Practical Reasoning with Qualified Number Restrictions: A Hybrid Abox Calculus for the Description Logic $\mathcal{SHQ}$

Nasim Farsiniamarj [a] and Volker Haarslev [a,*]

[a] *Department of Computer Science and Software Engineering, Concordia University, Montreal*

This article presents a hybrid Abox tableau calculus for $\mathcal{SHQ}$ which extends the basic description logic $\mathcal{ALC}$ with role hierarchies, transitive roles, and qualified number restrictions. The prominent feature of our hybrid calculus is that it reduces reasoning about qualified number restrictions to integer linear programming. The calculus decides $\mathcal{SHQ}$ Abox consistency w.r.t. a Tbox containing general axioms. The presented approach ensures a more informed calculus which adequately handles the interaction between numerical and logical restrictions in $\mathcal{SHQ}$ concept and individual descriptions. A prototype reasoner for deciding $\mathcal{ALCHQ}$ concept satisfiability has been implemented. An empirical evaluation of our hybrid reasoner and its integrated optimization techniques for a set of synthesized benchmarks featuring qualified number restrictions clearly demonstrates the effectiveness of our hybrid calculus.

Keywords: description logics, qualified number restrictions, integer linear programming

1. Introduction

Description Logics (DLs) [1] are formal languages to represent the knowledge about concepts, individuals, and their relationships (called roles). In the domain of the semantic web, the Web Ontology Language (OWL) [35] is based on description logics. The DL subset of OWL, called OWL-DL, is based on the DL $\mathcal{SHOIN}$ which is a subset of $\mathcal{SHOIQ}$ [21]. Developing techniques for optimized reasoning with qualified number restrictions has become an important goal because qualified number restrictions have been added to the forthcoming OWL 2 [31] which is based on the DL $\mathcal{SROIQ}$ [20]. In this investigation we focus our attention on the DL $\mathcal{SHQ}$, which is a subset of OWL 2 and complete enough to cover the characteristics of qualified number restrictions.

Using $\mathcal{SHQ}$ one can express number restrictions on the role fillers[1] of individuals. For instance, the expression $\forall \mathit{hasCredit}.(\mathit{Science} \sqcup \mathit{Engineering} \sqcup \mathit{Business})$ about engineering undergraduate students states that credits must be taken only from science, engineering, or business. Moreover, $(\geq 140\, \mathit{hasCredit})$ states that every student must have at least 140 credits, $(\geq 120\, \mathit{hasCredit}.(\mathit{Science} \sqcup \mathit{Engineering}))$ states that at least 120 credits must be from science or engineering, $(\leq 32\, \mathit{hasCredit}.(\mathit{Science} \sqcup \mathit{Business}))$ allows at most 32 credits from science or business, and $(\leq 91\, \mathit{hasCredit}.\mathit{Engineering})$ restricts the credits from engineering to at most 91. A typical question for a DL reasoner would be whether the conjunction of the concept expressions given above is satisfiable.

For example, deciding the satisfiability of such a $\mathcal{SHQ}$ concept consists of finding a model for a student with a set of at least 140 *hasCredit*-fillers that are only instances of the concept $(\mathit{Science} \sqcup \mathit{Engineering} \sqcup \mathit{Business})$, at least 120 *hasCredit*-fillers are instances of $(\mathit{Science} \sqcup \mathit{Engineering})$, at most 32 *hasCredit*-fillers are instances of $(\mathit{Science} \sqcup \mathit{Business})$, and at most 91 *hasCredit*-fillers are instances of *Engineering*. Most DL tableau algorithms [16,3,22] test the satisfiability of such a concept by first satisfying all at-least restrictions,

[*] Corresponding author: Volker Haarslev, Department of Computer Science and Software Engineering, Concordia University, 1455 de Maisonneuve Blvd. W., Montreal, Quebec H3G 1M8, Canada, Email: haarslev@cse.concordia.ca

[1] Informally speaking, if an individual a is related to an individual b via a role R, b is called a R role filler (or R-filler) of a (see Section 2 for more details).

e.g., by creating 260 *hasCredit*-fillers, of which 120 are instances of (*Science* $\sqcup$ *Engineering*). Eventually, a nondeterministic choose-rule assigns to each of these 260 individuals (*Science* $\sqcup$ *Business*) or $\neg$(*Science* $\sqcup$ *Business*), and *Engineering* or $\neg$*Engineering*. In case an at-most restriction is violated, e.g., a student has more than 91 *hasCredit*-fillers of *Engineering*, a nondeterministic merge-rule tries to reduce the number of these individuals by merging a pair of individuals until the upper bound specified in this at-most restriction is satisfied. Searching for a model in such an arithmetically uninformed or blind way is usually very inefficient.

An obvious question might be raised whether bigger numbers will be used in qualified number restrictions occurring in OWL 2 ontologies. First, this seems to be a chicken-and-egg problem because in the presence of inefficient reasoning techniques for cases with bigger numbers in qualified number restrictions ontology designers will most likely avoid the use of these constructs. So, an empirical analysis of existing ontologies might be misleading about the necessity of efficient reasoning techniques for these cases. Second, we argue that the use of bigger numbers in qualified number restrictions is very natural in many domains (e.g., in the example above) and conceptual restrictions for role fillers are essential. This is also motivated by examples from medical domains, for instance the human anatomy distinguishes hundreds of different kinds of bones as part of the human skeleton [28]. Our third argument is about the difficulty of conjunctions of interdependent qualified number restrictions, where the calculus proposed in this article is better informed and far superior due to the use of linear integer programming because such a conjunction is represented as a system of linear inequations capturing all numerical interdependencies (see below for more details).

Our hybrid calculus (see also [11]) is based on a standard tableau algorithm for $\mathcal{SH}$ [3] modified and extended to deal with qualified number restrictions and works with an inequation solver based on integer linear programming. The algorithm encodes number restrictions into a set of inequations using the so-called atomic decomposition technique [30]. The set of inequations is processed by the inequation solver which finds, if possible, a minimal non-negative integer solution (distribution of role fillers constrained by number restrictions) satisfying the inequations. The algorithm ensures that such a distribution of role fillers also satisfies the logical restrictions.

Since this hybrid algorithm collects all the information about arithmetic expressions before creating any role filler, it will not satisfy any at-least restriction by violating an at-most restriction and there is no need for a mechanism of merging role fillers. Moreover, it reasons about number restrictions by means of an inequation solver, thus its performance is not affected by the values of numbers occurring in number restrictions. Since the solution from the inequation solver satisfies all numerical restrictions imposed by at-least and at-most restrictions, our calculus needs to create only one so-called *proxy individual* (inspired by [13]) representing a set of role fillers. Considering all these features the proposed hybrid algorithm is well suited to improve average case performance. Furthermore, in [7, Chapter 6] it has been shown that a tableau procedure extended by global caching and an algebraic method similar to the one presented in [30,15] and in this article is worst-case optimal for $\mathcal{SHIQ}$. Although the calculus presented in our article is different from the one in [7] we conjecture that the result presented in [7] can be transferred to our work (if extended by global caching) because the use of integer linear programming was essential to proving worst-case optimality.

This calculus extends our work presented in [8] by (i) covering Abox consistency for $\mathcal{SHQ}$ with general Tboxes, and (ii) introducing the use of proxy individuals representing sets of role fillers. It complements the work in [9,10] (which extends the work in [8] to $\mathcal{ALCOQ}$) by additionally dealing with role hierarchies and transitive roles and also giving evidence that (i) the absence of nominals can lead to a more efficient calculus because the atomic decomposition can be applied locally on the fly, which significantly reduces the number of partitions to be considered, and (ii) restricted forms of nominals can be viewed as Abox individuals allowing a more efficient treatment. This article also discusses practical aspects of using the hybrid method and presents evaluation results based on an implemented prototype. It also significantly differs from the work presented in [29,30] where no formal calculus was proposed and the covered logic did not support Abox consistency for SHQ with general Tboxes. Our early work presented in

[15] did not deal with Aboxes and was based on a recursive calculus without using proxy individuals and did not provide any proof for soundness, completeness, or termination. This article also extends the work on the signature calculus presented in [13] because the signature calculus, although using proxy individuals to represent sets of identical individuals, first satisfies all relevant at-least restrictions and then tries to satisfy violated at-most restrictions by merging affected signatures.

The remainder of this article is structured as follows. After briefly introducing the DL $\mathcal{SHQ}$ and some preprocessing steps in the next section, we present the hybrid Abox calculus in Section 3, illustrate the rules with two examples, and conclude this section with formal proofs. Section 4 starts the second major part of this article, practical reasoning. We analyze the complexity of the standard and hybrid algorithm for dealing with number restrictions and present optimization techniques for the hybrid algorithm. In Section 5 the architecture of the implemented prototype reasoner is described. A detailed evaluation of the prototype is presented in Section 6. We conclude this article with a discussion and outline future work.

2. Description Logic $\mathcal{SHQ}$

In the following, three disjoint sets are defined; N_C is the set of concepts 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 $I_{\mathcal{A}} \subseteq I$ is the set of individuals occurring in an Abox $\mathcal{A}$. We use a standard Tarski-style semantics based on an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the non-empty domain and $\cdot^{\mathcal{I}}$ the interpretation function. The interpretation $\mathcal{I}$ gives meaning to the atomic constructs of $\mathcal{SHQ}$ and is extended to complex constructs as shown in Table 1, where we assume that C, D are arbitrary concept descriptions, $R, S \in N_R$, $a, b \in I_{\mathcal{A}}$, $n, m \in \mathbb{N}$, and $R^{\#}(x, C)$ denotes the cardinality of $\{x \mid (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$. A concept description C is said to be satisfiable by an interpretation $\mathcal{I}$ iff $C^{\mathcal{I}} \neq \emptyset$. We use $\top$ ($\bot$) as abbreviations for $A \sqcup \neg A$ ($A \sqcap \neg A$) for some $A \in N_C$.

A role hierarchy $\mathcal{R}$ is a set of axioms of the form $R \sqsubseteq S$ where $R, S \in N_R$. In the following, let $\sqsubseteq_*$ be the transitive, reflexive closure of $\sqsubseteq$ over N_R. For $R \sqsubseteq_* S$, R is called a sub-role of S and S a super-role of R. 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 with the exception of $\geq 1\, R.C$. However, recent investigations in [26] show that this condition could be relaxed in the absence of inverse roles. Moreover, $\leq 0\, R.C$ is obviously equivalent to $\forall R.\neg C$.

Table 1
Syntax and semantics of $\mathcal{SHQ}$.

Syntax	Semantics
Concepts	
A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, A is a concept name
$\neg C$	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\forall R.C$	$\{x \mid \forall y : (x, y) \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$
$\geq n\, R.C$	$\{x \mid \#R^{\mathcal{I}}(x, C) \geq n\}$
$\leq m\, R.C$	$\{x \mid \#R^{\mathcal{I}}(x, C) \leq m\}$
Roles	
$R \in N_R$	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
$R \in N_{RT}$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
Axioms	
$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Assertions	
$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
$(a, b) : R$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
$a \neq b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

Definition 1 (Tbox) A $\mathcal{SHQ}$ Tbox $\mathcal{T}$ w.r.t. a role hierarchy $\mathcal{R}$ is a finite set of axioms of the form $C \sqsubseteq D$ or $C \equiv D$ where C, D are concept expressions and $C \equiv D$ is the placeholder for $\{C \sqsubseteq D, D \sqsubseteq C\}$. A Tbox $\mathcal{T}$ is satisfiable by an interpretation $\mathcal{I}$ iff $\mathcal{I}$ satisfies all axioms in $\mathcal{T}$ and $\mathcal{R}$.

Definition 2 (Abox) A $\mathcal{SHQ}$ Abox $\mathcal{A}$ w.r.t. a Tbox $\mathcal{T}$ and a role hierarchy $\mathcal{R}$ is a finite set of assertions of the form $a : C$, $(a, b) : R$, and $a \neq b$. An Abox $\mathcal{A}$ is satisfiable by an interpretation $\mathcal{I}$ (or consistent) iff $\mathcal{I}$ satisfies $\mathcal{T}$ and $\mathcal{R}$ and all assertions in $\mathcal{A}$.

Let $a, b \in I$ be (possibly unnamed) individuals, given an assertion $(a, b) : R$, a is called a predecessor of b and b a successor or role filler of a. The R-fillers of a are defined as $\mathit{Fil}(a, R) = \{b \mid (a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}\}$.

Inspired by [30] we transform given qualified number restrictions to unqualified number restrictions with $(\geq n\, R)^{\mathcal{I}} = (\geq n\, R.\top)^{\mathcal{I}}$, $(\leq n\, R)^{\mathcal{I}} = (\leq n\, R.\top)^{\mathcal{I}}$, and add a new role-set difference operator $\forall(R\backslash R').C$ such that $(\forall(R\backslash R').C)^{\mathcal{I}} = \{x \mid \forall y\colon (x,y) \in (R^{\mathcal{I}} \backslash R'^{\mathcal{I}}) \Rightarrow y \in C^{\mathcal{I}}\}$. The resulting language is called $\mathcal{SHN}^{\backslash}$. Let $\dot{\neg}C$ denote the standard negation normal form (NNF)[2] of $\neg C$. We define a recursive function *unQ* which rewrites $\mathcal{SHQ}$ assertions or concept descriptions into $\mathcal{SHN}^{\backslash}$. It is important to note that this rewriting process always introduces for each transformed qualified number restriction a unique new role.

Definition 3 (*unQ*) This function transforms the input description into its NNF and replaces qualified number restrictions (if $C \neq \top$). 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
$unQ(\forall R.C) := \forall R.unQ(C)$
$unQ(C \sqcap D) := unQ(C) \sqcap unQ(D)$
$unQ(C \sqcup D) := unQ(C) \sqcup unQ(D)$
$unQ(\geq nR.C) := (\geq nR') \sqcap \forall R'.unQ(C)$ (1)
$unQ(\leq nR.C) := (\leq nR') \sqcap \forall(R\backslash R').unQ(\dot{\neg}C)$ (2)
$unQ(a\colon C) := a\colon unQ(C)$
$unQ((a,b)\colon R) := (a,b)\colon R$
$unQ(a \neq b) := a \neq b$

Remark According to [30] one can replace a qualified number restriction of the form $(\geq n\, R.C)$ by $(\exists R' : (R' \sqsubseteq R) \in \mathcal{R} \wedge \geq n\, R' \sqcap \forall R'.C)$ and $\leq n\, R.C$ by $\exists R'$ such that $(R' \sqsubseteq R) \in \mathcal{R}$, $\leq n\, R' \sqcap \forall R'.C \sqcap \forall R\backslash R'.(\neg C)$ ((1) and (2)). Therefore, $\neg(\geq n\, R.C)$ (which is equal to $\leq (n-1)\, R.C$) is equivalent to $(\forall R' \sqsubseteq R\colon \leq (n-1)\, R' \sqcup \exists R'.\neg C)$ which is not a formula expressible 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, the negated description needs to be converted to NNF before further applying *unQ*. 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.

[2]The negation normal form of $\neg(\geq n\, R.C)$ ($\neg(\leq n\, R.C)$) is defined as $\leq (n-1)\, R.C$ ($\geq (n+1)\, R.C$) respectively.

Since (2) is slightly different from what is proposed in [30], we prove this equivalence based on the semantics of the interpretation function $\mathcal{I}$.

Proposition 4 $(\leq n\, R.C)$ is equisatisfiable with the expression $(\forall(R\backslash R').\neg C \sqcap \leq n\, R')$ with $\exists R' : R' \sqsubseteq R$.

