cases where we have a disjunction in the sources of a clash, there may exist more than two sources for a clash. For example, assume $\{\forall R.(A \sqcup \neg B), \forall S.(B \sqcup \neg C), \forall T.(C \sqcup \neg A)\} \subseteq$

Consider the binary coding for the indices of the variables in which the ith digit represents R_i and the jth digit represents T_j. Therefore, all the variables, where the binary coding has 1 as its ith digit and 0 as its jth digit must be zero. Since the binary coding of the variable indices has a total of $p+q$ digits, the number of variables that must be zero will be 2^{p+q-2}. All other variables which are $2^{p+q}-2^{p+q-2}$, can freely take two types of constraints and open two branches in the search space. Therefore, the number of branches will reduce from $2^{|V_x|}$ to $2^{3/4|V_x|}$ which is a significant improvement. In fact, the *atomic decomposition* is a method to organize the search space and at the same time by means of numerical reasoning and proxy individuals remains unaffected by the value of numbers.

For example, consider the case when there are 7 numerical restrictions in $\mathcal{L}(x)$ and therefore 2^7 variables. Accordingly, the *ch*-Rule opens 2^{128} branches in the search space. If y is a successor of x which is created based on the solution $\sigma(v_{0011101}) = m$ and y ends up with a clash, the algorithm can conclude that $v_{0011101} \leq 0$ must be added to $\mathcal{L}_E(x)$. Therefore, based on simple backtracking, 2^{127} branches remain to be explored. Moreover, assume the clash in y is due to $\{A, \neg A\} \subseteq \mathcal{L}(y)$ where A is created because of $\forall R_3.A \in \mathcal{L}(x)$ and $\neg A$ is created because of $\forall R \backslash R_2.\neg A \in \mathcal{L}(x)$. Hence, the algorithm can conclude that for all the variables v where $R_3 \in \alpha(v)$ and $R_2 \notin \alpha(v)$, the same clash will occur. Namely, variables of the form $v_{\Box 01\Box\Box\Box\Box}$ where $\Box \in \{1, 0\}$ must be zero. Therefore, $2^{128-32} = 2^{3\times 32}$ branches remain to be explored.

5.3.1 Backtracking in the Arithmetic Reasoner

Normally there could be more than one solution for a set of inequations. According to Lemma 6 in Chapter 4, when we have a solution with respect to a set of restrictions

---

$\mathcal{L}(x)$ and we have $\{R, S, T\} \subseteq \mathcal{L}(\langle x, y\rangle)$ and y leads to a clash. In fact, all of these three role names in $\mathcal{L}(\langle x, y\rangle)$, together, are the sources of this clash. Therefore, the algorithm concludes that all the variables v for which $R \in \alpha(v) \wedge S \in \alpha(v) \wedge T \in \alpha(v)$ will end up with the same clash.

of the form $v_i \geq 1$, different solutions where the non-zero variables only differ in their values do not make any logical differences. In fact, the algorithm will create successors with the same logical labels but different cardinalities based on these different solutions. Since all the solutions minimize the sum of variables and satisfy all the numerical restrictions, they do not make any arithmetic differences (as long as the set of zero-value variables is the same).

In addition, notice that backtracking within arithmetic reasoning is not trivial due to the fact that the cause of an arithmetic clash cannot be easily traced back. In other words, the whole set of numerical restrictions together causes the clash. In the same sense as in a standard tableau algorithm, if all the possible merging arrangements end up with a clash, one can only conclude that the corresponding numerical restrictions are not satisfiable together.

5.3.2 Backjumping: Standard Algorithms vs. the Hybrid Algorithm

By comparing the effect of dependency-directed backtracking on the hybrid algorithm and on the standard algorithm, we can conclude:

1. In fact, the atomic decomposition is a mechanism of organizing role-fillers of an individual in partitions that are disjoint and yet cover all possible cases. Therefore, it is more suitable for dependency-directed backtracking. In other words, the whole tracking and recording that are performed in order to detect sources of a clash to prune the search space, are hard-coded in the hybrid algorithm by means of the atomic decomposition.

2. In the hybrid algorithm, the sources of nondeterminism are only the *ch*-Rule and the $\sqcup$-Rule, whereas in the standard algorithms we have three sources of

non-determinism: the $\sqcup$-Rule, the *choose*-Rule, and the $\leq$-Rule. Therefore, in contrast to the standard algorithms which have three non-deterministic rules, the hybrid algorithm can more easily backjump to the source of the clash. In other words, the nondeterminism due to the concept *choose*-Rule and the $\leq$-Rule is gathered just in one level which is the variable *ch*-Rule.

Chapter 6

Reasoner Description

In this chapter we explain the architecture of the implemented reasoner which is based on the hybrid algorithm presented in Chapter 4 and benefits from the optimization techniques proposed in Chapter 5. Moreover, the backtracking method can be turned off, switched to the simple level or complex level for the sake of comparison. After presenting the architecture of the whole reasoner, we zoom into the logical module and describe its mechanism of rule expansion. To overcome the high complexity of the *ch*-Rule (see Section 5.1.2), its application is moved to the arithmetic module which is responsible for finding a non-negative integer solution. We explain the arithmetic reasoner in more detail in the next section. Finally, we describe the problems we encountered during the implementation of the hybrid reasoner.

6.1 Architecture

As illustrated in Figure 13, the hybrid reasoner is composed of two major modules: the logical module and the arithmetic module. The input of the reasoner is an $\mathcal{ALCHQ}$[1]

[1]The language $\mathcal{ALCHQ}$ is equivalent to $\mathcal{SHQ}$ without transitive roles. Since transitive roles are assumed to have no interaction with qualified number restrictions, they were not be implemented in the hybrid reasoner.

concept expression. The output of the algorithm is either a complete and clash-free completion graph[2] if the input concept expression is satisfiable and otherwise it returns "unsatisfiable". The complete and clash-free completion graph can be considered as a pre-model based on which we can construct a tableau (see Figure 12).

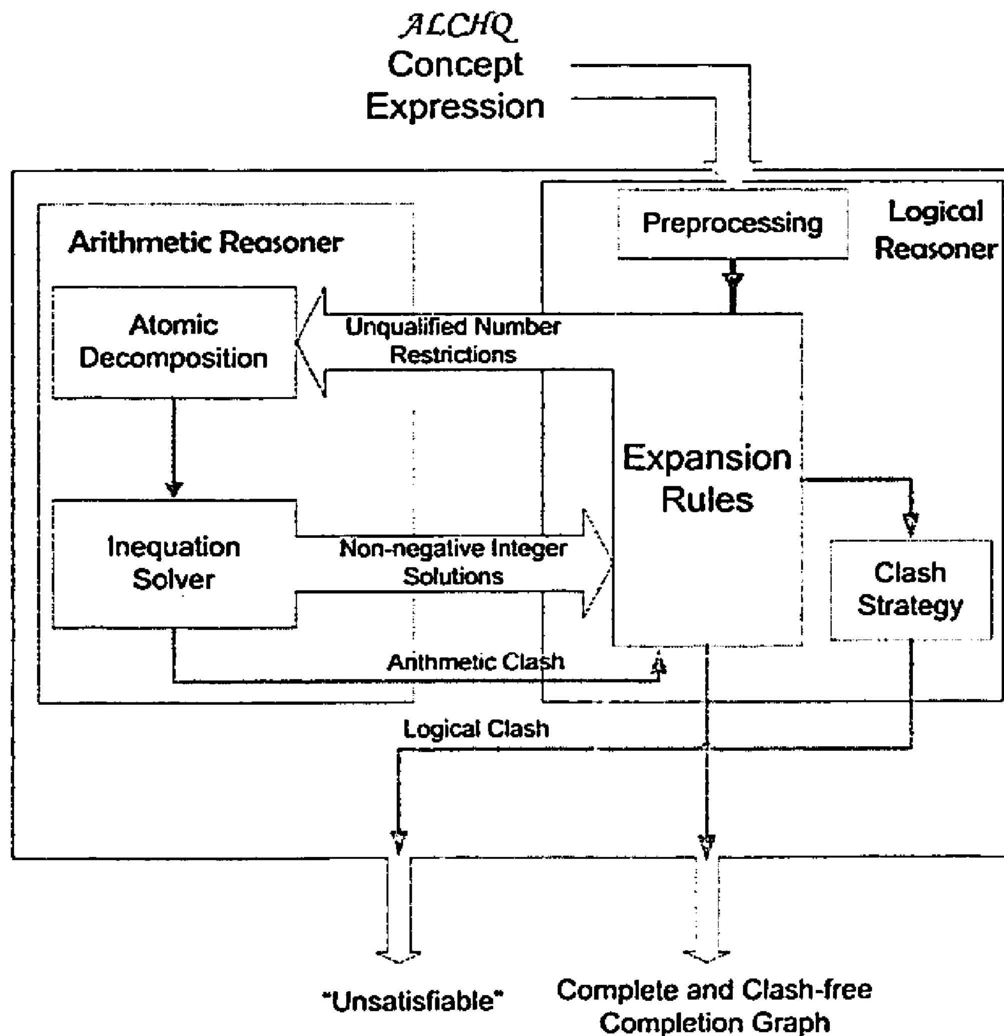

Figure 13: Reasoner architecture

The logical reasoner modifies the input concept expression according to the function *unQ* proposed in Section 4.1. It also provides the arithmetic module with a set

[2]Notice that since the input of the reasoner is not an ABox, the algorithm constructs a completion graph rather than a completion forest.

of unqualified number restrictions (UCRs). The arithmetic module either returns an arithmetic clash or a non-negative integer solution based on which the logical module generates the successors for an individual. In the following sections we describe applications of both modules in more detail.

6.2 Logical Module

The logical module can be considered as the main module which performs the expansion rules and calls the arithmetic reasoner whenever needed. It is composed of a preprocessing component which modifies the input ontology (.owl file) based on the *unQ* function. Therefore, it replaces qualified number restrictions with equisatisfiable unqualified ones which are also transformed to negation normal form. Notice that the converted language is not closed under negation. Accordingly, the reasoner never negates a concept expression that is a direct or indirect output of the preprocessing component. Moreover, the logical reasoner as illustrated in Figure 14 is composed of a set of expansion rules, clash strategy component, and some more auxiliary components.

The major data structure in the logical module is *state* which records a state of the completion graph. The logical reasoner builds a tree of states such that firing a deterministic rule creates only one child for a state. On the other hand, the application of a non-deterministic rule (such as $\sqcup$-Rule) can generate more than one child for a state. For example, if the reasoner fires the $\sqcup$-Rule for $C_1 \sqcup C_2 \sqcup \ldots C_n$ for an individual x in `state1`, the current state, `state1` will have n children each of which contains one of the disjuncts in $\mathcal{L}(x)$. In other words, each state contains a unique completion graph and if we had no non-deterministic rule, the output would be a single path of states. Moreover, every state contains all the information of its individuals, including their label, their cardinality, and the label of the edges.

