# Subsumption Algorithms for Concept Languages

Bernhard Hollunder Werner Nutt
German Research Center for Artificial Intelligence
Postfach 2080
D-6750 Kaiserslautern, West Germany
e-mail: {hollunde, nutt}@informatik.uni-kl.de

**Abstract**

We investigate the subsumption problem in logic-based knowledge representation languages of the KL-ONE family and give decision procedures. All our languages contain as a kernel the logical connectives conjunction, disjunction, and negation for concepts, as well as role quantification. The algorithms are rule-based and can be understood as variants of tableaux calculus with a special control strategy.

In the first part of the paper, we add number restrictions and conjunction of roles to the kernel language. We show that subsumption in this language is decidable, and we investigate sublanguages for which the problem of deciding subsumption is PSPACE-complete.

In the second part, we amalgamate the kernel language with feature descriptions as used in computational linguistics. We show that feature descriptions do not increase the complexity of the subsumption problem.

# Contents

# 1 Introduction

Concept languages of the KL-ONE family are a means of expressing taxonomical knowledge by describing hierarchies of concepts [BL84, Pat84, BS85, BPGL85, KBR86, Vil85, MB87, NvL88, Neb89]. In contrast to earlier knowledge representation formalisms like frames and semantic networks, KL-ONE languages have the advantage of a Tarski style declarative semantics that allows to conceive them as sublanguages of predicate logic [BL84]. Concepts are intended to be descriptions of classes of objects. Essentially, such a description is given in terms of primitive classes and attributes of objects. A related family of formalisms emerged in computational linguistics with unification based grammars. Here, constituents of sentences are described in terms of attributes (so-called features) and their values [Sh86, Smo88]. Recently, concept languages have been investigated as a means to describe object oriented data models [BBMR89].

A concept is built up of two kinds of primitive symbols, concepts and roles. An interpretation interprets them as subsets of a domain and binary relations over the domain. These primitives can be combined by various language constructs yielding complex concepts, which again are interpreted as subsets of the domain. Different languages are distinguished by the constructs they provide.

Examples for primitive concepts may be $\mathsf{person}$ and $\mathsf{female}$, examples for primitive roles may be $\mathsf{child}$ and $\mathsf{female\_relative}$. If logical connectives like conjunction, disjunction, and negation are present as description constructs, one can describe the concept of "persons that are not female" by the expression

$$\mathsf{person} \sqcap \neg\mathsf{female}.$$

Conjunction, disjunction, and negation are interpreted as set intersection, union, and complement. Most languages provide quantification over roles that allows for instance to describe the concepts of "individuals having a female child" and "individuals for which all children are female" by the expressions

$$\exists\mathsf{child}.\mathsf{female} \qquad \text{and} \qquad \forall\mathsf{child}.\mathsf{female}.$$

Number restrictions on roles denote sets of individuals having at least or at most a certain number of fillers for a role. For instance,

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

can be read as "all individuals having at least three friends and at most two children." We also provide a role-forming construct, namely conjunction of roles, that allows for instance to define the role

$$\mathsf{child} \sqcap \mathsf{female\_relative},$$

which intuitively yields the role "daughter." It is straightforward to give a formal semantics to role quantification, number restrictions and role conjunction that captures the intuitive meaning.

Only recently, the close relation between KL-ONE-languages and feature formalisms has been pointed out [Smo88]. In this paper we introduce a language that encorporates both KL-ONE-one constructs and features. Features are functional roles, that is they are supposed to have at most one filler. Natural examples for features may be $\mathsf{father}$ and $\mathsf{first\_name}$. The selection operator for features can be used to describe the concept of "all individuals whose father has at most one child" by the expression

$$\mathsf{father}.(\leq 1\,\mathsf{child}).$$

Agreement of feature chains allows to describe "all individuals whose father and grandfather have the same first name" by the expression

$$\mathsf{father}\ \mathsf{first\_name} \doteq \mathsf{father}\ \mathsf{father}\ \mathsf{first\_name}.$$

Interestingly, agreements of feature chains are computationally tractable, whereas agreements of arbitrary role chains cause undecidability [Sch89].

Concepts implicitly form a hierarchy: a concept $C$ is subsumed by a concept $D$ if in every interpretation the set denoted by $C$ is a subset of the set denoted by $D$. The basic reasoning facility provided by a KL-ONE system is a subsumption checker. For a long time, the KL-ONE community was content with sound, but incomplete subsumption algorithms. Such an algorithm delivers a correct answer when given $C$ and $D$ such that $C$ is not subsumed by $D$, but sometimes fails to recognize that one concept is subsumed by another one.

Until recently, a decision procedure was only known for the rather trivial language $\mathcal{FL}^-$, offering as constructs conjunction, $\forall P.C$, and $\exists P.\top$ ,where $P$ may be a primitive role and $\top$ is a concept denoting the entire domain of an interpretation [BL84]. Several complexity results showed that already for seemingly slight extensions of $\mathcal{FL}^-$ the subsumption problem is co-NP-hard [LB87, Neb88]. Other work identified languages with undecidable subsumption problem [Pat89, Sch89, Sch88].

The first nontrivial subsumption algorithm was given by Schmidt-Schauß and Smolka [SS88] for the language $\mathcal{ALC}$, which extends $\mathcal{FL}^-$ by allowing for arbitrary logical connectives and role quantification as constructs. The algorithm is even optimal, since it requires linear space and they show that subsumption checking in $\mathcal{ALC}$ is PSPACE-complete.

In contrast to former subsumption algorithms, their algorithm is formulated as a satisfiability checking algorithm. An interpretation $\mathcal{I}$ is a model of the concept $C$ if $C$ denotes a nonempty set in $\mathcal{I}$. A concept is satisfiable if it has a model and unsatisfiable otherwise. A satisfiability checking algorithm also yields a subsumption checking algorithm, since $C$ is subsumed by $D$ if and only if $C \sqcap \neg D$ is unsatisfiable.

The purpose of this paper is twofold. First, we give satisfiability—and therefore subsumption—checking algorithms for substantial extensions of the language $\mathcal{ALC}$. These extensions include number restrictions ($\mathcal{ALCN}$), role conjunction ($\mathcal{ALCR}$), and features ($\mathcal{ALCF}$). The algorithms for these three languages require polynomial space. Since the corresponding problems generalize the satisfiability problem for $\mathcal{ALC}$, which is known to be PSPACE-complete, these algorithms can be considered optimal. Furthermore, we give a procedure to decide satisfiability for $\mathcal{ALCNR}$, a language that contains both number restrictions and subroles.

Second, we want to show that there is a general technique of devising subsumption algorithms for most of the languages that have been reported in the literature. The declarative semantics of concepts allows to view primitive concepts as unary predicates and primitive roles and features as binary predicates. This identification can be extended to concepts by associating to every concept $C$ a predicate logic formula $\phi_C(x)$. For instance, to the concept

$$C = \exists \mathsf{child}.\mathsf{female} \sqcap \forall \mathsf{child}.\mathsf{person}$$

corresponds the formula

$$\phi_C(x) = \exists y.(\mathsf{child}(x,y) \land \mathsf{female}(y)) \land \forall z.(\mathsf{child}(x,z) \to \mathsf{person}(z)).$$

A model of the formula $\exists x.\phi_C(x)$ is a model of the concept $C$ and vice versa. In particular, $C$ is unsatisfiable if and only if $\exists x.\phi_C(x)$ is unsatisfiable.

A careful analysis shows that first order tableaux calculus [Sm68] always terminates for such formulas, and exhibits a model if the formula is satisfiable, or produces obvious contradictions if the formula is not satisfiable. In particular it follows that a formula $\exists x.\phi_C(x)$ has a finite model if it has a model at all. Based on this observation one could devise a simple minded satisfiability checker that consists of two components: a refutation theorem prover and a procedure that for a given formula enumerates all finite interpretations and tests whether they are models. If both processes start with input $\exists x.\phi_C(x)$ and run in parallel, the theorem prover will eventually find out that that formula is unsatisfiable if it is, and the interpretation tester will eventually exhibit a model if there is one. The tableaux calculus combines the characteristics of both processes.

