A Connection Method for Reasoning with the Description Logic *ALC*

Fred Freitas[1,2], Anne Schlicht[2], Heiner Stuckenschmidt[2]

[1]Informatics Center - Federal Universidade of Pernambuco (CIn - UFPE)
Av. Prof. Luis Freire, s/n, Cidade Universitária, 50740-540, Recife – PE, Brazil
[2]Knowledge Representation and Knowledge Management Research Group
Computer Science Institute, B6, 26, 68159, University of Mannheim, Germany

fred@cin.ufpe.br, {anne, heiner}@informatik.uni-mannheim.de

Abstract. The connection method was created by W. Bibel in the 70s, and earned good reputation in the field known as automated theorem proving for around three decades, due to practical reasons: it employs easy to understand and implement matrices that deal with formulae and theorems directly, in opposition to the classical refutation methods like resolution and tableaux, which require formulae to be negated. Such distinctive features attracted our attention in order to propose a connection method especially tailored to infer over the nowadays so popular Semantic Web ontologies which are written in a fragment of first order logic (FOL) known as description logic (DL). We are proposing a connection inference method for the DL *ALC*, mainly because this formalism is the basis of many other DLs. This paper is both a first formal attempt in this direction and discusses some practical issues. The classifier has some interesting features: it is able to perform all inference classification services and, as the most widespread DL tableaux systems, reduces them to subsumption.

Keywords: connection method, description logic inference, reasoning, description logic classification, *ALC*

1 Introduction

The problem of reasoning over ontologies written in Description Logic (DL) [Baader et al 2003] has been receiving strong interest from researchers since the Semantic Web inception. The necessity of creating reasoners to deal with the specific fragments of the first-order logic (FOL) of the OWL language [Patel-Schneider et al 2004] in its various versions posed interesting research questions for inference systems, making the problem earn the reputation of complex and mysterious, whose users see solutions as black boxes. Therefore, not many reasoners became well known to the public; indeed, tableaux based methods [Baader et al 2003] took over the field for many years, and recently one implementation of superposition calculus, the Hermit system, gained increasing popularity [Motik et al 2009]. Nevertheless, looking closely, the proposed tableaux systems for DL don't differ radically from its original, particularly if we use the DL notation without variables to represent first order logic formulae. This motivated us to try out other inference systems for the task of DL deduction.

This paper is both an attempt in this trend: we propose formally an inference system for the basic Description Logic *ALC* based on the connection calculus [Bibel 87], which is a simple, clear and effective inference method that has been used

successfully over FOL. The classifier defined here has some interesting features: it is able to perform all inference classification services (namely, subsumption, assertional queries, unsatisfiability, equivalence and disjointness), and, as the most widespread DL tableaux systems, reduces all of them to subsumption [Baader et al 2003]. Some practical issues, such as notations, properties and possible optimizations are also discussed throughout the paper, in particular strategies for implementation and possible gains in terms of memory usage.

The rest of the article is organized as follows: section 2 describes the connection method for first order formulae; section 3 defines the syntax of the Description Logic *ALC*; section 4 brings a method to translate *ALC* ontologies into first order, positive disjunctive normal form using matrices that are adequate for the connection method; section 5 constitutes the core of the article, by proposing a connection-based method for proving subsumption in *ALC*, proving the desired logical properties for inference systems, namely, soundness, completeness and termination, indicating notational improvements, clarifying the method with examples and discussing the benefits of the approach; section 6 some relevant issues for implementation; and section 7 presents future work and conclusions.

2 First-Order Logic Preliminaries

In order to describe the connection method as a formal inference system (in section 4), and the positive matricial normal form used in it, we will briefly describe the notation we use for first order logic, before examining the method. We are presuming readers to be acquainted to first order logic.

Definition 1 (First-order logic syntax).

The alphabets given by table 1 compose the FOL syntax notation. ■

Table 1. First order logic syntax notation.