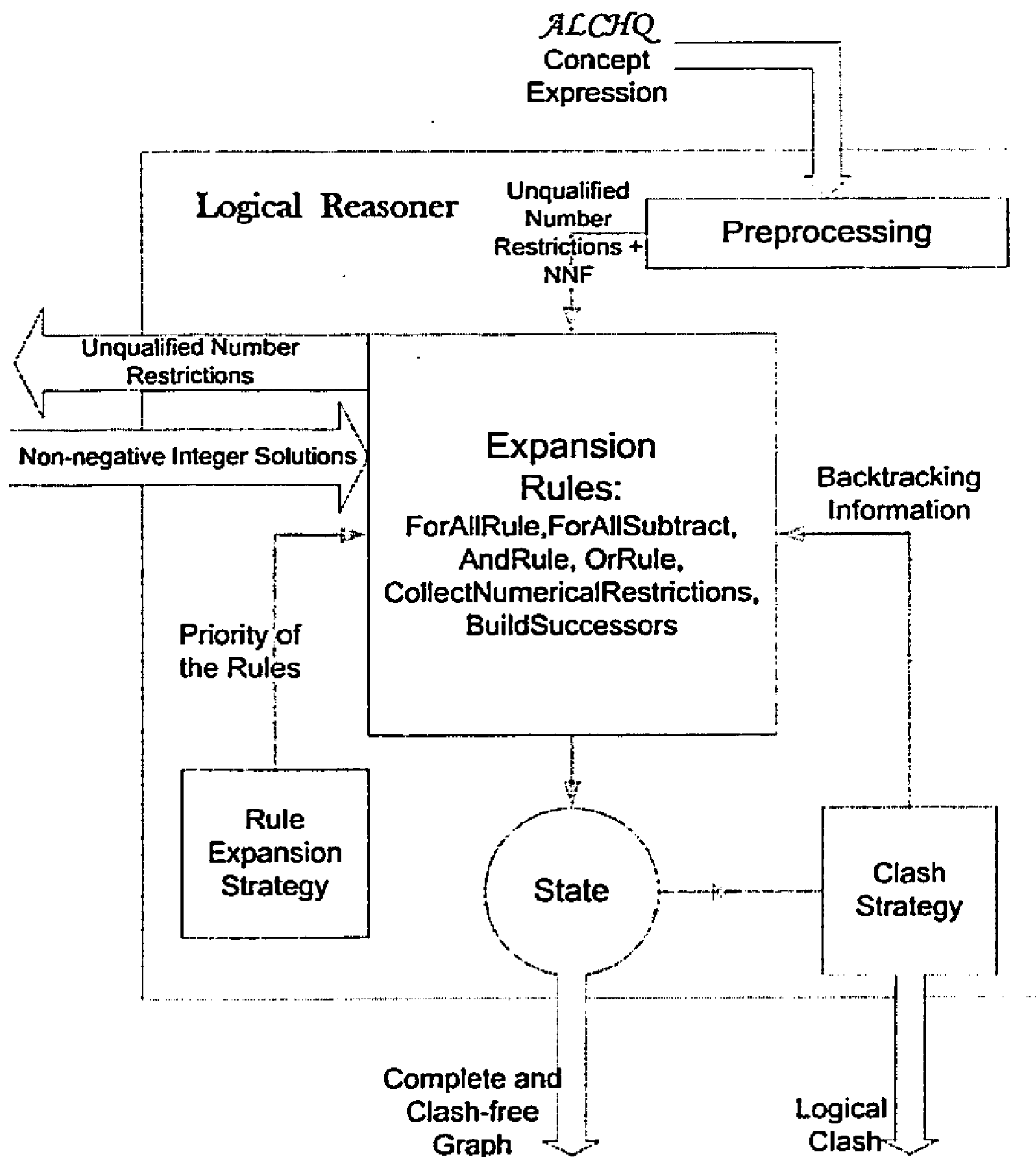

Figure 14: Architecture of the logical module

6.2.1 Expansion Rules

The set of expansion rules is based on the tableau rules presented in Figure 7. However, since the logical module has no information regarding the variables and inequations, the *ch*-Rule is moved to the arithmetic module. All the rules follow the general

template in Algorithm 1. Each rule has a precondition to be applicable to a state. Moreover, after its application, a rule modifies a copy of the current state to create a new state which will be a child of the current state. Furthermore, in each application of a rule, it will be fired for all of its individuals for which it is applicable. The logical module tries to apply the expansion rules due to their priority to every state that is not closed. If no rule is applicable to a state, it will be closed. If all of the states are clashed and closed the input concept expression is unsatisfiable.

Algorithm 1 Rule expansion template

```
canApply(State s) {
if s contains an individual for which Rule is applicable then
  return true
else
  return false
end if
}
apply(State s){
newState ← Copy(s)
newState.parent ← s
for all individual in s such that Rule is applicable do
  newInd ← apply Rule on indvidual
  replace individual with newInd in newState
end for
return newState
}
```

In the following, we assume that the current state on which the rule is applied is called `state1`. There are two variations of the set of expansion rules according to the use of backtracking.

Without backtracking: There are two rules which together function as the *fil*-Rule, the $\leq$-Rule, and the $\geq$-Rule: (i) The Collect-And-Initiate-Rule collects all the unqualified number restrictions in the label of individuals in `state1` and calls the arithmetic reasoner. The arithmetic reasoner computes all the cases for the variables based on the *ch*-Rule and returns all of the non-negative integer solutions it finds in

a list. This rule stores the list of solutions in `state2` which is a child of `state1`.

(ii) The Build-Arithmetic-Results-Rule which is a non-deterministic rule, creates successors of an individual based on the solutions provided by the Collect-And-Initiate-Rule (similar to the function of the fil-Rule). Therefore, it is applied to `state2` and creates a new state for each solution. For example, if there exist n different solutions for an individual x in `state2`, this rule creates n new states as children of `state2` and in each of them expands one solution.

With backtracking: In this case the reasoner does not find all the solutions at once. In fact it assumes the first solution will end up with a clash-free graph and if this assumption fails, it will modify its knowledge about the variables and tries to search a new solution. There are two other rules responsible for this task: (i) The Collect-And-Create-Rule, similar to the Collect-And-Initiate-Rule with the lowest priority, collects all the numerical restrictions for each individual in `state1`. Furthermore, it calls the arithmetic reasoner which returns the first solution it finds and generates successors of individuals based on this solution in `state2`.

(ii) For a detailed description of the *Build-in-Sibling-Rule*, assume an individual x in `state1` that has a set of numerical restrictions according to which the Collect-And-Create-Rule has created a set of successors $y_1, y_2, \ldots, y_n$ in `state2`. We call `state1` the *generating* state of y_is. All of the rules may modify labels of the y_is in the succeeding states. If for instance y_1 ends up with a clash in all the paths of states starting from `state2`, the reasoner can conclude that y_1 is a clashed individual and therefore the corresponding solution in `state1` is not valid and cannot survive. The Build-in-Sibling-Rule is applicable to the clashed states that are closed (i.e. cannot be expanded in another way according to the Or-Rule). When this rule finds a clashed individual such as y_1 in a state, it determines its generating state which is `state1` in this case. Furthermore, it calls the arithmetic reasoner in `state1` and sets the variable

related to y_1 to zero and gets a new solution. Afterwards, this rule will generate new successors of x in a new child state of `state1` (if any solution exists).

The *Rule Expansion Strategy* component imposes an order of the rules and the rules are prioritized by their order as in the following. In other words, the logical reasoner, before applying a rule in `state1`, ensures that no rule with higher priority is applicable to it.

1. The For-All-Rule ($\forall$-Rule).
2. The For-All-Subtract-Rule ($\forall_{\setminus}$-Rule).
3. The And-Rule ($\sqcap$-Rule).
4. The Or-Rule ($\sqcup$-Rule).
5. The Build-Arithmetic-Results-Rule or the Build-in-Sibling-Rule.
6. The Collect-And-Initiate-Rule or the Collect-And-Create-Rule.

For example, assume we have x, y as two individuals and we have $\{C_1 \sqcup C_2 \sqcup C_3\} \subseteq \mathcal{L}(x)$ and $\{D_1 \sqcup D_2\} \subseteq \mathcal{L}(y)$ in `state1`. If none of the first three rules is applicable to any of the individuals in the `state1`, the Or-Rule checks if C_1, C_2, and C_3 are not in $\mathcal{L}(x)$ (or similarly if D_1 and D_2 are not in $\mathcal{L}(y)$). Therefore, the Or-Rule is applicable for the `state1` and creates 6 states as children of `state1` such that in each of them one of the C_is and one of the D_js is selected. In other words, in one application of the Or-Rule, 6 new states are created.

The structure of the For-All-Rule, the For-All-Subtract-Rule, the And-Rule, and the Or-Rule is similar to their relevant tableau rule. However, the function of the last two rules is slightly different from their corresponding tableau rule. Consider the case when we have backtracking and the logical module uses the Build-in-Sibling-Rule and the Collect-And-Create-Rule. For example, if $\{\geq 2R, \leq 4S, \geq 3T\} \subseteq \mathcal{L}(x)$ in

`state1` and x has no successors, the Collect-And-Create-Rule collects passes the list $\{\geq 2R, \leq 4S, \geq 3T\}$ to the arithmetic reasoner and receives either *"no solution"* which means that `state1` contains a clash or the first non-negative integer solution that the arithmetic reasoner finds. Assume the first solution found by the arithmetic reasoner is of the form $v_1 = 2, v_2 = 1$ such that $\alpha(v_1) = \{R, S, T\}$ and $\alpha(v_2) = \{T, S\}$ (see Figure 15).

Afterwards, the Collect-And-Create-Rule creates a new state `state2` as a child of `state1`. In `state2`, it generates two new individuals x_1 and x_2 such that x_1 is an R_1-successor of x while $R_1 \sqsubseteq R$, $R_1 \sqsubseteq S$, $R_1 \sqsubseteq T$, and $card(x_1) = 2$. Similarly, x_2 is R_2-successor of x while $R_2 \sqsubseteq T$, $R_2 \sqsubseteq S$, and $card(x_2) = 1$. Notice that all the information in `state1` will be exactly copied to `state2` before generating any new individual.

Later on, assume the individual x_1 ends up with a clash in $\texttt{state}_{i+1}$ and all other possible states (such as in $\texttt{state}_i$). Therefore, the Build-in-Sibling-Rule will be invoked for state_{i+1}. This Rule sets $v_1 = 0$ and calls the arithmetic reasoner for another solution which will be generated in another child of `state1`, `state3`. Whenever the arithmetic reasoner cannot find another solution for the list of numerical restrictions for x, the `state1` will clash and the logical reasoner must search in another branch for a closed and clash-free state which therefore contains a complete and clash-free graph. Figure 15 illustrates the function of these two rules in this example.

Another component in the logical module is the *Clash Strategy Component* which triggers a clash for an individual x whenever (i) $\{A, \neg A\} \subseteq \mathcal{L}(x)$ for a concept name A, and (ii) an arithmetic clash is detected in the arithmetic component. The logical module returns the first non-clashed and closed state it finds as a complete and clash-free graph. Otherwise it will return "unsatisfiable".