The algorithmic technique which we propose basically consists in applying tableaux calculus with some control strategy to formulas obtained from concepts. The algorithms are described by tableaux calculus like rules operating on so-called constraints, which directly correspond to logical formulae. The control is incorporated into the conditions that allow to apply the rules. The idea of a rule based calculus operation on constraints was already underlying the calculus in [SS88], although its presentation obscured its intimate relation to tableaux calculus.

We feel that this technique could be applied as well to other KL-ONE-language constructs that have not been considered in this paper, like inverse roles or agreement of roles. Conversely, an algorithm for a very general language, like $\mathcal{ALCNR}$ or $\mathcal{ALCF}$, can be used as a starting point to devise algorithms for sublanguages. Finally, complexity results show that the algorithms obtained using this technique are often optimal. For instance, the ones described in this paper require polynomial space and solve PSPACE-complete problems.

In the next chapter we formally introduce syntax and semantics of the language $\mathcal{ALCNR}$. In chapter 3 we give a satisfiability checking algorithm for the entire language. In chapter 4 we consider two sublanguages of $\mathcal{ALCNR}$ and show that their satisfiability—and therefore subsumption—problems are PSPACE-complete. Finally in chapter 5 we amalgamate the language $\mathcal{ALC}$ with features and give a satisfiability checking algorithm for this language.

## 2 Concept Languages

In this section we introduce syntax and semantics of the concept language $\mathcal{ALCNR}$, which contains arbitrary logical connectives for concepts, role quantification, number restrictions, and intersection of roles.

We assume that two alphabets of symbols, called *primitive concepts* and *primitive roles*, are given. The letter $A$ will always denote a primitive concept, and the letter $P$ will always denote a primitive role.

Arbitrary *roles* (denoted by the letters $Q$ and $R$) are built out of primitive roles according to the syntax rule

$$Q, R \longrightarrow \; P \mid Q \sqcap R.$$

The *concepts* (denoted by the letters $C$ and $D$) of the language $\mathcal{ALCNR}$ are built out of primitive concepts and roles according to the syntax rule

$$\begin{array}{lcll} C, D & \longrightarrow & A \mid & \text{(primitive concept)} \\ & & \top \mid & \text{(top concept)} \\ & & \bot \mid & \text{(bottom concept)} \\ & & C \sqcap D \mid & \text{(intersection)} \\ & & C \sqcup D \mid & \text{(union)} \\ & & \neg C \mid & \text{(complement)} \\ & & \forall R.C \mid \exists R.C \mid & \text{(role quantification} \\ & & (\leq n\, R) \mid (\geq n\, R) & \text{(number restriction),} \end{array}$$

where $n$ ranges over the nonnegative integers coded as binary strings. In the following we will refer to $\mathcal{ALCNR}$-concepts simply as concepts. A *subconcept* of a concept $C$ is a substring of $C$ that is a concept.

Concepts are interpreted as subsets of a domain and roles are interpreted as binary relations over a domain. More precisely, an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$ (the *domain* of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the *interpretation function* of $\mathcal{I}$) that maps every concept to a subset of $\Delta^{\mathcal{I}}$ and every role to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that the following equations are satisfied:

$$\begin{array}{rcl} \top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\ \bot^{\mathcal{I}} & = & \emptyset \\ (C \sqcap D)^{\mathcal{I}} & = & C^{\mathcal{I}} \cap D^{\mathcal{I}} \end{array}$$

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

Observe that an interpretation is uniquely determined by the values that it gives to primitive concepts and primitive roles.

Intersection of roles can be used to model subroles. We say that $Q$ is a *subrole* of a $R$ and write $Q \preceq R$, if $Q^{\mathcal{I}} \subseteq R^{\mathcal{I}}$ for every interpretation $\mathcal{I}$. If $Q = P_1 \sqcap \ldots \sqcap P_k$ and $R = P'_1 \sqcap \ldots \sqcap P'_l$, then $Q$ is a subrole of $R$, if and only if for every primitive role $P'_j$ occurring in $R$ there is a role $P_i$ occurring in $Q$ such that $P_i = P'_j$.

An interpretation $\mathcal{I}$ is a *model* for a concept $C$ if $C^{\mathcal{I}}$ is nonempty. A concept is *satisfiable* if it has a model and *unsatisfiable* otherwise. We say $C$ is *subsumed* by $D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$, and $C$ is *equivalent* to $D$ if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$.

Since our language allows for general complements, satisfiability and subsumption can be reduced to each other in linear time. Therefore the two problems are of equal complexity, and to decide them it suffices to devise a decision procedure for only one of them.

**Proposition 2.1**

- *$C$ is subsumed by $D$ if and only if $C \sqcap \neg D$ is not satisfiable*
- *$C$ is satisfiable if and only if $C$ is not subsumed by $\bot$.*

In the following chapters we give algorithms that decide the satisfiability of concepts. By the above proposition, these algorithms can be used as well for subsumption checking.

To keep our algorithms simple, we do not allow arbitrary concepts as input, but only concepts in a certain normal form. A concept is called *simple* if it contains only complements of the form $\neg A$, where $A$ is a primitive concept. Simple concepts are the analogue of logical formulae in negation normal form. Arbitrary concepts can be rewritten to equivalent simple concepts by the following rules:

$$
\begin{array}{rcl}
\neg\top & \longrightarrow & \bot \\
\neg\bot & \longrightarrow & \top \\
\neg(C \sqcap D) & \longrightarrow & \neg C \sqcup \neg D \\
\neg(C \sqcup D) & \longrightarrow & \neg C \sqcap \neg D \\
\neg\neg C & \longrightarrow & C \\
\neg(\forall R.C) & \longrightarrow & \exists R.\neg C \\
\neg(\exists R.C) & \longrightarrow & \forall R.\neg C \\
\neg(\leq n\, R) & \longrightarrow & (\geq n+1\, R) \\
\neg(\geq n\, R) & \longrightarrow & \left\{ \begin{array}{cl} \bot & \text{if } n = 0 \\ (\leq n-1\, R) & \text{if } n > 0. \end{array} \right.
\end{array}
$$

**Proposition 2.2** *For every concept one can compute in linear time an equivalent simple concept.*

In this paper we consider three sublanguages of $\mathcal{ALCNR}$:

$\mathcal{ALCN}$ consists of concepts containing no intersections of roles.

$\mathcal{ALCR}$ consists of concepts containing no number restrictions.

$\mathcal{ALC}$ consists of concepts containing no number restrictions and no intersections of roles.

The language $\mathcal{ALC}$ has been introduced by Schmidt-Schauß and Smolka [SS88]. They proved that subsumption and satisfiability in this language are PSPACE-complete problems. The names of the other languages are chosen in such a way as to indicate that they extend $\mathcal{ALC}$ either by number restrictions or role intersection or both.

In this paper we will prove that $\mathcal{ALCNR}$ has a decidable satisfiability problem. For the sublanguages $\mathcal{ALCN}$ and $\mathcal{ALCR}$ we will give satisfiability checking algorithms, which need polynomial space. Since both languages generalize $\mathcal{ALC}$, we know that satisfiability and subsumption for $\mathcal{ALCN}$ and $\mathcal{ALCR}$ are PSPACE-complete problems.

# 3 Checking Satisfiability

We are going to devise a calculus for checking the satisfiability of concepts. The calculus will operate on constraints consisting of variables, concepts and roles. Concepts are supposed to be in normal form.

We assume that there exists an alphabet of variable symbols, which will be denoted by the letters $x$, $y$, and $z$. A constraint is a syntactic object of one of the forms

$$x{:}\,C, \quad xPy,$$

where $C$ is a simple concept and $P$ is a primitive role. Intuitively, $x{:}\,C$ says that $x$ is in the interpretation of $C$ and $xPy$ says that the pair $(x, y)$ is in in the interpretation of $P$.

Let $\mathcal{I}$ be an interpretation. An *$\mathcal{I}$-assignment* is a function $\alpha$ that maps every variable to an element of $\Delta^{\mathcal{I}}$. We say that $\alpha$ *satisfies* $x{:}\,C$ if $\alpha(x) \in C^{\mathcal{I}}$, and $\alpha$ *satisfies* $xPy$ if $(\alpha(x), \alpha(y)) \in P^{\mathcal{I}}$. A constraint $c$ is *satisfiable* if there is an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment $\alpha$ such that $\alpha$ satisfies $c$. A *constraint system* $S$ is a finite, nonempty set of constraints. An $\mathcal{I}$-assignment $\alpha$ *satisfies* a constraint system $S$ if $\alpha$ satisfies every constraint in $S$. A constraint system $S$ is *satisfiable* if there is an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment $\alpha$ such that $\alpha$ satisfies $S$.