Alphabets	
a, b, c, d	constants
x,y, z	variables
f, g, h	function symbols
$t_1, ..., t_n$	terms, composed with the three other alphabets
P, Q, R, S	predicate symbols
Connectives	
∧	conjunction
∨	disjunction
¬	negation (unary)
→	implication
Quantifiers	
∃	existencial
∀	universal

Definition 2 (Atom, literal, disjunctive normal form, clause, matrix).
All connectives are binary, except negation. *Atoms or atomic formulae* are predicates in the form of $P(t_1, \ldots)$. A first-order formula is a set of atoms linked by connectives, whose variables are quantified. A *literal* is either an atom or its negation, and is denoted by *L*. The complement of a literal *L* is its negation *L* and vice-versa.

Direct methods like CM are devoted to prove that a formula or a set of formulae is a theorem, iff in each interpretation a tautology is generated. Tautologies are usually in the form $L \vee \neg L$, therefore formulae are required to be in *disjunctive normal form (DNF).* A formula in DNF is a conjunction of disjunctions. Formulae are in the form

$$\bigcup_{i=1}^{n} C_i \text{ or , in a simpler way, } C_1 \vee \ldots \vee C_n ,$$

where each C_i is a *clause*, i. e., a conjunction of literals in the form

$$\bigcap_{j=1}^{m} L_{i,j} \text{ , or , } L_{i,1} \wedge \ \ldots \wedge L_{i,m} \text{ , also denoted as } \{L_{i,1}, \ldots, L_{i,m}\},$$

where each $L_{i,j}$ is a literal, resulting in formulae in the form

$$\bigcup_{i=1}^{n} \bigcap_{i=1}^{m} L_{i,j} \text{ , or simply } (L_{1,1} \wedge \ \ldots \wedge L_{1,m}) \vee \ldots \vee (L_{n,1} \wedge \ \ldots \wedge L_{n,m}) ,$$

also denoted in *disjunctive clausal form* as $\{\{(L_{1,1}, \ldots, L_{1,m}\}, \ldots, \{L_{n,1}, \ldots, L_{n,m}\}\}$ or $\{C_1, \ldots, C_n\}$. Formulae normalized in this way are said to be in *positive matricial form,* since they can be simply represented as a matrix like in Figure 1. Note that each clause is a conjunction and occupies a column in the matrix. ■

$$\begin{bmatrix} L_{1,1} & \cdots & L_{n,1} \\ \vdots & \ddots & \vdots \\ L_{1,m} & \cdots & L_{n,m} \end{bmatrix}$$

Figure 1. The formula $(L_{1,1} \wedge \ \ldots \wedge L_{1,m}) \vee \ldots \vee (L_{n,1} \wedge \ \ldots \wedge L_{n,m})$ represented as a matrix.

Definition 3 (Skolemization). In order to deal with formulae in DNF, quantifiers should be removed by skolemization. In contrast to the common skolemization that preserves satisfiability, formulae transformed into DNF are valid iff the original formula is valid. Hence, instead of existential quantifiers, universal quantifiers ( ) are replaced by constants or Skolem functions. Variables in the resulting DNF are then (implicitly) existentially quantified.

Example 1 (Disjunctive normal form, skolemization, clause, matrix).

The formula

$\forall x(Bird(x) \rightarrow Animal(x) \wedge \exists y(hasPart(x,y) \wedge Bone(y)) \wedge$
$\forall w(Animal(w) \wedge (\exists z\ (hasPart(w,z) \wedge Bone(z)) \rightarrow Vertebrate(w))$
$\rightarrow \forall t(Bird(t) \rightarrow Vertebrate(t)))$

in DNF can be expressed as
$\exists x \forall y((Bird(x) \wedge \neg Animal(x)) \vee (Bird(x) \wedge \neg hasPart(x,y)) \vee (Bird(x) \wedge \neg Bone(y)) \vee$
$\exists w \exists z(Animal(w) \wedge hasPart(w,z) \wedge Bone(z) \wedge \neg Vertebrate(w)) \vee$
$\forall t(\neg Bird(t) \vee Vertebrate(t)))$.

Continuing with the example, we should skolemize the variables y and t respectively by the function $f(x)$ and the constant c, in the clausal form, the formula reads as

$(Bird(x) \wedge \neg Animal(x)) \vee (Bird(x) \wedge \neg hasPart(x,f(x))) \vee (Bird(x) \wedge \neg Bone(f(x))) \vee$
$(Animal(w) \wedge hasPart(w,z) \wedge Bone(z) \wedge \neg Vertebrate(w)) \vee \neg Bird(c) \vee Vertebrate(c))$.