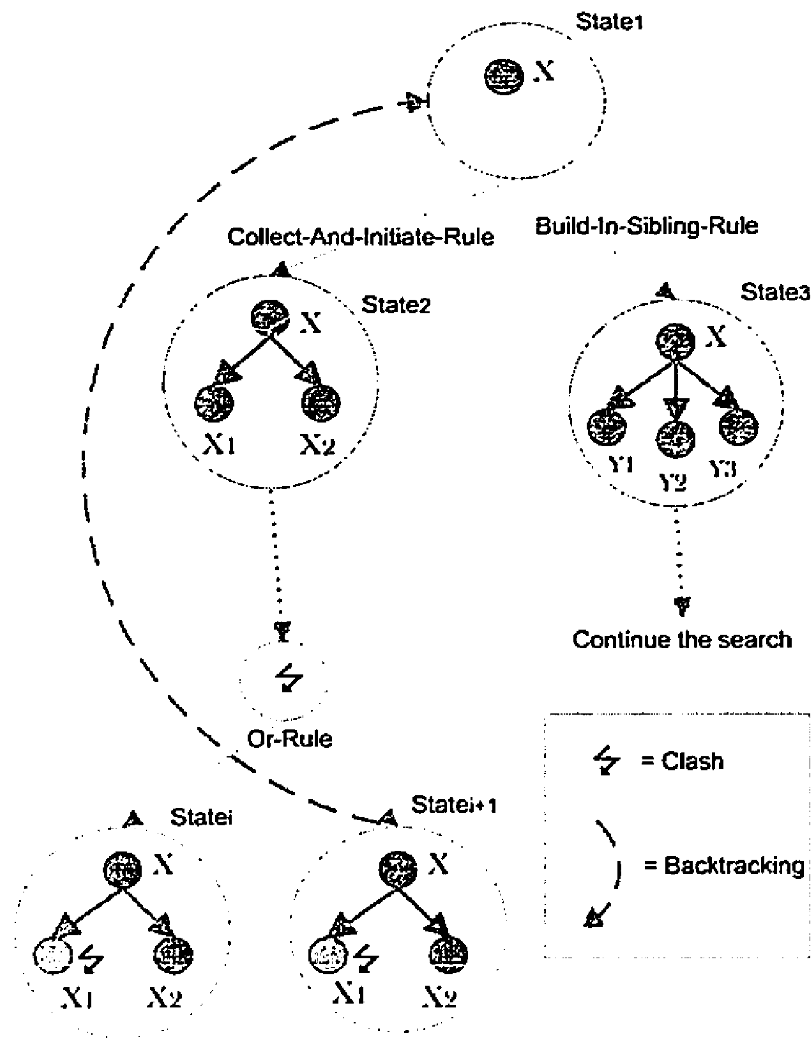

Figure 15: Illustration of the rules application when backtracking

6.3 Arithmetic Reasoner

The major function of the arithmetic reasoner is to find a non-negative integer solution for a set of unqualified number restrictions. Notice that the implemented arithmetic module is slightly different from the arithmetic reasoner proposed in Chapter 4. Firstly, in addition to an inequation solver, it performs the *ch*-Rule. Moreover, it contains a few heuristics to guide the search for a non-negative integer solution. In this section we describe the architecture of the arithmetic module which is illustrated

in Figure 16. Furthermore, we demonstrate the functionality of each component in more detail by means of pseudo code.

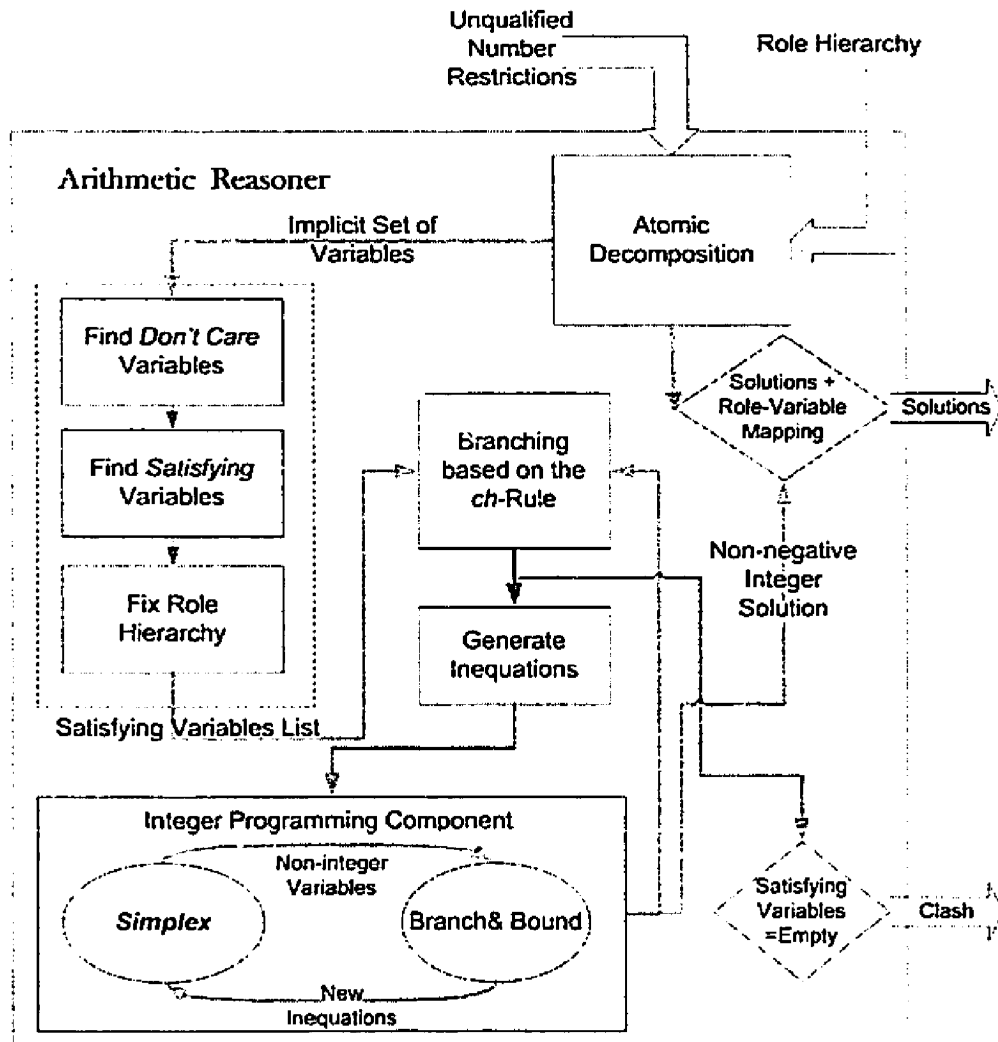

Figure 16: Architecture of the arithmetic module

6.3.1 Atomic Decomposition

Let UCR be the set of input unqualified number restriction. After reading UCR, the arithmetic module determines the number of different numerical restrictions which will later be the number of inequations. In the following we assume that the size

of UCR is equal to n. Therefore, the arithmetic module implicitly considers $2^n - 1$ variables such that for $UCR =< R_1, R_2, \ldots, R_n >$ we will have $R_i \in \alpha(v_m)$ if in the binary coding of m, the ith digit is equal to 1. For example, if $n = 4$ we can conclude that $\alpha(v_5) = \{R_1, R_3\}$ and $\alpha(v_{14}) = \{R_2, R_3, R_4\}$. To retrieve the role names related to a variable, the arithmetic module uses the `getRelatedRoles` function which has the same output as α.

6.3.2 Preprocessing

Before starting the application of the *ch*-Rule to search for an arithmetic solution, the arithmetic module classifies the variables according to the values that they can take. We define the *freedom*[3] of a variable such that:

- $freedom(v) = 2$ *iff* v must be zero due to logical reasons and cannot take any value other than zero,
- $freedom(v) = 0$ *iff* v is decided to be zero by the *ch*-Rule which can be changed later by the *ch*-Rule,
- $freedom(v) = 1$ *iff* v is decided to be greater or equal 1 by the *ch*-Rule, and
- $freedom(v) = -1$ *iff* v is a *don't care* variable and can be greater or equal zero. In other words, it can get any value except in the case that logical reasons impose a freedom of 2.

The following functions set the freedom of the variables before starting the branching:

Find *don't care* variables: We define *don't care* variables as the variables that occur in an at-least restriction but in no at-most restriction. Therefore, they are not

[3]This term should not be confused with any other well-known definition of freedom and the values assigned to *freedom* have no particular meaning.

bounded by any limitations due to the at-most restrictions and can take any value greater or equal zero. Although logical restrictions may force them to be zero, the arithmetic restrictions do not impose any restrictions on them.

Algorithm 2 Find *don't care* variables.

---

for $i = 1$ **to** n **do**
  if $UCR[i]$ is an at-least restriction **then**
    for all $j = 1$ to $2^n - 1$ such that its ith digit in binary coding $= 1$ **do**
      if (kth digit of j) $= 1$ AND $UCR[j]$ is not an at-most restriction **then**
        $freedom(v_j) = -1$
      end if
      if $freedom(v_j) \neq -1$ AND $freedom(v_j) \neq 2$ **then**
        add v_j to *satisfyingVariablesList*
      end if
    end for
  end if
end for

Find *satisfying variables*: In order to find an arithmetic solution for the input UCR list, the arithmetic module constructs a set of variables, called the *satisfying variables* on which it will apply the *ch*-Rule. In fact, these are the variables occurring in an at-least restriction which are not necessarily zero according to the logical reasons nor the *don't care* variables. The *find-don't-care* function presented in Algorithm 2 retrieves *don't care* variables and sets their freedom to -1. Moreover, whenever a variable v is neither *don't care* nor $freedom(v) = 2$, this function adds it to the *satisfying variable* list.

Remark 4. It is worth noticing that the order in which the variables are asserted in the *satisfying variables* list can significantly affect the performance of the arithmetic reasoner. In fact, by choosing the variables that are least probable to fail (either arithmetically or logically), the reasoner can speed up the procedure of searching for a complete and clash-free graph.

Therefore, it seems that the variables which lead to the individuals that are less

restricted (by the universal restrictions created by the unQ function), may be a better choice to be close to the head of the list. However, by means of the following example we will demonstrate why it is not trivial to find an optimum order of the variables.

Assume we have 4 UCRs in the input, three of which are at-least restrictions. Therefore, we will have the following general inequations:

$$\begin{aligned}
v_{0001} + v_{0011} + v_{0101} + v_{0111} + \underline{v_{1001}} + v_{1011} + v_{1101} + v_{1111} &\geq n_1 \\
v_{0010} + v_{0011} + v_{0110} + v_{0111} + \underline{v_{1010}} + v_{1011} + v_{1110} + v_{1111} &\geq n_2 \\
v_{0100} + v_{0101} + v_{0110} + v_{0111} + \underline{v_{1100}} + v_{1101} + v_{1110} + v_{1111} &\geq n_1 \\
v_{1000} + \underline{v_{1001}} + \underline{v_{1010}} + v_{1011} + \underline{v_{1100}} + v_{1101} + v_{1110} + v_{1111} &\leq m
\end{aligned}$$

In the set of variables from v_1 to v_{15}, the variables with the 1st digit (from right) equal to 1 are restricted by the universal restriction related to $UQR[1]$ (similarly for the 2nd and the 3rd restriction). Likewise, the variables with the 4th digit equal to 0 are restricted by the universal restriction related to $UQR[4]$[4].