**Proposition 3.1** *A simple concept $C$ is satisfiable if and only if the constraint system $\{x{:}\,C\}$ is satisfiable.*

Our calculus starts with a constraint system $S = \{x{:}\,C\}$. In successive steps it adds constraints to $S$ until either a contradiction is generated or an interpretation satisfying $C$ can be obtained from the resulting system. The calculus relies on a set of rules, which closely resemble the rules of the tableaux calculus for first order logic. In fact if one translates concepts into predicate logic formulas, and applies to them the tableaux calculus with a suitable control strategy, one obtains essentially the calculus described here.

Before we formulate the rules we need some notation. Let $S$ be a constraint system and $R = P_1 \sqcap \ldots \sqcap P_k$ be a role. We say that *$xRy$ holds in $S$* if the constraints $xP_1y, \ldots, xP_ky$ are in $S$.

In order to denote for a variable $x$ the number of variables $y$ such that $xRy$ holds we define

$$n_{R,S}(x) := |\{y \mid xRy \text{ holds in } S\}|.$$

With $[y/z]S$ we denote the constraint system obtained from $S$ by replacing each occurrence of $y$ by $z$. We say that the replacement of $y$ by $z$ is *safe in $S$* if for every variable $x$ and for every role $R$ such that $x\colon(\geq n\,R)$ is in $S$ and $xRy$ and $xRz$ hold in $S$, we have $n_{R,S}(x) > n$. Intuitively, the replacements of $y$ by $z$ is safe in $S$, if every number restriction that is satisfied in $S$ remains satisfied in $[y/z]S$.

The calculus is given by the following *completion rules*:

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

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

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

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

3. $S \rightarrow_{\exists} \{xP_1y, \ldots, xP_ky,\ y\colon C\} \cup S$

   if $x\colon \exists R.C$ is in $S$, $R = P_1 \sqcap \ldots \sqcap P_k$, there is no $z$ such that $xRz$ holds in $S$ and $z\colon C$ is in $S$, and $y$ is a new variable

4. $S \rightarrow_{\forall} \{y\colon C\} \cup S$

   if $x\colon \forall R.C$ is in $S$, $xRy$ holds in $S$, and $y\colon C$ is not in $S$

5. $S \rightarrow_{\geq} \{xP_1y, \ldots, xP_ky\} \cup S$

   if $x\colon(\geq n\,R)$ is in $S$, $R = P_1 \sqcap \ldots \sqcap P_k$, $n_{R,S}(x) < n$, and $y$ is a new variable

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

   if $x\colon(\leq n\,R)$ is in $S$, $xRy$ and $xRz$ holds in $S$, $n_{R,S}(x) > n$, and the replacement of $y$ by $z$ is safe in $S$.

Obviously, the $\rightarrow_{\sqcap}$-, $\rightarrow_{\sqcup}$-, $\rightarrow_{\exists}$-, and $\rightarrow_{\forall}$-rule imitate the tableaux rules for conjunctions, disjunctions, and existentially quantified formulas.

The $\rightarrow_{\geq}$- and the $\rightarrow_{\leq}$-rule operate in a more complicated way to cope with equality, that comes in through number restrictions. For instance, translating the number restriction $(\geq 2\,P)$ we obtain the the formula

$$\phi_{(\geq 2\,P)}(x) = \exists y_1, y_2.\, P(x, y_1) \wedge P(x, y_2) \wedge \neg(y_1 \doteq y_2),$$

which contains equality. Applying the $\rightarrow_{\geq}$-rule twice to a constraint $x: (\geq 2\, P)$ yields two constraints $xPy_1, xPy_2$. Now, the $\rightarrow_{\leq}$-rule is designed in such a way that after subsequent applications there will always be two constraints of the form $xPy_1', xPy_2'$, since it is allowed to replace one variable by another one only if the replacement is safe.

We distinguish two kinds of completion rules, *deterministic* ones ($\rightarrow_{\sqcap}$, $\rightarrow_{\exists}$, $\rightarrow_{\forall}$, $\rightarrow_{\geq}$) and *nondeterministic* ones ($\rightarrow_{\sqcup}$, $\rightarrow_{\leq}$). The nondeterministic rules correspond to concept constructs that contain disjunction, when translated into logic. Obviously, the union of concepts is the analogue of the disjunction of formulae. Although less obvious, disjunction is also present in "atmost" restrictions.

To see this, observe that for instance translating the concept $(\leq 2\, P)$ into logic yields the formula

$$\begin{aligned}\phi_{(\leq 2\, P)}(x) = \ & \forall y_1, y_2, y_3.\, P(x, y_1) \wedge P(x, y_2) \wedge P(x, y_3) \\ & \qquad\qquad \longrightarrow\ y_1 \doteq y_2 \vee y_1 \doteq y_3 \vee y_2 \doteq y_3.\end{aligned}$$

Consequently, when the $\rightarrow_{\leq}$-rule applies, there is a choice as to which variables are to be identified.

**Proposition 3.2 (Invariance)** *Let $S$ and $S'$ be constraint systems. Then:*

1. *If $S'$ is obtained from $S$ by application of a deterministic rule, then $S$ is satisfiable if and only if $S'$ is satisfiable.*

2. *If $S'$ is obtained from $S$ by application of a nondeterministic 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.*

**Proposition 3.3 (Termination)** *Let $C$ be a simple concept. Then:*

1. *There is no infinite chain of completion steps issuing from $\{x: C\}$.*

2. *The lenght of a chain of completion steps issuing from $\{x: C\}$ is bounded exponentially in the size of $C$.*

A constraint system is *complete* if no propagation rule applies to it. A *clash* is a constraint system having one of the following forms:

- $\{x: \bot\}$
- $\{x: A,\ x: \neg A\}$
- $\{x: (\leq 0\, R),\ xP_1y, \ldots, xP_ky\}$, where $R = P_1 \sqcap \ldots \sqcap P_k$
- $\{x: (\geq m\, Q),\ x: (\leq n\, R)\}$, where $m > n$ and $Q \preceq R$.

**Proposition 3.4** *A complete constraint system is satisfiable if and only if it contains no clash.*

*Proof.* Obviously, a system containing a clash is unsatisfiable. Conversely, if $S$ is a clash free complete system, then there is an interpretation $\mathcal{I}$ obtained by taking for $\Delta^{\mathcal{I}}$ all variables occurring in $S$, for $A^{\mathcal{I}}$ all $x$ such that $x: A$ is in $S$, for $P^{\mathcal{I}}$ all pairs $(x, y)$ such that $xPy$ is in $S$, and by taking the sets $C^{\mathcal{I}}$ for complex concepts $C$ and $R^{\mathcal{I}}$ for complex roles $R$ as required by the definition of an interpretation. The $\mathcal{I}$-assignment mapping every variable to itself satisfies $S$. $\square$

It is straightforward to turn the calculus into a decision procedure. To check a simple concept $C$ for satisfiability, one has to generate all complete constraint systems derivable from $\{x: C\}$, which are, up to variable renaming, finitely many. If all these systems contain a clash, then $C$ is unsatisfiable, otherwise it is satisfiable.

**Theorem 3.5** *Satisfiability and subsumption of $\mathcal{ALCNR}$-concepts can be decided with nondeterministic exponential time.*

**Corollary 3.6** *An $\mathcal{ALCNR}$-concept has a model if and only if it has a finite model.*

# 4 PSPACE-Complete Languages

Satisfiability of $\mathcal{ALC}$-concepts is a PSPACE-complete problem [SS88]. Since $\mathcal{ALC}$ is a sublanguage of $\mathcal{ALCNR}$ it follows that checking satisfiability of $\mathcal{ALCNR}$-concepts is PSPACE-hard. This gives a lower complexity bound. An upper bound is furnished by our calculus that requires nondeterministic exponential time. Better upper bounds exist for special cases. In Section 4.1

we show that satisfiability of $\mathcal{ALCN}$-concepts can be decided with quadratic space. In Section 4.2 we give a linear space algorithm for checking satisfiability of $\mathcal{ALCR}$-concepts. Since both problems are still more general than the satisfiability of $\mathcal{ALC}$-concepts, it follows that they are PSPACE-complete. The algorithms are inspired by a technique first applied in [SS88]. We conclude with a remark on the influence of binary and unary coding of number restrictions on the complexity of deciding satisfiability of $\mathcal{ALCNR}$-concepts.

