Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Research
Report
RR-91-03

Qualifying Number Restrictions in Concept Languages

Bernhard Hollunder Franz Baader

February 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ❑ Intelligent Engineering Systems
- ❑ Intelligent User Interfaces
- ❑ Intelligent Communication Networks
- ❑ Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Qualifying Number Restrictions in Concept Languages

Bernhard Hollunder, Franz Baader

DFKI-RR-91-03

A short version of this paper will be published in the Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1991.

© Deutsches Forschungszentrum für Künstliche Intelligenz 1991

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Qualifying Number Restrictions in Concept Languages

Bernhard Hollunder and **Franz Baader**
German Research Center for Artificial Intelligence (DFKI)
Projectgroup WINO
Postfach 2080, W-6750 Kaiserslautern, Germany
E-mail: {hollunde, baader}@dfki.uni-kl.de

Abstract

We investigate the subsumption problem in logic-based knowledge representation languages of the KL-ONE family. The language presented in this paper provides the constructs for conjunction, disjunction, and negation of concepts, as well as qualifying number restrictions. The latter ones generalize the well-known role quantifications (such as value restrictions) and ordinary number restrictions, which are present in almost all KL-ONE based systems. Until now, only little attemps were made to integrate qualifying number restrictions into concept languages. It turns out that all known subsumption algorithms which try to handle these constructs are incomplete, and thus detecting only few subsumption relations between concepts.

We present a subsumption algorithm for our language which is sound and complete. Subsequently we discuss why the subsumption problem in this language is rather hard from a computational point of view. This leads to an idea of how to recognize concepts which cause tractable problems.

Contents

1 Introduction 2

2 The Language 5

3 Checking Satisfiability 7

4 Implementation 22

5 Remarks on the Complexity of Qualifying Number Restrictions 27

6 Conclusion 30

1 Introduction

Knowledge representation systems of the KL-ONE family such as KL-ONE [Brachman and Schmolze, 1985], BACK [Nebel, 1990], CLASSIC [Borgida et al., 1989], KANDOR [Patel-Schneider, 1984], KL-TWO [Vilain, 1985], KRYPTON [Brachman et al., 1985], LOOM [MacGregor and Bates, 1987], provide so-called *concept languages* for expressing taxonomical knowledge. Concept languages allow the definition of *concepts* which are built out of primitive concepts and roles. The primitive concepts are interpreted as sets of individuals and the roles are interpreted as binary relations between individuals. Starting with primitive concepts and roles one can build concepts using various language constructs. Concepts are again interpreted as sets of individuals.

To give an example, assume that person, male, and shy are primitive concepts, and child is a role. If constructs such as conjunction, disjunction, and negation of concepts are available in the concept language, one can express "persons that are male or not shy" by $\mathsf{person} \sqcap (\mathsf{male} \sqcup \neg \mathsf{shy})$. Since concepts are interpreted as sets, conjunction of concepts ($\sqcap$) can be interpreted as set intersection, disjunction of concepts ($\sqcup$) as set union, and negation of concepts ($\neg$) as set complement.

Qualifying number restrictions provide restrictions on roles which for instance allow to describe "individuals with at least two children who are shy" by

$$(\geq 2\ \mathsf{child}\ \mathsf{shy}),$$

and "individuals with at most three children who are male and not shy" by

$$(\leq 3\ \mathsf{child}\ (\mathsf{male} \sqcap \neg \mathsf{shy})).$$

It turns out that the well-known role quantifications and ordinary number restrictions, which are available in almost all concept languages, are special cases of qualifying number restrictions.

Role quantifications are of the form $\exists \mathsf{child}.\mathsf{male}$ and $\forall \mathsf{child}.\mathsf{shy}$ (or (SOME child male) and (ALL child shy) in a Lisp-like notation). These expressions can be read as "individuals having (at least) one male child" and "individuals for whom all children are shy", respectively. Obviously, $\exists \mathsf{child}.\mathsf{male}$ means the same as $(\geq 1\ \mathsf{child}\ \mathsf{male})$, and $\forall \mathsf{child}.\mathsf{shy}$ means the same as $(\leq 0\ \mathsf{child}\ \neg \mathsf{shy})$.

Ordinary number restrictions are of the form $(\geq n\ R\ \top)$ and $(\leq n\ R\ \top)$, where the special concept symbol $\top$ denotes the set of all individuals (of the interpretation). Qualifying number restrictions generalize these ordinary

number restrictions as follows: a specified number of role fillers for a role can be restricted to arbitrary concepts rather than only to $\top$.

The main reasoning facilities concerning the taxonomical knowledge in the above mentioned systems are the

- determination whether a concept is ***satisfiable***, i.e., whether a concept denotes a nonempty set in some interpretation, and the
- computation of the ***subsumption*** relation between concepts.

A concept C ***subsumes*** (is more general than) a concept D iff in every interpretation the set denoted by C is a superset of the set denoted by D. For example,

$$(\leq 1 \ \mathsf{child} \ (\mathsf{male} \sqcap \neg \mathsf{shy}))$$

subsumes

$$(\geq 2 \ \mathsf{child} \ (\mathsf{male} \sqcap \mathsf{shy})) \sqcap (\leq 3 \ \mathsf{child} \ \mathsf{male}),$$

since every individual with at least two male and shy children and at most three male children has at most one child which is male and not shy. This example demonstrates that, in contrast to other concept languages, already for small concepts it is not that apparent whether there exists a subsumption relation between them.

The subsumption problem can be reduced to the satisfiability problem if the concept language provides conjunction and negation of concepts. In fact, a concept C subsumes a concept D if and only if the concept $\neg C \sqcap D$ is not satisfiable. Thus, an algorithm which checks satisfiability of concepts also yields an algorithm which checks subsumption between concepts.

If one considers the subsumption algorithms used in the above mentioned KL-ONE systems, then it turns out that these algorithms are *incomplete*. A reason for this fact is that until recently only incomplete subsumption algorithms were known for nontrivial concept languages. An incomplete subsumption algorithm has the property that it sometimes fails to recognize that a concept subsumes another one. [Schmidt-Schauß and Smolka, 1988] were the first who gave a sound and complete subsumption algorithm for the concept language $\mathcal{ALC}$ which provides conjunction, disjunction, and negation of concepts, as well as role quantifications. Actually, they presented an algorithm for checking satisfiability of concepts. Given a concept C the algorithm tries to generate a finite interpretation such that the set denoted by C is nonempty. If this process fails, i.e., if a contradiction occurs, such an interpretation cannot exist and C is not satisfiable, otherwise C is satisfiable.

[Hollunder, Nutt, and Schmidt-Schauß, 1990] demonstrate that the ideas underlying the algorithm which checks satisfiability for the language $\mathcal{ALC}$ can be applied to languages using also other constructs such as ordinary number restrictions and conjunction of roles. Moreover, they argue that these ideas yield a general methodology for devising satisfiability (and hence subsumption) algorithms for concept languages.

The present paper pursues several goals. Firstly, it describes a sound and complete algorithm for checking satisfiability for a very expressive concept language containing not only conjunction, disjunction, and negation of concepts, but also qualifying number restrictions. Qualifying number restrictions are not just artificial constructs, but they are partly available in some systems. For example, the concept language used in KANDOR allows qualifying number restrictions in the restricted form $(\geq n\ R\ C)$ and $(\leq n\ R\ \top)$. Recently, qualifying number restrictions have been included into the assertional part ("A-Box") of the system MESON [Owsnicki-Klewe, 1990].

Secondly, the paper exemplifies the claim that there is in fact a general methodology for devising satisfiability algorithms for concept languages. However, devising a satisfiability algorithm for concept languages with qualifying number restrictions is of a rather different quality. The reason is that qualifying number restrictions allow to generate inherently complex term structures causing subtle combinatorial problems.

Finally, the paper investigates what kind of language constructs, and in what kind of combination may lead to long computations of the presented algorithm. This leads to an idea of how to single out concepts which can be checked fast on satisfiability.

The paper is organized as follows. In the next section we formally introduce the syntax and semantics of our concept language. In Section 3 we present a calculus for deciding satisfiability of concepts of this language. A functional algorithm which is extracted from the calculus is given in Section 4. In Section 5 we give ideas of how to recognize concepts which cause tractable problems. Finally, in Section 6 we discuss the claim that there is a general methodology for devising satisfiability (and hence subsumption) algorithms for concept languages.

2 The Language

We assume two disjoint alphabets of symbols, called *primitive concepts* and *roles*. We usually denote primitive concepts by the letter A and roles by the letter R. The special primitive concepts $\top$ and $\bot$ are called *top* and *bottom*. The set of *concepts* is inductively defined as follows. Every primitive concept is a concept. Now let C and D be concepts already defined, let R be a role, and let n be a nonnegative integer. Then

$C \sqcap D$	(conjunction)
$C \sqcup D$	(disjunction)
$\neg C$	(negation)
$(\geq n\ R\ C)$	(at-least restriction)
$(\leq n\ R\ C)$	(at-most restriction)

are concepts. The at-least and at-most restrictions are also called *qualifying number restrictions.*

An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of our language consists of a set $\Delta^{\mathcal{I}}$ (the *domain* of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the *interpretation function* of $\mathcal{I}$). The interpretation function maps every primitive concept A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, and every role R to a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The special concepts $\top$ and $\bot$ are interpreted as $\Delta^{\mathcal{I}}$ and the empty set, respectively. The interpretation function—which gives an interpretation for primitive concepts and roles—can be extended to arbitrary concepts as follows. Let C and D be concepts, let R be a role, and let n be a nonnegative integer. Assume that $C^{\mathcal{I}}$ and $D^{\mathcal{I}}$ are already defined. Then

$$
\begin{aligned}
(C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &:= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\geq n\ R\ C)^{\mathcal{I}} &:= \{a \in \Delta^{\mathcal{I}} \mid \left|aR^{\mathcal{I}} \cap C^{\mathcal{I}}\right| \geq n\} \\
(\leq n\ R\ C)^{\mathcal{I}} &:= \{a \in \Delta^{\mathcal{I}} \mid \left|aR^{\mathcal{I}} \cap C^{\mathcal{I}}\right| \leq n\}
\end{aligned}
$$

where $aR^{\mathcal{I}} := \{b \in \Delta^{\mathcal{I}} \mid (a, b) \in R^{\mathcal{I}}\}$, and $|X|$ denotes the cardinality of the set X.

Our language allows to express the well-known *role quantifications* (see e.g. [Nebel and Smolka, 1990]), which are present in almost all concept languages, as follows:

$$
\begin{aligned}
\forall R.C &:= (\leq 0\ R\ \neg C) \\
\exists R.C &:= (\geq 1\ R\ C).
\end{aligned}
$$

The construct $\forall R.C$ is usually called *value restriction*, and the construct $\exists R.C$ is e.g. called *c-some* in [Nebel, 1990]. Note that the concept $(\leq 0\ R\ \neg C)$ denotes the set of all elements for which every role filler for R is in C. This shows that the definition for the construct $\forall R.C$ we have given above coincides with the semantics usually given for this construct.

Now we are able to define the language $\mathcal{ALC}$ of [Schmidt-Schauß and Smolka, 1988] as sublanguage of our language. This concept language allows conjunction, disjunction, and negation of concepts, as well as role quantifications.