In this example, we can conclude that the least restricted variable is v_{1000}. Nevertheless, not occurring in any at-least restriction, this variable is not even in the *satisfying variables* list. Another choice could be the case when variables have only one restriction such as v_{1001}, v_{1010} and v_{1100}. But this case is exactly similar to the standard tableau algorithms presented in Chapter 3. Although they seem to be logically less restricted, by not sharing any individuals between the at-least restrictions, they are highly probable to fail arithmetically (simply when $n_1 + n_2 + n_3 > m$).

Another strategy could be starting from variables that occur in more at-least restrictions (in this example v_{1111}) which is the case for the implemented arithmetic module. Therefore, (i) we obtain a faster arithmetic solution, (ii) we can ensure a

[4] For every at-least qualified number restriction ($\geq nR.C$), we will have $\geq nR' \sqcap \forall R'.C$. Thus, the existence of R' and therefore, appearance of 1 in its related digit will invoke the universal restriction $\forall R'.C$. However, in case of the at-most restrictions, we will replace $\leq mR.C$ by $\leq mR' \sqcap \forall R \backslash R'.(\neg C)$. Therefore, the absence of R' and accordingly, appearance of 0 in its related digit will invoke the universal restriction $\forall R \backslash R'.C$

minimum number of successors model property[5], (iii) although the probability of a logical clash may be high due to many restrictions, by choosing a highly restricted variable and detecting a clash, we can set many more variables to zero by backtracking.

Fix role hierarchy: By means of the `fix-role-hierarchy` function, the arithmetic module sets the freedom of the variables that cannot be satisfied due to the role hierarchy to 2. More precisely, if $R \sqsubseteq S$ and $R \in \alpha(v)$ but $S \notin \alpha(v)$ Algorithm 3 sets $freedom(v) = 2$.

Algorithm 3 Fix role hierarchy.

for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    if $R_i \sqsubseteq R_j$ AND $i \neq j$ **then**
      for all v such that R_i related to v and R_j not related to v **do**
        set $freedom(v) = 2$.
      end for
    end if
  end for
end for

Backtracking results: In the simple method of backtracking, the arithmetic module only needs to set the freedom of the variable related to the clashed individual to 2. In the case of complex backtracking, if the logical module discovers the fact that the existence (absence) of two or more role names in a variable may cause a clash, the arithmetic reasoner, before searching for an arithmetic solution, sets the freedom of all similar variables to 2.

Heuristics: In the case where we have no at-most restrictions or the numbers occurring in the at-most restrictions is so high that they cannot be violated by any at-least restriction, there exists a trivial solution. In this case, similar to standard tableau algorithms, we can generate successors according to the at-least restrictions and do not add any arithmetic complications. More precisely, for each at-least restriction

[5] A model has minimum number of successors property *iff* for all of its individuals we cannot have less number of successors without causing a clash.

$\geq n\ R$ we create n R-successors and we can be sure that this model will not fail.

Assuming N UCRs, M of which are at-least restrictions, the procedure presented in Algorithm 4 in fact has the same effect as the standard algorithms. For every at-least restriction $\geq n\ R$, this algorithm assigns n as the value of v for which we have $\alpha(v) = \{R\}$. It is worth noticing that in this case the algorithm violates the property of creating a model with minimal number of successors.

Algorithm 4 Heuristic 1

```
{Assume N UCRs such that UCR[1] to UCR[M] are at-least restrictions.}
for i = 1 to M do
  sumOfLimits ← sumOfLimits + UQR[i].limit
end for
MinAtMost ← ∞
for i = M to N do
  if UQR[i].limit < MinAtMost then
    MinAtMost ← UCR[i].limit
  end if
end for
if sumOfLimits ≤ MinAtMost then
  for i = 1 to M do
    value(v_{2^i}) ← UCR[i].limit
  end for
end if
```

6.3.3 Branching

After finalizing the ***satisfying variables*** list, the main function starts the application of the ch-Rule. As presented in Algorithm 5, the `branching` function starts letting the satisfying variables to have the freedom of 1 (i.e., being greater or equal 1). If there exist k variables in the ***satisfying variables*** list, in order to be complete, the algorithm must try all the 2^k cases regarding the freedom of the variables. In the case of disabled backtracking, the branching function tries all the 2^k cases and returns all the non-negative integer solutions found by the integer programming component.

However, when benefiting from backtracking, the algorithm returns to the logical module the first non-negative integer solution it finds. If the found solution logically fails, at least for one variable v, $freedom(v) = 1$ changes to $freedom(v) = 2$ which later will result in a totally different solution and the algorithm cannot compute the same solution again and falling in a cycle. If `branching` does not return any solution, the arithmetic module returns an arithmetic clash. The `branching` function in Algorithm 5 assumes the use of backtracking.

Algorithm 5 Branching over the *satisfying variables* based on the *ch*-Rule.

```
branching(satisfyingVariablesList){
if satisfyingVariablesList is empty then
  return null
else
  inequations ← build-inequations(satisfyingVariablesList)
  if result found by IntegerProgramming(inequations) then
    return result
  else
    branchingVariable ← remove the last element of satisfyingVariablesList
    freedom(branchingVariable) ← 1
    branching(satisfyingVariablesList)
    freedom(branchingVariable) ← 0
    branching(satisfyingVariablesList)
  end if
end if
}
```

6.3.4 Integer Programming

The integer programming or the equation-solver component, gets a set of linear inequations as an input. The goal function is always to minimize the sum of all the variables, while all of the variables are greater or equal zero. The set of constraints imposed by the freedom of the variables will also be part of the input in form of inequations. In other words, if $freedom(v) = 1$ for a variable, we will have $v \geq 1$ as a part of the input. Notice that in the cases where $freedom(v) = 0$ or $freedom(v) = 2$

the variable v never appears in the set of input inequations.

The integer programming component is composed of a linear programming algorithm according to *Simplex* method presented in [CLRS01] and branch-and-bound to obtain integer solutions when the linear solution contains fractional values.

6.4 Problems

This system was implemented in Java using OWL-API 2.1.1 which is a Java interface and implementation to parse the W3C Web Ontology Language OWL [HBN07]. Although choosing Java as the programming language gave us the opportunity to utilize OWL-API, the performance of the reasoner was significantly affected by the overhead due to Java features. On the other hand, no other major optimization techniques were implemented.

One of the major problems with this choice of language was the representation of float numbers. In fact, floating point numbers as a result of linear programming cannot be represented precisely. Therefore, sometimes rounding errors can result in a wrong solution. Especially when having a large number of variables, the sum of the errors may exceed 1 and may result in a wrong answer. This problem can be solved when representing fractional numbers by two integers: numerator and denominator. Unfortunately, these integers can grow very fast and use dynamic memory and also objects as float numbers can become very expensive in terms of time and memory. In fact, this error can hardly happen and basically never happened in any of the sample ontologies we used. Hence, we decided to leave this problem open for future work.

6.5 Summary

In this chapter we presented an architecture for a prototype experimental reasoner employing the hybrid algorithm. In order to overcome the high inefficiency due to the large number of variables, this implementation benefits from several optimization techniques regarding arithmetic reasoning. Moreover, in contrast with the tableau rules proposed in Chapter 4, the *ch*-Rule has been moved inside the arithmetic module. Thus, the logical module is unaware of the arithmetic reasoning and its variables.

Furthermore, we have presented the pseudo code for the major algorithms implemented in the arithmetic reasoner. Finally, we described the problems we encountered during the actual implementation of this experimental prototype.

Chapter 7

Evaluation

In this chapter we present the empirical results obtained from an implemented prototype based on the reasoner described in Chapter 6. Before presenting a set of test cases and the results, we briefly discuss the issue of benchmarking in OWL and description logics. Afterwards, we identify different parameters that may affect the complexity of reasoning with numerical restrictions. Consequently, based on these parameters we build a set of benchmarks for which we evaluate the hybrid reasoner.

7.1 Benchmarking

One major problem with benchmarking in OWL is the fact that there exist not many comprehensive real-world ontologies to utilize. In fact, as stated in [WLL+07], the current well-known benchmarks are not well suited to address typical real-world needs. On the other hand, qualified number restrictions are expressive constructs added to OWL 1.1[1] which has been recently renamed to called OWL 2 [MGH+08]. Therefore, the current well-known benchmarks do not contain qualified number restrictions. In fact, to the best of our knowledge there is no real-world benchmark available which

[1]The motivations for adding this feature are based on the requirements proposed by *Chemical Functional Groups*.

contains qualified number restrictions. Accordingly, we need to build a set of synthetic test cases to empirically evaluate the hybrid reasoner.

Since the hybrid algorithm follows the same rules to deal with the constructs other than numerical restrictions, we focus our evaluation on concept expressions only containing qualified number restrictions. In order to study the behavior of the hybrid reasoner, we need to develop a set of synthetic benchmarks. We identify the following parameters that may affect the complexity of reasoning with numerical restrictions:

1. The size of numbers occurring in the numerical restrictions. Namely, n and m in the restrictions of the form $\leq nR.C$ and $\geq mR.C$.

2. The number of qualified number restrictions.

3. The ratio of the number of at-least restrictions to the number of at-most restrictions.

4. Satisfiability versus unsatisfiability of the input concept expression.

7.2 Evaluation Results

In this section we briefly examine the performance of the hybrid reasoner with respect to the parameters identified in the previous section. Moreover, we present an evaluation to examine the effect of backtracking technique in different levels.

Tableau reasoning in expressive DLs is known to be a very time/memory-consuming procedure. Therefore, in order to remain practical, most of the reasoners benefit from numerous optimization techniques. A list of more than 70 optimization techniques which are widely used in DL reasoners is given in [Bec06].

Well-known reasoners that support qualified cardinality restrictions such as FaCT++ [TH06] or Pellet [SPG+07] implement numerous optimization techniques. Therefore, their performance is not fairly comparable to this hybrid prototype. Accordingly, we do not base our evaluations on a comparison with the existing reasoners and we try to study the behavior of the hybrid reasoner.

The following experiments are performed under Windows 32 on a standard PC with dual-core (2.10 GHz) processor and 3 GB of physical memory. To improve the precision, every test was executed in five independent runs. Furthermore, we set the timeout limit to 1000 seconds.