Proof. The hypothesis can be translated to:
$(\leq\, nR.C)^{\mathcal{I}} = \{a \mid \#\{y \mid \langle a,y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\} \Leftrightarrow$
$\exists \mathcal{I}' : \{a \mid \exists R' : R'^{\mathcal{I}'} \subseteq R^{\mathcal{I}'} \wedge \#\{y \mid \langle a,y\rangle \in R'^{\mathcal{I}'}\} \leq n \wedge \forall b : \langle a,b\rangle \in R^{\mathcal{I}'} \wedge \langle a,b\rangle \notin R'^{\mathcal{I}'} \Rightarrow b \in \Delta^{\mathcal{I}'} \backslash C^{\mathcal{I}'}\}$.
($\Leftarrow$): If $a \in \Delta^{\mathcal{I}'}$, $\langle a,y\rangle \in R^{\mathcal{I}'}$, $y \in C^{\mathcal{I}'}$, we can conclude that $\langle a,y\rangle \in R'^{\mathcal{I}'}$ (because if $\langle a,y\rangle \notin R'^{\mathcal{I}'}$ then $y \in \Delta^{\mathcal{I}'} \backslash C^{\mathcal{I}'}$). 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 $\langle a,b\rangle \in R''^{\mathcal{I}'}$. □

3. A Hybrid Abox Tableau Calculus for $\mathcal{SHQ}$

Extending $\mathcal{ALC}$ with qualified number restrictions provides the ability to express arithmetic restrictions on role fillers. This expressiveness can increase the practical complexity of the reasoning when employing arithmetically uninformed algorithms (e.g., see [16,3,22]). Therefore, a tableau calculus which benefits from arithmetic methods can improve the average case performance of reasoning about qualified cardinality restrictions. In this section we propose a hybrid tableau algorithm which benefits from integer linear programming to properly handle numerical features of the language. We present a tableau algorithm that accepts a general Abox $\mathcal{A}$ w.r.t. a general Tbox $\mathcal{T}$ and a role hierarchy $\mathcal{R}$ as input and either returns "inconsistent" if $\mathcal{A}$ is not consistent or otherwise "consistent" with a complete and clash-free completion forest.

For reasons of simplicity we assume in this article that standard Tbox inference services w.r.t. a Tbox $\mathcal{T}$ and a role hierarchy $\mathcal{R}$, e.g., concept satisfiability and subsumption testing, are reduced to Abox satisfiability in the usual way. For instance, a concept C is satisfiable iff the Abox $\{a\colon C\}$ is satisfiable and a concept C is subsumed by a concept D iff the Abox $\{a\colon \neg D \sqcap C\}$ is unsatisfiable.

3.1. Preprocessing

As a preprocessing step, the input Abox and Tbox are transformed into $\mathcal{SHN}^{\backslash}$ as defined in Section 2. Similar to [24], in order to propagate Tbox axioms to all individuals we define the concept $C_{\mathcal{T}} := \bigsqcap_{C \sqsubseteq D \in \mathcal{T}} unQ(\dot{\neg} C \sqcup D)$ and U as a new transitive role in the role hierarchy $\mathcal{R}$ extended by $\{R \sqsubseteq U \mid R \in N_R\}$. In order to examine the consistency of $\mathcal{A}$, the algorithm first extends $\mathcal{A}$ by $\{x_0 \colon (C_{\mathcal{T}} \sqcap \forall U.C_{\mathcal{T}})\} \cup \{(x_0, x) \colon U \mid x \text{ occurs in } \mathcal{A}\}$, where x_0 is new in $\mathcal{A}$. By this transformation, we impose the axioms in the Tbox on all individuals in $\mathcal{A}$. Furthermore, we rewrite the role assertions in $\mathcal{A}$ as follows.

Definition 5 If $x, y \in I_{\mathcal{A}}$ then there exists a unique role name $R_{xy} \in N_R$ only used to represent that y is an R-filler of x with $R_{xy} \sqsubseteq R \in \mathcal{R}$. In other words whenever $(x, z) : R_{xy}$ we have $y^{\mathcal{I}} = z^{\mathcal{I}}$.

We replace role assertions of the form $(b, c) : R$ by $b \colon (\geq 1\, R_{bc} \sqcap \leq 1\, R_{bc})$. The reason for translating Abox role assertions into number restrictions is due to the fact that they actually impose a numerical restriction on an individual. For example, an assertion $(b, c) \colon R$ implies there exists one and only one R_{bc}-filler for b which is c. Since the hybrid algorithm needs to consider *all* number restrictions before creating an arithmetic solution and generating successors, it is necessary to consider these restrictions as well.

3.2. Atomic Decomposition of Role Fillers

Atomic decomposition is a technique first proposed in [30] for reasoning about sets. Later it was applied for concept languages such as in description logics 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 ($|\cdot|$ denotes set cardinality). For example, assume we want to translate the following number restrictions occurring in an Abox $\mathcal{A}$ into arithmetic inequations: $\mathcal{A} = \{a : (\leq 3\, hasDaughter \sqcap \leq 4\, hasSon \sqcap \geq 5\, hasChild)\}$ (adapted from [30]).

Different partitions for this set of restrictions are defined as:

c	=	children, not sons, not daughters.
s	=	sons, not children, not daughters.
d	=	daughters, not children, not sons.
cs	=	children, sons, not daughters.
cd	=	children, daughters, not sons.
sd	=	sons, daughters, not children.
csd	=	children, sons, daughters.

Since it is an atomic decomposition and subsets are mutually disjoint, we can translate the three number restrictions for the individual a 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 first solution for the initial set of number restrictions. As there may exist more than one solution for this system, there may be a nondeterministic approach needed to handle it.

Due to the nature of $\mathcal{SHN}^{\backslash}$, a quasi tree-model property for concepts[3] [36] ensures that different individuals in a model do not share common role fillers and role fillers (as part of a concept model) cannot affect their predecessors due to the absence of inverse roles. The only exception are individuals from $I_{\mathcal{A}}$ occurring in an Abox $\mathcal{A}$. Thus, in the following we introduce the necessary notation to define the atomic decomposition of role fillers locally[4] for individuals, i.e., it is based for each individual on the roles of its role fillers.

3.3. Completion Forest

In the following we introduce the notion of a completion forest and some other notation to specify the completion rules that form the major part of our tableau algorithm.

[3]The tree model states that a concept C has a model (an interpretation $\mathcal{I}$ with $C^{\mathcal{I}} \neq \emptyset$) iff C has a tree-shaped model, i.e., one in which the interpretation of role filler relations defines a tree shaped directed graph. Transitive roles compromise the tree model property to some extent because they can cause "short-cuts" down the branches of a tree. This relaxed version is called quasi tree-model property.

[4]For instance, the DL $\mathcal{ALCOQ}$, which supports so-called nominals, requires a global decomposition (see [9,10] for more details).

Definition 6 (Closure) The closure $clos(E)$ of a concept expression 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 $D \in clos(E)$, $(\forall R.C) \in clos(E) \Rightarrow C \in clos(E)$, $(\geq n\, R.C) \in clos(E)$ or $(\leq m\, R.C) \in clos(E) \Rightarrow C \in clos(E)$.

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 $clos(\mathcal{A})$ such that if $(a : C) \in \mathcal{A}$ then $clos(C) \subseteq clos(\mathcal{A})$.

Definition 7 (Completion Forest) A completion forest $\mathcal{F} = (V, E, \mathcal{L}, \mathcal{L}_E)$ for a $\mathcal{SHQ}$ Abox $\mathcal{A}$ is composed of a set of arbitrarily connected nodes as the roots[5] of completion trees (if one ignores the connections between root nodes). 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 $(\sum_{i \in 1..k} v_i) \bowtie n$ with $\bowtie \in \{\leq, \geq\}$, $n \in \mathbb{N}$, and $v_i \in \mathcal{V}$ where $\mathcal{V}$ is a set of variables (see Definition 10 below); each edge $\langle x, y\rangle \in E$ is labeled by the set $\mathcal{L}(\langle x, y\rangle) \subseteq N_R$, and x (y) is called a predecessor (successor) of y (x), and the transitively closed set of successors (predecessors) is called descendants (ancestors) respectively. We maintain the distinction between nodes of a forest by the relation $\dot{\neq}$. In the following we also refer to elements of $\mathcal{L}$ and $\mathcal{L}_E$ as *constraints* of a node. Notice that our version of completion forests is slightly different from standard completion graphs (e.g., see [24]).

Definition 8 (Node-related Partitions) The set R_x of *related roles* for a node x is defined as $R_x = \{S \mid \{\leq n\, S, \geq m\, S\} \cap \mathcal{L}(x) \neq \emptyset\}$. We define a *partitioning* $\mathcal{RS}_x = (\bigcup_{RS \subseteq R_x} \{RS\}) \setminus \{\emptyset\}$ and for $RS_x \in \mathcal{RS}_x$ we define $RS_x^{\mathcal{I}} = (\bigcap_{S \in RS_x} Fil^{\mathcal{I}}(x, S)) \setminus (\bigcup_{S \in R_x \setminus RS_x} Fil^{\mathcal{I}}(x, S))$ with $Fil^{\mathcal{I}}(x, S) = \{y^{\mathcal{I}} \mid y \in Fil(x, S)\}$. $RS_x^{\mathcal{I}}$ represents the interpretation of fillers of x that are fillers for the roles in RS but not fillers for the roles in $R_x \setminus RS$. Therefore, by definition the fillers of x associated with the partitions in $\mathcal{RS}_x$ are mutually disjoint w.r.t. the interpretation $\mathcal{I}$.

Definition 9 (Partition Variables) We assume a set $\mathcal{V}$ of variables and by using a mapping $\alpha : \mathcal{V} \leftrightarrow \mathcal{RS}_x$ we associate a unique variable $v \in \mathcal{V}$ with each partition RS_x in $\mathcal{RS}_x$ such that $\alpha(v) = RS_x$.

[5]In contrast to the standard notion of root nodes of a tree we allow for root nodes incoming edges from other root nodes.

Definition 10 (Inequations) Let $\mathcal{V}^R$ be defined as the set of all variables related to a role R such that $\mathcal{V}^R = \{v \in \mathcal{V} \mid R \in \alpha(v)\}$. A function ξ is used to map number restrictions to inequations of the form $\xi(R, \bowtie, n) := (\sum_{v_i \in \mathcal{V}^R} v_i) \bowtie n$ where $\bowtie \in \{\leq, \geq\}$ and $n \in \mathbb{N}$.

Since all concept restrictions for node successors that are not root nodes are propagated through universal restrictions for roles and a new role is created for each role filler occurring in a role assertion, we can conclude that all nodes in a certain node partition share the same restrictions and can be dealt with as a unit. We call this unit *proxy node* which is a representative of possibly more than one node.

Definition 11 (Node Cardinality) The cardinality associated with proxy nodes is defined by the mapping $card : V \rightarrow \mathbb{N}$.

Definition 12 (Blocked Node) A node x in a forest F is blocked by a node y iff x, y are not root nodes and y is a ancestor of x such that $\mathcal{L}(x) \subseteq \mathcal{L}(y)$.

Definition 13 (Clash Triggers) A node x contains a clash iff there exists a concept name $A \in N_C$ such that $\{A, \neg A\} \subseteq \mathcal{L}(x)$ (logical clash) or $\mathcal{L}_E(x)$ does not have a non-negative integer solution (arithmetic clash).

Definition 14 (Arithmetic Solution) An arithmetic solution is represented using the function $\sigma : \mathcal{V} \rightarrow \mathbb{N}$ which assigns a non-negative integer value to each variable in $\mathcal{V}$. Let $\mathcal{V}_x$ be the set of variables assigned to a node x, $\mathcal{V}_x = \{v \in \mathcal{V} \mid v \text{ occurs in } \mathcal{L}_E(x)\}$. We define a set of solutions Ω for x as $\Omega(x) := \{\sigma(v){=}n \mid n \in \mathbb{N}, v \in \mathcal{V}_x\}$. Notice that the goal (or objective) function of the inequation solver is to minimize the sum of the variables occurring in the input inequations.

The algorithm starts with the forest $\mathcal{F}_{\mathcal{A}}$ with the individuals mentioned in $\mathcal{A}$ as root nodes. For each $a \in I_{\mathcal{A}}$ a root node x_a will be created with $\mathcal{L}(x_a) = \{C \mid (a : C) \in \mathcal{A}\}$. Additionally, for every root node x we set $card(x) = 1$.

We illustrate these definitions with a simple example shown in Figure 1. Assume for a node x that we have $\mathcal{L}(x) = \{\geq 3\, R, \leq 2\, T, \geq 1\, S, \leq 1\, S\}$. Therefore, we have $R_x = \{R, T, S\}$ and $\mathcal{RS}_x$ consists of 7 different partitions named $p_1, \ldots, p_7$ (see Figure 1).

$p_1 = \{R\}$, $p_2 = \{T\}$, $p_4 = \{S\}$,
$p_3 = \{R, T\}$, $p_5 = \{R, S\}$, $p_6 = \{S, T\}$,
$p_7 = \{R, S, T\}$

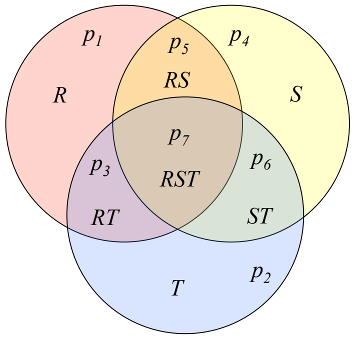

Fig. 1. Atomic Decomposition Example.

Assuming a binary coding of the indices of variables, where the first digit from the right represents R, the second T, and the last S, we define the variables such that each v_i is associated with its corresponding partition (see $\alpha(v)$ below and Figure 1).

$\alpha(v_{001}) = p_1$, $\alpha(v_{010}) = p_2$, $\alpha(v_{100}) = p_4$,
$\alpha(v_{011}) = p_3$, $\alpha(v_{101}) = p_5$, $\alpha(v_{110}) = p_6$,
$\alpha(v_{111}) = p_7$

Hence, the number restrictions in $\mathcal{L}(x)$ can be translated to the following set of inequations in $\mathcal{L}_E(x)$:

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

3.4. Partitioning for Root Nodes

In arbitrary Aboxes, similar to the effect of inverse roles, a root node in a corresponding completion forest can be influenced by its successors if they are also root nodes. When a new number restriction for a root node x is added to $\mathcal{L}(x)$, the algorithm needs to refine the partitioning assigned to x. However, the current state of the forest is a result of existing solutions based on the previous partitioning. In fact, the newly added number restriction has been added to $\mathcal{L}(x)$ after the application of a completion rule (in fact, by the *fil*-Rule which is explained in the next section). Therefore, we can conclude that the newly added number 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.

We considered two approaches to handle cycles between Abox individuals (represented as root nodes).

Global partitioning: One can treat an individual in an Abox similarly to a nominal. Because of the global effect of nominals, one has to consider a global partitioning for *all* roles occurring in the Abox (see [9,10] for more details). Hence, all possible partitions and therefore variables will be computed before starting the application of the rules. Consequently, whenever a number restriction is added to the label of a node, there is no need to recompute the partitioning to construct new inequations. Although global partitioning enables the algorithm to deal with cycles, according to the possibly large number of partitions/variables, it might impose a high nondeterminism on the tableau algorithm. Moreover, the number of added roles (and partitions) could become enormously large for Aboxes containing many role assertions because our algorithm introduces a new sub-role for each Abox assertion of the form $(a, b) : R$.

Incremental local partitioning: In contrast to nominals, which increase the expressiveness of the language and can have a global effect on other nodes, individuals in Aboxes have a local effect and can be handled locally. Moreover, one can usually assume that Aboxes contain a large number of individuals but only a relatively small number of nominals. Therefore, in the context of $\mathcal{SHQ}$ we decided to adopt incremental partitioning and deal with cycles between Abox individuals locally. This means that whenever a previously unknown number restriction for a new role is added to a root node, a new extended partitioning is computed and the corresponding arithmetic constraints are updated. Please note that this is only necessary for root nodes.