## 4.1 Satisfiability of $\mathcal{ALCN}$-Concepts

The reason why the general calculus is so cumbersome is in the $\rightarrow_{\geq}$-rule. If a constraint system contains a constraint $x\colon(\geq n\,R)$, the algorithm will apply the $\rightarrow_{\geq}$-rule to add constraints until the extended system will contain $n$ variables $y_1,\ldots,y_n$ such that $xRy_1,\ldots,xRy_n$ holds. Since $n$ is assumed to be coded in binary, the number of additional constraints is exponential in the length of the string representing $n$.

When all roles in a constraint system are primitive roles, a satisfiability checking procedure has to do less work when it encounters the above situation. We will show that in this case the $\rightarrow_{\geq}$-rule needs to be applied at most once, adding a single constraint of the form $xPy$.

A constraint system $S$ is *quasi-complete* if

1. $S$ is obtained from $\{x\colon C\}$, for some simple concept $C$, by application of the completion rules;

2. the $\rightarrow_{\sqcap}$-, $\rightarrow_{\forall}$-, $\rightarrow_{\exists}$-, $\rightarrow_{\sqcup}$- and $\rightarrow_{\leq}$-rules don't apply to $S$;

3. $x\colon(\geq n\,R)\in S$ always implies $n_{R,S}(x)>0$.

Note that every complete constraint system is quasi-complete. A quasi-complete constraint system is complete if it doesn't contain a constraint $x\colon(\geq n\,R)$ with $n_{R,S}(x)<n$.

A clash free quasi-complete constraint system that contains role intersections need not be satisfiable.

**Example 4.1** Consider the constraint system

$$S=\Big\{x\colon(\leq 2\,P),\ x\colon(\geq 1\,P\sqcap P_1),\ x\colon(\geq 2\,P\sqcap P_2),\ x\colon\forall P_1.A,\ x\colon\forall P_2.\neg A\Big\}.$$

Then $S' = S \cup \{xPy_1,\ xP_1y_1,\ y_1\colon A,\ xPy_2,\ xP_2y_2,\ y_2\colon \neg A\}$ is quasi-complete and clash free. But completing $S'$ will yield a clash, because the number restriction on $P \sqcap P_2$ forces us to introduce further constraints $xPy'_2$ and $xP_2y'_2$, and the number restriction on $P$ forces us to identify $y_1$ with $y_2$ or $y'_2$, thus producing a variable constrained to $A$ and $\neg A$.

If roles are primitive, number restrictions cannot interact via subroles.

**Theorem 4.2** *Let $S$ be a quasi-complete constraint system such that all roles occurring in $S$ are primitive. Then $S$ is satisfiable if and only if it contains no clash.*

*Proof.* Suppose $S$ is a clash free quasi-complete constraint system such that all roles occurring in $S$ are primitive. We show that there exists a clash free complete constraint system $S'$ such that $S \subseteq S'$.

Such an extension $S'$ can be obtained from $S$ constructing a sequence $S = S_1, S_2, \ldots, S_k = S'$ where $S_j$ is transformed into $S_{j+1}$ using the following steps:

- select a variable $x$ with $\{x\colon (\geq n\, P),\ xPy\} \subseteq S_j$ and $n_{P,S_j}(x) = m < n$;
- let $y_1, \ldots, y_{n-m}$ be new variables;
- let $S_{j+1}$ be obtained from $S_j$ by adding $xPy_1, \ldots, xPy_{n-m}$, by adding $y_1\colon C, \ldots, y_{n-m}\colon C$ for every constraint in $S_j$ of the form $y\colon C$, and by adding $y_1Pz, \ldots, y_{n-m}Pz$ for every constraint in $S_j$ of the form $yPz$.

The process eventually halts when a complete constraint system is reached. Since the newly introduced constraints are copies of constraints already occurring in $S$, there is no clash in $S'$. □

**General Assumption.** *In the rest of this section, all concepts are $\mathcal{ALCN}$-concepts and all constraint systems contain only $\mathcal{ALCN}$-concepts.*

A quasi-complete constraint system that originated from $\{x\colon C\}$ may still be exponential in the size of $C$. For a polynomial space algorithm it is therefore crucial not to keep an entire quasi-complete constraint system in the memory but to store only small portions of it at a time. To make this idea more precise we need the following definitions.

Let $S$ be a constraint system, $P$ a primitive role, and $x$ a variable occurring in $S$. In order to denote the number of constraints in $S$ of the form $x\colon \exists P.C$ we define

$$\mathit{ex}_S^P(x) := |\{C \mid x\colon \exists P.C \in S\}|.$$

In order to denote the minimal "atmost" constraint imposed on $x$ for $P$ in $S$ we let $N = \{n \mid x\colon (\leq n\, P) \in S\}$ and define

$$\mathit{atmost}_S^P(x) := \begin{cases} \min N & \text{if } N \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

Let $X$ be a finite set and $k$ a positive integer with $k < |X|$. A *$k$-partition* of $X$ is a set $\Pi$ containing $k$ pairwise disjoint subsets $\pi$ of $X$ such that $\bigcup_{\pi \in \Pi} \pi = X$.

Now we give transformation rules that build up portions of quasi-complete constraint systems. The *trace rules* consist of the $\rightarrow_{\sqcap}$-, $\rightarrow_{\forall}$- and the $\rightarrow_{\sqcup}$-rule together with the following three rules:

1. $S \rightarrow_{T\geq} \{xPy\} \cup S$
   if $x\colon (\geq n\, P)$ is in $S$, $\mathit{ex}_S^P(x) = 0$,
   there is no constraint $xP'z$, and $y$ is a new variable
2. $S \rightarrow_{T\exists} \{y\colon C,\ xPy\} \cup S$
   if $x\colon \exists P.C$ is in $S$, $\mathit{atmost}_S^P(x) \geq \mathit{ex}_S^P(x)$,
   there is no constraint $xP'z$, and $y$ is a new variable
3. $S \rightarrow_{T\exists\leq} \{xPy\} \cup \{y\colon C \mid C \in \pi\} \cup S$
   if $\mathit{ex}_S^P(x) = l$, $\mathit{atmost}_S^P(x) = k$, $l > k$,
   $\{x\colon \exists P.C_1,\ \ldots,\ x\colon \exists P.C_l\} \subseteq S$,
   $\pi \in \Pi$ where $\Pi$ is a $k$-partition of $\{C_1, \ldots, C_l\}$,
   there is no constraint $xP'z$, and $y$ is a new variable.

The trace rules are designed to produce for a variable $x$ at most one variable $y$ that is related to $x$ by constraints of the form $xPy$. The only completion rules introducing such constraints are the $\rightarrow_{\geq}$- and the $\rightarrow_{\exists}$-rule. The trace rules prescribe a controlled application of these rules. The $\rightarrow_{T\geq}$-rule allows the $\rightarrow_{\geq}$-rule to be applied only if no constraint $x\colon \exists P.C$ is present in $S$. The $\rightarrow_{T\exists}$-rule allows the $\rightarrow_{\exists}$-rule to be applied if there exist constraints of the form $x\colon \exists P.C$ in $S$, and the number of such constraints doesn't exceed the minimal "atmost" restriction imposed on $x$. However, if $l$ is the number of constraints in $S$ having the form $x\colon \exists P.C$, $k$ is the minimal "atmost" restriction for $x$ in $S$, and $l$ exceeds $k$, then applications of the $\rightarrow_\exists$-rule would introduce variables $y_1, \ldots, y_l$ satisfying constraints $\{xPy_j,\ y_j\colon C_j\}$, where $1 \leq j \leq l$. Subsequent applications of the $\rightarrow_\leq$-rule would identify some of the new variables, thus creating a $k$-partition of the set $\{C_1, \ldots, C_l\}$. The $\rightarrow_{T\exists\leq}$-rule models the combined application of the two rules.