As already mentioned in the introduction, a concept C is called *satisfiable* iff there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}}$ is nonempty. We say C *subsumes* D iff $C^{\mathcal{I}} \supseteq D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$, and C is *equivalent* to D iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$.

Since our language allows conjunction and negation of concepts, satisfiability and subsumption of concepts can be reduced in linear time to each other.

Proposition 2.1 *Let C and D be concepts. Then:*

1. *C subsumes D if and only if $\neg C \sqcap D$ is not satisfiable.*

2. *C is satisfiable if and only if $\bot$ does not subsume C.*

Thus an algorithm for checking satisfiability of concepts can also be used for deciding subsumption of concepts and vice versa.

In the following section we present an algorithm that decides satisfiability of concepts. To keep this algorithm simple it is convenient to transform concepts into normal forms. We say a concept C is in *negation normal form* if negation signs in C appear only immediately in front of primitive concepts different from $\top$ and $\bot$. Negation normal forms can be computed using the following *simplification rules*:

$$
\begin{aligned}
\neg\top &\longrightarrow \bot \\
\neg\bot &\longrightarrow \top \\
\neg(C \sqcap D) &\longrightarrow \neg C \sqcup \neg D \\
\neg(C \sqcup D) &\longrightarrow \neg C \sqcap \neg D \\
\neg\neg C &\longrightarrow C
\end{aligned}
$$

$$\neg(\leq n\ R\ C) \longrightarrow (\geq (n+1)\ R\ C)$$
$$\neg(\geq n\ R\ C) \longrightarrow \begin{cases} \bot & \text{if } n = 0 \\ (\leq (n-1)\ R\ C) & \text{if } n > 0. \end{cases}$$

Let C be a concept. By $NNF(C)$ we denote the concept which is obtained from C by applying the simplification rules in top-down, left to right order as long as possible. For example,

$$NNF(\neg(A \sqcup (\geq 3\ R\ B))) = (\neg A \sqcap (\leq 2\ R\ B)).$$

The following result can be proved easily.

Proposition 2.2 *Let C be a concept. Then $NNF(C)$*

1. *is in negation normal form,*
2. *is equivalent to C, and*
3. *can be computed in linear time.*

3 Checking Satisfiability

In this section we describe an algorithm for deciding satisfiability in our concept language. The algorithm uses so-called constraints, which are built out of variables, concepts, and roles. Sets of constraints will be modified with the help of completion rules. The notions constraints and completion rules have already been used in [Schmidt-Schauß and Smolka, 1988, Hollunder, Nutt, and Schmidt-Schauß, 1990].

We assume that there exists an alphabet of variable symbols, which will be denoted by the letters x, y, z, and u. A *constraint* has one of the following forms

$$x\colon C, \quad xRy,$$

where C is a concept in negation normal form and R is a role. For xRy we say that y is an *R-successor of x*. If R is clear from the context or irrelevant, we simply say that y is a *successor of x*.

The interpretation of constraints is defined as follows. Let $\mathcal{I}$ be an interpretation of the concept language. An *$\mathcal{I}$-assignment* is a function α that maps every variable to an element of $\Delta^{\mathcal{I}}$. We say that α *satisfies* $x\colon C$ iff

$$\alpha(x) \in C^{\mathcal{I}},$$

and α *satisfies* xRy iff

$$(\alpha(x), \alpha(y)) \in R^{\mathcal{I}}.$$

A constraint s is *satisfiable* iff there exists an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α such that α satisfies s. A *constraint system* S is a finite, nonempty set of constraints. An $\mathcal{I}$-assignment α *satisfies* a constraint system S iff α satisfies every constraint in S. A constraint system S is *satisfiable* iff there exists an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α such that α satisfies S.

Proposition 3.1 *A concept C in negation normal form is satisfiable if and only if the constraint system $\{x : C\}$ is satisfiable.*

Proof. Follows immediately from the definitions. □

Given a concept C in negation normal form which has to be checked for satisfiability, our calculus starts with the constraint system $\{x : C\}$. Then, by applications of completion rules, it adds constraints to this system until a contradiction occurs, or an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}}$ is nonempty can be immediately obtained from the actual constraint system.

Before we formulate the rules we need some more definitions.

- For a variable x in a constraint system S we want to count the R-successors which are in a certain concept C. We therefore define

$$n_{R,C,S}(x) := |\{y \mid \{xRy,\ y : C\} \subseteq S\}|.$$

- By $[y/z]S$ we denote the constraint system that is obtained from S by replacing each occurrence of y by z.

We have the following rules:

1. $S \rightarrow_{\sqcap} \{x : C_1,\ x : C_2\} \cup S$

    if $x : C_1 \sqcap C_2$ is in S, $x : C_1$ and $x : C_2$ are not both in S

2. $S \rightarrow_{\sqcup} \{x : D\} \cup S$

    if $x : C_1 \sqcup C_2$ is in S, neither $x : C_1$ nor $x : C_2$ is in S,
    and $D = C_1$ or $D = C_2$

3. $S \rightarrow_{choose} \{y : D\} \cup S$

    if $x : (\leq n\ R\ C)$ is in S, xRy is in S,
    neither $y : C$ nor $y : NNF(\neg C)$ is in S,
    and $D = C$ or $D = NNF(\neg C)$

4. $S \rightarrow_{\geq} \{xRy,\ y\colon C\} \cup S$

   if $x\colon (\geq n\ R\ C)$ is in S, $n_{R,C,S}(x) < n$, and y is a new variable

5. $S \rightarrow_{\leq} [y/z]S$

   if $x\colon (\leq n\ R\ C)$ is in S, xRy, $y\colon C$, xRz, $z\colon C$ are in S,

   $y \neq z$, $n_{R,C,S}(x) > n$,

   for every u with xRu either $u\colon C$ or $u\colon NNF(\neg C)$ is in S.

Note that to every language construct—except negation of concepts—there corresponds a rule. Since concepts are assumed to be in negation normal form, we do not need rules which handle concepts with negation as outermost symbol such as $x\colon \neg(C \sqcap D)$, $x\colon \neg(C \sqcup D)$ etc. Negation applied to primitive concepts will be treated by the definition of "clash" given below.

The rules are used to decide whether a given constraint system is satisfiable as follows. We apply the rules until we obtain a constraint system such that no rule is applicable to it (a so-called complete constraint system). For such a constraint system it is easy to decide whether it is satisfiable.

Now let us discuss the rules. Obviously, the $\rightarrow_{\sqcap}$- and $\rightarrow_{\sqcup}$-rules are decomposing constraints having the form $x\colon C \sqcap D$ and $x\colon C \sqcup D$, respectively. Both rules are defined as in [Schmidt-Schauß and Smolka, 1988]. The rules which treat qualifying number restrictions are similar to the rules for ordinary number restrictions used in [Hollunder, Nutt, and Schmidt-Schauß, 1990]. However, in the case of ordinary number restrictions one only has to know for a given variable x *all* its R-successors for some role R; whereas in the case of qualifying number restrictions one has to know for a given variable x all its R-successors *which are in a certain concept* C. This fact leads to the definition of $n_{R,C,S}(x)$ given above. However, the number $n_{R,C,S}(x)$ is only useful if the $\rightarrow_{choose}$-rule does not apply to constraints containing the variable x. To see this assume that S is a constraint system such that $x : (\leq n\ R\ C)$ and xRy are in S, but neither $y\colon C$ nor $y\colon NNF(\neg C)$ is in S. Then $n_{R,C,S}(x)$ would not count the variable y since $y : C$ is not in S. Nevertheless there may exist an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α such that $\alpha(y) \in C^{\mathcal{I}}$. That means that in this interpretation x may have more R-successors in C than are indicated by $n_{R,C,S}(x)$, i.e., we may have

$$n_{R,C,S}(x) < \left|\{y \mid xRy \text{ is in } S \text{ and } \alpha(y) \in C^{\mathcal{I}}\}\right|.$$

Thus $n \geq n_{R,C,S}(x)$ need not imply

$$n \geq \left|\{y \mid xRy \text{ is in } S \text{ and } \alpha(y) \in C^{\mathcal{I}}\}\right|.$$

This means that the constraint $x : (\leq n\ R\ C)$ can be violated in spite of the fact that $n \geq n_{R,C,S}(x)$. To overcome this problem we use the $\rightarrow_{choose}$-rule which is based on the following idea. We know that for every $\mathcal{I}$ and α we have $\alpha(y) \in C^{\mathcal{I}}$ or $\alpha(y) \in (\neg C)^{\mathcal{I}}$. An application of the $\rightarrow_{choose}$-rule nondeterministically adds the correct constraint $y : C$ or $y : NNF(\neg C)$ to S.

Now suppose that for every R-successor u of x either $u : C$ or $u : NNF(\neg C)$ is in S. If $x : (\leq n\ R\ C)$ is in S and $n_{R,C,S}(x) > n$, then there are too many R-successors of x in C. An application of the $\rightarrow_{\leq}$-rule reduces their number by identifying two previously different R-successors.

For constraints of the form $x : (\geq n\ R\ C)$ it is not necessary to apply the $\rightarrow_{choose}$-rule to all successors y of x. In fact, assume that xRy is in S, but neither $y : C$ nor $y : NNF(\neg C)$ is in S. In this case is does not lead to problems if the $\rightarrow_{\geq}$-rule introduces a new variable because y was not counted for $n_{R,C,S}(x)$. If $\mathcal{I}$ is an interpretation and α is an $\mathcal{I}$-assignment such that $\alpha(y) \in C^{\mathcal{I}}$, we may now have

$$n_{R,C,S}(x) < \left|\{y \mid xRy \text{ is in } S \text{ and } \alpha(y) \in C^{\mathcal{I}}\}\right|.$$

But then $n \leq n_{R,C,S}(x)$ also implies

$$n \leq \left|\{y \mid xRy \text{ is in } S \text{ and } \alpha(y) \in C^{\mathcal{I}}\}\right|.$$

If $x : (\geq n\ R\ C)$ is in S and $n_{R,C,S}(x) < n$, then there may not be enough R-successors of x in C. An application of the $\rightarrow_{\geq}$-rule generates a new variable y and adds the constraints xRy and $y : C$ to S. Thus, the number $n_{R,C,S}(x)$ is increased by one.

The rules as described above have the disadvantage that they allow infinite chains of rule applications.