3.5. Completion Rules

The completion rules are shown in Figure 2. The completion rules are always applied with the fol-

reset-Rule	**if** x is a root node, $\{(\leq nR), (\geq nR)\} \cap \mathcal{L}(x) \neq \emptyset$, and $\forall v \in V_x : R \notin \alpha(v)$ **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 root nodes z_x, z_a, z_b for $x, a, b \in I_{\mathcal{A}}$ such that $R' \sqsubseteq_* R_{xa}$, $R' \in \mathcal{L}(\langle z_x, z_b\rangle)$ **then** merge the nodes z_a, z_b and their labels and replace every occurrence of z_a in the completion graph by z_b
$\sqcap$-Rule	**if** $(C_1 \sqcap C_2) \in \mathcal{L}(x)$ and $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$ **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 $\mathcal{L}(x) = \mathcal{L}(x) \cup \{X\}$ with $X \in \{C_1, C_2\}$
$\forall$-Rule	**if** $\forall R.C \in \mathcal{L}(x)$ and there exists a y and R' with $R' \in \mathcal{L}(\langle x, y\rangle)$, $C \notin \mathcal{L}(y)$, $R' \sqsubseteq_* R$ **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' with $R' \in \mathcal{L}(\langle x, y\rangle)$, $R' \sqsubseteq_* R$, $C \notin \mathcal{L}(y)$, and there exists no S' such that $S' \sqsubseteq_* S$ and $S' \in \mathcal{L}(\langle x, y\rangle)$ **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 with $R' \in \mathcal{L}(\langle x, y\rangle)$, $R' \sqsubseteq_* S$, $S \sqsubseteq_* R$, $S \in N_{RT}$, and $\forall S.C \notin \mathcal{L}(y)$ **then** set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{\forall S.C\}$
ch-Rule	**If** there occurs v in $\mathcal{L}_E(x)$ with $\{v \geq 1,\ v \leq 0\} \cap \mathcal{L}_E(x) = \emptyset$ **then** set $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{X\}$ with $X \in \{v \geq 1,\ v \leq 0\}$
disjoint-Rule	**if** there occurs v in $\mathcal{L}_E(z_x)$ with $\{R', S'\} \subseteq \alpha(v)$, $R' \sqsubseteq_* R_{xa}$, $S' \sqsubseteq_* R_{xb}$, $x, a, b \in I_{\mathcal{A}}$, $a \neq b \in \mathcal{A}$, $v \leq 0 \notin \mathcal{L}_E(z_x)$ **then** set $\mathcal{L}_E(z_x) := \mathcal{L}_E(z_x) \cup \{v \leq 0\}$
equal-Rule	**if** there occurs v in $\mathcal{L}_E(z_x)$ with $R' \in \alpha(v)$, $S' \notin \alpha(v)$, $R' \sqsubseteq_* R_{xa}$, $S' \sqsubseteq_* S_{xa}$, $x, a, b \in I_{\mathcal{A}}$, $v \leq 0 \notin \mathcal{L}_E(z_x)$ **then** set $\mathcal{L}_E(z_x) := \mathcal{L}_E(z_x) \cup \{v \leq 0\}$
hierarchy-Rule	**if** there occurs v in $\mathcal{L}_E(x)$ with $R \in \alpha(v)$, $S \notin \alpha(v)$, $R \sqsubseteq_* S$, $v \leq 0 \notin \mathcal{L}_E(x)$ **then** set $\mathcal{L}_E(x) := \mathcal{L}_E(x) \cup \{v \leq 0\}$
$\geq$-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 $\mathcal{L}_E(x) = \mathcal{L}_E(x) \cup \{\xi(R, \leq, n)\}$
fil-Rule	**If** there exists v occurring in $\mathcal{L}_E(z_x)$ such that $\sigma(v) = n$ with $n > 0$: *(a)* **if** $n = 1$, z_x, z_b root nodes, and $R_{xb} \in \alpha(v)$ with $x, b \in I_{\mathcal{A}}$ **then if** $\mathcal{L}(\langle z_x, z_b\rangle) = \emptyset$ **then** set $\mathcal{L}(\langle z_x, z_b\rangle) := \alpha(v)$ **end if** **else** *(b)* **if** z_x is not blocked and $\neg\exists y : \mathcal{L}(\langle z_x, y\rangle) = \alpha(v)$ **then** create a new node y and set $\mathcal{L}(\langle z_x, y\rangle) := \alpha(v)$ and $card(y) = n$

Fig. 2. Completion rules for $\mathcal{SHQ}$ Abox consistency (listed in decreasing priority).

lowing priorities (given in decreasing priority). A rule of lower priority can never be applied if another one with a higher priority is applicable.

1. *reset*-Rule and *merge*-Rule
2. $\sqcap$-Rule, $\sqcup$-Rule, $\forall$-Rule, $\forall_{\backslash}$-Rule, $\forall_{+}$-Rule, *ch*-Rule, *disjoint*-Rule, *equal*-Rule, and *hierarchy*-Rule
3. $\leq$-Rule and $\geq$-Rule
4. *fil*-Rule

There are two limitations on the application of the rules:

– priority of the rules, and
– rules are only applicable to nodes that are not blocked.

All rules which have the two highest priorities among the completion rules, extend $\mathcal{L}(x)$ with new logical expressions or further constrain $\mathcal{L}_E(x)$. After the application of these rules the logical label of the node x cannot be expanded anymore. This is a consequence of the characteristics of $\mathcal{SHQ}$ because the labels of a non-root node can never be affected by its successors in the graph.

Definition 15 (Complete/Clash-free Forest) A completion forest $\mathcal{F}$ is called *complete* if no completion rule is applicable to any node of $\mathcal{F}$. A completion forest $\mathcal{F}$ is called *clash-free* if none of the clash triggers defined in Definition 13 is applicable to any node of $\mathcal{F}$.

The $\sqcap$-Rule, $\sqcup$-Rule, $\forall$-Rule, and the $\forall_+$-Rule are similar to the ones in standard tableau algorithms. The $\forall_+$-Rule preserves the semantics of transitive roles. The $\forall_\backslash$-Rule handles the new universal restriction expression introduced by the transformation function *unQ*.

The $\sqcup$-Rule and *ch*-Rule are nondeterministic while all other rules are deterministic.

$\leq$-Rule, $\geq$-Rule: Since all logical constraints of a node are collected by the rules with the two highest priorities, after their application the algorithm has collected all number 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 number restrictions, based on the atomic decomposition technique, into inequations. Consequently, they will add these inequations to $\mathcal{L}_E(x)$ for a node x. In case x is a root node with incoming edges from other root nodes, the partitioning might be revised later due to the interaction of the *reset*-Rule and *fil*-Rule.

ch-Rule: The intuition behind the *ch*-Rule is due to partitioning consequences. When we partition all successors for a node, we actually consider all possible cases for the role successors. If a partition p_x for a node x is logically unsatisfiable, the corresponding variable v with $\alpha(v) = p_x$ should be zero. But if it is logically satisfiable, nothing but the current set of inequations can restrict the number of nodes being fillers for the roles in the partition p_x. 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 and ensure completeness, the algorithm needs to distinguish between these two cases: $v \geq 1$ or $v \leq 0$.

disjoint-Rule: In order to preserve Abox assertions of the form $a \neq b$, if there exists a node z_x where a variable v occurs in $L_E(z_x)$ such that $\{R', S'\} \subseteq \alpha(v)$ and $R' \sqsubseteq_* R_{xa}$ and $S' \sqsubseteq_* R_{xb}$, then this variable needs to be equal to zero because otherwise a solution might exist where a and b need to be merged.

equal-Rule: Since fillers of R_{xy} and S_{xy} have to be equivalent, if there exists a node z_x where a variable v occurs in $L_E(z_x)$ such that $R' \in \alpha(v)$ but $S' \notin \alpha(v)$ with $R' \sqsubseteq_* R_{xy}$ and $S' \sqsubseteq_* S_{xy}$, then this variable needs to be equal to zero because otherwise a solution might exist where the fillers of R_{xy} and S_{xy} would not be equivalent.

hierarchy-Rule: In order to preserve role hierarchies, if there exists a node x where a variable v occurs in $L_E(x)$ such that $R \in \alpha(v)$ and $S \notin \alpha(v)$ but $R \sqsubseteq_* S$, i.e., the partition $\alpha(v)$ violates the role hierarchy, then this variable needs to be equal to zero in order to "disable" this partition.

reset-Rule: In case a number restriction with a new role R has been added to a root node x, the label $\mathcal{L}_E$ and the edge label of all successors of x are reset to $\emptyset$.

merge-Rule: This rule merges the successors (and root nodes) z_a, z_b of z_x into one node by replacing every occurrence of z_a in the completion graph by z_b. The labels $\mathcal{L}$ and $\mathcal{L}_E$ of z_b are unified with the corresponding labels of z_a. All incoming (outgoing) edges of z_a become now incoming (outgoing) edges of z_b. We deliberately do not specify the details of such a node merger but appeal to the intuition of the reader and refer to the following two section describing merging examples.

fil-Rule: The *fil*-Rule is the rule with the lowest priority and it is the only generating rule. It possibly creates one successor (proxy node) for a node z_x that is not blocked and sets the cardinality of the proxy node to $\sigma(v)$. If z_x is a root node and has an already existing successor x_b, which is a also a root node, and the role set for $\langle z_x, z_b \rangle$ is empty, i.e., it has not yet been initialized or has been reset by the *reset*-rule, the role set is set to $\alpha(v)$. This rule only creates successors based on the non-negative integer solution provided by the arithmetic reasoner. Hence, it will never create successors for a node that might violate number restrictions of this node. Therefore, there is no need for a mechanism of merging nodes created by this rule.

Due to possible cycles between root nodes, one has to consider incremental partitioning for root nodes because a previously unknown number restriction could be propagated to a root node. Whenever a concept expression of the form $(\leq n\, R)$ or $(\geq\; m\, R)$ is added to $\mathcal{L}(x)$ (which means it did not already exist in $\mathcal{L}(x)$), the following three tasks take place:

1. The *reset*-Rule becomes applicable for x, which sets $\mathcal{L}_E(x) := \emptyset$ and clears the arithmetic label constraining the outgoing edges of x.
2. Now that $\mathcal{L}_E(x)$ is empty, the $\leq$-Rule and $\geq$-Rule will be invoked again to recompute the partitions and variables. Afterwards, the set of inequations based on the new partitioning is added.
3. If $(\sigma(v_i) = n) \in \Omega(x)$, where $v_i \in \mathcal{V}_x$ corresponds to the previous partitioning, then set

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

where $\mathcal{V}_{x,i} := \{v'_j \in \mathcal{V}'_x \mid \alpha(v_i) \subseteq \alpha(v'_j)\}$ and $v'_j \in \mathcal{V}'_x$ are based on the new partitioning.

The third task in fact maintains the previous solutions in x and records it 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 3.7.

Remark 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 $v_s \in \mathcal{V}_{x,i}$. 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.

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

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 equal to 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 the new nodes to root nodes, the new nodes never propagate back any information in the forest to root nodes. Therefore, removing x_i from the forest does not violate restrictions on the root nodes.

3.6. Simple merging example

Let us assume an Abox $\mathcal{A} = \{x : \leq 1\, R, (x,y) : R, (x,z) : R\}$. The preprocessing converts the role assertions and we get a new $\mathcal{A}' = \{x : \leq 1\, R,\ x : \geq 1\, R_{xy},\ x : \leq 1\, R_{xy},\ x : \geq 1\, R_{xz},\ x : \leq 1\, R_{xz}\}$. From this we derive a forest $\mathcal{F}$ and the root nodes x, y, z with the following labels: $R \in \mathcal{L}(\langle x,z\rangle)$, $R \in \mathcal{L}(\langle x,y\rangle)$, $\mathcal{L}(x) = \{\leq 1\, R, \leq 1\, R_{xy}, \geq 1\, R_{xy}, \leq 1\, R_{xz}, \geq 1\, R_{xz}\}$ where $\{R_{xy} \sqsubseteq R, R_{xz} \sqsubseteq R\} \subseteq \mathcal{R}$. Consider the variables such that $\alpha(v_{001}) = \{R\}$, $\alpha(v_{010}) = \{R_{xy}\}$, $\alpha(v_{100}) = \{R_{xz}\}$, ..., $\alpha(v_{111}) = \{R, R_{xy}, R_{xz}\}$.

After applying the *hierarchy*-Rule, we will have $v_{010} \leq 0$, $v_{100} \leq 0$, and $v_{110} \leq 0$. Hence, after removing all variables that must be zero, the following system of inequations in $\mathcal{L}_E(x)$ needs to be solved:

$$\left.\begin{aligned} v_{001} + v_{011} + v_{101} + v_{111} \leq 1, \\ v_{011} + v_{111} = 1 \\ v_{101} + v_{111} = 1 \end{aligned}\right\} (*)$$

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 2 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 2 becomes applicable for the nodes y, z and merges them.

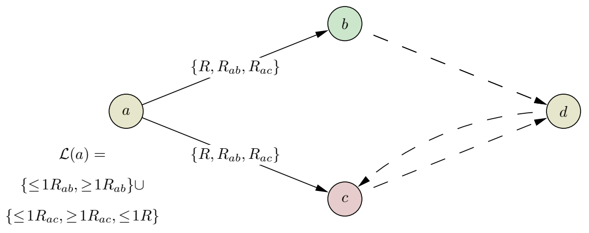

Fig. 3. Initial completion graph. Dashed edges do not exist yet.

3.7. Complex merging example

Consider the Abox $\mathcal{A}$ with an empty Tbox and role hierarchy: $\mathcal{A} = \{b : \forall S.(\forall R.(\ge 3S.C)), a : \le 1\,R, (a,b) : R, (a,c) : R, (b,d) : R, (c,d) : S, (d,c) : R\}$. 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 $(\ge 3\,S.C) \to \ge 3\,S' \sqcap \forall S'.C$
2. Converting 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 $\mathcal{L}(a) = \mathcal{L}(a) \cup \{\le 1\,R_{ab}, \ge 1\,R_{ab}, \le 1\,R_{ac}, \ge 1\,R_{ac}\}$, $\mathcal{L}(b) = \mathcal{L}(b) \cup \{\le 1\,R_{bd}, \ge 1\,R_{bd}\}$, $\mathcal{L}(c) = \mathcal{L}(c) \cup \{\le 1\,S_{cd}, \ge 1\,S_{cd}\}$, $\mathcal{L}(d) = \mathcal{L}(d) \cup \{\le 1\,R_{dc}, \ge 1\,R_{dc}\}$

Applying the rules: The $\le$-Rule and $\ge$-Rule are applicable for all of the nodes and translate the number restrictions into inequations (Figure 3). Node a is similar to node x in the example in the previous section and invokes the *merge*-Rule for c and b.

Assume the *merge*-Rule replaces every occurrence of c by b, we will have $\mathcal{L}(b) = \{\le 1\,R_{bd}, \ge 1\,R_{bd}, \le 1\,S_{bd}, \ge 1\,S_{bd}, \forall S.(\forall R.(\ge 3\,S' \sqcap \forall S'.C))\}$ and for d we will have $\mathcal{L}(d) = \{\le 1\,R_{db}, \ge 1\,R_{db}\}$ (Figure 4). We have four unqualified number restrictions in $\mathcal{L}(b)$ (equivalent to two equality restrictions) which will be transformed into inequations by the $\le$-Rule and the $\ge$-Rule. Assuming $\alpha(v_{01}) = \{R_{bd}\}$ and $\alpha(v_{10}) = \{S_{bd}\}$, we will have $v_{01} + v_{11} = 1$ and $v_{10} + v_{11} = 1$. After applying the *hierarchy*-Rule, we get $v_{01} \le 0$ and $v_{10} \le 0$. Thus, there is only one solution for $\mathcal{L}_E(b)$ which is $\sigma(v_{11}) = 1$ and the *fil*-Rule sets $\mathcal{L}(\langle b, d\rangle) = \{S_{bd}, R_{bd}\}$.

Afterwards, the $\forall$-Rule becomes applicable for the node b and adds $\forall R.(\ge 3\,S' \sqcap \forall S'.C)$ to $\mathcal{L}(d)$. There is only one equation for d which results in

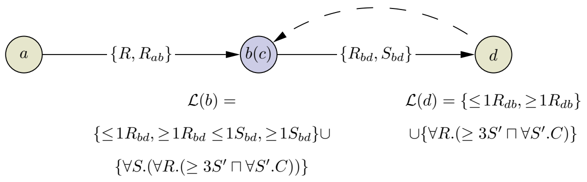
a
$\{R, R_{ab}\}$
$b(c)$
$\{R_{bd}, S_{bd}\}$
d
$\mathcal{L}(b) =$
$\{\le 1R_{bd}, \ge 1R_{bd} \le 1S_{bd}, \ge 1S_{bd}\}\cup$
$\{\forall S.(\forall R.(\ge 3S' \sqcap \forall S'.C))\}$
$\mathcal{L}(d) = \{\le 1R_{db}, \ge 1R_{db}\}$
$\cup\{\forall R.(\ge 3S' \sqcap \forall S'.C)\}$

Fig. 4. Completion graph after merging b and c.