Let $C$ be a simple concept and let $T$ be a constraint system obtained from $\{x\colon C\}$ by application of the trace rules. We call $T$ a *trace* of $\{x\colon C\}$, if no trace rule applies to $T$.

**Proposition 4.3** *Let $C$ be a simple concept and $S = \{x\colon C\}$. Then:*

1. *The length of a trace rule derivation issuing from $S$ is bounded linearly in the size of $C$.*
2. *Every trace of $S$ is contained in a quasi-complete constraint system extending $S$.*
3. *Every quasi-complete constraint system extending $S$ can be obtained as the union of finitely many traces of $S$.*

To detect clashes it suffices to inspect traces.

**Proposition 4.4** *Let $C$ be a simple concept, $S$ a quasi-complete constraint system extending $\{x\colon C\}$, and $\mathcal{T}$ a finite set of traces such that $S = \bigcup_{T\in\mathcal{T}} T$. Then $S$ contains a clash if and only if some $T \in \mathcal{T}$ contains a clash.*

Suppose $C$ is a simple concept, and the recursive function *sat1* in Figure 1 is called with arguments $x$ and $S = \{x\colon C\}$. Then *sat1* returns *true* if and only if there exists a clash free quasi-complete constraint system extending $S$.

Nijenhuis and Wilf [NW75] give a linear space algorithm that enumerates all partitions of a finite set. We can use a slightly modified version of their algorithm enumerating only $k$-partitions to find an appropriate $k$-partition of $\{C_1, \ldots, C_l\}$. We assume that such an algorithm is called as a subprocedure by *sat1*.

The function *sat1* can be executed such that at most one trace needs to be kept in memory. Furthermore, for every variable in such a trace we *sat1*: *variable x constraint system* $\rightarrow$ *bool*

*sat1*$(x, S) =$
*if* $S$ *contains a clash*
*then false*
*elsif* $x{:}\, C \sqcap D \in S$ *and* $x{:}\, C \notin S$ *or* $x{:}\, D \notin S$
*then sat1*$(x, S \cup \{x{:}\, C,\ x{:}\, D\})$
*elsif* $x{:}\, C \sqcup D \in S$ *and* $x{:}\, C \notin S$ *and* $x{:}\, D \notin S$
*then sat1*$(x, S \cup \{x{:}\, C\})$ *or sat1*$(x, S \cup \{x{:}\, D\})$
*else let* $y$ *be a new variable in:*
*forall* $x{:}\, (\geq n\, P) \in S$
*with* $ex_S^P(x) = 0$*:*
*sat1*$(y, S \cup \{y{:}\, C \mid x{:}\forall P.C \in S\})$
*and*
*forall* $x{:}\, \exists P.C \in S$
*with* $ex_S^P(x) \leq atmost_S^P(x)$*:*
*sat1*$(y, S \cup \{y{:}\, C\} \cup \{y{:}\, D \mid x{:}\forall P.D \in S\})$
*and*
*forall* $\{x{:}\, \exists P.C_1, \ldots, x{:}\, \exists P.C_l\} \subseteq S$
*with* $l = ex_S^P(x)$, $k = atmost_S^P(x)$, $l > k$*:*
*exists a* $k$*-partition* $\Pi$ *of* $\{C_1, \ldots, C_l\}$ *such that:*
*forall* $\pi \in \Pi$*:*
*sat1*$(y, S \cup \{y{:}\, C \mid C \in \pi\} \cup \{y{:}\, D \mid x{:}\forall P.D \in S\})$

Figure 1: A functional algorithm deciding the satisfiability of $\mathcal{ALCN}$-concepts. The call *sat1*$(x, \{x{:}\, C\})$ returns *true* if and only if $C$ is satisfiable.

have to store information on the corresponding $k$-partition that is currently investigated, which can be achieved using linear space. Thus, *sat1* can be executed using space quadratic in the size of $C$.

Since satisfiability of $\mathcal{ALC}$-concepts is PSPACE-complete [SS88] and $\mathcal{ALC}$ is a sublanguage of $\mathcal{ALCN}$, we have proven the main result of this section.

**Theorem 4.5** *Satisfiability and subsumption of* $\mathcal{ALCN}$*-concepts are*

*PSPACE-complete problems, which can be decided with quadratic space.*

## 4.2 Satisfiability of $\mathcal{ALCR}$-Concepts

In this section we give an algorithm for checking the satisfiability of $\mathcal{ALCR}$-concepts.

**General Assumption.** *In this section, all concepts are $\mathcal{ALCR}$-concepts and all constraint systems contain only $\mathcal{ALCR}$-concepts.*

Since now concepts do not contain number restrictions, we can complete constraint systems without using the $\rightarrow_{\geq}$- and the $\rightarrow_{\leq}$-rule.

**Proposition 4.6**

1. *A constraint system is complete if and only if the $\rightarrow_{\sqcap}$-, $\rightarrow_{\forall}$-, $\rightarrow_{\exists}$-, and the $\rightarrow_{\sqcup}$-rule don't apply.*
2. *A complete constraint system is satisfiable if and only if it contains no clash.*

Similarly as in the previous section, we first define a set of trace rules, then consider the traces that can be computed with them, and finally give a functional algorithm that checks for a given simple concept $C$ the satisfiability of $\{x{:}\, C\}$.

The *trace rules* consist of the $\rightarrow_{\sqcap}$-, $\rightarrow_{\sqcup}$-, $\rightarrow_{\forall}$-rule and the following rule:

$$S \rightarrow_{T\exists} \{xP_1y, \ldots, xP_ky,\ y{:}\, C\} \cup S$$
if $x{:}\, \exists R.C$ is in $S$, $R = P_1 \sqcap \ldots \sqcap P_k$, there is no constraint of the form $xPz$ in $S$, and $y$ is a new variable.

The $\rightarrow_{T\exists}$-rule ensures that in a trace for a variable $x$ at most one constraint of the form $xPy$ is generated.

Let $C$ be a simple $\mathcal{ALCR}$-concept and let $T$ be a constraint system obtained from $S = \{x{:}\, C\}$ by application of the trace rules. We call $T$ a *trace* of $S$, if no trace rule applies to $T$. The traces defined in this section have properties similar to those stated in the previous section. In particular, Propositions 4.3 and 4.4 hold again if quasi-complete constraint systems are replaced by complete constraint systems.

```
sat2: variable x constraint system → bool

sat2(x, S) =
    if S contains a clash
      then false
    elsif x: C ⊓ D ∈ S and x: C ∉ S or x: D ∉ S
      then sat2(x, S ∪ {x: C, x: D})
    elsif x: C ⊔ D ∈ S and x: C ∉ S and x: D ∉ S
      then sat2(x, S ∪ {x: C}) or sat2(x, S ∪ {x: D})
      else let y be a new variable in:
         forall x: ∃R.C ∈ S
            sat2(y, S ∪ {y: C} ∪ {y: D | x: ∀Q.D ∈ S and R ⪯ Q})
```

Figure 2: A functional algorithm deciding the satisfiability of $\mathcal{ALCR}$-concepts. The call *sat2*$(x, \{x\colon C\})$ returns *true* if and only if $C$ is satisfiable.

The recursive function *sat2* in Figure 2 returns *true* if and only if for the constraint system $\{x\colon C\}$ given as argument there exists a clash free complete system extending $\{x\colon C\}$. The function can be executed such that at most one trace needs to be kept in memory. When implemented naively, the algorithm needs quadratic space to store a trace, since it creates constraints of the form $x\colon D$ where $D$ is a subconcept of $D$. But if subconcepts are represented by pointers instead of copies, then a trace needs at most space linear in the size of $C$. Thus, *sat2* needs at most space linear in the size of the input.

Since deciding the satisfiability of $\mathcal{ALC}$-concepts is a special case of deciding the satisfiability of $\mathcal{ALCR}$-concepts, we have proven the following result.

**Theorem 4.7** *Satisfiability and subsumption of $\mathcal{ALCR}$-concepts are PSPACE-complete problems, which can be decided with linear space.*

## 4.3 Numbers as a Source of Complexity

We don't know if a PSPACE-algorithm for checking the satisfiability of $\mathcal{ALCNR}$-concept descriptions exists. Here we give arguments why there is no straightforward way of applying the trace technique to the problem.