Example 3.2 Consider the constraint system

$$S = \{x : (\geq 2\ R\ A) \sqcap (\leq 1\ R\ A)\}.$$

We obtain the following constraint systems by applications of the rules:

$$\begin{aligned} S &\rightarrow_{\sqcap} S \cup \{x : (\geq 2\ R\ A),\ x : (\leq 1\ R\ A)\} =: S_1 \\ &\rightarrow_{\geq} S_1 \cup \{xRy,\ y : A\} =: S_2 \\ &\rightarrow_{\geq} S_2 \cup \{xRz,\ z : A\} =: S_3 \\ &\rightarrow_{\leq} [z/y]S_3 = S_2. \end{aligned}$$

Thus we have the infinite chain $S \rightarrow_{\sqcap} S_1 \rightarrow_{\geq} S_2 \rightarrow_{\geq} S_3 \rightarrow_{\leq} S_2 \rightarrow_{\geq} S_3 \ldots$

The example demonstrates that alternating applications of the $\rightarrow_{\geq}$- and $\rightarrow_{\leq}$-rule are responsible for infinite chains. The idea to avoid such infinite chains is as follows. We only apply the $\rightarrow_{\leq}$-rule to a constraint system S (yielding the constraint system S') if every constraint of the form $x : (\geq n\ R\ C)$ that is satisfied in S, i.e., $n_{R,C,S}(x) \geq n$, remains satisfied in S'. Note that this condition is violated in the above example since $S_3 \rightarrow_{\leq} S_2$, $n_{R,A,S_3}(x) = 2$ and $n_{R,A,S_2}(x) = 1$.

Before we can formulate the modified $\rightarrow_{\leq}$-rule we need the following definition. Let S be a constraint system. We say that the replacement of y by z is *safe in* S if for all x, R, C, n

$$\{xRy,\ y\colon C,\ xRz,\ z\colon C,\ x\colon (\geq n\ R\ C)\} \subseteq S$$

implies $n_{R,C,S}(x) > n$.

We reformulate the $\rightarrow_{\leq}$-rule as follows. Let S be a constraint system.

5*. $S \rightarrow_{*\leq} [y/z]S$
if $x\colon (\leq n\ R\ C)$ is in S, xRy, $y\colon C$, xRz, $z\colon C$ are in S,
$y \neq z$, $n_{R,C,S}(x) > n$,
for every u with xRu either $u\colon C$ or $u\colon NNF(\neg C)$ is in S,
and the replacement of y by z is safe in S.

We will see that the additional condition ensures that there is no infinite chain of rule applications.

The *completion rules* consist of the $\rightarrow_{\sqcap}$-, $\rightarrow_{\sqcup}$-, $\rightarrow_{choose}$-, $\rightarrow_{\geq}$-, and $\rightarrow_{*\leq}$-rule. The following proposition, which one can prove easily, shows that the completion rules are defined in an appropriate manner.

Proposition 3.3 *Let S and S' be constraint systems. Then:*

1. *If S' is obtained from S by application of the (deterministic) $\rightarrow_{\sqcap}$- or $\rightarrow_{\geq}$-rule, then S is satisfiable if and only if S' is satisfiable.*

2. *If S' is obtained from S by application of the (nondeterministic) $\rightarrow_{\sqcup}$-, $\rightarrow_{choose}$-, or $\rightarrow_{*\leq}$-rule, then S is satisfiable if S' is satisfiable. Furthermore, if a nondeterministic rule applies to S, then it can be applied in such a way that it yields a constraint system S' such that S' is satisfiable if and only if S is satisfiable.*

Thus we know that the completion rules are locally sound and complete. However, to prove the overall soundness and completeness of the calcalus it will not be necessary to use Proposition 3.3.

The proof of termination of the completion rules—which is less obvious—will employ techniques which have been developed for proving termination of term rewriting systems (see [Dershowitz and Manna, 1979]).

Proposition 3.4 *Let C_0 be a concept in negation normal form. Then there is no infinite chain of applications of completion rules issuing from $\{x_0 : C_0\}$.*

A constraint system S which can be obtained from $\{x_0 : C_0\}$ by a finite number of applications of completion rules will be called *derived system* in the following. In order to prove the proposition, any derived system S will be mapped on an element $\Psi(S)$ of a set Q which is equipped with a well-founded strict partial ordering $\gg$. Since the ordering is well-founded, i.e., has no infinitely decreasing chains, termination will follow immediately as soon as one has established the following property. Whenever S' is obtained from the derived system S by application of a rule, one has $\Psi(S) \gg \Psi(S')$.

The elements of the set Q will have a rather complex structure. They are finite multisets of 5-tuples. Each component of the tuples is either a finite multiset of nonnegative integers (for the second, third, and fifth component) or a nonnegative integer (for the first and fourth component). Multisets are like sets, but allow multiple occurrences of identical elements. For example, $\{2,2,2\}$ is a multiset which is distinct from the multiset $\{2\}$. A given ordering on a set T can be extended to form an ordering on the finite multisets over T. In this ordering, a finite multiset M is larger than a finite multiset M' iff M' can be obtained from M by replacing one or more elements in M by any finite number of elements taken from T, each of which is smaller than one of the replaced elements. For example, $\{2,2,2\}$ is larger than $\{2\}$ and $\{2,2,1,1,0\}$. [Dershowitz and Manna, 1979] show that the induced ordering on finite multisets over T is well-founded if the original ordering on T is so.

The nonnegative integer components of our 5-tuples are compared with respect to the usual ordering on integers, and the finite multiset components by the multiset ordering induced by this ordering. The whole tuples are ordered lexicographically from left to right, i.e., $(c_1, ..., c_5)$ is larger than $(c'_1, ..., c'_5)$ iff there exists $i, 1 \leq i \leq 5$, such that $c_1 = c'_1, ..., c_{i-1} = c'_{i-1}$, and c_i is larger than c'_i. Since the orderings on the components are well-founded, the lexicographical ordering on the tuples is also well-founded. Finite multisets of these tuples are now compared with respect to the multiset ordering induced

by this lexicographical ordering. This is the well-founded ordering $\gg$ on Q mentioned above.

Before we can define the mapping Ψ from derived systems to elements of Q, we need two more definitions. For two nonnegative integers n, m we denote by $n \dot{-} m$ the asymmetrical difference between n and m, i.e., $n \dot{-} m := n - m$ if $n \geq m$, and $n \dot{-} m := 0$ if $n < m$. For a concept C the size $|C|$ is inductively defined as

- $|A| = 1$ for all primitive concepts A,
- $|\neg C| = |C|$,
- $|(\geq n\ R\ C)| = |(\leq n\ R\ C)| = 1 + |C|$,
- $|C \sqcap D| = |C \sqcup D| = |C| + |D|$.

Definition 3.5 *Let S be a constraint system. Then $\Psi(S)$ is the multiset which contains for each variable x occurring in S the following 5-tuple $\psi_S(x)$:*

1. *The* first component *of $\psi_S(x)$ is the nonnegative integer* $\max\{|C| \mid x\colon C \text{ is in } S\}$.
2. *The* second component *of $\psi_S(x)$ is a multiset which contains, for each constraint $x\colon C \sqcap D$ (resp. $x\colon C \sqcup D$) in S for which the $\rightarrow_\sqcap$-rule (resp. $\rightarrow_\sqcup$-rule) is applicable, the nonnegative integer $|C \sqcap D|$ (resp. $|C \sqcup D|$).*
3. *The* third component *of $\psi_S(x)$ is a multiset which contains, for each constraint $x\colon (\geq n\ R\ C)$ in S, the nonnegative integer $n \dot{-} n_{R,C,S}(x)$.*
4. *The* fourth component *of $\psi_S(x)$ is the number of all successors of x in S.*
5. *The* fifth component *of $\psi_S(x)$ is a multiset which contains, for each constraint $x\colon (\leq n\ R\ C)$ in S, the number of all R-successors y of x such that neither $y\colon C$ nor $y\colon NNF(\neg C)$ is in S.*

For the constraint systems of Example 3.2 we have

$$
\begin{aligned}
\Psi(S) &= \{(4,\{4\},\emptyset,0,\emptyset)\},\\
\Psi(S_1) &= \{(4,\emptyset,\{2\},0,\{0\})\},\\
\Psi(S_2) &= \{(4,\emptyset,\{1\},1,\{0\}),(1,\emptyset,\emptyset,0,\emptyset)\},\\
\Psi(S_3) &= \{(4,\emptyset,\{0\},2,\{0\}),(1,\emptyset,\emptyset,0,\emptyset),(1,\emptyset,\emptyset,0,\emptyset)\}.
\end{aligned}
$$

Thus the chain of rule applications $S \rightarrow_{\sqcap} S_1 \rightarrow_{\geq} S_2 \rightarrow_{\geq} S_3$ corresponds to the decreasing chain $\Psi(S) \gg \Psi(S_1) \gg \Psi(S_2) \gg \Psi(S_3)$ in Q. However, the unsafe application of the $\rightarrow_{\leq}$-rule transforms S_3 into the system S_2 which has a larger Ψ-image.

The following facts will be important in the termination proof.

Lemma 3.6

1. *For any concept C we have $|C| \geq |NNF(\neg C)|$.*

2. *For any variable y in a derived system S there exists at most one pair (x, R) consisting of a variable x and a role R such that xRy is a constraint in S. That means that y has at most one predecessor in S.*

3. *Let xRy be a constraint in the derived system S. Then we have*

$$\max\{|C| \mid x\colon C \textit{ is in } S\} > \max\{|D| \mid y\colon D \textit{ is in } S\}.$$

Proof. The first fact can easily be proved by induction on the number of applications of simplification rules needed to compute the negation normal form. The reason why we do not always have $|C| = |NNF(\neg C)|$ is that a concept of the form $\neg(\geq 0\ R\ C)$ is replaced by the usually much smaller concept $\perp$.

To see the second fact, note that if a variable y is newly introduced, then it is introduced as R-successor of exactly one variable x. In addition, if two different variables are identified by the $\rightarrow_{*\leq}$-rule, then they must have been R-successor of the same variable, and for the same role R.

The third fact can be shown as follows. By the second fact we know that x is the only predecessor of y. If a new constraint $y\colon D$ is imposed on y, then it either comes from a larger constraint on y itself (for the $\rightarrow_{\sqcap}$- or $\rightarrow_{\sqcup}$-rule), or from a larger constraint on x (for the $\rightarrow_{choose}$- or $\rightarrow_{\geq}$-rule). If y gets additional constraints because it replaces a variable z (in an application of the $\rightarrow_{*\leq}$-rule) then y and z have been successors of the same variable x. For that reason, all the constraints on z have also been smaller than the maximal constraint on x. □

Proposition 3.4 is now an immediate consequence of the next lemma.

Lemma 3.7 *If S' is obtained from the derived system S by application of a completion rule, then $\Psi(S) \gg \Psi(S')$.*

Proof. (1) Assume that S' is obtained from S by applying the $\rightarrow_{\sqcap}$-rule to the constraint $x: C \sqcap D$.
First, we consider how the tuple $\psi_S(x)$ is changed. The first component remains the same. In the multiset of the second component, the number $|C \sqcap D|$ is removed. If C or D have conjunction or disjunction as uppermost symbol, then we may have to include the numbers $|C|$ or $|D|$, which are however smaller than the removed one. This shows that the second component of $\psi_{S'}(x)$ is smaller than the second component of $\psi_S(x)$. Since the tuples are compared with respect to the lexicographical ordering, we thus have shown that the whole tuple becomes smaller, independently of what happens for the other components.
Now we consider a tuple $\psi_S(y)$ for a variable $y \neq x$. By the definition of the tuples, the change in the constraints for x can only influence the tuple for y if x is an R-successor of y for some role R. In this case it may affect the third and fifth component of $\psi_S(y)$. However, by adding constraints to a successor of y these components may at the most become smaller in $\psi_{S'}(y)$.
This shows that $\Psi(S')$ can be obtained from $\Psi(S)$ by replacing some (but at least one) of the tuples by smaller ones.