The corresponding negated DNF clausal form is below or alternatively represented as a matrix as shown in figure 2.

$\{\{Bird(x), \neg Animal(x)\}, \{Bird(x), \neg hasPart(x,f(x))\}, \{Bird(x), \neg Bone(f(x))\},$
$\{Animal(w), hasPart(w,z), Bone(z), \neg Vertebrate(w)\}, \{\neg Bird(c)\}, \{Vertebrate(c))\}\}$,

$$\begin{bmatrix} Bird(x) & Bird(x) & Bird(x) & Animal(w) & \neg Bird(c) & Vertebrate(c) \\ \neg Animal(x) & \neg hasPart(x, f(x)) & \neg Bone(f(x)) & hasPart(w,z) & & \\ & & & Bone(z) & & \\ & & & \neg Vertebrate(w) & & \end{bmatrix}$$

Figure 2. A formula represented as a matrix.

3 The Description Logic *ALC*

Description Logic (DL for short) is a family of knowledge representation formalisms that have been gaining strong interest from researchers in the last two decades, mainly because it has been approved as the W3C standard for representing the most expressive layer of the Semantic Web. We say a family due to the fact that many different formalisms can be assembled according to the set of constructors included. In this paper we are particularly interested in the Description Logic *ALC* (Attributive Concept Language with Complements), because it constitutes the foundations of many other DLs.

3.1 DL *ALC* Syntax

An ontology or knowledge base in *ALC* is a set of axioms defined over the ordered triple (N_C, N_R, N_O) [Baader et al 2003], where:

- N_C is the set of *concept names* (or *atomic concepts*), or unary predicates,
- N_R is the set of *role or property names* or binary relations,
- N_O is the set of *individual names* (or *objects*), instances of N_C and N_R,

N_C is composed of:

- $\top$ (the *top* concept, to which each instance $a \in N_O$ belongs to),
- $\bot$ (the *bottom concept or empty set*)
- letting r be a *role* ($R \in N_R$) *and* C and D be *concepts* ($C, D \in N_C$) then the following are *concepts*:
  - $C \sqcap D$ (the intersection of two *concepts*)
  - $C \sqcup D$ (the union of two *concepts*)
  - $\neg C$ (the complement of a *concept*)
  - $\forall r.C$ (the universal restriction of a *concept* by a *role* is a *concept*)
  - $\exists r.C$ (the existential restriction of a *concept* by a *role* is a *concept*)

N_O is composed of:

- *Concept assertions C(a),* where $C \in N_C$ and
- *Role assertions R(a,b),* where $r \in N_R$.

The axioms are composed of N_O and any finite set of GCIs (*general concept inclusion*) in one of the forms $C \sqsubseteq D$ or $C \equiv D$ (meaning both $C \sqsubseteq D$ and $D \sqsubseteq C$), where C and D are *concepts* (C, D ∈).

An ontology or knowledge base (KB) is also referred as a pair (T,A), where T is a terminological box (or Tbox in the DL jargon), which stores all but N_O, which on its turn is stored in the assertional box (ABox) A.

3.2 Semantics

There are two major ways of defining the semantics of *ALC*. The most used and well known is defining it in terms of interpretations over a domain Δ, together with its models, fixpoints, etc [Baader et al 2003]. The other and easiest way, which we adopt here, is simply mapping the *ALC* constructs to first order logic, and then relying over the semantics defined for it, for instance in [III.2, Bibel 1987]. We chose the latter, with the purpose of inheriting FOL semantics to take advantage of the formalized completeness and soundness theorems for our work.

4 Translation of $\mathcal{ALC}$ Ontologies into a First Order, Disjunctive Normal Form using Matrices

In order to guarantee an easy translation from $\mathcal{ALC}$ ontologies into matrices of first order predicates in the Disjunctive Normal Form (DNF), first we must normalize axioms and then apply translations. These steps are described in the next four subsections.

In any case, all inference services of any DL formalism, namely ontology querying, subsumption (or logical consequence) and consistency, instance and equivalence checking, will be reduced to proving subsumption (see subsection 5.2., Proposition 2) in the form of KB $\rightarrow \alpha$ (which comes from KB $\models \alpha$) and the deduction theorem [Bibel 93]), where α is an axiom as defined in the subsection 3.1. Just as many other inference procedures, like resolution, we work with the consequent α negated ($\neg \alpha$).