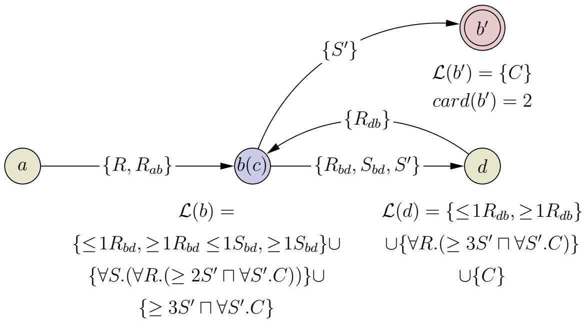
$\{S'\}$
b'
$\mathcal{L}(b') = \{C\}$
$card(b') = 2$
$\{R_{db}\}$
a
$\{R, R_{ab}\}$
$b(c)$
$\{R_{bd}, S_{bd}, S'\}$
d
$\mathcal{L}(b) =$
$\{\le 1R_{bd}, \ge 1R_{bd} \le 1S_{bd}, \ge 1S_{bd}\}\cup$
$\{\forall S.(\forall R.(\ge 2S' \sqcap \forall S'.C))\}\cup$
$\{\ge 3S' \sqcap \forall S'.C\}$
$\mathcal{L}(d) = \{\le 1R_{db}, \ge 1R_{db}\}$
$\cup\{\forall R.(\ge 3S' \sqcap \forall S'.C)\}$
$\cup\{C\}$

Fig. 5. Final completion graph.

setting $\mathcal{L}(\langle d, b\rangle) = \{R_{db}\}$. After this change, as $R_{db} \sqsubseteq R$ the $\forall$-Rule adds $(\ge 3\,S' \sqcap \forall S'.C)$ to $\mathcal{L}(b)$. Since $\ge 3\,S'$ did not exist in $\mathcal{L}(b)$, the algorithm performs $reset(b)$. The $\le$-Rule and the $\ge$-Rule will be fired again to recompute the partitions, considering the new number restriction $\ge 3\,S'$. 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:

$$\left.\begin{aligned} \underline{v'_{001}} + v'_{011} + \underline{v'_{101}} + v'_{111} &= 1 \\ \underline{v'_{010}} + v'_{011} + \underline{v'_{110}} + v'_{111} &= 1 \\ v'_{100} + \underline{v'_{101}} + \underline{v'_{110}} + v'_{111} &\ge 3 \\ v'_{011} + v'_{111} &= 1 \end{aligned}\right\} (**)$$

After applying the *hierarchy*-Rule, the variables v'_{001}, v'_{010}, v'_{101}, and v'_{110} (underlined in $(**)$) must be less than or equal to zero. One of the solutions for $(**)$ can be achieved, after the *ch*-Rule decided $v'_{111} \ge 1$, $v'_{011} \le 0$, and $v'_{100} \ge 1$, 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 $\mathcal{L}(d)$ and $\mathcal{L}(b')$ (see Figure 5).

3.8. Proof of Termination, Soundness, and Completeness

In this section we prove the termination, soundness, and completeness of the proposed hybrid calculus for the Abox consistency test. Weaker problems such as the Tbox satisfiability or concept satisfiability test can be reduced to the Abox satisfiability test with respect to a given Tbox and role hierarchy (see also at the beginning of Section 3).

3.8.1. Tableau

In order to prove the soundness and completeness of our calculus, a *tableau* is defined as an abstraction of a model to facilitate relating the result of the calculus 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 [21]. The similarity between tableau and completion graphs, which are the result of the completion rules, makes it easier to prove the soundness and completeness of the calculus.

Due to the fact that after preprocessing the input of the hybrid tableau calculus is in $\mathcal{SHN}^{\backslash}$, we define a tableau for $\mathcal{SHN}^{\backslash}$ Aboxes with respect to a role hierarchy $\mathcal{R}$. The following definition is similar to the one in [24].

Definition 16 (Tableau for $\mathcal{SHN}^{\backslash}$) Let $R_{\mathcal{A}}$ be the set of role names and $I_{\mathcal{A}}$ the set of individuals in an 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:

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

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

P1 If $A \in \mathcal{L}^{\mathcal{T}}(s)$, then $\neg A \notin \mathcal{L}^{\mathcal{T}}(s)$.
P2 If $C_1 \sqcap C_2 \in \mathcal{L}^{\mathcal{T}}(s)$, then $C_1 \in \mathcal{L}^{\mathcal{T}}(s)$ and $C_2 \in \mathcal{L}^{\mathcal{T}}(s)$.
P3 If $C_1 \sqcup C_2 \in \mathcal{L}^{\mathcal{T}}(s)$, then $C_1 \in \mathcal{L}^{\mathcal{T}}(s)$ or $C_2 \in \mathcal{L}^{\mathcal{T}}(s)$.
P4 If $\forall R.C \in \mathcal{L}^{\mathcal{T}}(s)$ and $\langle s,t\rangle \in \mathcal{E}(R)$, then $C \in \mathcal{L}^{\mathcal{T}}(t)$.
P5 If $\forall R.C \in \mathcal{L}^{\mathcal{T}}(s)$, for some $S \sqsubseteq R$ we have S is transitive, and $\langle s,t\rangle \in \mathcal{E}(S)$, then $\forall S.C \in \mathcal{L}^{\mathcal{T}}(t)$.
P6 If $\forall R\backslash S.C \in \mathcal{L}^{\mathcal{T}}(s)$, $\langle s,t\rangle \in \mathcal{E}(R)$, but $\langle s,t\rangle \notin \mathcal{E}(S)$, then $C \in \mathcal{L}^{\mathcal{T}}(t)$.
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 $\geq nR \in \mathcal{L}^{\mathcal{T}}(s)$, then $\#R^T(s) \geq n$
P9 If $\leq mR \in \mathcal{L}^{\mathcal{T}}(s)$, then $\#R^T(s) \leq m$
P10 If $(a : C) \in \mathcal{A}$ then $C \in \mathcal{L}^{\mathcal{T}}(\mathcal{J}(a))$
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 17 A $\mathcal{SHQ}$ Abox $\mathcal{A}$ has a tableau iff $unQ(\mathcal{A})$ has a $\mathcal{SHN}^{\backslash}$ tableau, where $unQ(\mathcal{A})$ denotes $\mathcal{A}$ after applying unQ to every assertion in $\mathcal{A}$.

Lemma 17 is a straightforward consequence of the equisatisfiability of C and $unQ(C)$ for every concept expression C in $\mathcal{SHQ}$ (see Section 2 and [30]).

3.8.2. 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 nondeterministic rules, the $\sqcup$-Rule and the *ch*-Rule, we must also prove that the algorithm creates finitely many forests due to nondeterminism.

Lemma 18 (Termination) The hybrid algorithm terminates for a given Abox $\mathcal{A}$ with respect to a role hierarchy $\mathcal{R}$[7].

Proof. Let $m = |clos(\mathcal{A})|$ and k be the number of different number 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:

[7]Since Tbox axioms are propagated through the universal transitive role, we do not mention the Tbox as an input of the algorithm (see Section 2).

1. There are only two nondeterministic 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 $2^{\mathcal{V}_x}$ times and $\mathcal{V}_x$ is bounded by 2^k. Accordingly, we can conclude that the nondeterministic rules can only be fired finitely often 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 $|\mathcal{V}_x|$ successors for a node x. Therefore, the out degree of the forest is bounded by $|\mathcal{V}_x| \leq 2^k$.
4. According to the blocking condition, 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 only finitely many forests. Considering (2) and (5), we can conclude the termination of the hybrid algorithm. □

3.8.3. Soundness

To prove the soundness, we must prove that the constructed model, based on a complete and clash-free completion forest, does not violate the semantics of the input language. According to Lemma 17 and the fact that a model can be obtained from a $\mathcal{SHQ}$ tableau (proven in [24]), it is sufficient to prove that a $\mathcal{SHN}^\backslash$ tableau can be obtained from a complete and clash-free forest.

Lemma 19 (Abox semantics) The hybrid algorithm preserves the semantics of Abox assertions; i.e., assertions of the form $(a,b) : R$ and $a \neq b$.

Proof. The algorithm replaces assertions of the form $(a,b) : R$ with $a : (\leq 1\, R_{ab} \sqcap \geq 1\, R_{ab})$. 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}, the cardinality restrictions, and assuming the fact that the algorithm correctly handles unqualified number restrictions, one can conclude that for some $v \in V_{x_a}$ that $\sigma(v) = 1$ with $R_{ab} \in \alpha(v)$. Therefore, according to the condition *(a)* of the *fil*-Rule in Figure 2, R_{ab} will be added to $\mathcal{L}(\langle x_a, x_b\rangle)$. Since the preprocessing and the *hierarchy*-Rule preserve the role hierarchy and $R_{ab} \sqsubseteq R$ holds, the assertion $(a,b) : R$ is satisfied.

On the other hand, due to the restriction $\leq 1\, R_{ab}$, 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.

Let us assume an assertion of the form $a \dot{\neq} b$ is violated, i.e., $a^\mathcal{I} = b^\mathcal{I}$. This could only happen if the *merge*-Rule became applicable to the corresponding nodes $x_a, x_b, x_y \in V$ such that w.l.o.g. $\{R_{ya}, R_{yb}\} \subseteq \mathcal{L}(\langle x_y, x_b\rangle)$ and $\mathcal{A}$ contains $\{(y,a) : R, (y,b) : R\}$. Then, the *disjoint*-Rule would have fired for a $v \in \mathcal{V}_{x_y}$ and added $v \leq 0$ to $\mathcal{L}_E(x_y)$ such that $\{R_{xa}, R_{xb}\} \subseteq \alpha(v)$ because due to our preprocessing and its resulting role hierarchy and $\{R_{xa}, R_{xb}\} \subseteq \mathcal{L}(\langle x_y, x_b\rangle)$, $\mathcal{L}_E(x_y)$ must contain in at least one of its inequations a variable v representing the possibility that a and b need to be merged. Due to $v \leq 0 \in \mathcal{L}_E(x_y)$, it is impossible for the arithmetic reasoner to create a solution which would require the algorithm to merge a and b. □

Lemma 20 (Soundness) If the completion 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 $\mathcal{F}$ by $T := (\mathbf{S}, \mathcal{L}^T, \mathcal{E}, \mathcal{J})$ as shown in Figure 6.

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.

$\mathbf{S} :=$	$\{x_1, \ldots, x_m \mid x \in \mathcal{F}, card(x) = m\}$
$\mathcal{L}^T(x_i) :=$	$\mathcal{L}(x)$ 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 into x_a such that every occurrence of x_b is replaced by x_a, then $\mathcal{J}(b) = x_a$.

Fig. 6. Converting forest $\mathcal{F}$ to tableau 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** and **P5** are respectively guaranteed by the $\forall$-Rule and $\forall_+$-Rule.

– If $\forall R\backslash S.C \in \mathcal{L}^T(x_i)$ it means that $\forall R\backslash S.C \in \mathcal{L}(x)$. Also, $\langle x_i, y_i\rangle \in \mathcal{E}(R)$ means that there exists a $R' \sqsubseteq_* R$ where $R' \in \mathcal{L}\langle x, y\rangle$. Similarly, $\langle x_i, y_i\rangle \notin \mathcal{E}(S)$ means that there exists no $S' \sqsubseteq_* S$ such that $S' \in \mathcal{L}\langle x, y\rangle$. Therefore, the $\forall_\backslash$-Rule becomes applicable for x and adds C to $\mathcal{L}(y)$ which is equivalent with $C \in \mathcal{L}^T(y_i)$ and **P6** holds for T.

– 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 *hierarchy*-Rule maintains 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.

– Due to the priorities of rules, if the $\leq$-Rule and the $\geq$-Rule are invoked, the logical label, $\mathcal{L}(x)$, cannot be extended anymore. In other words, a correct partitioning based on all the number restrictions for a node has been created. Therefore, the solution created by the arithmetic reasoner satisfies all 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 $x_i \in \mathbf{S}$. 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 represents 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 19. □

3.8.4. 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 completion rules in such a way that it yields a forest $\mathcal{F}$ from which we can obtain T as shown in Figure 6.

Lemma 21 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 way to extend 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 the following 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 completion rules. Therefore, when extended similarly, they will remain identical after the application of the tableau rules.

(ii) If w.l.o.g. 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 22 According to the mapping from a forest $\mathcal{F}$ to a $\mathcal{SHN}^\backslash$ tableau T (Figure 6), every $\mathcal{L}^T(s)$ in T is equal to $\mathcal{L}(x)$ in $\mathcal{F}$ if x is mapped to s. Moreover, every $R \in \mathcal{L}(\langle x, y\rangle)$ is mapped to $\langle s, t\rangle \in \mathcal{E}(R)$. Therefore $\mathcal{L}(\langle x, y\rangle) = \mathcal{L}(\langle x, z\rangle)$ is equivalent to $\{R \in N_R \mid \langle s, t\rangle \in \mathcal{E}(R)\} = \{R \in$

$N_R \mid \langle s, t' \rangle \in \mathcal{E}(R)\}$ 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 Lemma 21, we can conclude: $\{R \in N_R \mid \langle s, t \rangle \in \mathcal{E}(R)\} = \{R \in N_R \mid \langle s, t' \rangle \in \mathcal{E}(R)\} \Rightarrow \mathcal{L}^T(t) = \mathcal{L}^T(t')$.

Lemma 23 If for a node z_x a set of non-negative integer solutions $\Omega(z_x)$ based on the set of inequations $\mathcal{L}_E(z_x)$ causes a logical clash, all other non-negative integer solutions for $\mathcal{L}_E(z_x)$ will also trigger the same logical clash.

Proof. Assume we have a solution set $\Omega(z_x) = \{\sigma(v_1) = m_1, \sigma(v_2) = m_2, \ldots, \sigma(v_n) = m_n\}$, which can only occur if $v_i \geq 1$ for $1 \leq i \leq n$ is decided by the *ch*-Rule and all other variables in $\mathcal{V}_{z_x}$ are equal to zero. Let $\mathcal{V}_{z_x}^{\geq 1} = \{v \in \mathcal{V}_{z_x} \mid \sigma(v) \geq 1\}$ denote this subset of variables. Suppose we also have a different solution $\Omega'(z_x)$ for $\mathcal{L}_E(z_x)$ and $\mathcal{V}_{z_x}^{\geq 1}$ such that $\Omega'(z_x) = \{\sigma'(v_1) = p_1, \sigma'(v_2) = p_2, \ldots, \sigma'(v_n) = p_n\}$. Both solutions $\Omega(z_x)$ and $\Omega'(z_x)$ have the same the set $\mathcal{V}_{z_x}^{\geq 1}$ of non-zero variables. So, for both solutions the *fil*-Rule will fire for the variables in $\mathcal{V}_{z_x}^{\geq 1}$ but for some of these variables their assignment given by σ and σ' will be different. We have to distinguish the following two cases for a given $v \in \mathcal{V}_{z_x}^{\geq 1}$: (i) condition *(a)* of the *fil*-Rule (see Figure 2) is true, then z_x and its successor z_b are root nodes representing a role assertion $(x, b) : R_{xb}$ (see Definition 5) and z_x must have for all possible arithmetic solutions an R_{xb}-filler, so, since $\sigma(v) = \sigma'(v) = 1$ the possibly assigned value $\alpha(v)$ of $\mathcal{L}_E(\langle z_x, z_b \rangle)$ will only depend on the partitioning $\mathcal{RS}_{z_x}$ (see Definition 8) and not on $\Omega(z_x)$ or $\Omega'(z_x)$; (ii) condition *(b)* is true, then the *fil*-Rule will create a new node y and, again, the assigned value $\alpha(v)$ of $\mathcal{L}_E(\langle z_x, y \rangle)$ will only depend on the partitioning $\mathcal{RS}_{z_x}$ and not on $\Omega(z_x)$ or $\Omega'(z_x)$.

This concludes that the edge labels $\mathcal{L}_E(\langle z_x, \cdot \rangle)$, created by the *fil*-Rule, only depend on the composition of $\mathcal{V}_{z_x}^{\geq 1}$ and not on the values assigned to the variables in $\mathcal{V}_{z_x}^{\geq 1}$. Therefore, by considering Lemma 21 the forest generated based on $\Omega'(x)$ will contain the same nodes as $\Omega(x)$, however, with different associated cardinalities. Since logical clashes do not depend on the cardinality of the nodes, we can conclude that the selection of $\Omega'(x)$ will result in the same clash as for $\Omega(x)$. $\square$

Corollary 24 According to Lemma 23, all 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 25 (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 completion 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 obtaining T from $\mathcal{F}$ is shown in Figure 6. We prove this by induction on the set of nodes in V.