It is possible to split a complete constraint system into traces, whenever it is sufficient to consider for a variable $x$ only one variable $y$ such that $xPy$ holds. As seen in Example 4.1, this is not the case when number restrictions and role intersections together are present. Our method to cope with the interaction of number restrictions and role intersections required to introduce for every constraint of the form $x\colon (\geq n\, R)$ occurring in a constraint system at least $n$ new constraints of the form $xPy$ such that $xRy_1, \ldots, xPy_n$ holds. If $n$ is coded in binary, this method yields a number of additional constraints that is exponential in the length of the string representing $n$. To store these constraints, one would need exponential space.

However, if numbers are coded in unary– that is, the number $n$ is represented by a string of $n$ equal symbols– then linear space would suffice to store the new constraints. In this case one could employ the trace technique to devise a quadratic space algorithm as follows: for a variable $x$ in a constraint system $S$, the algorithm would

1. introduce all constraints of the form $xPy$ as required by the constraints of the form $x\colon (\geq n\, R)$ and $x\colon \exists R.C$ in $S$,

2. identify individual variables as required by the constraints of the form $x\colon (\leq n\, R)$ in $S$,

3. proceed with a variable $y$ where $xPy$ is in the updated constraint system.

# 5 Combining Concepts and Feature Logic

Feature descriptions emerged in computational linguistics as a device to describe constituents of sentences in terms of attributes, which are called features in this context [Sh86]. Examples for such features may be gender, number, or voice. The main operation on feature descriptions is a test for consistency.

In his paper on "Feature Logic", Smolka gave a Tarsky style semantics for feature descriptions and showed that they are closely related to concepts of KL-ONE-like languages [Smo88], the main difference being that in Feature

Logic attributes (called features) are interpreted as partial functions while in KL-ONE attributes (called roles) are interpreted as arbitrary binary relations.

Features come with the constructs selection, and agreement and disagreement of feature chains. Selection corresponds to existential quantification of roles. Agreement and disagreement are counterparts of KL-ONE's so-called "role value maps" [BS85]. Smolka shows that a feature language that contains intersection, union and complement of feature descriptions, as well as selection, agreement and disagreement on features, has an NP-complete satisfiability problem and a co-NP-complete subsumption problem. Interestingly, if agreement is used with roles, then it causes undecidability [Sch89]. Notationally very similar, one minor difference in the semantics causes major computational differences.

In this chapter we amalgamate the language $\mathcal{ALC}$ with feature logic by adding selection, agreement and disagreement operators for features and show that a tableaux like calculus is also applicable to this combination.

## 5.1 Syntax and Semantics of $\mathcal{ALCF}$-Concepts

We assume a further alphabet of symbols, called *features*, that is disjoint from the alphabets of primitive concepts and primitive roles. The letter $f$ will always denote a feature.

A *path*, denoted by the letter $p$ or $q$, is a sequence $f_1 \cdots f_n$ of features. The empty path is denoted by $\varepsilon$.

Concepts (denoted by $C$ and $D$) of the language $\mathcal{ALCF}$ are formed out of primitive concepts, primitive roles, and features according to the syntax rule

$$
\begin{array}{rcll}
C, D & \longrightarrow & A \mid \top \mid \bot \mid C \sqcap D \mid C \sqcup D \mid \neg C \mid \forall P.C \mid \exists P.C \mid & \\
 & & p\uparrow \mid & \text{(undefinedness)} \\
 & & f.C \mid & \text{(selection)} \\
 & & p \doteq q \mid & \text{(agreement)} \\
 & & p \neq q & \text{(disagreement).}
\end{array}
$$

Thus, $\mathcal{ALCF}$ adds to $\mathcal{ALC}$ undefinedness, selection, agreement and disagreement of features.

A *feature interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$ and a function $\cdot^{\mathcal{I}}$ mapping concepts to subsets of $\Delta^{\mathcal{I}}$, roles to binary relations on $\Delta^{\mathcal{I}}$, and features and paths to partial functions from $\Delta^\mathcal{I}$ to $\Delta^\mathcal{I}$ such that the following equations are satisfied (by *dom* we denote the domain of partial functions):

$$\begin{aligned}
\varepsilon^\mathcal{I}(a) &= a \text{ for every } a \in \Delta^\mathcal{I}\text{, i.e. } \varepsilon^\mathcal{I} \text{ is the identity function on } \Delta^\mathcal{I} \\
(fp)^\mathcal{I}(a) &= p^\mathcal{I}(f^\mathcal{I}(a)) \\
(p\uparrow)^\mathcal{I} &= \Delta^\mathcal{I} \setminus dom\, p^\mathcal{I} \\
(f.C)^\mathcal{I} &= \{a \in dom\, f^\mathcal{I} \mid f^\mathcal{I}(a) \in C^\mathcal{I}\} \\
(p \doteq q)^\mathcal{I} &= \{a \in dom\, p^\mathcal{I} \cap dom\, q^\mathcal{I} \mid p^\mathcal{I}(a) = q^\mathcal{I}(a)\} \\
(p \not\doteq q)^\mathcal{I} &= \{a \in dom\, p^\mathcal{I} \cap dom\, q^\mathcal{I} \mid p^\mathcal{I}(a) \neq q^\mathcal{I}(a)\}.
\end{aligned}$$

An interpretation that interprets features as binary relations is a feature interpretation if for every feature $f$ it satisfies the axiom

$$\forall x, y, z.\, f(x, y) \wedge f(x, z) \;\longrightarrow\; y \doteq z.$$

Observe that feature axioms contain equality. This will show up again in the rules of the calculus.

The selection $f.C$ denotes the set of all elements of the domain for which the feature $f$ is defined and for which the application of $f$ yields an element of the set denoted by $C$. The agreement $p \doteq q$ of two paths $p$ and $q$ denotes the set of all elements of the domain for which $p$ and $q$ are both defined and the application of $p$ and $q$ yields the same element as result. The disagreement $p \not\doteq q$ of $p$ and $q$ denotes the set of all elements of the domain for which $p$ and $q$ are both defined and the application of $p$ and $q$ yields different elements as result.

**General Assumption.** *In the following concepts are always understood to be $\mathcal{ALCF}$-concepts.*

A feature interpretation $\mathcal{I}$ is a *feature model* for $C$ if $C^\mathcal{I}$ is nonempty. Furthermore, satisfiability, subsumption and equivalence of concepts are defined with respect to feature interpretations instead of arbitrary interpretations.

Suppose $f$ is a feature, $P$ a primitive role, and $\mathcal{I}$ a feature interpretation with domain $\Delta^\mathcal{I}$ such that for all $a$, $b \in \Delta^\mathcal{I}$ we have $(a, b) \in P^\mathcal{I}$ if and only if $a \in dom\, f^\mathcal{I}$ and $f^\mathcal{I}(a) = b$. That is, $P^\mathcal{I}$ is the graph of the partial function $f^\mathcal{I}$. Then the following equations hold for all concepts $C$:

$$\begin{aligned}
(\exists P.C)^\mathcal{I} &= (f.C)^\mathcal{I} \\
(\forall P.C)^\mathcal{I} &= (f.C \sqcup \neg f.\top)^\mathcal{I}.
\end{aligned}$$

Thus, role quantification of $P$ can be expressed in terms of the selection operator, if $P$ is interpreted as a partial function. Conversely, feature selection can be expressed by existential quantification.

A new element comes into the language via agreements and disagreements of feature paths. The equations

$$\begin{array}{rcl}(p \doteq q)^{\mathcal{I}} & = & (p.\top \sqcap q.\top \sqcap \neg(p \neq q))^{\mathcal{I}} \\ (p \neq q)^{\mathcal{I}} & = & (p.\top \sqcap q.\top \sqcap \neg(p \doteq q))^{\mathcal{I}}\end{array}$$

show, that the agreements can be expressed by disagreements and vice versa.

As in Chapter 2 we single out a special class of concepts as normal forms. A concept is called *simple* if it contains only complements of the form $\neg A$, where $A$ is a primitive concept, and no subconcepts of the form $p\uparrow$ where $p$ is not a feature.