The explanation is the following: In order to get DNF clauses, we see the formula KB $\rightarrow \alpha$ as $\neg$KB $\vee \alpha$, where KB is in the form

$$\neg \bigcap_{i=1}^{n} A_i \vee \alpha, \text{ (or in a longer form, } \neg(A_1 \wedge \ \ldots \wedge A_n) \vee \alpha, \ A_i, \text{ i=1,…,n, axioms),}$$

what yields a formula already close to DNF, in the form

$$\bigcup_{i=1}^{n} \neg A_i \vee \alpha \text{ (or } \neg A_1 \vee \ldots \vee \neg A_n \vee \alpha, \ A_i, \text{ i=1,…,n, axioms).}$$

For establishing a single uniform set of rules to apply over this formula, we will deal with $\neg \alpha$ instead of α, considering the formula as $\neg A_1 \vee \ldots \vee \neg A_n \vee \neg\neg \alpha$. This means that the translation rules have to be applied over all A_i and $\neg \alpha$, therefore the axiom to be proved α should be first negated.

The first action to be made over the axioms now is splitting equivalence axioms of the form $C \equiv D$ into two subsumptiom clauses $C \sqsubseteq D$ and $D \sqsubseteq C$. After that all axioms either are subsumption axioms or ABox definitions.

The use of negation throughout DNF requires that, before or during any normalization and translation, no negation is found over an expression, only in atoms, instead. Definition 4 describes how this is carried out.

Definition 4 (Positive Disjunctive Normal Form TBox). The first step is converting all the axioms into negated normal form using the following simple rewriting rules:

$$\begin{array}{lcl}
\neg(C \sqcap D) & \leftrightarrow & \neg C \sqcup \neg D \\
\neg(C \sqcup D) & \leftrightarrow & \neg C \sqcap \neg D \\
\neg \forall r.C & \leftrightarrow & \exists r.\neg C \\
\neg \exists r.C & \leftrightarrow & \forall r.\neg C
\end{array}$$

■

Next, we present two ways of normalizing and translating. The first describe a simpler, albeit more verbose way of normalization and translation, while the latter introduces a more complex normalization, which makes use of lesser additional artificial concepts. Note that their translations to FOL are distinct, since the definitions of normal forms differ.

However, before presenting the mappings, some additional remarks should be stated about the direct matrix mappings.

First, in both cases, after introducing additional artificial concepts, the normalized resulting TBoxes of both normalizations are still *conservative extensions* [Ghilardi et al 2006] of the original TBoxes, since every model of the former is a model of the latter [Baader et al 2005, 2005a].

Moreover, since we are working with a L2 FOL fragment (which contains only binary relations) [Baader et al 2003], similarly to the standard description logic notation, we can eliminate variables from our negated DNF matricial representation without any losses, iff we supply a way of linking roles to concepts in the case of qualified restrictions of the type $\exists r.C$ or $\forall r.C$. Aiming to that, a dashline links roles to their qualified concepts in order to do so but also to distinguish the identifier employed in the qualified concepts (y or $f(x)$) from the ones used in other concepts (x). There are two types of links: in $\mathcal{ALC}$, *horizontal* dashlines connect *universally* quantified roles to skolemized concepts ($\forall r.C$, designated by $r(x,f(x))$ and $f(x)$), while *vertical* dashlines links *existentially* quantified roles ($\exists r.C$), to concepts ($C(y)$).

Nevertheless, there are interesting aspects on universally quantified roles' representation, which are even clearer than other representations. The predicate $\forall r.C$ has the following semantics:

$$(\forall r.C)^I = \{\forall b, (a,b) \in R^I \rightarrow b \in C^I\}$$

Hence, for an axiom of the form $A \equiv \forall r.C$, it does not oblige the concept A to dispose of instances –indeed a very common error from description logic novice users. But maybe it is not their fault: it is hard to grasp that meaning if even tableaux proofs over such axioms don't stress its semantics, in the sense that it allows instances of A without any role instances from r associated to it. The same does not happen to $\mathcal{ALC}$ CM; the matricial representation explicits this situation, as shows figure 3.

$$\begin{bmatrix} A \\ [\neg r \quad \neg C] \end{bmatrix}$$

Figure 3. The representation of the axiom $A \equiv \forall r.C$. An advantage of this representation is stressing the possibility of allowing instances of A without any associated role instances from r.