Consider a node x in $\mathcal{F}$ and the completion rules in Figure 2 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 completion 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, and $\forall_+$-Rule, which are deterministic rules, are similar to the $\sqcup$-Rule. In fact these rules are built exactly based on their relevant tableau property.
- $\forall_{\backslash}$-Rule: If $\forall R \backslash S \in \mathcal{L}(x)$ then $\forall R \backslash S.C \in \mathcal{L}^T(s)$ which means if $\langle s, t \rangle \in \mathcal{L}(R)$ but $\langle s, t \rangle \notin \mathcal{E}(S)$ then $C \in \mathcal{L}^T(t)$. If t is a mapping from y it is equivalent to say if there exists $R' \sqsubseteq R$ and no $S' \sqsubseteq S$ such that $R' \in \mathcal{L}(\langle x, y \rangle)$ and $S' \in \mathcal{L}(\langle x, y \rangle)$ then $C \in \mathcal{L}(y)$. This condition is satisfied by means of the $\forall_{\backslash}$-Rule.
- The *ch*-Rule: Consider in T 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 22, $N_{kl} := \{R \in N_R \mid \langle s, t_j \rangle \in \mathcal{E}(R)\}$ will be identical for t_j, $k \leq j \leq l$. We define a variable v_{kl} for such a set of role names with $\alpha(v_{kl}) = N_{kl}$. 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, $\leq n\, R$ and $\geq m\, R$ are satisfied in T for

s. Therefore, the inequations based on these variables will have a non-negative integer solution. Notice that the created set of variable constraints 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 23 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 $a, b, c \in I_\mathcal{A}$ are such that b and c are successors of a that must be merged according to an at-most restriction $a\colon (\leq n\, R)$. Since T is a tableau the restriction $\leq n\, R \in \mathcal{L}^T(\mathcal{J}(a))$ 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, $\geq$-Rule, *equal*-Rule, *disjoint*-Rule, and the *hierarchy*-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 $card(x) = m$ and $m > 1$ will be mapped to m elements in $\mathbf{S}$.

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 $(\geq n\, R)$ or $(\leq m\, R)$ in $\mathcal{L}(x)$ and therefore $\mathcal{L}^T(s)$ that cannot be satisfied which violates property **P8** and/or **P9** of a tableau. □

4. 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 mentioned above, a standard algorithm to deal with qualified number restrictions must extend its tableau rules with at least two nondeterministic rules; i.e., the *choose*-Rule and the $\leq$-Rule (see Figure 7). In order to achieve an acceptable performance, a tableau algorithm needs to employ effective optimization techniques. As stated in [19], the performance of tableau algorithms even for simple logics is a problematic issue.

In this section we briefly analyze the complexity of both the standard and hybrid algorithms. Based on the complexity analysis, we address some sources of inefficiency in DL reasoning. Moreover, we propose some new or adapted optimization techniques for the hybrid algorithm. In the last part 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.

4.1. Complexity Analysis

To analyze the complexity of the concept satisfiability test with respect to qualified number restrictions, we count the number of branches that the algorithm creates in the search space.[8]

In the following we assume a node $x \in V$ in the completion forest that contains p at-least restrictions and q at-most restrictions in its label ($R_i, R'_j \in N_R$ and $C_i, C'_j \in clos(\mathcal{T})$):

[8]Notice that every nondeterministic rule that can have k outcomes opens k new branches in the search space.

$\geq$-Rule	**if** $(\geq n\, R.C) \in \mathcal{L}(x)$ and there are no R-successors $y_1, y_2, \ldots, y_n$ for x such that $C \in \mathcal{L}(y_i)$ and $y_i \neq y_j$ **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 $y_i \neq y_j$ for $1 \leq i \leq j \leq n$ and $i \neq j$
choose-Rule	**if** $(\leq m\, R.C) \in \mathcal{L}(x)$ 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** $\mathcal{L}'(y) := \mathcal{L}(y) \cup \{\neg C\}$
$\leq$-Rule	**if** (i) $(\leq n\, R.C) \in \mathcal{L}(x)$ and x has m R-successors such that $m > n$ and, (ii) there exist R-successors y, z for x and $y \neq z$ does not hold **then** replace every occurrence of y by z

Fig. 7. Standard tableau rules handling $\mathcal{Q}$.

$$\{\geq n_1\, R_1.C_1, \ldots, \geq n_p\, R_p.C_p\} \subseteq \mathcal{L}(x)$$
$$\{\leq m_1\, R'_1.C'_1, \ldots, \leq m_q\, R'_q.C'_q\} \subseteq \mathcal{L}(x)$$

4.1.1. Standard Tableau

The rules of a standard tableau algorithm dealing with qualified number restrictions as shown in Figure 7 create n R-successors in C for each at-least restriction of the form $\geq n\, R.C$. Moreover, in order to avoid that the successors are being merged, they are asserted as mutually distinct individuals. Assuming that no C_i is subsumed by a C_j, there will be $N := n_1 + \cdots + n_p$ 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 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 $f(M, m_i) := \binom{M}{2}\binom{M-1}{2}\ldots\binom{m_i+1}{2}/(M - m_i)!$ 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 $m := \min_{i \in 1..q} m_i$ 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.

4.1.2. Hybrid Tableau

During the preprocessing step, the hybrid algorithm converts all qualified number restrictions into unqualified ones which introduces $p + q$ new role names. According to the atomic decomposition presented in Section 3.2, the hybrid algorithm defines $2^{p+q} - 1$ partitions and consequently variables for x; i.e. $|\mathcal{V}_x| = 2^{p+q} - 1$. The *ch*-Rule opens two branches for each variable in $\mathcal{V}_x$. Therefore, there will be totally $2^{|\mathcal{V}_x|}$ cases to be examined by the arithmetic reasoner, which considers only a single solution out of many possible solutions. The *ch*-Rule will be invoked $|\mathcal{V}_x| = 2^{p+q} - 1$ 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 $p + q$. In [27] a polynomial-time algorithm for integer programming with a fixed number of variables is given. However, our prototype implementation employed the Simplex method which is known to be NP in the worst case but usually behaves very well on average.

4.1.3. Hybrid vs. Standard

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

- The complexity of the standard algorithm is a function of N and therefore the numbers occurring in the at-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 cases (whether worst or 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, based on an arithmetically correct solution for a set of inequations, generates and merges the successors of an individual deterministically and.[9]
– Whenever a clash occurs, the algorithm needs to explore an open choice point to choose a new branch. The sources of nondeterminism due to number 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.

4.2. Partition Optimization Techniques

For various DL reasoning services different optimization techniques have been developed. For example, axiom absorption [25] or lazy unfolding [2] are optimization techniques for Tbox services such as classification or subsumption testing. These optimization techniques facilitate subsumption testing and by avoiding unnecessary steps in 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 sources of inefficiency are due to high nondeterminism. In fact, nondeterministic rules such as the $\sqcup$-Rule in Figure 2 or the *choose*-Rule in Figure 7 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 it is that its decision will fail later.

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

Although it seems that the hybrid algorithm is double-exponential and the large number of variables seems to be hopelessly inefficient, there are some effective heuristics and optimization 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 average case.

4.2.1. Default Value for Variables

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 based on the *choose*-Rule, in variable branching we can ignore the existence of variables that are less or equal to zero. In other words, the arithmetic reasoner only considers variables that are greater or equal to one.

Therefore, by setting the default value to $v \leq 0$ for every variable, the algorithm does not need to invoke the *ch*-Rule $|\mathcal{V}_x|$ 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 variables in $|\mathcal{V}_x|$. Obviously, the solution for this set of inequations, which is $\forall\, v_i \in \mathcal{V}_x : \sigma(v_i) = 0$, cannot satisfy any at-least restriction. Therefore, the algorithm must choose some variables in $\mathcal{V}_x$ to make them greater or equal to one. Although in the worst case the algorithm still needs to try $2^{|\mathcal{V}_x|}$ 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 (see Section 4.1.3 for a definition) of the hybrid algorithm is no longer 2^{p+q}.

4.2.2. 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 find a solution faster, if one exists.

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, any non-negative integer value observing the arithmetic limitations can be assigned to these variables and we can leave them unchanged in all inequations unless a logical clash is triggered.

Moreover, we define *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 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 their corresponding number restriction. For example, in the inequation $v_1 + v_{01} + \cdots + v_{10000000} \geq 5$, although we have 128 variables in the inequation, not more than five of the v_i can be greater or equal to one at the same time.

4.2.3. Variable 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.

4.3. Dependency-Directed Backtracking

Backjumping [12] or conflict-directed-directed backjumping [32] are improved backtracking methods originally developed in areas such as constraint reasoning. These techniques were adapted to DL reasoning as dependency-directed backtracking [23]. These techniques detect the sources of an encountered clash and try to bypass during backtracking 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 [23,18], this method improved the performance of the FaCT system to deal much more effectively with qualified number restrictions. It is nowadays an optimization technique that is inevitable in any practically usable DL reasoner [1, Chapter 9]. It is beyond the scope of this article to prove the completeness of this well-known optimization technique.

By analogy, dependency-directed backtracking was also adapted to the hybrid algorithm. Whenever a logical clash for a successor y of x is encountered, one can conclude that the corresponding variable v_y for the partition in which y resides must be zero.

Assume, y is based on a solution $\sigma(v_y) = k$ by the *fil*-Rule and the algorithm encounters a clash in the node y. This solution can only occur whenever $v_y \geq 1$ is decided by the *ch*-Rule for the node x. Any other completion forest in this branch, where $v_y \geq 1$, will end up with a solution in the form $\sigma(v_y) = k'$ $(k' \geq 1)$. Assume the successor of x, created based on this solution is y'. Since $\mathcal{L}\langle x, y\rangle = \alpha(v_y)$ and $\mathcal{L}\langle x, y'\rangle = \alpha(v_y)$ are equal and all the concepts in $\mathcal{L}(y)$ and $\mathcal{L}(y')$ are created based on the roles from x, we can conclude that $\mathcal{L}(y) = \mathcal{L}(y')$. Therefore, y' will contain the same clash as y. Consequently, we can conclude that all the branches in the search space where $v_y \geq 1$ will end up with the same clash.

Therefore, we can prune all branches for which $v_y \geq 1 \in \mathcal{L}_E(x)$. We call this method, *simple backtracking* which can exponentially decrease the size of the search space by pruning half of the branches each time the algorithm detects a clash. For example, for an arbitrary $\mathcal{L}(x)$, by pruning all the branches where $v_y \geq 1$, we will in fact prune $2^{|\mathcal{V}_x|-1} = 2^{2^{p+q}-1}$ branches w.r.t. the *ch*-Rule, which amounts to half of the branches.

We can improve this by a more complex version of dependency-directed backtracking in which we prune all branches that have the same reason for the clash caused by v_y. We call this method *complex backtracking.* 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, w.l.o.g. 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.[10] This is due to the fact that whenever the arithmetic reasoner assigns to a variable v (for which $R_i \in \alpha(v) \wedge T_j \notin \alpha(v)$) occurring in $\mathcal{L}_E(x)$ a value $k \geq 1$, the *fil*-Rule will eventually fire for x and create a corresponding successor y with $R_i \in \mathcal{L}(\langle x,y \rangle)$ and $T_j \notin \mathcal{L}(\langle x,y \rangle)$. This would make the $\forall$-Rule and $\forall_\backslash$-Rule applicable to x and cause a clash for y.

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 requires a total of $p+q$ digits, the number of variables that must be zero will be 2^{p+q-2}. The $2^{p+q} - 2^{p+q-2}$ other variables are free to be constrained and will open two branches in the search space. Therefore, the number of branches will be reduced from $2^{|\mathcal{V}_x|}$ to $2^{3/4|\mathcal{V}_x|}$ which is a significant improvement. In fact, the atomic decomposition technique can be considered as a method to organize the search space and at the same time by means of numerical reasoning and proxy individuals remain unaffected by the value of numbers.

For example, consider the case when there are 7 number 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 $v_{\square 01 \square\square\square\square}$, where $\square \in \{1,0\}$, must be zero. Therefore, $2^{128-32} = 2^{3 \times 32}$ branches remain to be explored.

4.3.1. Backtracking in the Arithmetic Reasoner

Normally there could be more than one solution for a set of inequations. According to Lemma 23 in Section 3, when we have a solution with respect to a set of restrictions 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 number 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 number restrictions are not satisfiable together.

4.3.2. Backjumping: Standard vs. Hybrid Algorithm

By comparing the effect of dependency-directed backtracking on the hybrid and 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 standard algorithms we have three sources of nondeterminism: the $\sqcup$-Rule, the *choose*-Rule, and the $\leq$-Rule. Therefore, in contrast to standard algorithms

[10]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 $\{\forall R.(A \sqcup \neg B), \forall S.(B \sqcup \neg C), \forall T.(C \sqcup \neg A)\} \subseteq \mathcal{L}(x)$ 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 branches in the search space for which $v \geq 1$ and $R \in \alpha(v) \wedge S \in \alpha(v) \wedge T \in \alpha(v)$ will end up with the same clash.

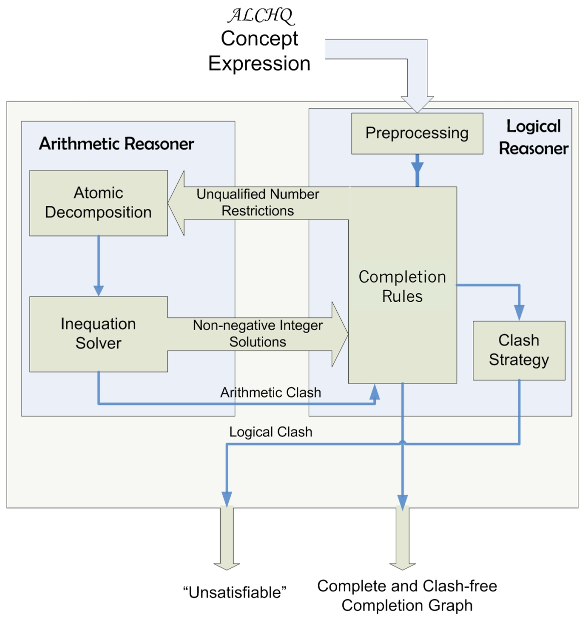

Fig. 8. Overall reasoner architecture.

which have three nondeterministic 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 in standard algorithms is integrated into one rule, the variable *ch*-Rule, in the hybrid algorithm.

5. Architecture of the Prototype Reasoner

As illustrated in Figure 8, the hybrid reasoner is composed of two major modules: the logical module and the arithmetic module. The input of the reasoner is an $\mathcal{ALCHQ}$[11] concept expression. The output of the algorithm is "satisfiable" provided a complete and clash-free completion graph[12] has been constructed or otherwise "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 6). The 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 [17]. Although choosing Java as the programming language gave us the opportunity to utilize the OWL-API, the performance of the reasoner was significantly affected by the overhead due to Java features such as garbage collection, no major (compiler-level) optimizations in numerical functions, and inefficiencies in representing fractional numbers. On the other hand, no other major optimization techniques were implemented.

[11]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 implemented in the prototype reasoner.

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

Algorithm 1 $main(state_1)$

```
$root = state_1$
$preprocess(state_1)$
**if** $expand(state_1) = TRUE$ **then**
    **return** satisfiable
**else**
    **return** unsatisfiable
```

Algorithm 2 $expand(state)$

```
**if** $state$ contains clash **then**
    $updateVariables(Backtracking\_Strategy)$
**else**
    **for all** $rule$ in Rule-Priority-List **do**
        $expandedList = apply(state, rule)$
        **if** $expandedList = null$ **then**
            $close(state)$
            **if** $state$ has no clash **then**
                **return** $TRUE$
        **else**
            **for all** $state'$ in $expandedList$ **do**
                $expand(state')$
```

The overall structure of the control flow for the whole system is sketched in Algorithms 1 and 2. The hybrid algorithm, by expanding the nodes using the implemented completion rules, tries to find a clash-free completion graph for the input concept. Also, whenever it encounters a clash, the hybrid algorithm prunes the search space based on the backtracking strategy it had adopted. The following sections describe the two major modules in more detail.

5.1. Logical Module

The logical module can be considered as the main module which performs the completion rules and calls the arithmetic reasoner whenever needed. It is composed of a preprocessing component which

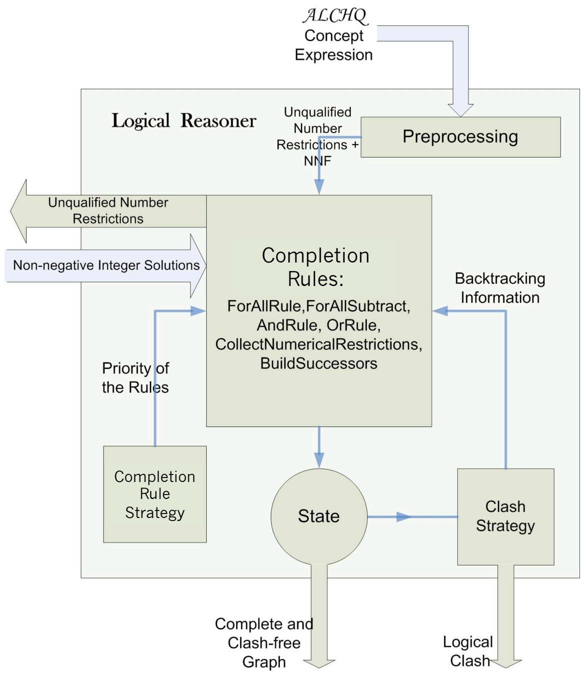
$\mathcal{ALCHQ}$ Concept Expression
Logical Reasoner
Unqualified Number Restrictions + NNF
Preprocessing
Unqualified Number Restrictions
Non-negative Integer Solutions
Completion Rules:
ForAllRule,ForAllSubtract, AndRule, OrRule, CollectNumericalRestrictions, BuildSuccessors
Backtracking Information
Priority of the Rules
Completion Rule Strategy
State
Clash Strategy
Complete and Clash-free Graph
Logical Clash

Fig. 9. Detailed architecture of the logical module.

modifies the input ontology (.owl file) based on the *unQ* function (see Definition 3). 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 9 is composed of a set of completion rules, clash strategy component, and some other auxiliary components.