7.2.1 Increasing Numbers

The major advantage of benefiting from an arithmetic method is the fact that reasoning is unaffected by the size of numbers. In fact, it translates the numerical restrictions to a set of inequations. For example, for the concept expression $\geq 3 hasChild.Female$ the size of the number is three which is relatively small. However, when expressing the concept $(\geq 141 hasCredit \sqcap \leq 45\ hasCredit.ComputerScience)$ to model a university undergraduate engineering program or $(\geq 1200\ hasSeat \sqcap \leq 600\ hasSeat.(Arena \sqcup GrandCircle))$ to model the structure of a theater, larger numbers come into play.

In order to observe this major advantage which is the scalability of the hybrid algorithm with respect to the size of the numbers, we decided to compare its performance with Pellet. The reasons that we choose Pellet[2] as a representative implementation of the standard algorithm are:

- it is a free open-source reasoner that handles qualified cardinality restrictions,
- similar to our prototype it is a Java-based implementation, and

[2]We used Pellet 1.5.2, released on May 1, 2008.

- in contrast with FaCT++ which sometimes turned out to be unsound when dealing with numerical restrictions, Pellet returned correct answers in all of the experiments.

Notice that FaCT++ has no specific optimization technique for qualified cardinality restrictions. Therefore, since the goal is to compare the hybrid algorithm with the standard algorithm, we considered Pellet as a representative implementation of the standard algorithm.

Test case description:

The concept expressions for which we executed the concept satisfiability test are

$$(\geq (2i)RS.(A \sqcup B)) \sqcap (\leq iS.A) \sqcap (\leq iR.B) \sqcap (\leq (i-1)T.(\neg A)) \sqcup (\leq iT.(\neg B))$$

and

$$(\geq (2i)RS.(A \sqcup B)) \sqcap (\leq iS.A) \sqcap (\leq iR.B) \sqcap (\leq (i-1)T.(\neg A)) \sqcup (\leq (i-1)T.(\neg B))$$

with respect to a role hierarchy $\{R \sqsubseteq T, S \sqsubseteq T, RS \sqsubseteq R, RS \sqsubseteq S\}$ where i is a number incremented for each benchmark. We abbreviate the first concept expression with C_{SAT} and the second expression with C_{UNSAT}.

The concept expression C_{SAT} is a satisfiable concept where for an assertion $x : C_{SAT}$, the individual x has $(2 \times i)$ RS-successors in $(A \sqcup B)$, i of which must be in $\neg A$ and i must be in $\neg B$ (according to the at-most restrictions $(\leq iS.A)$ and $(\leq iR.B)$). Therefore, it can be concluded that i of them are in $(\neg A \sqcap B)$ and the other i successors are in $(\neg B \sqcap A)$. The disjunction $(\leq (i-1)T.(\neg A)) \sqcup (\leq iT.(\neg B))$ can be satisfied when choosing $(\leq iT.(\neg B))$ which is not violated due to the fact that x has exactly i successors in $\neg B$. According to a similar explanation, since none of

the disjuncts can be satisfied, C_{UNSAT} is an unsatisfiable concept.

In fact, C_{SAT} is not trivially satisfied neither by the hybrid algorithm nor by the standard algorithm. In the hybrid algorithm, the set of inequations is only satisfied in the case that all of the variables are zero except $v, v' \geq 1$ where $\alpha(v) = \{RS', S', T'\}$ and $\alpha(v') = \{RS', R'\}$.[3] The standard algorithm to examine the satisfiability of C_{SAT} creates $(2 \times i)$ RS-successors for x in $(A \sqcup B)$ and according to three at-most restrictions it opens 8 new branches for each successor. However, since $(2 \times i)$ is much larger than i or $i - 1$, the reasoner must start merging the extra successors when an at-most restriction is violated which happens in all the branches in this test case.

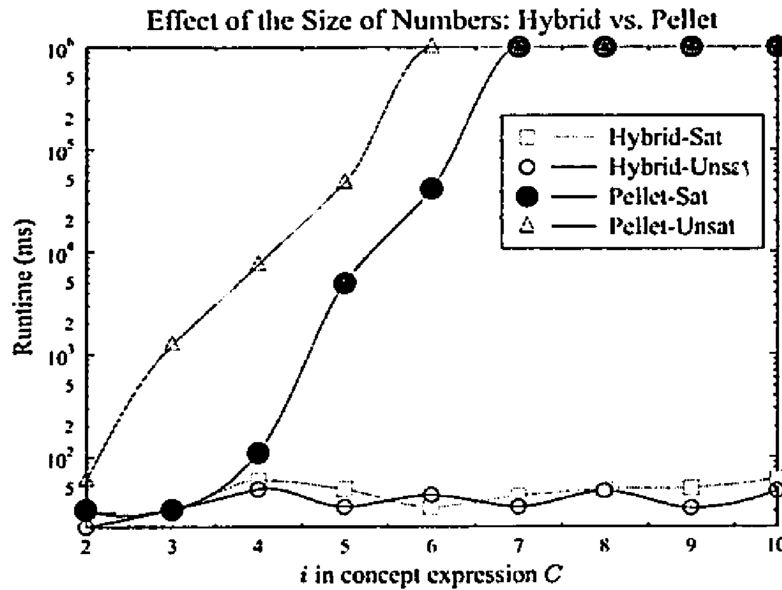

Figure 17: Comparing with standard algorithm: Effect of the value of numbers

As illustrated in Figure 17,[4] linear growth of i from 2 to 10 has almost no effect on the hybrid reasoner while it kills the standard algorithm for numbers as small as

[3]Assume R', S', RS', and T' are new sub-roles of R, S, RS, and T.

[4]Note that Figure 17 is a log-linear plot and time values are in logarithmic scale. In Figure 18 we can observe the behavior of the hybrid reasoner in a linear plot.

6 and 7. Moreover, we can observe that for $i = 6$ the satisfiability of C_{SAT} is decided in about 40s while for C_{UNSAT} which has a very slight difference this time increases up to more than 1000s. Therefore, this gap reveals the fact that by decreasing i to $i - 1$ in just one at-most restriction, which leads to unsatisfiability, the complexity of the problem increases tremendously. In Figure 18 we zoom into the hybrid part of the Figure 17 to present the behavior of the hybrid reasoner in more detail.

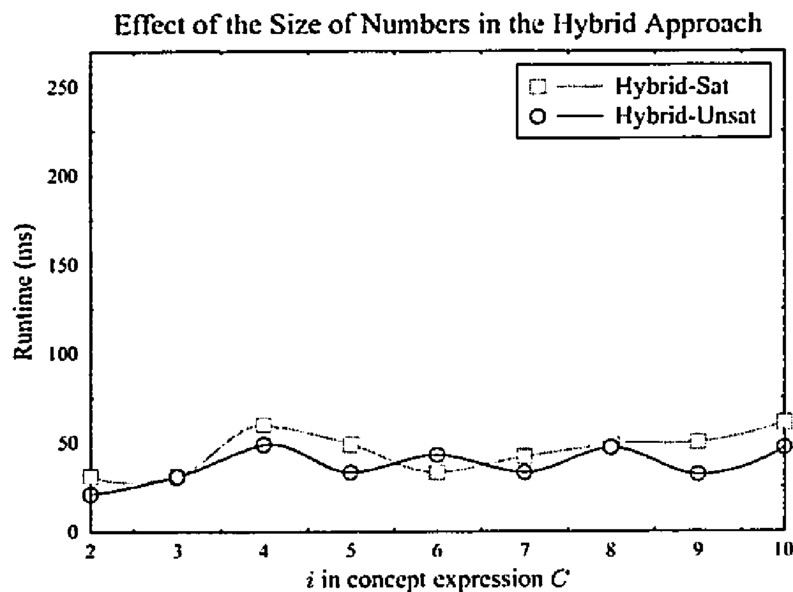

Figure 18: Behavior of the hybrid reasoner.

In contrast with the standard reasoner, the performance of the hybrid reasoner is unaffected by the value of the numbers. In Figure 19 we illustrate the linear behavior of the hybrid algorithm with respect to a linear growth of the size of the numbers in the qualified number restrictions. Furthermore, to assure that this independence will be preserved also with respect to exponential growth of i, in Figure 20 we present the performance of the hybrid reasoner for $i = 10^n, n = 1, 2, 3, 4, 5, 6$.

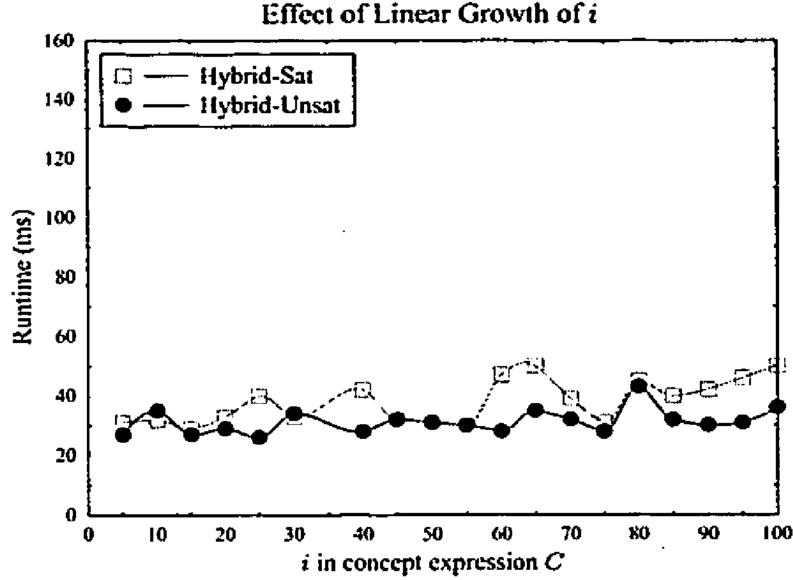

Figure 19: Behavior of the hybrid reasoner: Linear growth of i

7.2.2 Backtracking

One of the major well-known optimization techniques addressing complexity of the reasoning with numerical restrictions is dependency-directed backtracking or back-jumping. In this experiment we observe the effect of backtracking on the performance of the hybrid reasoner. In three different levels we firstly turn off the backtracking, secondly include backtracking at a simple level, and finally utilize the complex version of backtracking (see Section 5.3).

In order to better observe the impact of backtracking, we tested an unsatisfiable concept D_{UNSAT} which follows the pattern

$$(\geq 3R.D_1) \sqcap (\geq 3R.D_2) \sqcap \ldots \sqcap (\geq 3R.D_i) \sqcap (\leq (3i-1)T)$$

where $D_j \sqcap D_k = \bot$ for $1 \leq j, k \leq i, j \neq k$ with respect to a role hierarchy $R \sqsubseteq T$.