We transform concepts into simple concepts preserving equivalence by rewriting with the rules in Chapter 2 and the following rules:

$$\begin{array}{rcl}\neg f.C & \longrightarrow & f\uparrow \sqcup f.\neg C \\ \neg p \doteq q & \longrightarrow & p\uparrow \sqcup q\uparrow \sqcup p \neq q \\ \neg p \neq q & \longrightarrow & p\uparrow \sqcup q\uparrow \sqcup p \doteq q \\ \varepsilon\uparrow & \longrightarrow & \bot \\ (fp)\uparrow & \longrightarrow & f\uparrow \sqcup f.(p\uparrow)\end{array}$$

**Proposition 5.1** *For every concept one can compute in linear time an equivalent simple concept.*

## 5.2 Checking Satisfiability

Next, we extend our calculus for checking the satisfiability of $\mathcal{ALCNR}$-concepts so that it can cope with features. First we introduce two new kinds of constraints. Constraints now have one of the following forms:

$$x{:}\,C,\ \ xPy,\ \ xpy,\ \ x \neq y,$$

where $C$ is a simple concept, $P$ is a primitive role, and $p$ is a path. Let $\mathcal{I}$ be a feature interpretation and let $\alpha$ be an $\mathcal{I}$-assignment. We say that $\alpha$ *satisfies*

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

$xPy$, if $(\alpha(x), \alpha(y)) \in P^{\mathcal{I}}$

$xpy$, if $\alpha(x) \in dom\, p^{\mathcal{I}}$ and $p^{\mathcal{I}}(\alpha(x)) = \alpha(y)$

$x \neq y$, if $\alpha(x) \neq \alpha(y)$.

As before, constraint systems are nonempty finite sets of constraints. Satisfiability of constraints and constraint systems are now defined in terms of feature interpretations instead of interpretations as in Chapter 2. Again, a simple concept $C$ is satisfiable if and only if the constraint system $\{x: C\}$ is satisfiable.

To deal with the new language constructs involving features we define the following *feature completion rules*:

1. $S \rightarrow_{selection} \{xfy,\ y: C\} \cup S$

   if $x: f.C$ is in $S$, there is no variable $z$ such that $xfz$ and $z: C$ are in $S$, and $y$ is a new variable

2. $S \rightarrow_{\doteq} \{xpy,\ xqy\} \cup S$

   if $x: p \doteq q$ is in $S$, there is no variable $z$ such that $xpz$ and $xqz$ are in $S$, and $y$ is a new variable

3. $S \rightarrow_{\not\doteq} \{xpy,\ xqz,\ y \neq z\} \cup S$

   if $x: p \neq q$ are in $S$, there are no variables $y'$, $z'$ such that $xpy'$, $xqz'$ and $y' \neq z'$ are in $S$, and $y$, $z$ are new variables

4. $S \rightarrow_{path} \{xfz,\ zpy\} \cup S$

   if $xfpy$ is in $S$, $p \neq \varepsilon$, there is no variable $z'$ such that $xfz'$ and $z'py$ are in $S$, and $z$ is a new variable

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

   if $xfy$ and $xfz$ are in $S$, and $y \neq z$.

The $\rightarrow_{selection}$-rule for features is the analogue of the $\rightarrow_{\exists}$-rule for roles. The $\rightarrow_{\doteq}$- and $\rightarrow_{\not\doteq}$-rule work on agreement and disagreement constraints. Path constraints like $xpy$, that are produced by application of these rules, are stepwise shortened by the $\rightarrow_{path}$-rule by stripping off the first feature symbol. The $\rightarrow_{function}$-rule reflects the assumption that features are interpreted as partial functions. It corresponds to the feature axioms coming in implicitly with every feature symbol.

The next proposition says that the feature rules are deterministic.

**Proposition 5.2 (Invariance)** *If the constraint system $S'$ is obtained from the constraint system $S$ by application of a feature completion rule, then $S'$ is consistent if and only if $S$ is consistent.*

The $\mathcal{ALCF}$*-completion rules* consist of $\rightarrow_{\sqcap}$-, $\rightarrow_{\forall}$-, $\rightarrow_{\exists}$-, and $\rightarrow_{\sqcup}$-rule together with the feature completion rules. A constraint system is *complete* if no $\mathcal{ALCF}$-completion rule applies to it.

**Proposition 5.3 (Termination)** *Let $C$ be a simple concept. Then:*

1. *There is no infinite chain of $\mathcal{ALCF}$-completion steps issuing from $\{x\colon C\}$.*
2. *The lenght of a chain of $\mathcal{ALCF}$-completion steps issuing from $\{x\colon C\}$ is bounded exponentially in the size of $C$.*

A *clash* is a constraint system having one of the following forms:

$$\{x\colon \bot\}, \quad \{x\colon A,\ x\colon \neg A\}, \quad \{x\colon f\uparrow\}, \quad \{x \neq x\}.$$

**Proposition 5.4** *A complete constraint system is satisfiable if and only if it contains no clash.*

*Proof.* Let $S$ be a constraint system. If $S$ contains a clash, then it is obviously unsatisfiable.

If $S$ is complete, then for every variable $x$ and every feature $f$ there is at most one variable $y$ such that $xfy$ is in $S$. Now it is easy to see that there is an interpretation $\mathcal{I}$ obtained by taking for $\Delta^{\mathcal{I}}$ all variables occurring in $S$, for $A^{\mathcal{I}}$ all $x$ such that $x\colon A$ is in $S$, for $P^{\mathcal{I}}$ all pairs $(x,y)$ such that $xPy$ is in $S$, for $f^{\mathcal{I}}$ the partial function with

$$dom\, f^{\mathcal{I}} = \{x \mid \exists y.\ xfy \in S\}$$

that is defined by $f^{\mathcal{I}}(x) = y$ if $xfy$ is in $S$. The $\mathcal{I}$-assignment mapping every variable to itself satisfies $S$. $\square$

If $C$ is a simple concept, then, up to variable renaming, one can obtain only finitely many complete constraint systems from $\{x: C\}$ using the $\mathcal{ALCF}$-completion rules. With the preceding proposition it follows that $C$ is satisfiable if and only if there is one system among these complete systems that contains no clash.

**Theorem 5.5** *Satisfiability and subsumption of $\mathcal{ALCF}$-concepts are decidable.*

**Corollary 5.6** *An $\mathcal{ALCF}$-concept has a feature model if and only if it has a finite feature model.*

## 5.3 PSPACE-Completeness

In this section we show that satisfiability of $\mathcal{ALCF}$-concepts is a PSPACE-complete problem. Since satisfiability of $\mathcal{ALC}$-concepts, which are contained in $\mathcal{ALCF}$, is known to PSPACE-complete, it is sufficient to give a polynomial space algorithm solving this problem.

To this purpose, the trace technique applied in the previous chapter has to be modified. The crucial observation that led to traces was that to detect clashes it is sufficient to generate for a variable $x$ at most one variable $y$ that is related to $x$ by a constraint of the form $xPy$ at time. For the $\mathcal{ALCF}$-algorithm however, it is important to make a distinction between roles and features. As before, the algorithm will introduce for a given $x$ only one variable $y$ and one constraint of the form $xPy$. But it has to introduce as many variables as required by the feature completion rules in order to compute all consequences of agreement and disagreement constraints.

We first define feature trace rules, then we consider feature traces computed with them, and finally we give a functional algorithm for checking the satisfiability of $\mathcal{ALCF}$-concepts.

The *feature trace rules* consist of the $\rightarrow_\sqcap$-, $\rightarrow_\forall$-, $\rightarrow_\sqcup$-rule, the feature completion rules, and the following rule:

$$S \rightarrow_{T\exists} \{y: C,\ xPy\} \cup S$$

if $x: \exists P.C$ is in $S$, there is no constraint of the form $xPz$ in $S$, and $y$ is a new variable

The $\rightarrow_{T\exists}$-rule is a restriction of the $\rightarrow_{\exists}$-rule designed such that for every variable $x$ at most one constraint of the form $xPy$ is produced.

Let $C$ be a simple concept and let $T$ be a constraint system obtained from $\{x: C\}$ by application of the feature trace rules. We call $T$ a *feature trace* of $\{x: C\}$ if no feature trace rule applies to $T$.

Feature traces have properties similar to those of traces stated in Section 4.1.