The major data structure in the logical module is a *state* which records the 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 nondeterministic rule (such as $\sqcup$-Rule) can generate more than one child for a state. For example, if the reasoner fires the $\sqcup$-Rule for $C_1 \sqcup C_2 \sqcup \cdots C_n$ for an individual x in the current state, *state₁*, it 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 nondeterministic rule, the output would be a single path of states. Moreover, every state contains all the information

Algorithm 3 $canApply(state, individual, rule)$

```
if $state$ contains $individual$ for which $rule$ is applicable then
    return $true$
else
    return $false$
```

Algorithm 4 $apply(state, rule)$

```
$newState \leftarrow Copy(state)$
$newState.parent \leftarrow state$
for all $individual$ in $newState$ do
    if $canApply(newState, individual, rule)$ then
        $newInd \leftarrow$ apply $rule$ on $indvidual$
        replace $individual$ with $newInd$ in $newState$
return $newState$
```

about its individuals, including their label, their cardinality, and the label of their edges.

The set of implemented completion rules is based on the completion rules presented in Figure 2. However, since the logical module has no information regarding the variables and inequations, the *ch*-Rule is implemented in the arithmetic module. All the rules follow the general templates in Algorithms 3 and 4. 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, each rule will be fired for all individuals for which it is applicable. The logical module tries to apply the completion rules in accordance to their priority to every state that is not closed. A state is called *closed* (*clashed*) if its corresponding completion graph is complete (contains a clash). If no rule is applicable to a state, it will be closed. If all of the states have clashed and are closed, the input concept expression is unsatisfiable.

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

Without backtracking: There are two rules which together function as the *fil*-Rule, the *ch*-Rule, the $\leq$-Rule, and the $\geq$-Rule:

– The Collect-And-Initiate-Rule collects all unqualified number restrictions in the label of each individual in *state₁* and calls the arithmetic reasoner. The arithmetic reasoner computes all cases for the variables based on the *ch*-Rule and returns all possible non-negative integer solutions in a list. This rule stores the list of solutions in *state$_2$* which is a child of *state$_1$*.

– The Build-Arithmetic-Results-Rule which is a nondeterministic 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 *state$_2$* and creates a new state for each solution. For example, if there exist n different solutions for an individual x in *state$_2$*, this rule creates n new states as children of *state$_2$* and in each of them expands one solution.

With backtracking: In this case the reasoner does not search for all 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 try to search for a new solution. There are two rules responsible for this task:

– The Collect-And-Create-Rule, similar to the Collect-And-Initiate-Rule with the lowest priority, collects all number restrictions for each individual in *state$_1$*. Furthermore, it calls the arithmetic reasoner which returns the first solution it finds and generates successors of individuals in *state$_2$* based on this solution.

– For a detailed description of the *Build-in-Sibling-Rule*, assume an individual x in *state$_1$* that has a set of number restrictions according to which the Collect-And-Create-Rule has created a set of successors $y_1, y_2, \ldots, y_n$ in *state$_2$*. We call *state$_1$* the *generating* state of the y_i. All of the rules may modify labels of the y_i in the succeeding states. For instance, if y_1 ends up with a clash in all the paths of states starting from *state$_2$*, the reasoner can conclude that y_1 caused a clashed state and therefore the corresponding solution in *state$_1$* 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 an individual such as y_1 in a clashed state, it determines its generating state which is *state$_1$* in this case. Furthermore, it calls the arithmetic reasoner in *state$_1$* and sets the variable related to y_1 to zero and gets a new solution. Since all the paths from *state$_2$* will end up with a clash and no rule is applicable to it, *state$_2$* will be automatically closed. Afterwards, this rule will generate new successors of x in a new child state of *state$_1$* (if any solution exists).

The *Completion Rule Strategy* component imposes an order on the rules and they are prioritized by their order as in the following. In other words, the logical reasoner, before applying a rule in *state$_1$*, ensures that no rule with a higher priority is applicable.

1. The For-All-Rule ($\forall$-Rule).
2. The For-All-Subtract-Rule ($\forall_\backslash$-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 *state$_1$*. If none of the first three rules is applicable to any of the individuals in the *state$_1$*, 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 to *state$_1$* and creates 6 states as children of *state$_1$* 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 functioning of the last two rules is slightly different from their corresponding tableau rule. Consider the case when we have simple backtracking enabled and the logical module uses the Build-in-Sibling-Rule and the Collect-And-Create-Rule. For example, if $\{\geq 2\, R, \leq 4\, S, \geq 3\, T\} \subseteq \mathcal{L}(x)$ in *state$_1$* and x has no successors, the Collect-And-Create-Rule passes the set $\{\geq 2\, R, \leq 4\, S, \geq 3\, T\}$ to the arithmetic reasoner and receives either *"no solution"* which means that *state$_1$* 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 10).

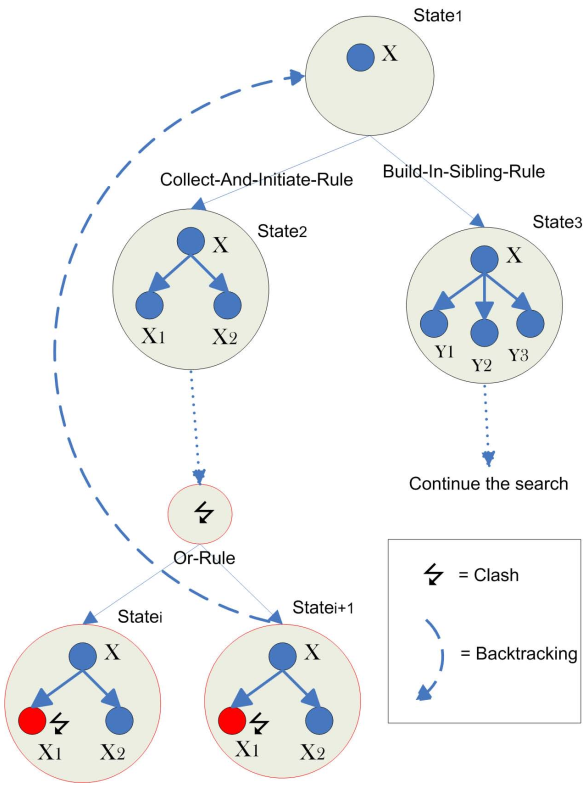

Fig. 10. Illustration of the rules application when backtracking.

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. It asserts R and S in the label of the role from x to x_1 and similarly R, S, and T in the label of the role from x to x_2. Also, it sets $card(x_1) = 2$ and $card(x_2) = 1$. Notice that all information in *state₁* will be copied to *state₂* before generating any new individual.

Later on, assume the individual x_1 ends up with a clash in $state_{i+1}$ and all other possible states (such as in $state_i$). 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 $state_1$, $state_3$. Whenever the arithmetic reasoner cannot find another solution for the list of numerical restrictions for x, the $state_1$ 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 10 illustrates the functioning of these two rules in this example.

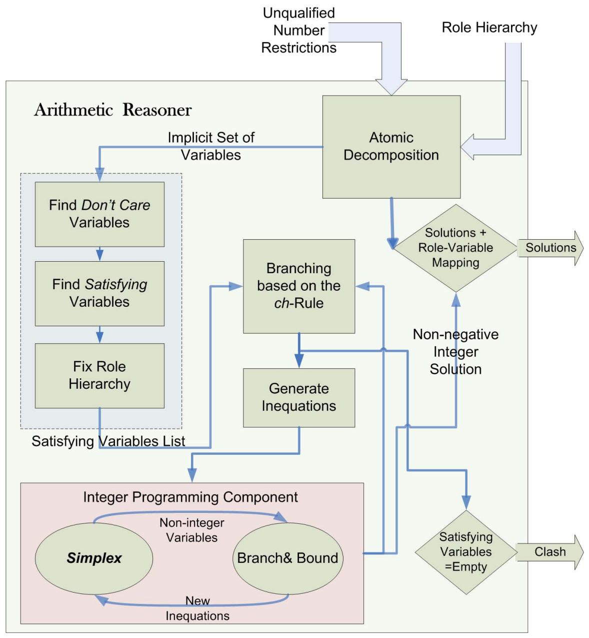
Unqualified Number Restrictions
Role Hierarchy
Arithmetic Reasoner
Implicit Set of Variables
Atomic Decomposition
Find Don't Care Variables
Find Satisfying Variables
Fix Role Hierarchy
Branching based on the ch-Rule
Solutions + Role-Variable Mapping
Solutions
Non-negative Integer Solution
Generate Inequations
Satisfying Variables List
Integer Programming Component
Non-integer Variables
Simplex
Branch& Bound
New Inequations
Satisfying Variables =Empty
Clash

Fig. 11. Detailed architecture of the arithmetic module.

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, or (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”.

5.2. *Arithmetic Module*

The major function of the arithmetic module 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 Section 3. Firstly, in addition to an inequation solver, it implements the *ch*-Rule. Moreover, it contains a few heuristics to guide the search for a non-negative integer solution. In the following we describe the architecture of the arithmetic module which is illustrated in Figure 11. Furthermore, we describe the functionality of each component in more detail using pseudo code snippets.

5.2.1. *Integer Linear 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 variables are required to have integer values, the problem is called *Integer Programming* (IP) or Integer Linear Programming (ILP).

Definition 26 (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: maximize (minimize) C^TX subject to $AX \leq b$ where the x_i can only have integer values and $X^T = [x_1\, x_2 \cdots x_n]$, C is the matrix of coefficients in the goal function, $A_{m\times n}$ is the matrix of coefficients in the constraints, and $b^T = [b_1\, b_2 \cdots b_m]$ contains the limit values in the inequations.

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.[13] 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 [5]. Although LP is known to be solvable in polynomial time, the Simplex method can behave exponentially for certain problems.

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

1. **Branch-and-bound:** Whenever a fractional value appears in the solution set, this method splits the search into 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 constraint $x_3 \leq 2$ 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 all 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 but 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.

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

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

Atomic Decomposition: Let *UNR* be the set of input unqualified number restrictions. After reading *UNR*, the arithmetic module determines the number of different number restrictions which will later be the number of inequations. In the following we assume that the size of *UNR* is equal to n. Therefore, the arithmetic module implicitly considers $2^n - 1$ variables such that for $UNR = \langle R_1, R_2, \ldots, R_n\rangle$ 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 $\alpha(v_{101}) = \{R_1, R_3\}$ and $\alpha(v_{1110}) = \{R_2, R_3, R_4\}$. To retrieve the role names related to a variable, the arithmetic module uses the *getRelatedRoles* function which has the same output as α.

5.2.2. 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 type of a variable (*v_type*) such that:

Algorithm 5 *find-don't-care-variables(UNR)*

```
for $i = 1$ to $n$ do
  if $UNR[i]$ is an at-least restriction then
    for all $j = 1$ to $2^n - 1$ such that its $i$th digit
    in binary coding $= 1$ do
      for $k = 1$ to $n$ do
        if ($k$th digit of $j$) $= 1 \wedge UNR[k]$ is not
        an at-most restriction then
          $v\_type(v_j) = -1$
      if $v\_type(v_j) \neq -1 \wedge v\_type(v_j) \neq 2$ then
        add $v_j$ to $satisfyingVariablesList$
```

- $v\_type(v) = 2$ if v must be zero due to logical reasons and cannot take any value other than zero,
- $v\_type(v) = 0$ if v is decided to be zero by the *ch*-Rule which can be changed later by the *ch*-Rule,
- $v\_type(v) = 1$ if v is decided to be greater or equal to 1 by the *ch*-Rule, and
- $v\_type(v) = -1$ if 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 type of 2.

The following tasks are performed by Algorithm 5 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 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.

Find satisfying variables: In order to find an arithmetic solution for the input *UNR* 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-variables* function presented in Algorithm 5 retrieves *don't care* variables and sets their type to -1. Moreover, whenever a variable v is neither *don't care* nor $v\_type(v) = 2$, this function adds it to the *satisfying variable* list.

Remark It is worth noticing that the order in which the variables are added to the *satisfying variables* list can significantly affect the performance of the arithmetic reasoner. In fact, by choosing the variables first 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 optimal ordering of the variables.

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

$v_{0001} + v_{0011} + v_{0101} + v_{0111} + \underline{v_{1001}} + v_{1011} + v_{1101} + v_{1111} \geq n_1$

$v_{0010} + v_{0011} + v_{0110} + v_{0111} + \underline{v_{1010}} + v_{1011} + v_{1110} + v_{1111} \geq n_2$

$v_{0100} + v_{0101} + v_{0110} + v_{0111} + \underline{v_{1100}} + v_{1101} + v_{1110} + v_{1111} \geq n_1$

$v_{1000} + \underline{v_{1001}} + \underline{v_{1010}} + v_{1011} + \underline{v_{1100}} + v_{1101} + v_{1110} + v_{1111} \leq m$

In the set of variables from v_1 to v_{1111}, the variables with the 1st digit (from right) equal to 1 are restricted by the universal restriction related to $UNR[1]$ (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 $UNR[4]$[14].

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 at-least restriction such as v_{1001}, v_{1010} and v_{1100} (see the underlined variables above). But this case is exactly similar to the standard tableau algorithms presented in Section 4.1.1. 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 $n_1 + n_2 + n_3 > m$).

[14]For every qualified at-least restriction $(\geq n\, R.C)$, we will have $\geq n\, R' \sqcap \forall R'.C$. 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 at-most restrictions, we will replace $\leq m\, R.C$ by $\leq m\, R' \sqcap \forall R \backslash R'.\neg C$. Therefore, the absence of R' and accordingly, the appearance of 0 in its related digit will invoke the universal restriction $\forall R \backslash R'.C$

Algorithm 6 *fix-role-hierarchy*($\mathcal{R}$)

```
  for all $R \sqsubseteq S \in \mathcal{R}$ do
    for all $v$ such that $R$ related to $v$ and $S$ not related to $v$ do
      set $v\_type(v) = 2$
```

Another strategy could be starting from variables that occur in many 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 *minimum number of successors* model property[15], (iii) although the probability of a logical clash may be high due to many restrictions, by choosing a highly restricted variable and after having detected a clash, we can set many more variables to zero to improve backtracking.

Fix role hierarchy: By means of the *fix-role-hierarchy* function (see Algorithm 6), the arithmetic module sets the type 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 6 sets $v\_type(v) = 2$. This algorithm implements the *hierarchy*-Rule.

Backtracking results: In the case of simple backtracking (see Section 4.3 for both backtracking types), the arithmetic module only needs to set the type of the variable related to the clashed individual to 2. In the case of complex backtracking, if the logical module discovers 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 type 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 are 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. More precisely, for each at-least restriction $\geq n\,R$ we create n R-successors and we can be sure that this model will not fail due to number restrictions for these successors.

[15]A model has the *minimum number of successors* property iff for all of its individuals one cannot reduce the number of successors without causing a clash.

Algorithm 7 *Heuristics*(*UNR*)

```
  {Assume $N$ UNRs such that UNR[1] to UNR[M] are at-least restrictions}
  for $i = 1$ to $M$ do
    $sumOfLimits \leftarrow sumOfLimits + UNR[i].limit$
  $MinAtMost \leftarrow \infty$
  for $i = M$ to $N$ do
    if $UNR[i].limit < MinAtMost$ then
      $MinAtMost \leftarrow UNR[i].limit$
  if $sumOfLimits \leq MinAtMost$ then
    for $i = 1$ to $M$ do
      $value(v_{2^i}) \leftarrow UNR[i].limit$
```

Assuming N UNRs, M of which are at-least restrictions, the procedure presented in Algorithm 7 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 a minimal number of successors.