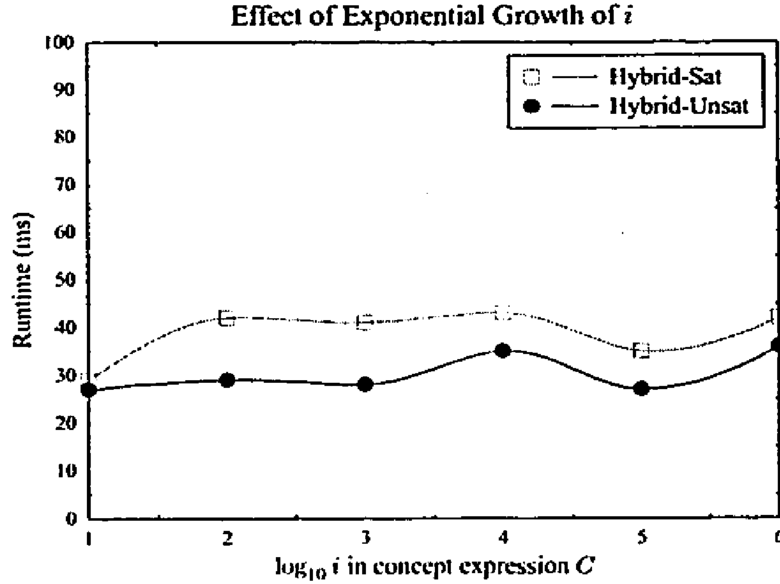

Figure 20: Behavior of the hybrid reasoner: Exponential growth of i

The assertion $x : D_{UNSAT}$ implies that x has 3 R-successors in D_1, 3 R-successors in D_2, ... and 3 R-successors in D_i. Since these $3i$ successors are instances of mutually disjoint concepts we can conclude that x has $3i$ distinct (not mergeable) successors. Therefore, the at-most restriction in D_{UNSAT} cannot be satisfied.

In this experiment in each step we increase i which will result in more numerical restrictions and therefore a larger number of variables. As the log-linear plot in Figure 21 suggests, the double-exponential nature of the hybrid algorithm and in general the high non-determinism of the ch-Rule makes it inevitable to utilize backtracking. Moreover, we can conclude that by using a more comprehensive and informed method of backtracking we can improve the performance of the reasoning significantly. For example, in Figure 21 we can observe that for $i = 6$ reasoning without backtracking results in a timeout while benefiting from simple backtracking the reasoner concludes

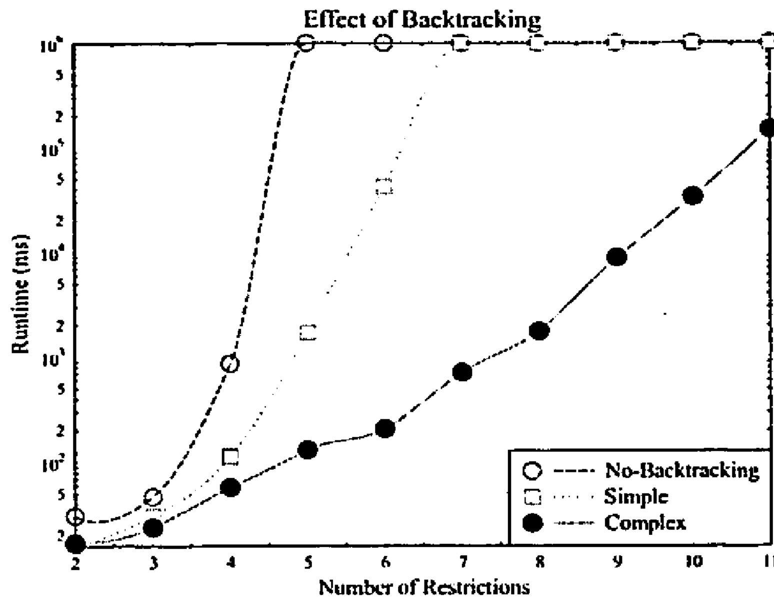

Figure 21: Effect of backtracking in different levels

unsatisfiability in about 41s and for the complex backtracking reasoning time is reduced to 206ms.

In fact, a better method of backtracking can prune a larger number of branches in the search space. In other words, the unsatisfiability of a concept can be concluded earlier after facing less number of clashes. In Figure 22, by observing the number of logical clashes each method produces before returning the result, we can compare their success in narrowing the search space.

7.2.3 Satisfiable vs. Unsatisfiable Concepts

In this experiment the test cases are concepts containing four qualified at-least restrictions and one unqualified at-most restriction according to the following pattern. We abbreviate the concept

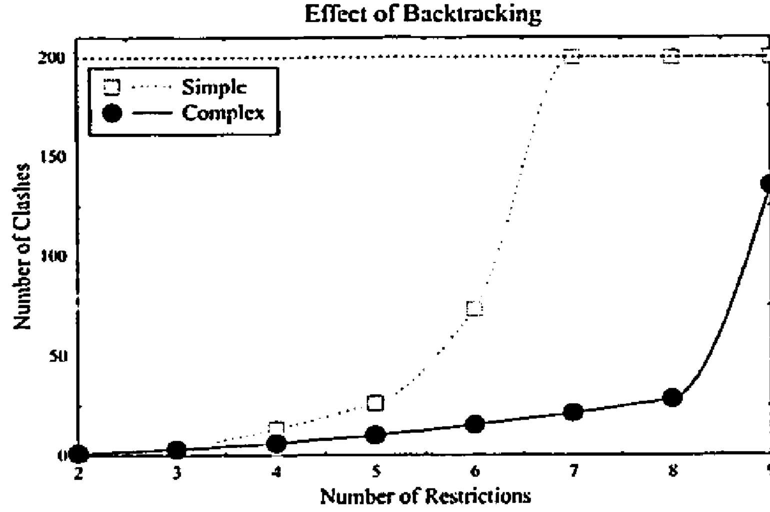

Figure 22: Effect of backtracking in different levels: Number of clashes

$$\geq 30R.(B \sqcap C) \sqcap \geq 30R.(B \sqcap \neg C) \sqcap$$
$$\geq 30R.(\neg B \sqcap C) \sqcap \geq 30R.(\neg B \sqcap \neg C) \sqcap \leq iT,$$

with E_i where $R \sqsubseteq T$ for $i = 1, 20, 40, \ldots, 220, 240$. Since the concept fillers of the four at-least restrictions are mutually disjoint, assuming the assertion $x : E_i$, we can conclude that x must have 120 nonmergeable R-successors. According to the role hierarchy $R \sqsubseteq T$, every R-successor is also a T-successor. Therefore, the concept E_i is satisfiable for $i \geq 120$ and unsatisfiable for $i < 120$.

As illustrated in Figure 23, the standard algorithm can easily infer that E_{20} is unsatisfiable since $20 < 30$ and x has at least 30 distinguished successors. However, from E_{30} to E_{120} it becomes very difficult for the standard algorithm to merge all the 120 successors in i individuals. Moreover, Figure 23 provides the fact that no matter which value i takes from 30 to 119, the standard algorithm performs similarly. In other words, we can conclude that it tries the same number of possible ways of merging which is all the possibilities to merge 4 sets of mutually distinguished

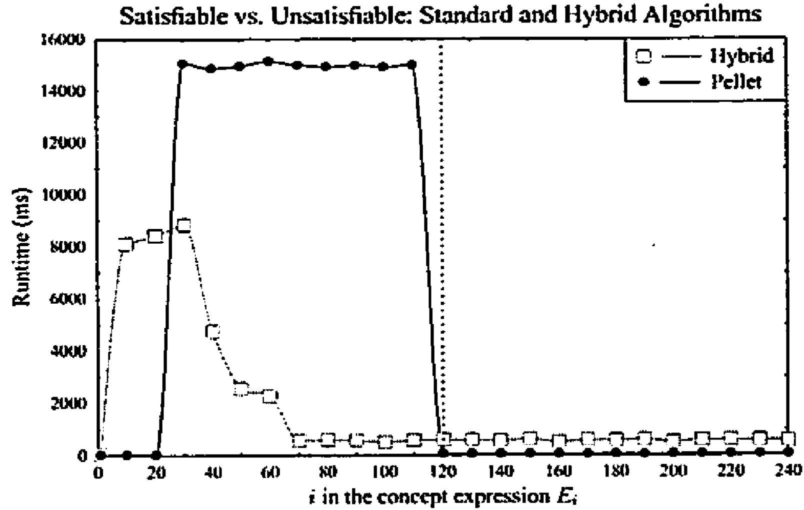

Figure 23: The effect of satisfiability

individuals. As soon as i becomes greater or equal 120, since the at-most restriction is not violated, the standard algorithm simply ignores it and reasoning becomes trivial for the standard algorithm.

Furthermore, we can conclude from Figure 23 that for the hybrid algorithm $i = 1$ is a trivial case since not more than one variable can have the freedom of $v \geq 1$ which is the case that easily leads to unsatisfiability for E_1. However, it becomes more difficult as i grows and reaches its maximum for $i = 30$ and starts to decrease gradually until $i = 120$. In fact, this unexpected behavior does not correspond to the formal analysis of the hybrid algorithm and needs to be analyzed more comprehensively and precisely.

Therefore, we extended our analysis by observing the time spent on arithmetic reasoning and logical reasoning as well as the number of different clashes. The reason that $i = 30$ is a break point is the fact that for $i < 30$ no arithmetic solution exists for the set of inequations. Therefore, it seems that for the arithmetic reasoner it is very difficult to realize the fact that a set of inequations has no solution. Moreover,

as i grows from 30 to 120, the arithmetic reasoner finds more solutions for the set of inequations which will fail due to logical clashes. In other words, the backtracking in the logical reasoner is much stronger than the arithmetic reasoner that whenever more logical clashes exist, the hybrid reasoner can accomplish the reasoning faster.

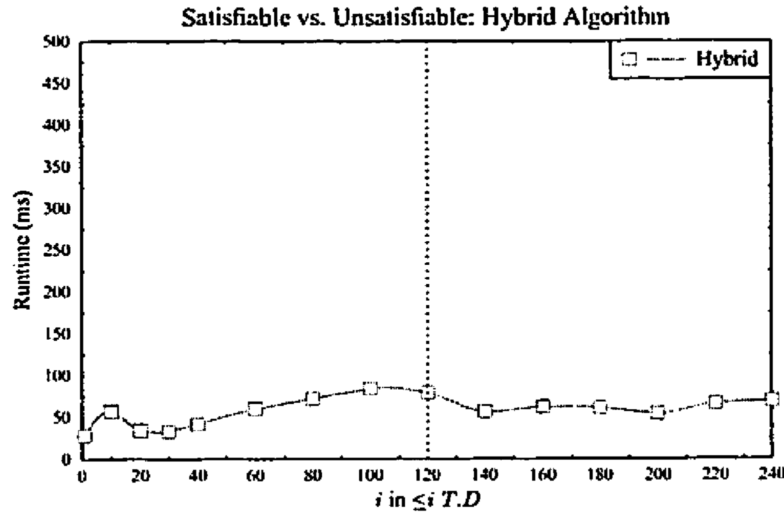

Figure 24: The effect of satisfiability: The hybrid algorithm

In order to verify this hypothesis, we built another pattern which is slightly different from E_i and we abbreviate it with F_i:

$$\geq 30R.(B \sqcap C \sqcap D) \sqcap \geq 30R.(B \sqcap \neg C \sqcap D) \sqcap$$
$$\geq 30R.(\neg B \sqcap C \sqcap D) \sqcap \geq 30R.(\neg B \sqcap \neg C \sqcap D) \sqcap \leq iT.D,$$

where $R \sqsubseteq T$ for $i = 1, 20, 40, \ldots, 220, 240$. The major difference between E_i and F_i is the fact that in F_i the at-most restriction is also a qualified restriction and concept D is added to the fillers of at-least restrictions. Therefore, the set of inequations always has an arithmetic solution, however, for $i < 120$ it will logically fail. In other words, dependency-directed backtracking discovers the unsatisfiability of the concept. Since

the clashes and therefore backtracking results are independent from the arithmetic nature of the problem, as presented in Figure 24, the performance of the hybrid reasoner stays almost constant for $1 \leq i \leq 240$. It is worth noticing, as expected due to its nature, the behavior of the standard algorithm for F_i remains exactly similar to E_i.