(2) The $\rightarrow_{\sqcup}$-rule can be treated in a similar way.

(3) Assume that S' is obtained from S by applying the $\rightarrow_{choose}$-rule to an R-successor y of x because of the constraint $x: (\leq n\ R\ C)$.
First, we consider how the tuple $\psi_S(x)$ is changed. The first and the second component remain the same. By adding constraints to a successor of x the third component may at the most become smaller. The fourth component remains unchanged. The fifth component becomes smaller because y is no longer counted for the constraint $x: (\leq n\ R\ C)$. This shows that the whole tuple $\psi_{S'}(x)$ is smaller than $\psi_S(x)$.
Now let us consider the tuple $\psi_S(y)$. Since y is a successor of x, the third of the facts stated in Lemma 3.6 implies that the first component of $\psi_{S'}(y)$ is smaller than the first component of $\psi_{S'}(x)$, which is the same as the first component of $\psi_S(x)$.
Assume that u is a variable different from x and y. A change in the constraints for y can only influence the tuple for u if y is a successor of u. But by the second fact of Lemma 3.6, x is the only predecessor of y.
Thus $\Psi(S')$ can be obtained from $\Psi(S)$ as follows. The tuple $\psi_S(y)$ is removed, and the tuple $\psi_S(x)$ is replaced by two smaller tuples, namely the tuples $\psi_{S'}(x)$ and $\psi_{S'}(y)$.

(4) Assume that S' is obtained from S by applying the $\rightarrow_{\geq}$-rule to the constraint $x: (\geq n\ R\ C)$.

First, we consider how the tuple $\psi_S(x)$ is changed. The first and the second component remain the same. Since the $\rightarrow_{\geq}$-rule introduces a new R-successor y of x and adds the constraint $y: C$, we have $n_{R,C,S'}(x) > n_{R,C,S}(x)$. In addition, we know that $n > n_{R,C,S}(x)$ because otherwise the rule would not have been applicable. These two facts imply that $n \dot{-} n_{R,C,S'}(x) < n \dot{-} n_{R,C,S}(x)$. If S also contains a constraint $x: (\geq m\ R\ C)$ for some nonnegative integer $m \neq n$ then we obviously have $m \dot{-} n_{R,C,S'}(x) \leq m \dot{-} n_{R,C,S}(x)$ (equality holds for $m \dot{-} n_{R,C,S}(x) = 0$). The contributions of constraints of the form $x: (\geq m\ R'\ C')$ for $R' \neq R$ or $D \neq C$ to the third component are not changed. To sum up, we have thus shown that there third component of $\psi_{S'}(x)$ is smaller than the third component of $\psi_S(x)$. For this reason, changes in the fourth and fifth component are irrelevant.
Now let us consider the tuple $\psi_{S'}(y)$. Since y is newly introduced, there is no corresponding tuple in $\Psi(S)$. But the first component of $\psi_{S'}(y)$ is smaller than the first component of $\psi_S(x)$.
As before, the tuples for variables u different from x and y are not changed. Thus $\Psi(S')$ can be obtained from $\Psi(S)$ by replacing $\psi_S(x)$ by the two smaller tuples $\psi_{S'}(x)$ and $\psi_{S'}(y)$.

(5) Finally, assume that S' is obtained from S by applying the $\rightarrow_{*\leq}$-rule to the constraint $x: (\leq n\ R\ C)$, and thereby replacing the variable y by z.
First, we consider how the tuple $\psi_S(x)$ is changed. The first and the second component remain the same. Let us now turn to the the third component. Assume that Assume that one of the numbers in the multiset of the third component increases. That means that there exist m, R', C' such that $x: (\geq m\ R'\ C')$ is in S, and $m \dot{-} n_{R',C',S'}(x) > m \dot{-} n_{R',C',S}(x)$. Obviously, this can only be the case if $R = R'$, and $y: C'$ and $z: C'$ are in S. Since the replacement of y by z is safe, we have $n_{R,C',S}(x) > m$, and thus $n_{R,C',S'}(x) \geq m$. This shows that $m \dot{-} n_{R,C',S'}(x) = 0 = m \dot{-} n_{R,C',S}(x)$, which contradicts our assumption. Thus we have shown that the third component also remains the same. Obviously, the fourth component, i.e., the number of successors of x, is decreased by one.
Now let us consider the tuple $\psi_{S'}(z)$. In S', the variable z has both its original constraints and the constraints y had in S. However, since y and z are successors of x we know that the size of each of these constraints is smaller than the first component of $\psi_S(x)$.
The variable y does not occur in S'.
Finally, consider a variable u in S' which is different from x, y, z. The changes for y and z can only affect the tuple of u if u is a predecessor of y or z. But this would mean that $u = x$.
Thus $\Psi(S')$ can be obtained from $\Psi(S)$ by removing the tuples $\psi_S(y)$ and

$\psi_S(z)$, and by replacing the tuple $\psi_S(x)$ by two smaller tuples $\psi_{S'}(x)$ and $\psi_{S'}(z)$. □

Thus we have shown that for every constraint system $\{x\colon C\}$, where C is a concept in negation normal form, we obtain after finitely many applications of completion rules a constraint system to which no completion rule applies.

A constraint system is called *complete* iff no completion rule applies to it. A constraint system S contains a *clash* iff

- $\{x\colon \bot\} \subseteq S$ for some variable x, or
- $\{x\colon A,\ x\colon \neg A\} \subseteq S$ for some variable x and some primitive concept A, or
- $\{x : (\leq n\ R\ C)\} \subseteq S$ and $n_{R,C,S}(x) > n$ for some variable x, role R, concept C, and nonnegative integer n.

In the remainder of this section we assume that C_0 is a concept in negation normal form. The satisfiability of a constraint system $S_0 = \{x_0 : C_0\}$, and thus of a concept C_0, can be characterized by using the completion rules and the notion of clash.

Proposition 3.8 *A constraint system $S_0 = \{x_0 : C_0\}$ is satisfiable if and only if there exists a clash free complete constraint system which can be derived from S_0 by applying the completion rules.*

In the case of ordinary number restrictions this proposition is proved as follows. One shows that a complete constraint system is satisfiable if and only if it is clash free [Hollunder, Nutt, and Schmidt-Schauß, 1990]. The proposition is then an immediate consequence of local soundness and completeness and the termination of the completion rules. To see why this simple method cannot be used in the case of qualifying number restrictions consider the constraint system

$$\begin{aligned} S \;=\; & \{x\colon (\geq 1\ R\ (A \sqcap B)) \sqcap (\geq 1\ R\ (B \sqcap A)) \sqcap \\ & (\leq 1\ R\ (A \sqcap B)) \sqcap (\geq 2\ R\ A)\}. \end{aligned}$$

We can obtain the complete constraint system

$$\begin{aligned} S' \;=\; & S \cup \{xRy,\ y\colon (A \sqcap B),\ y\colon A,\ y\colon B, \\ & xRz,\ z\colon (B \sqcap A),\ z\colon (A \sqcap B),\ z\colon A,\ z\colon B\}, \end{aligned}$$

if we first apply the $\rightarrow_{\geq}$-rule to $x\colon (\geq 1\ R\ (A \sqcap B))$ and $x\colon (\geq 1\ R\ (B \sqcap A))$, and then apply the $\rightarrow_{\sqcap}$-rule to $y : (A \sqcap B)$ and $z : (B \sqcap A)$. Note that the replacement of y by z and of z by y is not safe in S'. Since $(\leq 1\ R\ (A \sqcap B))$ is in S' and $n_{R,A\sqcap B,S'}(x) = 2$, S' contains a clash. Nevertheless it is easy to see that S' is satisfiable. By using an appropriate control strategy, the following clash free complete constraint system

$$\begin{aligned} S'' &= S \cup \{xRy,\ y\colon (A \sqcap B),\ y\colon (B \sqcap A),\ y\colon A,\ y\colon B, \\ &\qquad xRu,\ u\colon A,\ xRv,\ v\colon A\} \end{aligned}$$

can be derived from S. This strategy is described in the proof of the next lemma, which shows one direction of Proposition 3.8.

Lemma 3.9 *If a constraint system $S_0 = \{x_0\colon C_0\}$ is satisfiable, then there exist constraint systems $S_1, S_2, \ldots, S_l$ such that*

- *S_{i+1} is obtained from S_i by application of a completion rule,*
- *every S_i is satisfiable, and*
- *S_l is complete and does not contain a clash.*

Proof. If $S_0 = \{x_0\colon C_0\}$ is satisfiable, then there exist an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α_0 such that α_0 satisfies S_0. We use $\mathcal{I}$ and α_0 to guide the construction of a sequence $S_1, S_2, \ldots, S_l$ of constraint systems which satisfy the requirements of the lemma. For the construction of the sequence we need auxiliary sets $T_0, T_1, \ldots, T_l$ which consist of tuples of the form (x, y). We say that an $\mathcal{I}$-assignment α_i satisfies T_i iff for all $(x, y) \in T_i$ we have $\alpha_i(x) \neq \alpha_i(y)$. The sets T_i will be called inequality constraints. The inital set T_0 will be the empty set which obviously is satisfied by any $\mathcal{I}$-assignment.

In the first step of the construction we inductively define a sequence $S_1, S_2, \ldots, S_{l_1}$ of constraint systems satisfied by α_0 as follows. Assume that S_i $(i \geq 0)$ is a constraint system already obtained. If the $\rightarrow_{\sqcap}$-rule is applicable to S_i, we apply it, and thus get S_{i+1}. Obviously, α_0 also satisfies S_{i+1} since by induction it satisfies S_i. Otherwise, assume that the $\rightarrow_{\sqcup}$-rule is applicable to a constraint of the form $x_0\colon C_1 \sqcup C_2$. Since α_0 satisfies S_i we have $\alpha_0(x_0) \in C_1^{\mathcal{I}}$ or $\alpha_0(x_0) \in C_2^{\mathcal{I}}$. If $\alpha_0(x_0) \in C_1^{\mathcal{I}}$, then we choose $S_{i+1} := S_i \cup \{x_0 : C_1\}$; otherwise we take $S_{i+1} := S_i \cup \{x_0\colon C_2\}$. If neither $\rightarrow_{\sqcap}$- nor $\rightarrow_{\sqcup}$-rules apply to S_i, then we are done with the first step, and define $l_1 := i$. We also define $\alpha_{l_1} := \alpha_{l_1-1} := \ldots := \alpha_0$ and $T_{l_1} := T_{l_1-1} := \ldots := T_0 := \emptyset$. Obviously we

have for all $0 \leq i \leq l_1$, that α_i satisfies S_i and T_i. In addition, S_{l_1} is obtained from S_0 by applications of the $\rightarrow_\sqcap$- and $\rightarrow_\sqcup$-rules.