5.2.3. Branching

After finalizing the *satisfying variables* list, the main function starts the application of the *ch*-Rule. As presented in Algorithm 8, the *branching* function starts assigning the type of 1 (i.e., being greater or equal 1) to the satisfying variables. 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 type of the variables. In the case of disabled backtracking, the branching function tries all 2^k cases and returns all 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, $v\_type(v) = 1$ changes to $v\_type(v) = 2$ 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 8 assumes the use of backtracking. Note that different backtracking strategies will result in different values for the *satisfyingVL* list as input for the *branching* algorithm.

Algorithm 8 *branching*(*satisfyingVL*)

```
{Branch over the list of satisfying variables ($satisfyingVL$) based on the $ch$-Rule}
if $satisfyingVL = \emptyset$ then
  return $null$
else
  $inequations \leftarrow build\_inequations(satisfyingVL)$
  $result \leftarrow IntegerProgramming(inequations)$
  if $result \neq null$ then
    return $result$
  else
    $branchingVariable \leftarrow$ remove last element of $satisfyingVL$
    $v\_type(branchingVariable) \leftarrow 1$
    $result \leftarrow branching(satisfyingVL)$
    if $result \neq null$ then
      return $result$
    else
      $v\_type(branchingVariable) \leftarrow 0$
      return $branching(satisfyingVL)$
```

5.2.4. Integer Programming

The integer programming or the equation-solver component gets a set of linear inequations as input. The goal function is always to minimize the sum of all variables, while all variables must be greater or equal zero. The set of constraints imposed by the type of the variables will also be part of the input in form of inequations. In other words, if $v\_type(v) = 1$ for a variable, we will have $v \geq 1$ as a part of the input. Notice that in the cases where $v\_type(v) = 0$ or $v\_type(v) = 2$ the variable v never appears in the set of input inequations.

The integer programming component is composed of a linear programming algorithm according to the *Simplex* method presented in [5] and branch-and-bound to obtain integer solutions when the linear solution contains fractional values.

6. Evaluation

In this section we present the empirical results obtained from an implemented prototype as described in the previous section. 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 number restrictions. Consequently, based on these parameters we build a set of benchmarks for which we evaluate the hybrid reasoner.

6.1. Benchmarking

One major problem with benchmarking in OWL is due to the fact that there exist not many comprehensive real-world OWL ontologies to utilize as benchmarks. In fact, as stated in [37], 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 the forthcoming OWL 2 [31]. 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 $\mathcal{SHQ}$ benchmark available which contains non-trivial qualified number restrictions. Furthermore, our prototype reasoner does not implement any optimization technique except the one introduced above targeting reasoning with the $\mathcal{Q}$-component of $\mathcal{SHQ}$. So, any $\mathcal{SHQ}$ knowledge base requiring other optimization techniques than the one implemented in our reasoner would not demonstrate any improvements, especially if the difficulty is due to the missing optimizations techniques and not due to the occurring number restrictions. Accordingly, we needed to build a set of synthetic test cases to empirically evaluate the hybrid reasoner. Although they are synthetic and did not emerge (yet) from real-world ontologies, we claim that they reflect patterns that are very likely to be encountered in forthcoming OWL 2 ontologies.

For these reasons we focus our evaluation on concept expressions only containing qualified number restrictions. We identify the following parameters that may affect the complexity of reasoning with number restrictions:

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

6.2. Evaluation Results

In this section we briefly examine the performance of the hybrid reasoner with respect to the parameters identified above. Moreover, we present an evaluation to examine the effectiveness of the two proposed backtracking techniques.

Tableau reasoning in expressive DLs is known to be a very time and 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 [4].

Well-known reasoners that support qualified number restrictions such as Racer [14], FaCT++ [34] or Pellet [33] implement numerous optimization techniques. Therefore, their performance is not fairly comparable to this prototype. Accordingly, we base our evaluations only partially on a comparison with one of the existing reasoners and focus on the study of the behavior of the hybrid reasoner. Moreover, it is well-known that benchmarks can only compare systems but not algorithms because benchmark results are often affected by factors (e.g., index structures, heuristics, etc) that do not reveal any insight in comparing the efficiency of algorithms, e.g., the standard vs. hybrid algorithm for number restrictions.

The following experiments were performed under Windows XP Professional (32-bit) on a standard PC with an Intel Core 2 Duo E6400 processor and 3 GB of physical memory. To improve the precision, every test was executed in five independent runs and the average of these runs is presented. Furthermore, we set the timeout limit to 1000 seconds.

6.2.1. Increasing Values of 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 number restrictions to a set of inequations. For example, for the concept expression $\geq 3\, \mathit{hasChild}.\mathit{Female}$ the size of the number is three which is relatively small. However, when expressing the concept $(\geq 141\, \mathit{hasCredit} \sqcap \leq 45\, \mathit{hasCredit}.\mathit{ComputerScience})$ to model a university undergraduate engineering program or $(\geq 1200\, \mathit{hasSeat} \sqcap \leq 600\, \mathit{hasSeat}.(\mathit{Arena} \sqcup \mathit{GrandCircle}))$ 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 chose Pellet[16] as a representative implementation of the standard algorithm are:

– it is a free open-source reasoner that handles (qualified) number restrictions,
– similar to our prototype it is a Java-based implementation, and
– in contrast to FaCT++ which sometimes turned out to be unsound when dealing with number restrictions, Pellet returned correct answers in all our experiments.

Furthermore, to the best of our knowledge Pellet as well as FaCT++ have no specific optimization technique for dealing with qualified number restrictions. Therefore, since the goal is to compare implementations of the hybrid and standard algorithm, we considered Pellet's implementation of the standard algorithm as a adequate representative for a state-of-the-art DL reasoner.

Test case description: The concept expressions for which we executed the concept satisfiability test are as follows (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}.

$$(\geq 2i\, RS.(A \sqcup B)) \sqcap (\leq i\, S.A) \sqcap (\leq i\, R.B) \sqcap (\leq (i-1)\, T.(\neg A)) \sqcup (\leq i\, T.(\neg B)) \qquad (C_{SAT})$$
$$(\geq 2i\, RS.(A \sqcup B)) \sqcap (\leq i\, S.A) \sqcap (\leq i\, R.B) \sqcap (\leq (i-1)\, T.(\neg A)) \sqcup (\leq (i-1)\, T.(\neg B)) \qquad (C_{UNSAT})$$

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 $(\leq i\, S.A)$ and $(\leq i\, R.B)$). 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 $(\leq (i-1)\, T.(\neg A)) \sqcup (\leq i\, T.(\neg B))$ can be satisfied when choosing $(\leq i\, T.(\neg B))$ 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 satisfiable, neither by the hybrid nor the standard algorithm. In the hybrid algorithm, the set of inequations is only satisfied in the case where all variables are zero except $v, v' \geq 1$ and $\alpha(v) = \{RS', S', T'\}$ and $\alpha(v') = \{RS', R'\}$.[17] The standard algorithm in

[16]We used Pellet 2.0 RC7 from June 11, 2009.

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

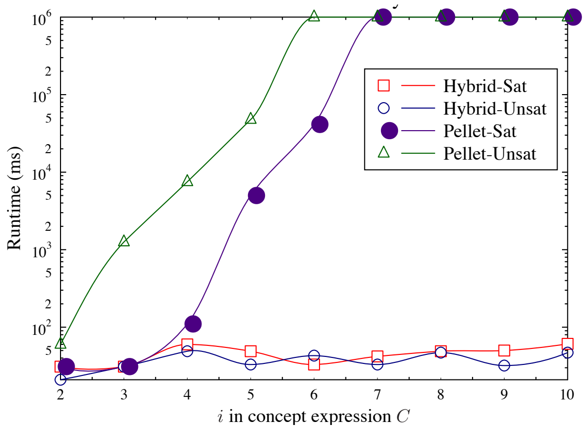

Fig. 12. Comparing the hybrid with the standard algorithm: Effect of increasing the value of numbers.

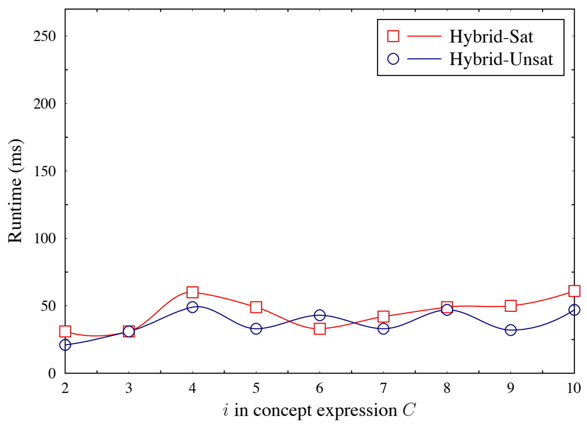

Fig. 13. Hybrid reasoner: Effect of increasing the value of numbers.

order to examine the satisfiability of C_{SAT} creates $(2 \times i)$ RS-successors for x in $(A \sqcup B)$ and according to the 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 branches in this test case.

As illustrated in Figure 12, the linear growth of i from 2 to 10 has almost no effect on the hybrid reasoner while it dramatically increases the runtime of the standard algorithm for numbers as small as 6 and 7.[18] Moreover, we can observe that for $i = 6$ the satisfiability of C_{SAT} is decided in about 40s while for C_{UNSAT}, which is only slightly different to C_{SAT}, the time increases to more than 1000s. This sudden jump reveals that by decreasing i to $i-1$ in just one at-most restriction, which leads to unsatisfiability, the practical complexity of the problem increases tremendously. In Figure 13 we zoom into the hybrid part of the Figure 12 to present the behavior of the hybrid reasoner in more detail.

In contrast to the standard reasoner, the performance of the hybrid reasoner is unaffected by the value of the numbers. In Figure 14 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 an exponential growth

[18]Note that Figure 12 is a log-linear plot and time values are in logarithmic scale. In Figure 13 one can observe the behavior of the hybrid reasoner in a linear plot.

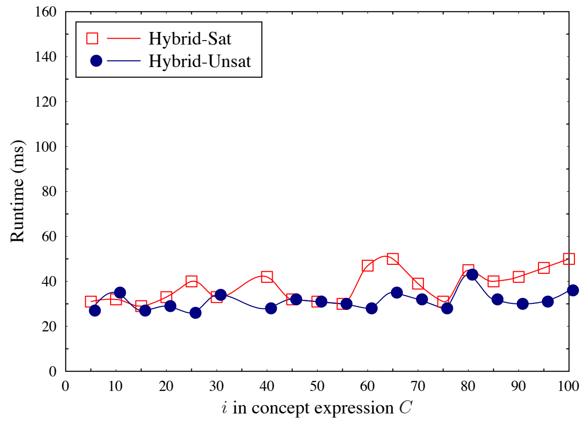

Fig. 14. Hybrid reasoner: Effect of linear growth of i.

of i, in Figure 15 we present the performance of the hybrid reasoner for $i = 10^n, n \in 1..6$.

6.2.2. Backtracking

One of the major well-known optimization techniques addressing complexity of the reasoning with number restrictions is dependency-directed backtracking or backjumping. In this experiment we observe the effect of backtracking on the performance of the hybrid reasoner. In three settings, we first turn off backtracking, second enable simple backtracking, and finally enable complex backtracking (see Section 4.3).

In order to observe the impact of backtracking, we tested an unsatisfiable concept D_{UNSAT} which follows the pattern where $D_j \sqcap D_k = \bot$ for $1 \leq j,k \leq i$, $j \neq k$ with respect to a role hierarchy $R \sqsubseteq T$: $(\geq 3\, R.D_1) \sqcap \ldots \sqcap (\geq 3\, R.D_i) \sqcap (\leq (3i-1)\, T)$.

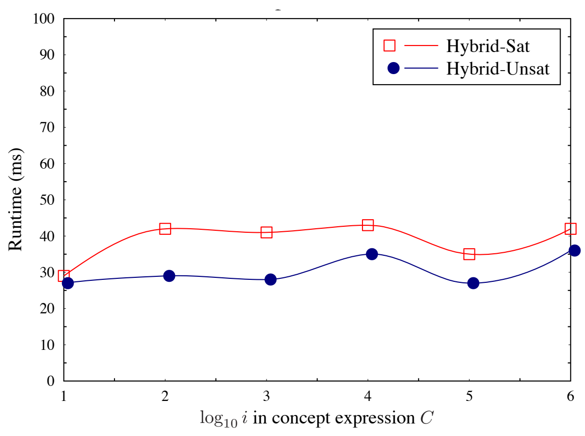

Fig. 15. Hybrid reasoner: Effect of exponential growth of i.

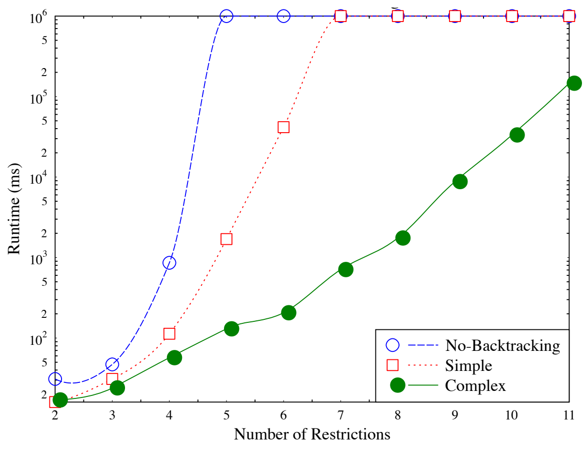

Fig. 16. Effect of backtracking at different levels.

The assertion $x : D_{UNSAT}$ implies that x has 3 R-successors in D_1, ..., 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 results in more number restrictions and therefore a larger number of variables. As the log-linear plot in Figure 16 suggests, the double-exponential nature of the hybrid algorithm and in general the high nondeterminism of the *ch*-Rule makes it inevitable to utilize at least simple backtracking. Moreover, we can conclude that by using complex backtracking we can improve the performance of the reasoning significantly. For example, in Figure 16 we can observe that for $i = 6$ reasoning without backtracking results in a timeout while benefiting from simple backtracking the reasoner concludes unsatisfiability in about 41s and for complex backtracking the reasoning time is reduced to 206ms.

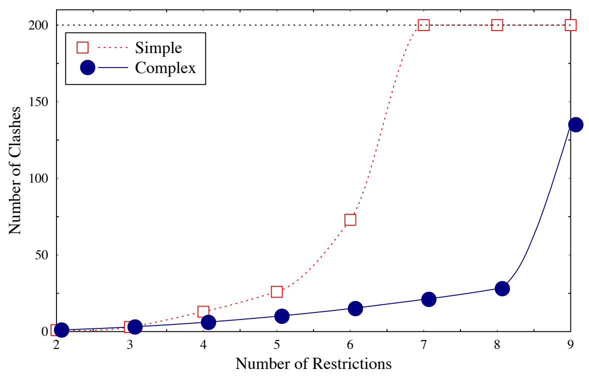

Fig. 17. Effect of backtracking at different levels: Number of clashes.

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 a smaller number of clashes. In Figure 17, by observing the number of logical clashes each method produces before returning the result, we can compare their success in pruning the search space.

6.2.3. 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 these concepts with E_i where $R \sqsubseteq T$ for $i = 1, 20, 40, \ldots, 220, 240$:

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

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 18, the standard algorithm can easily infer for $i \in 1..29$ that E_i is unsatisfiable since x has at least 30 distinguished successors. However, from E_{30} to E_{120} it becomes very time-consuming for the standard algorithm to merge all 120 successors into i individuals. Moreover, Figure 18 shows that no matter which value

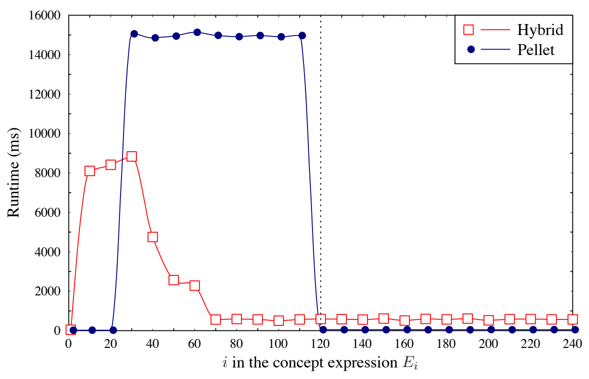

Fig. 18. Standard and Hybrid algorithm: Effect of (un)satisfiability for E_i.