There it is clear, that either the concept cannot dispose of instances of the role by definition ($\neg r$) or, when it has a role instance (a,b), b has to be of an instance of the concept C, *because it represents the formula* $\neg A(x)) \wedge (\neg r(x,f(x)) \vee \neg C(f(x))$. Since this can be also misleading, another possible representation, maybe even more explicit, can be seen on figure 4.

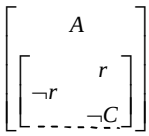

Figure 4. A possible representation of the axiom $A \equiv \forall r.C$. An advantage of this representation is stressing the possibility of allowing instances of A without any associated role instances from r by definition($\neg r$).

Nonetheless, we preferred now not to adopt it because we would have to make use of the symbols $\forall$ and $\exists$ to logically distinguish between skolemized and normal role representations for the matrices. Moreover, it is more complex and the former representation follows directly the formula $\neg r(x,f(x)) \vee C(f(x))$, which is the logical correspondence of $\forall r.C$.

4.1 A Simpler Normalization

In order to be directly translatable into DNF clauses and thus into a matricial form, all remaining subsumption axioms should be in one of the forms of figure 5, where $C_i \in N_C$, i=1,…,3, and C_i are negated or not concepts names:

$$C_1 \sqsubseteq C_2$$

$$C_1 \sqcap C_2 \sqsubseteq C_3 \qquad C_1 \sqcup C_2 \sqsubseteq C_3$$

$$C_1 \sqsubseteq C_2 \sqcap C_3 \qquad C_1 \sqsubseteq C_2 \sqcup C_3$$

$$\exists r.C_1 \sqsubseteq C_2 \qquad \forall r.C_1 \sqsubseteq C_2$$

$$C_2 \sqsubseteq \exists r.C_1 \qquad C_2 \sqsubseteq \forall r.C_1$$

Figure 5. ALC simpler DNFs.

On table 2, we enlist the rules to normalize the subsumption axioms as a preparation step to translate to the negated DNF form, where B and D are concept names, negated or not, A is an introduced concept name, $\hat{C}$ is a complex concept.

4.2 Translation Rules

With all the axioms in negated DNF, it is easy to map them both to FOL and to the matricial form, by simply applying the rules given in table 3.

Example 2 (Simpler normalization and its translation to FOL).

Let's normalize and translate the following formula to DNF FOL:

$PizzaMargherita \equiv Pizza \sqcap \exists hasTopping.Tomate \sqcap \exists hasTopping.Muzzarella \sqcap$
$\neg\exists hasTopping.\ \neg(Tomate \sqcup Muzzarella)$

Table 2. Normalization rules.

Rule	Entry	Transformed form
R1	$\hat{C} \sqsubseteq \hat{D}$	$\hat{C} \sqsubseteq A, A \sqsubseteq \hat{D}$
R2	$B \sqcap \hat{C} \sqsubseteq D$	$B \sqcap A \sqsubseteq D, \hat{C} \sqsubseteq A$
R3	$B \sqsubseteq D \sqcap \hat{C}$	$B \sqsubseteq D, B \sqsubseteq \hat{C}$
R4	$B \sqcup \hat{C} \sqsubseteq D$	$B \sqsubseteq D, \hat{C} \sqsubseteq D$
R5	$B \sqsubseteq D \sqcup \hat{C}$	$B \sqsubseteq D \sqcup A, A \sqsubseteq \hat{C}$
R6	$\exists r.\hat{C} \sqsubseteq D$	$\exists r.A \sqsubseteq D, \hat{C} \sqsubseteq A$
R7	$D \sqsubseteq \exists r.\hat{C}$	$D \sqsubseteq \exists r.A, A \sqsubseteq \hat{C}$
R8	$\forall r.\hat{C} \sqsubseteq D$	$\forall r.A \sqsubseteq D, \hat{C} \sqsubseteq A$
R9	$D \sqsubseteq \forall r.\hat{C}$	$D \sqsubseteq \forall r.A, A \sqsubseteq \hat{C}$