Now let us define the constraint system S_{l_2} and the set T_{l_2}. For every pair R, C such that $x_0\colon (\geq n\ R\ C)$ is in S_{l_1} for some n we add

- the constraints $x_0Ry_1, \ldots, x_0Ry_{n_{R,C}}, y_1\colon C, \ldots, y_{n_{R,C}}\colon C$ to S_{l_1}, where $n_{R,C} := \max\{n \mid x_0\colon (\geq n\ R\ C) \in S_{l_1}\}$ and the y_i are new variables for each pair R, C, and
- the tuples (y_i, y_j), $1 \leq i, j \leq n_{R,C}$, $i \neq j$ to T_{l_1}.

The systems thus obtained are S_{l_2} and T_{l_2}. Obviously, S_{l_2} can be obtained from S_{l_1} by finitely many applications of the $\rightarrow_\geq$-rule. In fact, for every pair R, C such that $x_0\colon (\geq n\ R\ C)$ is in S_{l_1} for some n we apply the $\rightarrow_\geq$-rule $n_{R,C}$ times. Each application adds the constraints x_0Ry and $y\colon C$ where y is a new variable. We still have to show that there exists an $\mathcal{I}$-assignment α_{l_2} which satisfies S_{l_2} and T_{l_2}. Consider a constraint $x_0\colon (\geq n_{R,C}\ R\ C)$ occurring in S_{l_1}. Since α_{l_1} satisfies S_{l_1} we have $\alpha_{l_1}(x_0) \in (\geq n_{R,C}\ R\ C)^\mathcal{I}$. Thus there exist $n_{R,C}$ pairwise different $d_1, \ldots, d_{n_{R,C}} \in \Delta^\mathcal{I}$ such that $(\alpha_{l_1}(x_0), d_i) \in R^\mathcal{I}$ and $d_i \in C^\mathcal{I}$. For the variables x already in S_{l_1}, we define $\alpha_{l_2}(x) := \alpha_{l_1}(x)$. Let $y_1, \ldots, y_{n_{R,C}}$ be the variables introduced for the pair R, C. We define $\alpha_{l_2}(y_i) := d_i$. It is easy to see that α_{l_2} satisfies S_{l_2}. Since $\alpha_{l_2}(y_i) = d_i \neq d_j = \alpha_{l_2}(y_j)$ for $i \neq j$ we conclude that α_{l_2} satisfies T_{l_2}. Thus α_{l_2} satisfies S_{l_2} and T_{l_2}.

In the next step, the constraint system S_{l_3} is defined as follows. For every pair $x_0\colon (\leq n\ R\ C)$, x_0Ry in S_{l_2} such that neither $y\colon C$ nor $y\colon \mathit{NNF}(\neg C)$ is in S_{l_2}, we add $y\colon C$ if $\alpha_{l_2}(y) \in C^\mathcal{I}$; otherwise we add $y\colon \mathit{NNF}(\neg C)$. Obviously, S_{l_3} can be obtained from S_{l_2} by finitely many applications of the $\rightarrow_{choose}$-rule. We set $\alpha_{l_3} := \alpha_{l_2}$, and $T_{l_3} := T_{l_2}$. It is easy to see that α_{l_3} satisfies S_{l_3} and T_{l_3}.

Now we define a sequence $S_{l_3+1}, S_{l_3+2}, \ldots, S_{l_4}$ such that

- S_{i+1} is obtained from S_i by applying the $\rightarrow_{*\leq}$-rule to a constraint of the form $x_0\colon (\leq n\ R\ C)$ occurring in S_i, and
- for every $x_0\colon (\leq n\ R\ C)$ in S_{l_4} we have $n_{R,C,S_{l_4}}(x_0) \leq n$.

For the first step, suppose $x_0\colon (\leq n\ R\ C), x_0Ry_1, \ldots, x_0Ry_k, y_1\colon C, \ldots, y_k\colon C$ are in S_{l_3} where $k > n$. Since α_{l_3} satisfies S_{l_3} there exist y_i and y_j, $i \neq j$ such that $\alpha_{l_3}(y_i) = \alpha_{l_3}(y_j)$. We replace y_i by y_j, i.e., $S_{l_3+1} := [y_i/y_j]S_{l_3}$.

Furthermore, we set $\alpha_{l_3+1} := \alpha_{l_3}$ and $T_{l_3+1} := [y_i/y_j]T_{l_3}$, i.e., T_{l_3+1} is obtained from T_{l_3} by replacing each occurrence of y_i by y_j. We now prove the following facts:

1. *α_{l_3+1} satisfies S_{l_3+1}.*
   We know that α_{l_3} satisfies S_{l_3}, and $\alpha_{l_3}(y_i) = \alpha_{l_3}(y_j)$. Thus α_{l_3+1} satisfies $S_{l_3+1} = [y_i/y_j]S_{l_3}$.

2. *α_{l_3+1} satisfies T_{l_3+1}.*
   Suppose (y_i, u) or (u, y_i) is in T_{l_3} for some u. Since $\alpha_{l_3}(y_j) = \alpha_{l_3}(y_i) \neq \alpha_{l_3}(u)$ we have $\alpha_{l_3}(y_j) \neq \alpha_{l_3}(u)$. Thus α_{l_3+1} satisfies T_{l_3+1}.

3. *For every constraint $x_0 : (\geq n\ R\ C)$ occurring in S_{l_3+1} there exist variables $u_1, \ldots, u_n$ such that $x_0Ru_1, \ldots, x_0Ru_n, u_1 : C, \ldots, u_n : C$ are in S_{l_3+1}, and (u_i, u_j) is in T_{l_3+1} for $1 \leq i, j \leq n$, $i \neq j$.*
   By construction of S_{l_3} we know that for every $x_0 : (\geq n\ R\ C)$ in S_{l_3} there exist variables $z_1, \ldots, z_n$ such that $x_0Rz_1, \ldots, x_0Rz_n, z_1 : C, \ldots, z_n : C$ are in S_{l_3}, and (z_i, z_j) is in T_{l_3} for $1 \leq i, j \leq n$, $i \neq j$. If $(y_i, z) \in T_{l_3}$ or $(z, y_i) \in T_{l_3}$ for some z, then we have $(y_j, z) \in T_{l_3+1}$ or $(z, y_j) \in T_{l_3+1}$. If in addition $x_0Ry_i \in T_{l_3}$ and $y_i : C \in T_{l_3}$, then $x_0Ry_j \in T_{l_3+1}$ and $y_j : C \in T_{l_3+1}$. Thus S_{l_3+1} contains for every $x_0 : (\geq n\ R\ C)$ at least n variables $u_1, \ldots, u_n$ such that

   - $x_0Ru_1, \ldots, x_0Ru_n, u_1 : C, \ldots, u_n : C$ are in S_{l_3+1}, and
   - (u_i, u_j) is in T_{l_3+1} for $1 \leq i, j \leq n$, $i \neq j$.

   Since we have already seen that α_{l_3+1} satisfies T_{l_3+1}, the variables $u_1, \ldots, u_n$ are pairwise distinct.

4. *The replacement of y_i by y_j is safe in S_{l_3}. This shows that S_{l_3+1} is really obtained by the $\rightarrow_{*\leq}$-rule from S_{l_3}.*
   Suppose the replacement of y_i by y_j is not safe in S_{l_3}. Then there exists $x_0 : (\geq m\ R\ D)$ in S_{l_3} such that $n_{R,D,S_{l_3}}(x_0) = m$, and x_0Ry_i, x_0Ry_j, $x_0Rz_3, \ldots, x_0Rz_m, y_i : D, y_j : D, z_3 : D, \ldots, z_m : D$ are in S_{l_3}. By construction of S_{l_3} we know that $(y_i, y_j) \in T_{l_3}$. Thus $\alpha_{l_3}(y_i) \neq \alpha_{l_3}(y_j)$ which contradicts to $\alpha_{l_3}(y_i) = \alpha_{l_3}(y_j)$.

We can now continue this process with S_{l_3+1} in place of S_{l_3} until we obtain a constraint system—which is called S_{l_4}—such that for every $x_0 : (\leq n\ R\ C)$ occurring in it we have $n_{R,C,S_{l_4}}(x_0) \leq n$. By induction one can proof that the four properties hold for every i, $l_3 \leq i \leq l_4$. Since S_{l_3} contains only finitely many variables we obtain after finitely many applications of the $\rightarrow_{*\leq}$-rule

a constraint system S_{l_4} such that the $\rightarrow_{*\leq}$-rule does not apply to $x_0 : (\leq n\ R\ C)$ in S_{l_4}. Note that $n_{R,C,S_{l_4}}(x_0) \leq n$ for every $x_0 : (\leq n\ R\ C)$ in S_{l_4}. Furthermore, it is easy to see that the $\rightarrow_{\sqcap}$-, $\rightarrow_{\sqcup}$-, $\rightarrow_{\geq}$-, and $\rightarrow_{choose}$-rules do not apply to constraints which contain the variable x_0.

This shows that all constraints imposed on x_0 are explicitly present in S_{l_4}.

We can now apply the strategy of rule applications as performed for the variable x_0 to all successors of x_0, then to all successors of these successors, and so on. With Proposition 3.4 we know that after finitely many applications of the completion rules a complete constraint system S_l is obtained. Since S_l is satisfiable, it does not contain constraints of the form $x : \bot$ or $x : A$, $x : \neg A$. Furthermore, we know that for every $x : (\leq n\ R\ C)$ in S_l we have $n_{R,C,S_l}(x) \leq n$ by construction of the sequence $S_0, S_1, \ldots, S_l$. Thus S_l is clash free. □

Now we show the other direction of Proposition 3.8.

Lemma 3.10 *If there exists a clash free complete constraint system issuing from $S_0 = \{x_0 : C_0\}$, then S_0 is satisfiable.*

Proof. We first show that a clash free complete constraint system S is satisfiable. To prove this claim we construct an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α which satisfies S as follows. The domain $\Delta^{\mathcal{I}}$ of $\mathcal{I}$ consists of all variables occurring in S. For all primitive concepts A different from $\top$ and $\bot$ we define $A^{\mathcal{I}} := \{x \mid x : A \in S\}$, and for all roles R we define $R^{\mathcal{I}} := \{(x, y) \mid xRy \in S\}$. The $\mathcal{I}$-assignment α is defined by mapping variables to themselves, i.e. $\alpha(x) := x$. We now prove that α satisfies every constraint s in S. If s has the form xRy, then α satisfies xRy by definition of $\mathcal{I}$ and α. If s has the form $x : C$, we show by *induction on the structure* of C that $\alpha(x) \in C^{\mathcal{I}}$.

Base case: If C is a primitive concept different from $\top$ and $\bot$, then $\alpha(x) \in A^{\mathcal{I}}$ by definition of $\mathcal{I}$ and α. If $C = \top$, then obviously $\alpha(x) \in \top^{\mathcal{I}}$. Since S is clash free we have $C \neq \bot$.

Induction step: If $C = \neg A$ for a primitive concept A, the constraint $x : A$ is not in S since S is clash free. Then $\alpha(x) \notin A^{\mathcal{I}}$ and $\alpha(x) \in \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$. Hence $\alpha(x) \in (\neg A)^{\mathcal{I}}$.

If $C = C_1 \sqcap C_2$, then the constraints $x : C_1$ and $x : C_2$ are in S since S is complete. By the induction hypothesis we know that $\alpha(x) \in C_1^{\mathcal{I}}$ and $\alpha(x) \in C_2^{\mathcal{I}}$, which implies $\alpha(x) \in C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$, and hence $\alpha(x) \in (C_1 \sqcap C_2)^{\mathcal{I}}$.