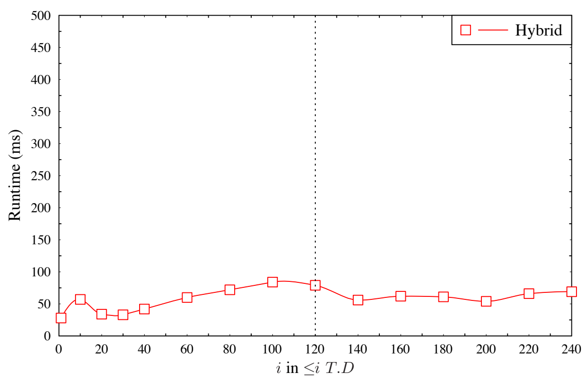

Fig. 19. Hybrid algorithm: Effect of (un)satisfiability for F_i.

i takes between 30 and 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 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 18 that for the hybrid algorithm $i = 1$ is a trivial case since not more than one variable can have the type 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 $i = 70$ and remains unchanged until $i = 120$. In fact, this unexpected behavior did not correspond to the formal analysis of the hybrid algorithm and needed 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 breaking point is the fact that for $i < 30$ no arithmetic solution exists for the set of inequations. Therefore, it seems to be very difficult for the arithmetic reasoner to realize 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 seems to be much stronger than in the arithmetic reasoner. Whenever more logical clashes exist, the hybrid reasoner can accomplish the reasoning faster.

In order to verify this hypothesis, we built another pattern which is slightly different from E_i and we abbreviate it with F_i where $R \sqsubseteq T$ for $i = 1, 20, 40, \ldots, 220, 240$:

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

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 19, the performance of the hybrid reasoner stays almost constant for $1 \leq i \leq 240$. It is worth noticing that the behavior of the standard algorithm for F_i remains exactly similar to E_i (not shown here).

6.2.4. *Number of Qualified Number Restrictions*

According to the complexity analysis of the hybrid algorithm in Section 4.1.2 one can conclude that the number of qualified number 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 number restrictions in the worst case.

In this experiment we built a concept containing one at-least restriction and extend it gradually. In order to keep the ratio between the number of at-least and at-most restrictions fixed, in each step

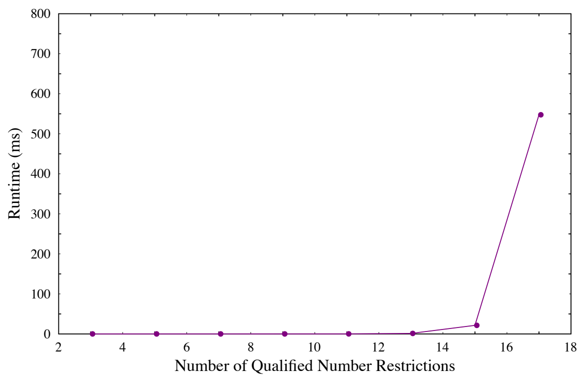

Fig. 20. Effect of increasing the number of qualified number restrictions.

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 following form with respect to the role hierarchy $\{RS \sqsubseteq R\}$:

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

Therefore, in each concept G_i we have $2i + 1$ number restrictions. It is essential for this problem that the roles participating in these number restrictions share the same role hierarchy. Otherwise, one could partition different role names from different role hierarchies and deal with each partition separately. Note that the hybrid algorithm encounters no clashes when deciding satisfiability of G_i. As presented in Figure 20, the maximum number of qualified number restrictions that the hybrid prototype currently can handle (in less than 1000s) is 17 although these concepts are easily satisfiable. The Pellet reasoner could decide the satisfiability of each concept G_i with a runtime well below one second. It is currently unclear whether this performance degradation of the hybrid reasoner is inevitable or could be avoided by an improved implementation. For instance, Algorithm 5, which finds *don't-care-variables*, still requires $\mathcal{O}(2^n)$ steps for n at-least restrictions. This dependency might be improved by better indexing techniques and might become linear to n with a smarter implementation.

6.2.5. Number of At-least vs. At-most Restrictions

In the following we denote the ratio between the number of at-least restrictions and at-most restrictions by $R_{Min/Max}$. In addition to the pure number of number restrictions, the ratio $R_{Min/Max}$ seems to also 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, i.e., the concept expressions are easily satisfiable.

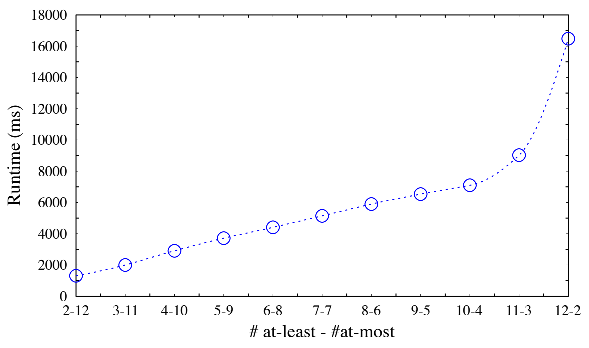

Fig. 21. Effect of the ratio between the number of at-least and at-most restrictions.

From Figure 21 we can conclude that the growth of $R_{Min/Max}$ increases the complexity of the reasoning for the hybrid reasoner while the Pellet reasoner could decide the satisfiability of these concept with a runtime well below one second. In fact, the hybrid reasoner tries to satisfy at-least restrictions while not violating any at-most restriction. Therefore, the length of the *satisfying variables* list, which is the list of variables for which the *ch*-Rule is applied, depends on the number of at-least restrictions.[19] 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 reasoner always computes 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 prototype accomplished the reasoning process in about 235 seconds. By ignoring the *minimal number of successors* property, this problem is trivial in the absence of at-most restrictions (see Section 5.2.2 where we presented a solution for this ineffi-

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

Table 2
Effect of number of at-least restrictions vs. number of at-most restrictions.

$\geq - \leq$	0-14	1-13	2-12	3-11	4-10	5-9	6-8	7-7	8-6	9-5	10-4	11-3	12-2	13-1	14-0
Time (s)	0.17	0.59	1.3	2	2.9	3.7	4.4	5.1	5.9	6.5	7.1	9	16.5	53	235

ciency, which is not yet implemented in our prototype).

7. Discussion and Future Work

Based on the evaluation results presented in Section 6 and the complexity analysis in Section 4 we identify the following advantages of the hybrid algorithm in comparison with the standard approaches:

– 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 number 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 the variables associated with partitions, the 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 4.3).
– Minimal 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 6.2.5, one major source of inefficiency in the hybrid reasoning is that it not only searches for a model but also always for a model with a minimal number of successors. This feature of the hybrid algorithm could be of interest 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 elements and therefore costs less is of great importance.

The following disadvantages of the hybrid algorithm can be observed:

– 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 nondeterministic *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 collects all number restrictions before generating any successor for an individual. 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 the advantages and disadvantages of the hybrid algorithm, one can conclude that this approach is inevitable whenever large numbers occur in numbers restrictions. This is also strongly supported by the results in [7] about a worst-case optimal tableau algorithm for $\mathcal{SHIQ}$ using algebraic reasoning and global caching. Moreover, the hybrid algorithm builds a well-structured search space which makes it well-suited for non-trivial concepts. However, in the case of trivially satisfiable concepts the current prototype performs slower than the standard algorithms. In other words, we can conclude that the overhead in the hybrid algorithm w.r.t. the current implementation is too high for trivial situations. Moreover, the fact that the number of the successors is minimized takes a possibly unnecessary extra effort.

As mentioned in the introduction, the signature calculus [13] was designed to address the problem of large numbers in number restrictions. However, it still has two highly nondeterministic rules (to split and merge proxy individuals) and does not process all at-most restrictions in one step. The algebraic method proposed in [30] 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.

Although the early work presented in [15] and implemented in Racer [14] deals with $\mathcal{SHQ}$, it is based on a recursive procedure not suited for formal proofs. Furthermore, it needs to examine the satisfiability of all partitions before initializing the Simplex component and does not support Aboxes. Whenever the number of qualified numbers restrictions grows and respectively the number of partitions exponentially grows, Racer becomes very inefficient.

We strongly believe that the current inefficiency of our implemented prototype w.r.t. the number of number restrictions should not be used to come to the premature conclusion that the hybrid algorithm is inefficient or even unusable for problems with many number restrictions. The inefficiency is due to the exponential growth of the variables in the inequations and the straight-forward implementation of the algebraic module which contains some algorithmic parts that still exhibit a best-case exponential behavior. A more advanced implementation could use standard techniques from linear programming such as (delayed) column generation (e.g., see [6]) that depends on the insight to consider only variables which have the potential to improve the objective function (in our case minimization of the sum of all variables occurring in the inequations of a given node in the completion forest). We conjecture that such an optimization technique would greatly improve the efficiency of the algebraic reasoner in the average case and would make the algorithmic parts obsolete that are currently best-case exponential.

We plan to address the following other topics in our future work.

- Turning off minimality: Since the *minimal number of successor* property is unnecessary in many cases, we could provide a switch to turn this property on and off. Therefore, whenever it is switched 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 $\geq n\, R.C$.
- Optimizing the arithmetic reasoner: In Section 6.2.3 we learned that one major source of inefficiency is due to the non-optimized arithmetic reasoner:
    - To optimize the arithmetic reasoner we could support 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 procedure. In fact, the Simplex module can *continue* its search for an integer solution, incrementally considering the newly discovered constraints on the variables.
    - One other possibility to improve the performance of the arithmetic reasoner is caching arithmetic solutions or clashes to avoid solving the same set of inequations more than once.
    - As explained in Section 5, 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 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 nominals affect all individuals. A hybrid calculus for $\mathcal{ALCOQ}$ is presented in [9,10]. In the presence of inverse roles the labels of individuals may be modified at any time. Therefore, a hybrid algorithm handling $\mathcal{SHIQ}$ will need to deal with incremental updates of labels of individuals due to the backpropagation of knowledge.

In general, one cannot expect that the algebraic tableau approach will work best for all possible input, especially since concept and Abox satisfiability are known to be ExpTime-complete for $\mathcal{SHQ}$. In Section 4.1 we illustrated that the worst-case complexity of the hybrid tableau is a function of the number of variables and, thus, of the number of at-least and at-most restrictions, while the complexity of the standard tableau is a function of the size of the numbers used in qualified number restrictions and the number of at-most restrictions. Given this analysis, in the future, after having improved the performance of the algebraic tableau approach, it seems to be promising to explore a combination with an alternative calculus for number restrictions such as the signature calculus [13], which pioneered the use of proxy individuals and can be considered as an improvement over standard tableau for dealing with number restrictions. A heuristic could be devised that decides whether the algebraic tableau or the signature calculus should be applied to a given problem.

A calculus equipped with the ability of informed arithmetic reasoning could be a motivation for introducing more complex numerical DL constructors. As proposed in [30], one possible extension could be used to restrict the ratio of cardinalities of fillers of two different roles. For example, in a taxonomy describing the structure of a theater, the concept expression $50 \times |hasRoom.WashRoom| \geq |hasSeat.\top|$ could express the restriction that there must exist at least one washroom for every 50 seats. Similarly, a percentage restriction such as $\leq 20\%\, hasCredit.Business$ could describe a concept where at most 20% of the *hasCredit* fillers are instances of the concept *Business*.

Since qualified number restrictions can be translated to inequations, such expressions can be transformed to linear inequations. However, the decidability of a DL extended by such constructors needs to be investigated.

Acknowledgements

We express our gratitude to our colleagues Jocelyne Faddoul and Ralf Möller for their valuable advice during this investigation. We are also very thankful for the very detailed and thoughtful reviews by the anonymous reviewers that greatly helped to improve this article.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook, 2nd edition.* Cambridge University Press, 2007.

[2] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H. Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.

[3] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[4] S. Bechhofer. DIG optimisation techniques, March 2006. `http://dl.kr.org/dig/optimisations.html` (last visited July 2009).

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[6] G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors. *Column Generation.* Mathematics of Decision Making. Springer-Verlag, 2005.

[7] Y. Ding. *Tableau-based Reasoning for Description Logics with Inverse Roles and Number Restrictions.* PhD thesis, Department of Computer Science and Software Engineering, Concordia University, April 2008. Available at `http://users.encs.concordia.ca/~haarslev/students/Yu_Ding.pdf`.

[8] J. Faddoul, N. Farsinia, V. Haarslev, and R. Möller. A hybrid tableau algorithm for ALCQ. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008), Patras, Greece, July 21-25*, pages 725–726, 2008. An extended version appeared in Proceedings of the 2008 International Workshop on Description Logics (DL-2008), Dresden, Germany, May 13-16, 2008.

[9] J. Faddoul, V. Haarslev, and R. Möller. Hybrid reasoning for description logics with nominals and qualified number restrictions. Technical report, Institute for Software Systems (STS), Hamburg University of Technology, 29 pages, 2008. See also `http://www.sts.tu-harburg.de/tech-reports/papers.html`.

[10] J. Faddoul, V. Haarslev, and R. Möller. Algebraic tableau algorithm for $\mathcal{ALCOQ}$. In *Proceedings of the 2009 International Workshop on Description Logics (DL-2009), Oxford, United Kingdom, July 27–30*, 2009.

[11] N. Farsiniamarj. Combining integer programming and tableau-based reasoning: A hybrid calculus for the description logic $\mathcal{SHQ}$. Master's thesis, Concordia University, Department of Computer Science and Software Engineering, 2008. Available at `http://users.encs.concordia.ca/~haarslev/students/Nasim_Farsinia.pdf`.

[12] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[13] V. Haarslev and R. Möller. Optimizing reasoning in description logics with qualified number restriction. In *Proceedings of the International Workshop on Description Logics (DL'2001), Aug. 1-3, 2001, Stanford, CA, USA*, pages 142–151, August 2001.

[14] V. Haarslev and R. Möller. RACER system description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy*, Lecture Notes in Computer Science, pages 701–705. Springer-Verlag, June 2001.

[15] V. Haarslev, M. Timmann, and R. Möller. Combining tableaux and algebraic methods for reasoning with qualified number restrictions. In *Proceedings of the International Workshop on Description Logics (DL'2001), Aug. 1-3, Stanford, USA*, pages 152–161, 2001.

[16] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *Second International Conference on Principles of Knowledge Representation, Cambridge, Mass., April 22-25, 1991*, pages 335–346, April 1991.

[17] M. Horridge, S. Bechhofer, and O. 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.

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

[19] I. Horrocks. Implementation and optimization techniques. In Baader et al. [1], chapter 9.

[20] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

[21] I. Horrocks and U. Sattler. A tableau decision procedure for $\mathcal{SHOIQ}$. *JAR*, 39(3):249–276, 2007.

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

[23] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[24] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In D. MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, Lecture Notes in Computer Science, pages 482–496, Germany, 2000. Springer Verlag.

[25] I. Horrocks and S. Tobies. Reasoning with axioms: Theory and practice. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'2000), Breckenridge, Colorado, USA, April 11-15, 2000*, pages 285–296, April 2000.

[26] Y. Kazakov, U. Sattler, and E. Zolin. How many legs do I have? Non-simple roles in number restrictions revisited. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2007, Yerevan, Armenia, Oct. 15-19*, volume 4790 of *LNAI*, pages 303–317. Springer-Verlag, 2007.

[27] H. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, November 1983.

[28] E. N. Marieb, K. Hoehn, P. B. Wilhelm, and N. Zanetti. *Human Anatomy & Physiology: International Edition with Human Anatomy and Physiology Atlas, 7/E.* Pearson Higher Education, 7 edition, 2006.

[29] H. Ohlbach and J. Koehler. Role hierarchies and number restrictions. In *Proceedings of the International Workshop on Description Logics (DL-97), Sept. 27 - 29, Gif sur Yvette (Paris), France*, 1997.

[30] H. Ohlbach and J. Köhler. Modal logics, description logics and arithmetic reasoning. *Artificial Intelligence*, 109(1-2):1–31, 1999.

[31] OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, W3C Working Draft, 02 December 2008, October 2008. `http://www.w3.org/TR/owl2-syntax/` (last visited July 2009).

[32] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[33] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: a practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2005.

[34] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[35] F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference, 2003. `http://www.w3.org/TR/owl-guide/` (last visited July 2009).

[36] M. Y. Vardi. Why is modal logic so robustly decidable? In N. Immerman and P. G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 149–184. American Mathematical Society, 1996.

[37] T. Weithöner, T. Liebig, M. Luther, S. Böhm, F. von Henke, and O. Noppens. Real-world reasoning with OWL. In *Proceedings of 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, LNCS 4519*, pages 296–310. Springer-Verlag, 2007.