Eliminating equivalence:

(PizzaMargherita $\sqsubseteq$ *Pizza* $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\neg\exists$*hasTopping.* $\neg$*(Tomate* $\sqcup$ *Muzzarella))*
$\sqcap$
(Pizza $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\neg\exists$*hasTopping.* $\neg$*(Tomate* $\sqcup$ *Muzzarella)* $\sqsubseteq$ *PizzaMargherita)*

Solving inner negations:

((PizzaMargherita $\sqsubseteq$ *Pizza* $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\forall$*hasTopping.* $\neg\neg$*(Tomate* $\sqcup$ *Muzzarella))*
$\sqcap$
(Pizza $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\forall$*hasTopping.* $\neg\neg$*(Tomate* $\sqcup$ *Muzzarella)* $\sqsubseteq$ *PizzaMargherita))*

which yields

((PizzaMargherita $\sqsubseteq$ *Pizza* $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\forall$*hasTopping.* (*Tomate* $\sqcup$ *Muzzarella))*
$\sqcap$
(Pizza $\sqcap \exists$*hasTopping.Tomate* $\sqcap \exists$*hasTopping.Muzzarella* $\sqcap$
$\forall$*hasTopping.* (*Tomate* $\sqcup$ *Muzzarella)* $\sqsubseteq$ *PizzaMargherita))*

Table 3. Translation rules to map *ALC* into DNF FOL and matrices in various forms.

Axiom type	**FOL DNF Negated mapping**	**DNF Negated Matrix mapping**	**Direct Matrix mapping**
Sub-sumption $A \sqsubseteq B$	$A(x) \wedge \neg B(x)$	$\begin{bmatrix} A(x) \\ \neg B(x) \end{bmatrix}$	$\begin{bmatrix} A \\ \neg B \end{bmatrix}$
Disjunction $A \sqsubseteq \neg B$	$A(x) \wedge B(x)$	$\begin{bmatrix} A(x) \\ B(x) \end{bmatrix}$	$\begin{bmatrix} A \\ B \end{bmatrix}$
LHS Union: $A \sqcup B \sqsubseteq C$	$(A(x) \wedge C(x)) \vee (B(x) \wedge \neg C(x))$	$\begin{bmatrix} A(x) & B(x) \\ \neg C(x) & \neg C(x) \end{bmatrix}$	$\begin{bmatrix} A & B \\ \neg C & \neg C \end{bmatrix}$
RHS Union: $A \sqsubseteq B \sqcup C$	$A(x) \wedge \neg B(x) \wedge \neg C(x)$	$\begin{bmatrix} A(x) \\ \neg B(x) \\ \neg C(x) \end{bmatrix}$	$\begin{bmatrix} A \\ \neg B \\ \neg C \end{bmatrix}$
LHS Intersection $A \sqcap B \sqsubseteq C$	$A(x) \wedge B(x) \wedge \neg C(x)$	$\begin{bmatrix} A(x) \\ B(x) \\ \neg C(x) \end{bmatrix}$	$\begin{bmatrix} A \\ B \\ \neg C \end{bmatrix}$
RHS Intersection $A \sqsubseteq B \sqcap C$	$(A(x) \wedge \neg B(x)) \vee (A(x) \wedge \neg C(x))$	$\begin{bmatrix} A(x) & A(x) \\ \neg B(x) & \neg C(x) \end{bmatrix}$	$\begin{bmatrix} A & A \\ \neg B & \neg C \end{bmatrix}$
LHS existencial restriction: $\exists r.C \sqsubseteq A$	$r(x,y) \wedge C(y) \wedge \neg A(x)$	$\begin{bmatrix} r(x,y) \\ C(y) \\ \neg A(x) \end{bmatrix}$	$\begin{bmatrix} r \\ C \\ \neg A \end{bmatrix}$
RHS existencial restriction: $A \sqsubseteq \exists r.C$	$(A(x) \wedge \neg r(x,f(x))) \vee (\neg A(x) \wedge \neg C(f(x)))$	$\begin{bmatrix} A(x) & A(x) \\ \neg r(x, f(x)) & \neg C(f(x)) \end{bmatrix}$	$\begin{bmatrix} A & A \\ \neg r & \neg C \end{bmatrix}$
LHS universal restriction: $\forall r.C \sqsubseteq A$	$(\neg r(x,f(x)) \wedge \neg A(x)) \vee (\neg C(f(x)) \wedge \neg A(x))$	$\begin{bmatrix} \neg r(x, f(x)) & C(f(x)) \\ \neg A(x) & \neg A(x) \end{bmatrix}$	$\begin{bmatrix} \neg r & C \\ \neg A & \neg A \end{bmatrix}$
RHS universal restriction: $A \sqsubseteq \forall r.C$	$A(x) \wedge r(x,y) \wedge \neg C(y)$	$\begin{bmatrix} A(x) \\ r(x,y) \\ \neg C(y) \end{bmatrix}$	$\begin{bmatrix} A \\ r \\ \neg C \end{bmatrix}$