Similarly, it can be shown that constraints of the form $x : C_1 \sqcup C_2$ are satisfied by $\mathcal{I}$ and α.

Suppose $C = (\leq n\ R\ D)$. Since the $\rightarrow_{choose}$-rule is not applicable to S, for every y with xRy either $y : D$ or $y : NNF(\neg D)$ is in S. By the induction hypothesis we know that $\alpha(y) \in D^{\mathcal{I}}$ if $y : D$ is in S, or $\alpha(y) \in (\neg D)^{\mathcal{I}}$ if $y : NNF(\neg D)$ is in S. Since S is complete and does not contain a clash, we have $n_{R,D,S}(x) \leq n$. Hence $\alpha(x) \in (\leq n\ R\ D)^{\mathcal{I}}$. Similarly, it can be shown that constraints of the form $x : (\geq n\ R\ D)$ are satisfied by $\mathcal{I}$ and α.

Thus we have shown that every clash free complete constraint system is satisfiable. To complete the proof of the lemma let S be a clash free complete constraint system issuing from $S_0 = \{x_0 : C_0\}$ by applications of the completion rules. With the above observation we know that S is satisfiable. Since $S_0 \subseteq S$ we also know that S_0 is satisfiable. □

Now we can formulate and prove the main result of this section.

Theorem 3.11 *Satisfiability and subsumption of concepts are decidable.*

Proof. We obtain a decision procedure for the satisfiability problem of concepts as follows. Let C be a concept. First we transform C into its negation normal form C_0, which can be done in linear time. Then we generate the finitely many complete constraint systems issuing from $\{x_0 : C_0\}$. If one of these constraint systems is clash free, then C_0 (and hence C) is satisfiable (Lemma 3.10); otherwise C is not satisfiable (Lemma 3.9).

Since our concept language allows negation of concepts the subsumption problem can be reduced to the satisfiability problem. □

Our concept language generalizes the concept language $\mathcal{ALC}$ in which deciding satisfiability and subsumption of concepts are PSPACE-complete problems [Schmidt-Schauß and Smolka, 1988]. Thus we have

Proposition 3.12 *Satisfiability and subsumption of concepts are PSPACE-hard problems.*

4 Implementation

The algorithm for deciding satisfiability of concepts given in the previous section has the nice property that soundness and completeness can be proved

in a relatively simple manner. However, the rule-based algorithm as described is not convenient for implementation purposes. The reasons will be discussed in the first part of this section. One can obtain an algorithm which is more suitable for an actual implementation by imposing an appropriate control on the applications of completion rules. Based on the idea of [Schmidt-Schauß and Smolka, 1988] who gave an optimized algorithm for the language $\mathcal{ALC}$ we will present in the second part of this section an optimized algorithm for our language.

The following considerations show how the above presented algorithm may be optimized.

(1) Contradictions may occur early in the completion process. If a constraint system S contains clashes of the form $\{x\colon \bot\}$ or $\{x\colon A,\ x\colon \neg A\}$, then every complete constraint system extending S also contains a clash. Thus, the completion process can be stopped since S is not satisfiable.

(2) In general, constraint systems contain redundant constraints. For example, suppose that a constraint system S contains $x\colon C \sqcap D$. If $x\colon C$ and $x\colon D$ are also in S, then obviously no completion rule will apply to $x\colon C \sqcap D$ in the further completion process. Hence it is not necessary to keep $x\colon C \sqcap D$ in S if $x : C$ and $x : D$ are in S. Similar arguments hold for $x : C \sqcup D$. Constraints of the form $x\colon (\geq n\ R\ C)$ or $x\colon (\leq n\ R\ C)$ can be removed from S under the following circumstances:

- No constraints having the form $x\colon C \sqcap D$ or $x\colon C \sqcup D$ are in S.
- The $\rightarrow_{choose}$-, $\rightarrow_{\geq}$-, and $\rightarrow_{*\leq}$-rules do not apply to $x\colon (\geq n\ R\ C)$ or $x\colon (\leq n\ R\ C)$. That means, every successor of x in the final complete system is already contained in S.

Both conditions guarantee that all constraints which are implicitly imposed on x by the system S are already explicitly present in S. Since all variables which are introduced in the further completion process are new and hence different from x, we will not get new unsatisfied constraints on x later on.

(3) Only small portions of a constraint system need to be kept in memory at a time. The idea for this comes from the following observation. Suppose x is a variable in S and no completion rule applies to constraints of the form $x\colon C$. Consequently, there will be no more replacements between successors of x. Suppose y and z are different successors of x. Since neither y will be replaced by z nor z will be replaced by y in the further completion process, there does not exist a variable u in the final constraint system such that u

can be reached both from y and z by role chains. This means that we can inspect the constraints which will be introduced by restrictions imposed on y independently from those introduced by restrictions imposed on z without loosing possible clashes. This is so because clashes are defined w.r.t. a single variable.

(4) To detect clashes of the form $x\colon (\leq n\ R\ C)$ in S, where $n_{R,C,S}(x) > n$, we have to consider only the R-successors of x. Thus an actual implementation may store for a variable x only its R-successors at a time. Furthermore, there is no need for explicitly storing constraints of the form xRy.

Taking these optimizations into account we will formulate a functional algorithm that decides the satisfiability of concepts. This algorithm is obtained from our rule-based calculus by imposing some control on the application of the completion rules.

The algorithm described in Figure 1 uses the functions *or* and *find*. We assume that the binary function *or* behaves as the LISP-*or*, i.e., the second argument is evaluated if and only if the first argument does not evaluate to *true*. Let L be a list and let f be a function. The call *find* l *in* L *such that* $f(l)$ iteratively takes an element l of L until the function f applied to l evaluates to *true*. In this case *find* immediately returns *true*. Otherwise, if there does no exist an element l in L such that f applied to l evaluates to *true*, *find* returns *false*.

Let C be a concept in negation normal form. Suppose the function *satisfiable* in Figure 1 is called with arguments x and $S = \{x\colon C\}$. Then *satisfiable* proceeds as follows. First, it checks whether S contains a clash. This check is restricted to constraints containing the variable x. If S is clash free, the $\rightarrow_{\sqcap}$- and $\rightarrow_{\sqcup}$-rules are applied until all constraints of the form $x\colon C \sqcap D$ or $x : C \sqcup D$ are decomposed. If no at-least restriction is imposed on x, i.e., there is no constraint $x : (\geq n\ R\ C)$ with $n > 0$ in S, we are done, and *satisfiable* returns *true*. (Note that in this case every at-most restriction imposed on x is trivially satisfied since we have not yet introduced successors of x.) Otherwise we generate for every role R which occurs in a constraint $x : (\geq n\ R\ C)$ with $n > 0$ R-successors for x. This is done by the call *generate-successors*(x, R, S). Note that successors of x which are related by different roles to x can be inspected independently from each other without loosing possible clashes.

Suppose that *generate-successors* is called with arguments x, R, and S. Then R-successors for the variable x are generated, which will be constrained according to the constraints imposed on x. The functions *satisfiable* and

satisfiable$(x, S) =$

    if $\{x\colon \bot\} \subseteq S$ *or* $\{x\colon A, x\colon \neg A\} \subseteq S$ *for some primitive concept* A
      then false
    elsif $x\colon C \sqcap D \in S$ *and* $(x\colon C \notin S$ *or* $x\colon D \notin S)$
      then satisfiable$(x, S \cup \{x\colon C,\ x\colon D\} \setminus \{x\colon C \sqcap D\})$
    elsif $x\colon C \sqcup D \in S$ *and* $x\colon C \notin S$ *and* $x\colon D \notin S$
      then satisfiable$(x, S \cup \{x\colon C\} \setminus \{x\colon C \sqcup D\})$ *or*
          satisfiable$(x, S \cup \{x\colon D\} \setminus \{x\colon C \sqcup D\})$
      else let $\mathcal{R} = \{R \mid x\colon (\geq n\ R\ C) \in S$ *for* $n > 0\}$
        for all $R \in \mathcal{R}$ *do:*
          generate-successors(x, R, S)

generate-successors$(x, R, S) =$

    if $x\colon (\leq n\ R\ C) \in S$, $|\{z \mid z$ *is a variable in* $S, z \neq x\}| > n$,
      $y\colon C \notin S$, $y\colon NNF(\neg C) \notin S$, *and* $x \neq y$
      then if $n = 0$
          then generate-successors$(x, R, S \cup \{y\colon NNF(\neg C)\})$
          else generate-successors$(x, R, S \cup \{y\colon C\})$ *or*
              generate-successors$(x, R, S \cup \{y\colon NNF(\neg C)\})$
    elsif $x\colon (\geq n\ R\ C) \in S$, $|\{z \mid z\colon C \in S, z \neq x\}| = m$ *and* $m < n$
      then let $y_{m+1}, \ldots, y_n$ *new variables:*
          generate-successors$(x, R, S \cup \{y_{m+1}\colon C, \ldots, y_n\colon C\})$
    elsif $x\colon (\leq n\ R\ C) \in S$, $|\{z \mid z\colon C \in S, z \neq x\}| > n$
      then let L *be a list of all pairs* (y, z) *such that*
          $y\colon C \in S$, $z\colon C \in S$, $y \neq x$, $z \neq x$, $y \neq z$, *and*
          the replacement of y *by* z *is safe in* S
        find (y, z) *in* L *such that:*
          generate-successors$(x, R, [y/z]S)$
    elsif $x\colon (\leq n\ R\ C) \in S$ *and* $|\{z \mid z\colon C \in S, z \neq x\}| > n$
      then false
      else for all variables $y \neq x$ *in* S*:*
        satisfiable$(y, \{y\colon C \mid y\colon C \in S\})$

Figure 1: A functional algorithm which decides satisfiability of concepts. The call *satisfiable*$(x, \{x : C\})$ returns *true* if and only if the concept C in negation normal form is satisfiable.

generate-successors are defined in such a way that every variable occurring in S which is not equal to x is an R-successor of x. Suppose $x: (\leq n\ R\ C)$ is in S. If the number of R-successors of x is not greater than n, then $x: (\leq n\ R\ C)$ is obviously satisfied in S. Thus, it is checked whether $|\{z \mid z \text{ is a variable in } S, z \neq x\}|$ (i.e., the number of R-successors of x) is greater than n. If this is the case, it is tested whether every R-successor of x is either in C or in $NNF(\neg C)$. Suppose y is a variable in S not equal to x. Thus y is an R-successor of x. Suppose further that neither $y: C$ nor $y: NNF(\neg C)$ is in S. If $x: (\leq 0\ R\ C)$ is in S, we add $y: NNF(\neg C)$ to S. Obviously, adding $y: C$ to S would lead to a clash. Otherwise, if $n > 0$, we cannot exclude one of the two possibilities. We only know that y is either in C or in $\neg C$. This nondeterminism is solved by generating two recursive calls of *generate-successors* covering both cases.