7.2.4 Number of Cardinality Restrictions

According to the complexity analysis of the hybrid algorithm in Section 5.1.2 one can conclude that the number of cardinality restrictions significantly influences the complexity of reasoning. More specifically, the complexity of the hybrid algorithm seems to be characterized by a double-exponential function of the number of cardinality restrictions.

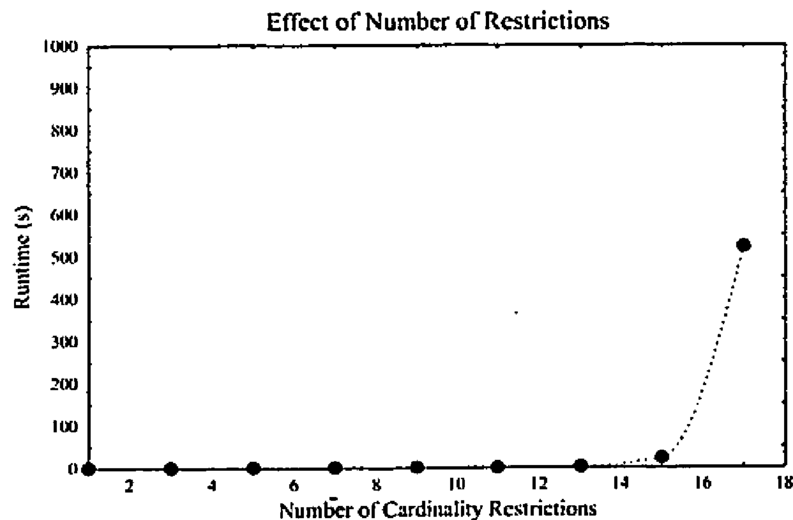

Figure 25: The effect of number of qualified number restrictions

In this experiment we build a concept containing one at-least restriction and

extend it gradually. In order to keep the ratio of the number of at-least to the at-most restrictions fixed, in each step we added one qualified at-least restriction and one qualified at-most restriction. In step i the concept which we abbreviate with G_i is of the form

$$\geq 20 RS \sqcap \geq 10 R.C_1 \sqcap \geq 10 R.C_2 \sqcap \ldots \sqcap \geq 10 R.C_i \sqcap$$
$$\leq 5 R.(\neg C_1 \sqcup \neg C_2) \sqcap \geq 5 R.(\neg C_2 \sqcup \neg C_3) \sqcap \ldots \sqcap \geq 5 R.(\neg C_i \sqcup \neg C_{i+1})$$

with respect to role hierarchy $RS \sqsubseteq R$. Therefore, in concept C_i we have $2i + 1$ cardinality restrictions. Note that the hybrid algorithm encounters no clashes when deciding satisfiability of G_i.

As presented in Figure 25, the maximum number of qualified cardinality restrictions that the hybrid prototype can handle (in less than 1000s) is 17. Notice the fact that the roles participating in these cardinality restrictions share the same role hierarchy. Otherwise, we can partition different role names from different role hierarchies and deal with each partition separately.

7.2.5 Number of At-least vs. Number of At-most

In this section we mention the ratio of the number of at-least restrictions to the number of at-most restrictions by $R_{Min/Max}$. In addition to the number of cardinality restrictions, $R_{Min/Max}$ seems to affect the complexity of reasoning. Therefore, in this experiment for a fixed total number of restrictions we evaluate the performance of the hybrid prototype with respect to this ratio. The structure of the concept expression is similar to G_i for which no clashes occur during the reasoning.

From Figure 26 we can conclude that the growth of $R_{Min/Max}$ increases the complexity of the reasoning for the hybrid reasoner. In fact, the hybrid reasoner tries to satisfy at-least restrictions while not violating any at-most restriction. Therefore, the

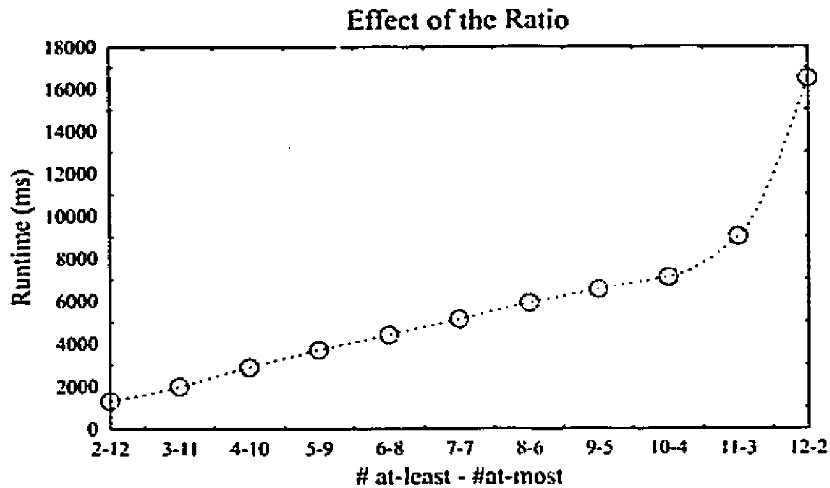

Figure 26: Ratio between number of at-least to the number of at-most restrictions

Table 2: Number of at-least restrictions - number of at-most restrictions

$\geq - \leq$	0-14	1-13	2-12	3-11	4-10	5-9	6-8	7-7
Time (s)	0.174	0.589	1.310	1.999	2.907	3.716	4.408	5.143
$\geq - \leq$	8-6	9-5	10-4	11-3	12-2	13-1	14-0	
Time (s)	5.898	6.534	7.101	9.028	16.481	53.138	235.098	

satisfying variables[5] list that is the list of variables for which the *ch*-Rule is applied, is a function of the number of at-least restrictions.[6] Therefore, the more at-least restrictions exist in G_i, the harder it becomes for the arithmetic reasoner to find a solution for the set of inequations.

Note that the at-most restrictions are not the only source of complexity. The fact that the arithmetic always searches for a minimal solution, significantly affects the complexity of the reasoning even when no at-most restriction exists. For example, in Table 2, when G_i has 14 at-least restrictions and no at-most restrictions, the hybrid

[5]See Section 6.3 for more detailed descriptions.

[6]In fact, it contains the variables participating in at least one at-least cardinality restriction.

prototype accomplished the reasoning process in about 235 seconds. Not considering the *minimal number of successors* property, this problem is trivial in absence of at-most restrictions.[7]

7.3 Conclusion

In this chapter we evaluated the implemented prototype hybrid reasoner through different experiments to study its performance with respect to different parameters of reasoning with cardinality restrictions. The major superiority of the hybrid reasoner, according to its arithmetic nature, is confirmed to be its scalability with respect to the size of the numbers appeared in the cardinality restrictions. Moreover, in case of unsatisfiable concepts or concept expressions that require a lot of merging, the hybrid algorithm performs significantly faster than the standard algorithm.

During the experiment of *satisfiability vs. unsatisfiability*, we realized the importance of an efficient arithmetic reasoner whenever no logical information can assist the process of reasoning. In other words, we need to equip the arithmetic reasoner with several effective heuristics or caching to increase its efficiency in comparison with the logical reasoner. Moreover, we observed how the *minimal number of successors* property can affect the reasoning even if no at-most restriction is violated. Therefore, we can introduce an option of turning off the *minimal number of successors* property where the reasoner simply checks if any at-most restriction is violated before continuing with arithmetic reasoning.

[7]See Section 6.3.2 where we propose a method to overcome this complexity in trivial cases.

Chapter 8

Conclusion and Future Work

Qualified cardinality restrictions extend basic DL languages with the ability of expressing numerical constraints about relationships. However, standard tableau algorithms deal with these numerical restrictions in a non-arithmetic manner. For example, they create 100 R-successors for x whenever we have the assertion $x : (\geq 100 R.C)$. Moreover, if they encounter a clash due to the violation of an at-most restriction, they try to seek a model by means of a trial and error method. Therefore, as soon as numbers become large or a kernel source of unsatisfiability exists in the problem, standard algorithms fail to terminate in a reasonable amount of time. Hence, an structured and arithmetically informed approach is needed to address this incompetency.

In this thesis we proposed such a tableau calculus based on the atomic decomposition method introduced in [OK99]. Moreover, we formally proved the correctness of the presented algorithm for $\mathcal{SHQ}$ ABox consistency. Furthermore, we analyzed the complexity of the hybrid algorithm. Since the proposed algorithm contains a high nondeterminism due to the ch-Rule, we proposed a set of optimization techniques addressing the complexity of the hybrid approach.

Afterwards, we described the architecture of a prototype reasoner, implementing

the hybrid calculus for $\mathcal{SHQ}$ concept satisfiability. Finally, we evaluated the hybrid reasoner through a set of experiments. In the following section we explain the advantages and disadvantages of the hybrid algorithm according to the evaluation results.

8.1 Discussion

Based on the evaluation results presented in Chapter 7 and the complexity analysis in Chapter 5 we identify the following advantages and disadvantages of the hybrid algorithm in comparison with the standard approaches.

8.1.1 Advantages

Insensitivity to the value of numbers: According to the nature of Linear Integer Programming, the value of numbers do not affect the hybrid algorithm. More precisely, larger numbers for the same variable only affect the cardinality of its relevant proxy individual.

Comprehensive reasoning: Since the hybrid algorithm collects all of the numerical restrictions before expanding the completion graph, its solution is more comprehensive and therefore more probable to survive. In other words, in contrast with standard algorithms it never creates extra successors which later need to be merged.

Structured search space: By means of the atomic decomposition and variables, hybrid approach searches for a model in a very structured and well-organized search space. As a result, when encountering a clash, it can efficiently backtrack to the source of the clash and optimally prune the branches which lead to the same clash (see Section 5.3).

Minimum number of successors: According to the fact that the goal function in the arithmetic reasoner is to minimize the sum of variables, the number of successors generated for an individual is always minimized. In fact, as mentioned in Section 7.2.5, one major source of inefficiency in the hybrid reasoning is that it always not only searches for a model, but also a model with minimum number of successors. This feature of the hybrid algorithm can become interesting for a set of problems where the number of successors affects the quality of the solution. For example, in configuration problems, not only a consistent and sound model is of interest, but also a model which requires less items and therefore costs less is of great importance.