**Proposition 5.7** *Let $C$ be a simple concept, $S = \{x: C\}$, and $T$ a feature trace of $S$. Then:*

1. *If $xPy$ and $xP'y'$ are in $T$, then $P = P'$ and $y = y'$, and if $xfy$ and $xfy'$ are in $T$, then $y = y'$.*
2. *The length of a feature trace rule derivation transforming $S$ into $T$ is bounded linearly in the size of $S$.*
3. *Every feature trace of $S$ is contained in a complete constraint system extending $S$.*
4. *Every complete constraint system extending $S$ can be obtained as the union of finitely many feature traces of $S$.*

**Proposition 5.8** *Let $C$ be a concept, $S'$ be a complete constraint system extending $\{x: C\}$, and let $\mathcal{T}$ be a finite set of traces such that $S' = \bigcup_{T \in \mathcal{T}} T$. Then $S'$ contains a clash if and only if some $T \in \mathcal{T}$ contains a clash.*

Let $S$ and $S'$ be constraint system. We say that $S'$ is a *feature completion* of $S$, if $S'$ is obtained from $S$ by application of the feature completion rules, and the feature completion rules don't apply to $S'$. Observe that two feature completions of a constraint system $S$ are equal up to variable renaming.

Let $C$ be a simple concept. The recursive function *sat3* employs a strategy in generating feature traces of $\{x: C\}$. It applies all feature completion rule as long as possible and only then applies the $\rightarrow_{T\exists}$-rule. The function checks whether $\{x: C\}$ has a clash free $\mathcal{ALCF}$-completion and can therefore be used to decide the satisfiability of $C$. Again, one can choose a suitable data structure to represent the subconcepts of $C$ occurring in feature traces, that does not use copies of subconcepts but represents them by pointers. With such a data structure a trace of $C$ can be stored using space linear in the size of $C$. From this observation we conclude the main result of this chapter.

```
sat3: variable x constraint system → bool

sat3(x, S) =
    if S contains a clash
      then false
    elsif x: C ⊓ D ∈ S and x: C ∉ S or x: D ∉ S
      then sat3(x, S ∪ {x: C, x: D})
    elsif x: C ⊔ D ∈ S and x: C ∉ S and x: D ∉ S
      then sat3(x, S ∪ {x: C}) or sat3(x, S ∪ {x: D})
    elsif a feature rule is applicable to S
      then let S' be a feature completion of S in:
            forall new variables y in S':
            sat3(y, S')
    else let y be a new variable in:
            forall x: ∃R.C ∈ S:
            sat3(y, S ∪ {y: C} ∪ {y: D | x: ∀R.D ∈ S})
```

Figure 3: A functional algorithm deciding the satisfiability of $\mathcal{ALCF}$-concepts. The call $sat3(x, \{x{:}\,C\})$ returns *true* if and only if $C$ is satisfiable.

**Theorem 5.9** *Satisfiability and subsumption of $\mathcal{ALCF}$-concepts are PSPACE-complete problems, which can be decided with linear space.*

# 6 Conclusion

This paper is a contribution to exploring the frontier between concept languages with decidable and such with undecidable subsumption problem. Former efforts concentrated on finding minimal languages with undecidable subsumption problem [Pat89, Sch88, Sch89]. We complement this work by givingv satisfiability and subsumption checking algorithms for languages that are, to the best of our knowledge, the richest for which these problems are known to be decidable. Nevertheless, we feel that they can still be extended by further constructs, like inverse roles and agreement of arbitrary roles. For an extended algorithm one would have to introduce constraints and completion rules corresponding to the new constructs, and a control structure governing the rule application.

A second contribution of this paper is that it exemplifies a method of designing subsumption algorithms. The method is based on the observation that subsumption can always be reduced to satisfiability. This problem can then be decided with a calculus based on inference rules that closely resemble those of the tableaux calculus for first order logic. Complexity results show that in most cases an algorithm based on these ideas is optimal. The algorithms described in this paper require polynomial space and solve PSPACE-complete problems. Similar results hold for sublanguages of $\mathcal{ALCNR}$ with respect to other complexity classes [DHL*90].

Finally, we showed that concept languages of the KL-ONE-family and feature based description languages as developed in computational linguistics not only are intimately related as regards their semantics (cf. [Smo88]) but also can be treated with similar algorithmic techniques. We presented the language $\mathcal{ALCF}$ that combines concepts and feature terms, and showed that adding features did not increase the complexity of the satisfiability and the subsumption problem.

## Acknowledgements

We are grateful to Manfred Schmidt-Schauß, Francesco M. Donini, Maurizio Lenzerini, and Daniele Nardi for many discussions on the topics of this paper. Francesco's idea to apply tableaux calculus to subsumption problems made us aware that the rule based algorithm developed by Schmidt-Schauß and Smolka was in fact a disguised tableaux calculus.

# References


[BBMR89] A. Borgida, R. J. Brachmann, 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.

[BL84] R. J. Brachmann, H. J. Levesque. "The tractability of subsumption in frame based description languages." In *Proceedings of the 4th National Conference of the AAAI*, pp. 34–37, Austin, Tex., 1984.

[BPGL85] 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.

[BS85] R. J. Brachman, J. G. Schmolze. "An Overview of the KL-ONE Knowledge Representation System." Cognitive Science 9(2), pp. 171–216, 1985.

[DHL*90] F. Donini, B. Hollunder, M. Lenzerini, A. Marchetti Spaccamela, Daniele Nardi, W. Nutt. *A Source of Complexity in Terminological Reasoning*. DFKI-Report, DFKI, Postfach 2080, D-6750 Kaiserslautern, West Germany. Forthcoming.

[Hol89] B. Hollunder. *Subsumption Algorithms for Some Attributive Concept Description Languages*. SEKI Report SR-89-16, FB Informatik, Universität Kaiserslautern, D-6750, Kaiserslautern, West Germany, 1988.

[KBR86] T. S Kaczmarek, R. Bates, G. Robins. "Recent developments in NIKL." In *Proceedings of the 5th National Conference of the AAAI*, pp. 578–587, Philadelphia, Pa., 1986.

[LB87] H. J. Levesque, R. J. Brachman. "Expressiveness and tractability in knowledge representation and reasoning." *Computational Intelligence*, **3**:78–93, 1987.

[MB87] 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.

[Neb88] B. Nebel. "Computational complexity of terminological reasoning in BACK." *Artificial Intelligence*, **34**(3):371–383, 1988.

[Neb89] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, PhD thesis, Universität des Saarlandes, Saarbrücken, West Germany, 1989. To appear in Lecture Notes in Artificial Intelligence, Springer Verlag.

[NvL88] B. Nebel, K. von Luck. "Hybrid reasoning in BACK." In Z. W. Ras, L. Saitta (editors), *Methodologies for Intelligent Systems*, pp. 260–269, North Holland, Amsterdam, Netherlands, 1988.

[NW75] A. Nijenhuis, H.Wilf. *Combinatorial Algorithms.* Academic Press, 1975.

[Pat84] 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.

[Pat89] P. Patel-Schneider. "Undecidability of subsumption in NIKL." *Artificial Intelligence*, 1989.

[Sch88] K. Schild. "Undecidability of $\mathcal{U}$." KIT Report 67, TU Berlin, D-1000 Berlin, West Germany, 1988.

[Sch89] M. Schmidt-Schauß. "Subsumption in KL-ONE is undecidable." In R. J. Bachmann, H. J. Levesque, R. Reiter (editors), *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pp. 421–431, Toronto, Ont., 1989.

[SS88] M. Schmidt-Schauß, G. Smolka. *Attributive Concept Descriptions with Unions and Complements.* SEKI Report SR-88-21, FB Informatik, Universität Kaiserslautern, D-6750, Kaiserslautern, West Germany,1988. To appear in Artificial Intelligence.

[Sh86] S. M. Shieber, *An Introduction to Unification-Based Approaches to Grammar. CSLI Lecture Notes 4*, Center for the Study of Language and Information, Stanford University, 1986.

[Smo88] G. Smolka. *A Feature Logic with Subsorts.* LILOG Report 33, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, West Germany, May 1988. To appear in the Journal of Automated Reasoning.

[Sm68] R. M. Smullyan. *First-Order Logic.* Springer Verlag, Berlin 1968.

[Vil85] M. B. Vilain. "The restricted language architecture of a hybrid representation system." In R. J. Bachmann, H. J. Levesque, R. Reiter (editors), *Proceedings of the 9th IJCAI*, pp. 547–551, Los Angeles, Cal., 1985.