Next it is checked whether the $\rightarrow_{\geq}$-rule is applicable to S. We therefore have to consider the R-successors of x which are in C. Since every variable in S which is not equal to x is an R-successor of x, we compute the number $|\{z \mid z: C \in S, z \neq x\}|$. If this number is not less than the number n in $x: (\geq n\ R\ C)$, we are done since x has enough R-successors which are in C. Otherwise, constraints of the form $y_i: C$ are added to S, where y_i is a new variable.[1]

Now assume that every constraint of the form $x: (\geq n\ R\ C)$ is satisfied. Thus all possible R-successors of x are already in S. Next we consider constraints of the form $x: (\leq n\ R\ C)$. If there are too many R-successors for x which are in C, we reduce them by replacing a variable by another one. Safe replacements of variables guarantee that every at-least restriction imposed on x remains satisfied. Obviously, we are confronted with another nondeterminism since we do not know which replacements may lead to a clash free complete constraint system. Thus all possibilities are tested (in the worst case) with the use of *find*.

Now assume that no more safe replacements are possible in S. Obviously, we have now reached a configuration where the $\rightarrow_{\sqcap}$-, $\rightarrow_{\sqcup}$-, $\rightarrow_{choose}$-, $\rightarrow_{\geq}$-, and $\rightarrow_{*\leq}$-rule are not applicable to constraints containing the variable x. We now have to check whether an at-most constraint imposed on x is violated. If $x: (\leq n\ R\ C) \in S$ and $|\{z \mid z: C \in S, z \neq x\}| > n$, then there are too

[1] At this point one could further optimize the algorithm. Suppose $x: (\geq n_1\ R\ C_1)$, $\ldots$, $x: (\geq n_l\ R\ C_l)$ are all the at-least constraints imposed on x. If there is no at-most constraint imposed on x, or if each number occurring in an at-most constraint imposed on x is not less than $n_1 + \ldots + n_l$, there is no need to really introduce n_i R-successors of x which are in C_i for a constraint $x: (\geq n_i\ R\ C_i)$. In this case it is sufficient to generate exactly *one* R-successor as specimen.

many R-successors of x in C, and hence S contains a clash. Otherwise S is clash free w.r.t. x and our algorithm proceeds with inspecting the constraints imposed on the R-successors of x. Thus *generate-successors* calls, for every R-successor y of x, the function *satisfiable* with arguments y and S', where S' contains all the constraints imposed on y.

Theorem 4.1 *A concept C in negation normal form is satisfiable if and only if the call satisfiable($x, \{x: C\}$) returns true.*

The complexity of *satisfiable* strongly depends on how the numbers occurring in at-least restrictions are coded. If we assume that these numbers are written in the decimal system as usual, the function may need space which is exponential in the size of the input concept for the following reason. Given a constraint $x : (\geq n\ R\ C)$ the function adds the constraints $y_1 : C, \ldots, y_n : C$, which obviously needs exponential space in the size of the decimal representation of n. However, if we assume that the numbers are coded unary, i.e., a number n is represented by a string of the length n, then we strongly conjecture that *satisfiable* needs only polynomial space in the size of the input concept.

5 Remarks on the Complexity of Qualifying Number Restrictions

In this section we will first show that already some very small concepts of the presented language may cause long computations of the function *satisfiable*. Then we will discuss under which circumstances qualifying number restrictions behave better with respect to the computation time, and may thus be used in applications.

Since the satisfiability problem in our language is PSPACE-hard, one may ask whether a concept language providing qualifying number restrictions can be used in applications. To discuss this question let us reconsider why the satisfiability (and hence the subsumption) problem in the presented language has such a high complexity.

To decide whether a concept C is satisfiable our algorithm starts with the constraint system $S = \{x : C\}$ and computes (in the worst case) all complete constraint systems issuing from S using the completion rules. One can distinguish two forms of complexities:

- Complete constraint systems issuing from S may contain exponentially many constraints. In this case, the computation of such complete constraint systems needs exponential time w.r.t. the size of C.

- There may be exponentially many different complete constraint systems issuing from S.

In [Donini et al., 1991a] it has been pointed out that both forms of complexity are present in the concept language $\mathcal{ALC}$. Intuitively, one can say that the first form of complexity is responsible for NP-hardness and the second form for co-NP-hardness of the subsumption problem; if both forms come together they cause PSPACE-hardness. Though the satisfiability problem in $\mathcal{ALC}$ is PSPACE-complete, experience with an implemented system (see [Baader and Hollunder, 1990]) has revealed the following result. If an $\mathcal{ALC}$-concept which should be checked for satisfiability is not very long (i.e., if it contains less than 20 to 30 symbols of the form $\sqcap$, $\sqcup$, $\exists$, or $\forall$), the implemented algorithm for checking satisfiability is rather fast.[2]

In spite of the fact that $\mathcal{ALC}$ and our language are similar w.r.t. worst case complexity, our language does not show the nice behaviour of $\mathcal{ALC}$ for small concepts. It turns out that there exist concepts containing only a few symbols of the form $\sqcap$, $\geq$, or $\leq$, and small numbers for which the function *satisfiable* already needs a rather long time for computation. The reason for this difference seems to be that the second of the above mentioned sources of complexity has more influence for our language. This comes from the presence of the highly nondeterministic $\rightarrow_{choose}$- and $\rightarrow_{*\leq}$-rules. In contrast to constraints of the form $x\colon C \sqcup D$, where an application of the nondeterministic $\rightarrow_{\sqcup}$-rule produces exactly two alternatives, constraints containing at-most restrictions in general are responsible for a lot more alternatives.

Example 5.1 Suppose that *satisfiable*$(x, \{x\colon C\})$ is called with the concept

$$\begin{aligned} C &= (\geq 2\ R\ (A \sqcap B)) \sqcap (\geq 2\ R\ (A \sqcap C)) \sqcap \\ &\quad\ (\leq 2\ R\ A) \sqcap (\leq 2\ R\ B) \sqcap (\leq 2\ R\ C). \end{aligned}$$

After applying the $\rightarrow_{\sqcap}$-rule the function *generate-successors* is called with arguments x, R, and

$$\begin{aligned} S &= \{x\colon (\geq 2\ R\ (A \sqcap B)), x\colon (\geq 2\ R\ (A \sqcap C)), \\ &\quad\ x\colon (\leq 2\ R\ A), x\colon (\leq 2\ R\ B), x\colon (\leq 2\ R\ C)\}. \end{aligned}$$

[2] It is however not yet clear how the algorithm behaves for longer concepts which may occur in practical applications.

Then the constraints $y_1:(A\sqcap B)$, $y_2:(A\sqcap B)$, and $y_3:(A\sqcap C)$, $y_4:(A\sqcap C)$ are added to S to satisfy the at-least restrictions. Afterwards *generate-successors* checks whether the at-most restrictions imposed on x can be satisfied. Thus, by application of the $\rightarrow_{choose}$-rule for each y_i, $1 \leq i \leq 4$, $y_i: A$ or $y_i: \neg A$, $y_i: B$ or $y_i: \neg B$, and $y_i: C$ or $y_i: \neg C$ are added. Obviously, since there are 4 such successors of x, $8^4 = 4096$ alternatives are tested (in the worst case) in order to find a clash free complete constraint system.
Applications of the $\rightarrow_{*\leq}$-rule create additional alternatives. Assume that for each i, $1 \leq i \leq 4$, $y_i: A$ is added by the $\rightarrow_{choose}$-rule. Since there are 4 successors of x in the concept A, *generate-successors* has to replace 2 variables to satisfy $x: (\leq 2\, R\, A)$. In this case there are $\binom{4}{2} * \binom{3}{2} = 6 * 3$ possible replacements, which however need not all be safe.
Note that the overall number of alternatives caused by applications of the $\rightarrow_{choose}$- and $\rightarrow_{*\leq}$-rule is determined by the number of R-successors of x (which depends on the at-least restrictions imposed on x), the number of at-most restrictions imposed on x, and the numbers occurring in the at-most restrictions.

This example demonstrates that—in contrast to concepts of the $\mathcal{ALC}$-language—already small concepts containing qualifying number restrictions cause intractable problems. This fact is not necessarily an argument in favour of excluding qualifying number restrictions from concept languages. If it turns out that *in applications* subsumption relations can often be computed fast with the presented algorithm, this algorithm should be employed. Nevertheless, one cannot simply rely on the fact that the input concepts are "well-behaved".

To avoid long computations we propose the following modification of the functions *satisfiable* and *generate-successors* given in the previous section. The idea is to precompute the number of alternatives which have to be inspected in the worst case in order to find a clash free complete constraint system. The number of alternatives caused by applications of the $\rightarrow_{choose}$- and $\rightarrow_{*\leq}$-rule to a variable can be estimated as demonstrated in Example 5.1. If there are too many such alternatives, then *satisfiable* stops the computation and returns *fail*. Otherwise, *satisfiable* proceeds as described in the previous section.

Thus the modified algorithm behaves as follows. If it is confronted with concepts where only few alternatives have to be inspected, then it behaves similarly to the algorithm for $\mathcal{ALC}$. In this case it hopefully returns the correct answer in short time. Otherwise, if there arise too many alternatives

in the computation, then it would probably take a long time to compute the correct answer. In this case the algorithm stops and returns *fail.*

Using such a modified algorithm instead of an *incomplete* algorithm—as used in almost all implemented KL-ONE systems—has the following advantage. If a fast but incomplete algorithm cannot detect a subsumption relation between two concepts, we do not know anything. A subsumption relation may or may not exist. Consequently, we do not really know whether there is *no* subsumption relation between concepts, or whether the computation of this relationship would just take too long.

On the other hand, assume that we have a subsumption algorithm as proposed above. If the algorithm returns *yes* (*no*), then we really know that there exists (does not exist) a subsumption relation. That means, both positive *and* negative answers are correct. Otherwise the algorithm returns the answer *fail.* However, in this case the computation of the correct answer is a hard problem, and hence fast incomplete algorithms would probably fail too.

Summing up, we have argued that one may use qualifying number restrictions although the satisfiability and subsumption problem in a concept language providing these constructs is rather hard from a computational point of view. The proposed method uses a sound and complete algorithm which in advance recognizes concepts probably causing long computations. In this case *fail* is returned. However, if the algorithm returns *yes* or *no*, then we know that this answer is *correct.*

6 Conclusion

The present paper is a contribution to clarifying the subsumption problem in concept languages. A sound and complete satisfiability and hence subsumption algorithm has been presented for a language providing qualifying number restrictions. We have seen that qualifying number restrictions are very expressive language constructs; they generalize role quantifications and ordinary number restrictions. Nevertheless, the presented algorithm has a very simple structure. In the following we will discuss the methodology for devising subsumption algorithms for concept languages which led to this algorithm.

The completion technique as used in the present paper was first described in [Schmidt-Schauß and Smolka, 1988] for devising a subsumption algorithm

for the language $\mathcal{ALC}$, and then extended to other concept languages (see e.g. [Hollunder, Nutt, and Schmidt-Schauß, 1990, Donini et al., 1991b]). The ideas underlying these algorithms are as follows. Since the subsumption problem can be reduced to the satisfiability problem[3], it is sufficient to devise algorithms which check satisfiability of concepts, i.e., which check whether a given concept denotes a nonempty set in some interpretation. If C is a concept which should be checked for satisfiability, the algorithms start with the constraint system $\{x : C\}$. Then constraints are added by applications of completion rules until either a contradiction occurs, or an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}}$ is nonempty can be immediately obtained from the actual constraint system.