8.1.2 Disadvantages

Exponential number of variables: According to the nature of the atomic decomposition, in order to have mutually disjoint sets, the hybrid algorithm introduces an exponential number of variables. Considering the non-deterministic rule, *ch*-Rule, the search for a model can become expensive for the algorithm whenever large numbers of cardinality restrictions occur in the label of an individual.

Long initialization time: The hybrid algorithm needs to perform a preprocessing step before starting the algorithm. Moreover, it always collects all of the numerical restrictions before generating any successor for an individual. In fact, this delay is due to the fact that the hybrid algorithm spends some time on choosing an efficient branch to proceed However, this initialization time is unnecessary for trivially satisfiable or unsatisfiable concepts.

Considering all the advantages and disadvantages of the hybrid algorithm, we can conclude that this approach is inevitable whenever large numbers occur in the cardinality restrictions. Moreover, it builds a well-structured search space which makes

it well-suited for non-trivial concepts. However, in the case of trivially decidable concepts, it performs slower than the standard algorithms.

In other words, we can conclude that the overhead in the hybrid algorithm is too high for trivial situations. Moreover, the fact that it minimizes the number of the successors takes an extra effort even when it is unnecessary. Therefore, it is more reasonable to use the hybrid method in cases where numbers are large and whenever the satisfiability of the input concept is not a trivial problem.

8.1.3 Comparison with Related Work

As explained in Chapter 3, the Signature calculus was also meant to address the problem of large numbers in the cardinality restrictions. However, it still has two highly nondeterministic rules (to split and merge the proxy individuals) and processes the cardinality restrictions separately. The algebraic method proposed in [OK99] cannot be considered as a calculus. It neither handles TBoxes with arbitrary axioms or terminological cycles nor directly deals with disjunctions and full negation. It is unclear how this methodology could be extended to handle more expressive description logics.

The recursive optimization proposed in [HTM01] which is implemented by Racer [HM03], examines the satisfiability of all the partitions before initializing the Simplex component. Therefore, whenever the number of qualified cardinality restrictions grows and respectively the number of partitions exponentially grows, Racer becomes very inefficient. Moreover, in the presence of ABoxes, Racer turns off the Simplex method and applies the signature calculus approach.

8.2 Future Work

We can divide the possible future works in two sets: (i) tasks which extend and optimize the current hybrid calculus on both tableau and implementation level, (ii) introducing new constructs and expressions which extend the expressiveness of current DL languages.

8.2.1 Extending and Refining the current research

Turning off minimality: Since the minimal number of successor property is unnecessary in many cases, we can put a tag for the reasoner to turn on and off this property. Therefore, whenever this tag is set off, we can consider the least restricted variables first in order to find a solution faster. For example, if no at-most restriction is violated, similar to the standard algorithm, we can create n successors for every at-least restriction $\geq nR.C$.

Optimizing the arithmetic reasoner: In Section 7.2.3 we realized the fact that one major source of inefficiency is due to non-optimized arithmetic reasoner:

- To optimize the arithmetic reasoner we can have incremental arithmetic reasoning; i.e., whenever a solution fails due to logical reasons and the arithmetic module modifies its knowledge about the variables, it does not restart the Simplex module. In fact, the Simplex module can *continue* its search for an integer solution, considering the newly discovered constraints on the variables.
- One other possibility to improve the performance of the arithmetic reasoner is caching the arithmetic solutions or clashes to avoid solving the same set of inequations several times.

- As we explained in Chapter 6, one important factor which affects the performance of the arithmetic module significantly, is the order of variables in the *satisfying-variables* list. Modifying this list according to the input concept expression and also the results gained during the backtracking, can help the arithmetic module find a surviving solution faster.

Extending to more expressive languages: There are two well-known constructors which increase the expressiveness of the language: Nominals ($\mathcal{O}$) and inverse roles ($\mathcal{I}$). In the presence of nominals, implied numerical restrictions due to the nominals affect all of the individuals. Therefore, when dealing with nominals, one has to consider these global restrictions as well as the local restrictions in the label of the nodes. Developing a hybrid algorithm for DL $\mathcal{SHOQ}$ is an ongoing research in our lab [FHM08].

In the presence of inverse roles the label of individuals may be modified at any time. Therefore, we can never assume that we have the complete set of cardinality restrictions. Therefore, a hybrid algorithm handling $\mathcal{SHIQ}$ needs to deal with incremental updates of labels of individuals due to the back propagation of knowledge.

8.2.2 Introducing New Structures

A calculus equipped with the ability of arithmetic reasoning can be a motivation for introducing more complex numerical constructors. As proposed in [OK99], one possible constraint can address restrictions on the ratio of cardinality of fillers of two different roles. For example, in the taxonomy which describes the structure of a theater, the concept expression $50 \times |\mathsf{hasRoom.WashRoom}| \geq |\mathsf{hasSeat.}\top|$ describes the restriction that there must exist at least one washroom for every 50 seats. Similarly, a percentage restriction such as $\leq 20\%\mathsf{hasCredit.Business}$ can express a concept which

at most 20% of its hasCredit successors are in the concept Business.

Since qualified number restrictions can be translated to an inequation, such an expression can be transformed to a linear inequation. However, decidability of adding such constructors needs to be investigated.

Bibliography

[Baa03] Franz Baader. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter 2, pages 306–346. Cambridge University Press, 2003.

[BBH96] F. Baader, M. Buchheit, and B. Hollunder. Cardinality restrictions on concepts. *Artificial Intelligence*, 88(1–2):195–213, 1996.

[Bec06] Sean Bechhofer. Dig optimisation techniques. http://dl.kr.org/dig/optimisations.html, March 2006. Last viewed in September 2008.

[BHN+94] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, and Enrico Franconi. An emperical analysis of optimization techniques for terminological representation systems or: "making kris get a move on". *Appl. Intell.*, 4(2):109–132, 1994.

[BS00] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:2001, 2000.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[FFHM08a] Jocelyne Faddoul, Nasim Farsinia, Volker Haarslev, and Ralf Möller. A hybrid tableau algorithm for $\mathcal{ALCQ}$. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, Patras, Greece, July 2008.

[FFHM08b] Jocelyne Faddoul, Nasim Farsinia, Volker Haarslev, and Ralf Möller. A hybrid tableau algorithm for $\mathcal{ALCQ}$. In *Proceedings of the 2008 International Workshop on Description Logics (DL-2008)*, Dresden, Germany, May 2008.

[FHM08] Jocelyne Faddoul, Volker Haarslev, and Ralf Möller. Hybrid Reasoning for Description Logics with Nominals and Qualified Number Restrictions. Technical report, Institute for Software Systems (STS), Hamburg University of Technology, Germany, 2008. See http://www.sts.tu-harburg.de/tech-reports/papers.html.

[HB91] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, KR-91*, pages 335–346, Boston (USA), 1991.

[HBN07] Matthew Horridge, Sean Bechhofer, and Olaf Noppens. Igniting the OWL 1.1 touch paper: The OWL API. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*, volume 258, Innsbruck, Austria, June 2007.

[HM01] Volker Haarslev and Ralf Möller. Optimizing reasoning in description logics with qualified number restrictions. In *Proceedings of the International Workshop on Description Logics (DL-2001)*, pages 142–151, 2001.

[HM03] Volker Haarslev and Ralf Möller. Racer: A core inference engine for the semantic web. In *EON*, 2003.

[Hor02] Ian Horrocks. Backtracking and qualified number restrictions: Some preliminary results. In *Proc. of the 2002 Description Logic Workshop (DL 2002)*, volume 63 of *CEUR*, pages 99–106, 2002.

[Hor03] I. Horrocks. Implementation and optimisation techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter 9, pages 306–346. Cambridge University Press, 2003.

[HS05] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 448–453, 2005.

[HS07] Ian Horrocks and Ulrike Sattler. A tableau decision procedure for $\mathcal{SHOIQ}$. *J. Autom. Reasoning*, 39(3):249–276, 2007.

[HST99] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 161–180. Springer, 1999.

[HST00] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David McAllester, editor, *Proc. of the 17th Int. Conf. on Automated Deduction (CADE 2000)*, volume 1831 of *Lecture Notes in Computer Science*, pages 482–496. Springer, 2000.

[HT00a] Ian Horrocks and Stephan Tobies. Optimisation of terminological reasoning. In *Proceedings of the 2000 International Workshop on Description Logics (DL2000)*, pages 183–192, Aachen, Germany, August 2000.

[HT00b] Ian Horrocks and Stephan Tobies. Reasoning with axioms: Theory and practice. In *Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 285–296, 2000.

[HTM01] Volker Haarslev, Martina Timmann, and Ralf Möller. Combining tableaux and algebraic methods for reasoning with qualified number restrictions. In *International Workshop on Methods for Modalities 2 (M4M-2)*, 2001.

[MGH+08] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, and Carsten Lutz. OWL 1.1 web ontology language: Structural specification and functional-style syntax. http://www.w3.org/TR/owl2-profiles/, October 2008. Last visited October 2008.

[OK99] Hans Jürgen Ohlbach and Jana Koehler. Modal logics, description logics and arithmetic reasoning. *Artif. Intell.*, 109(1-2):1–31, 1999.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *J. Web Sem.*, 5(2):51–53, 2007.

[SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

[TH06] Dmitry Tsarkov and Ian Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, pages 292–297, 2006.

[THPS07] Dmitry Tsarkov, Ian Horrocks, and Peter F. Patel-Schneider. Optimizing terminological reasoning for expressive description logics. *J. Autom. Reasoning*, 39(3):277–316, 2007.

[Tob01] Stephan Tobies. *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* PhD thesis, Aachen, June 2001.

[WLL+07] Timo Weithöner, Thorsten Liebig, Marko Luther, Sebastian Böhm, Friedrich W. von Henke, and Olaf Noppens. Real-world reasoning with OWL. In *Proceedings of The Semantic Web: Research and Applications, 4th European Semantic Web Conference*, pages 296–310, June 2007.

[YK07] E. Zolin Y. Kazakov, U. Sattler. How many legs do I have? Non-simple roles in number restrictions revisited. In *Proc. of the 14th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2007)*, volume 4790 of *Lecture Notes in Computer Science*. Springer, 2007.

[Zol07] Evgeny Zolin. Complexity of reasoning in description logics. `http://www.cs.man.ac.uk/~ezolin/dl/`, October 2007. Last visited October 2008.

Index

R-successor, 8

α, 34

$\mathcal{R}$, 15

card, 35

clos, 12

ABox, 6

ABox consistency test, 11

atomic decomposition, 32

blocking, 14

completion forest, 13

completion graph, 12, 37

concept inclusion axiom, 6

concept satisfiability test, 10

don't care, 70

generating rules, 12

initial complexity, 67

integer linear programming, 31

Integer Programming, 19

interpretation, 6

Linear Programming, 19

negation normal form, 11, 32

nominal, 47

practical complexity, 31

proxy individual, 35

proxy individuals, 39

Simplex, 20

tableau, 11, 53

TBox, 6

TBox consistency test, 11

unQ, 32