Each language construct gives rise to a particular completion rule. In general, it is relatively easy to determine for a given language construct a "corresponding" completion rule. This rule should satisfy two properties. On the one hand, it should be sound, i.e., it should satisfy one of the two properties stated in Proposition 3.3, depending on whether it is a deterministic or nondeterministic rule. On the other hand, it should be complete, i.e., if it is no longer applicable, the constraints containing the construct as uppermost operator should be "satisfied".

Now suppose that for each language construct of the concept language we have a "corresponding" sound and complete rule. We have seen that a concept C is not satisfiable if and only if every complete constraint system (i.e., a constraint system such that no completion rule is applicable) issuing from $\{x : C\}$ contains a clash[4]. Thus, to get a decision procedure for the satisfiability problem one has to guarantee that there is no infinite chain of applications of completion rules issuing from $\{x : C\}$. In general, it is more complicated to prove termination of the completion rules than to prove soundness and completeness of these rules (see Section 3). In fact, it would be relatively easy to generate sound and complete rules for constructs which cause undecidability such as *role value maps* [Schmidt-Schauß, 1989]. However, it is clear that in this case termination cannot be guaranteed in general.

For implementation purposes one should not be content with the rule-based algorithm obtained by applying this methodology. Practical experience has shown that—for the sake of efficiency—it is important to impose an

[3]Recall that C subsumes D if and only if $\neg C \sqcap D$ is not satisfiable. In [Donini et al., 1991a] it is shown how the reduction works if negation of concepts is not available in the concept language.

[4]Recall that clashes are obvious contradictions in constraint systems such as $x : A$, $x : \neg A$ or $x : \bot$.

appropriate control on the rule applications. In the best case one may obtain a functional algorithm as demonstrated in Section 4.

We have pointed out that deciding satisfiability in our language has a high complexity. To avoid long computations we have modified a sound and complete satisfiability algorithm such that concepts probably causing long computations are recognized. In this case *fail* is returned. Otherwise, if the algorithm returns *yes* or *no*, then we know that—in both cases—this answer is *correct*.

Concept languages are used in KL-ONE systems to represent terminological knowledge. However, almost all of these systems have, in addition to the terminological component ("T-Box"), an assertional component ("A-Box") which allows to describe knowledge concerning particular individuals. To reason with both the T-Box and A-Box one may need algorithms for inferences such as consistency checking of the represented knowledge and "realization". It has been demonstrated in [Hollunder, 1990] that the completion technique can also be used to devise sound and complete algorithms for these inferences.

Acknowledgements

We are grateful to our colleagues Hans-Jürgen Bürckert and Werner Nutt for reading earlier drafts of the present paper. This research has been supported by the German Bundesministerium für Forschung und Technologie under grant ITW 8903 0.

References

[Baader and Hollunder, 1990] F. Baader, B. Hollunder. *KRIS : Knowledge Representation and Inference System—System Description.* DFKI Technical Memo TM-90-03, DFKI Kaiserslautern.

[Borgida et al., 1989] A. Borgida, R. J. Brachman, D. L. McGuinness, L. A. Resnick. "CLASSIC: A Structural Data Model for Objects." In *Proceedings of the International Conference on Management of Data*, Portland, Oregon, 1989.

[Brachman et al., 1985] R. J. Brachman, V. Pigman Gilbert, H. J. Levesque. "An essential hybrid reasoning system: knowledge and symbol level accounts in KRYPTON." In *Proceedings of the 9th IJCAI*, pp. 532–539, Los Angeles, Cal., 1985.

[Brachman and Schmolze, 1985] R. J. Brachman, J. G. Schmolze. "An Overview of the KL-ONE knowledge representation system." *Cognitive Science*, 9(2):171-216, 1985.

[Dershowitz and Manna, 1979] N. Dershowitz, Z. Manna. "Proving Termination with Multiset Orderings." *C. ACM*, 22(8):465-476, 1979.

[Donini et al., 1991a] F. Donini, B. Hollunder, M. Lenzerini, A. Marchetti Spaccamela, D. Nardi, W. Nutt. *The Complexity of Existential Quantification in Concept Languages.* DFKI Research Report RR-91-02, DFKI Kaiserslautern.

[Donini et al., 1991b] F. Donini, M. Lenzerini, D. Nardi, W. Nutt. "Complexities in Terminological Reasoning." In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, Mas., 1991.

[Hollunder, Nutt, and Schmidt-Schauß, 1990] B. Hollunder, W. Nutt, M. Schmidt-Schauß. "Subsumption Algorithms for Concept Description Languages." In *Proceedings of the 9th ECAI*, pp. 348–353, Stockholm, Sweden, 1990.

[Hollunder, 1990] B. Hollunder. "Hybrid Inferences in KL-ONE-based Knowledge Representation Systems." In *Proceedings of the 14th German Workshop on Artificial Intelligence*, pp. 38–47, Eringerfeld, Germany, 1990.

[MacGregor and Bates, 1987] R.MacGregor, R.Bates. *The* LOOM *Knowledge Representation Language.* Technical Report ISI/RS-87-188, University of Southern California, Information Science Institute, Marina del Rey, Cal., 1987.

[Nebel, 1990] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in AI, LNAI 422, Springer Verlag, 1990.

[Nebel and Smolka, 1990] B. Nebel, G. Smolka. "Representation and Reasoning with Attributive Descriptions." In K. H. Bläsius, U. Hedtstück and C.-R. Rollinger (editors.), *Sorts and Types for Artificial Intelligence*, Lecture Notes in AI, LNAI 418, Springer-Verlag, 1990.

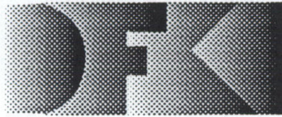
[Owsnicki-Klewe, 1990] B. Owsnicki-Klewe. "A Cardinality-Based Approach to Incomplete Knowledge." In *Proceedings of the 9th ECAI*, pp. 491-496, Stockholm, Sweden, 1990.

[Patel-Schneider, 1984] P. Patel-Schneider. "Small can be beautiful in knowledge representation." In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, pp. 11–16, Denver, Colo., 1984.

[Schmidt-Schauß, 1989] M. Schmidt-Schauß. "Subsumption in KL-ONE is undecidable." In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pp. 421–431, Toronto, Ont., 1989.

[Schmidt-Schauß and Smolka, 1988] M. Schmidt-Schauß, G. Smolka. *Attributive Concept Descriptions with Complements.* SEKI Report SR-88-21, FB Informatik, Universität Kaiserslautern, 1988. To appear in Artificial Intelligence, 47, 1991.

[Vilain, 1985] M. B. Vilain. "The restricted language architecture of a hybrid representation system." In *Proceedings of the 9th IJCAI*, pp. 547–551, Los Angeles, Cal., 1985.



DFKI
-Bibliothek-
Stuhlsatzenhausweg 3
6600 Saarbrücken 11
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:

1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka :
On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt:
The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger:
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

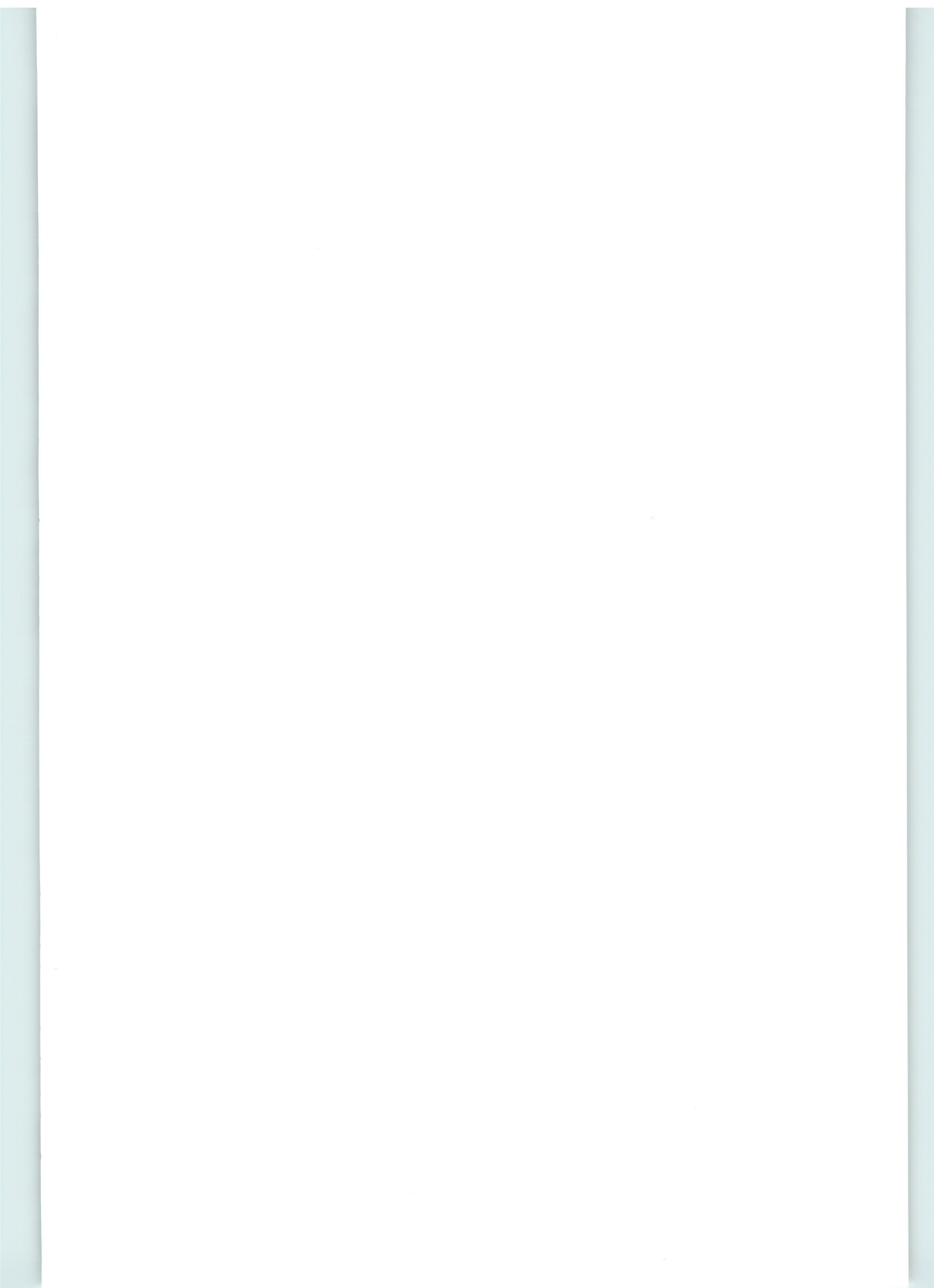
D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten



Qualifying Number Restrictions in Concept Languages

Bernhard Hollunder, Franz Baader

RR-91-03

Research Report