Normalizing the first axiom (step by step):

$PizzaMargherita \sqsubseteq Pizza \sqcap \exists hasTopping.Tomate \sqcap \exists hasTopping.Muzzarella \sqcap$
$\forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R3: $PizzaMargherita \sqsubseteq Pizza$ *(NF)*

$PizzaMargherita \sqsubseteq \exists hasTopping.Tomate \sqcap \exists hasTopping.Muzzarella \sqcap$
$\forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R3: $PizzaMargherita \sqsubseteq \exists hasTopping.Tomate$ *(NF)*

$PizzaMargherita \sqsubseteq \exists hasTopping.Muzzarella \sqcap$
$\forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R3: $PizzaMargherita \sqsubseteq \exists hasTopping.Muzzarella$ *(NF)*

$PizzaMargherita \sqsubseteq \forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R9: $PizzaMargherita \sqsubseteq \forall hasTopping.A$ *(NF)*

$A \sqsubseteq Tomate \sqcup Muzzarella$ *(NF)*

Therefore, the first axiom is transformed into a conjunction of the following clauses:

$PizzaMargherita \sqsubseteq Pizza$

$PizzaMargherita \sqsubseteq \exists hasTopping.Tomate$

$PizzaMargherita \sqsubseteq \exists hasTopping.Muzzarella$

$PizzaMargherita \sqsubseteq \forall hasTopping.A$

$A \sqsubseteq Tomate \sqcup Muzzarella$

Translating to matricial DNF using the rules of Table 3:

$$\left[\begin{array}{cccccccc} PMargh & PMargh & PMargh & PMargh & PMargh & PMargh & A \\ \neg Pizza & \neg hasTopp & \neg Tomate & \neg hasTopp & \neg Muzza & hasTopp & \neg Tomate \\ & & & & & \neg A & \neg Muzza \end{array}\right]$$

Figure 3. Translating axioms to matricial DNF.

Normalizing the second axiom (step by step):

$Pizza \sqcap \exists hasTopping.Tomate \sqcap \exists hasTopping.Muzzarella \sqcap$
$\forall hasTopping.\ (Tomate \sqcup Muzzarella) \sqsubseteq PizzaMargherita)\ )$

R2:$PizzaMargherita \sqsubseteq Pizza \sqcap A'$ *(NF)*

$A' \sqsubseteq \exists hasTopping.Tomate \sqcap \exists hasTopping.Muzzarella \sqcap$
$\forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R2:$A' \sqsubseteq \exists hasTopping.Tomate \sqcap B$

$B \sqsubseteq \exists hasTopping.Muzzarella \sqcap \forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R2:A' $\sqsubseteq$ $C \sqcap B$ *(NF)*

$C \sqsubseteq \exists hasTopping.Tomate$ *(NF)*

R2: $B \sqsubseteq \exists hasTopping.Muzzarella \sqcap D$

$D \sqsubseteq \forall hasTopping.\ (Tomate \sqcup Muzzarella))$

R2: $B \sqsubseteq E \sqcap D$ *(NF)*

$E \sqsubseteq \exists hasTopping.Muzzarella$ *(NF)*

R9: $D \sqsubseteq \forall hasTopping.F$ *(NF)*

$F \sqsubseteq Tomate \sqcup Muzzarella$ *(NF)*

Therefore, the second axiom is transformed into a conjunction of the following clauses:

$PizzaMargherita \sqsubseteq Pizza \sqcap A'$

$A' \sqsubseteq C \sqcap B$

$C \sqsubseteq \exists hasTopping.Tomate$

$B \sqsubseteq E \sqcap D$

$E \sqsubseteq \exists hasTopping.Muzzarella$

$D \sqsubseteq \forall hasTopping.F$

$F \sqsubseteq Tomate \sqcup Muzzarella$

Translation to matricial DNF:

PMargh	*PMargh*	*A'*	*A'*	*C*	*C*	*B*	*B*	*E*	*E*	*D*	*F*
$\neg Pizza$	$\neg A'$	$\neg C$	$\neg B$	$\neg hasTopp$	$\neg Tomate$	$\neg E$	$\neg D$	$\neg hasTopp$	$\neg Muzza$	*hasTopp*	$\neg Tomate$
										$\neg F$	$\neg Muzza$

Figure 6. Translating axioms to matricial DNF.

Figure 5 presents the representation of the initial equivalence axiom in the DNF matricial form, i.e., the merging of the two matrices.

4.2 A more complex and efficient normalization

In order to be directly translatable into DNF clauses and thus into a matricial form, the easiest way is to convert all axioms should be in one of the forms given below in definition 9. The first two forms are the only ones which generate new columns in the matrix, while the third generate only a single column, as we will soon see.

Figure 7. Translating the example of an equivalence axiom into matricial DNF.

Definition 5 (*ALC* disjunction, *ALC* conjunction). An ***ALC*** disjunction is either a literal, a disjunction $E_0 \sqcup$ or an universal restriction $\forall r.E$ An *ALC* conjunction is either a literal, a conjunction $E_0 \sqcap$ or an existential restriction $\exists r.$. and are arbitrary concept expressions. ■

Definition 6 (*ALC* pure disjunction). An ***ALC*** *pure disjunction* is an ***ALC*** disjunction, that in negation normal form, contains no existential quantifier ( ) and no conjunctions ( ). The set of ***ALC*** *pure disjunctions* is denoted by , Elements $\check{D} \in$ are marked by a circumflex for readability. An ***ALC*** *non-pure disjunction* is an ***ALC*** disjunction that is not a pure disjunction. ■

Definition 7 (*ALC* pure conjunction). Analogously, an ***ALC*** *pure conjunction* is an ***ALC*** conjunction that in negation normal form, contains no universal quantifier ( ) and no disjunctions ( ). The set of ***ALC*** *pure conjunctions* is denoted by , Elements $\hat{C} \in$ are marked by a caron for readability. An ***ALC*** *non-pure conjunction* is an ***ALC*** conjunction that is not an ***ALC*** pure conjunction. ■

Definition 8 (Impurity on a non-pure expression). During a lazy evaluation parsing [Watt 90], impurities of non-pure *ALC* DL expressions are either conjunctive expressions in a non-pure disjunction or disjunctive expressions in a non-pure conjunction. The set of impurities is called *ALC impurity set*, and is denoted by .

Example 2 (Impurities on non-pure expressions).
The expression ($\forall r.(D_0 \sqcup \ldots \sqcup D_n \sqcup (C_0 \sqcap \ldots \sqcap C_m) \sqcup (A_0 \sqcap \ldots \sqcap A_p$ a non-pure disjunction, contains two impurities: $(C_0 \sqcap \ldots \sqcap C_m)$ and $(A_0 \sqcap \ldots \sqcap A$

On the other hand, the expression $\exists r.(\forall r'.((D_0 \sqcup \ldots \sqcup D_m) \sqcup (A_0 \sqcap \ldots \sqcap A_p$, a non-pure conjunction, has one impurity $\forall r'.((D_0 \sqcup \ldots \sqcup D_n) \sqcup (A_0 \sqcap \ldots \sqcap A_p$

Definition 9 (Positive normal form). An *ALC* axiom is in positive normal form iff it is in one of the following forms: $(i)\ \hat{C} \sqsubseteq \check{D}$; $(ii)\ C \sqsubseteq \exists r.\hat{C}$; and $(iii)\ \forall r.\check{D} \sqsubseteq$; where C is a concept name, a pure conjunction and a pure disjunction. ■

4.2.1 Normalization Algorithm

The aim of this normal form is to assure creating only the unavoidably necessary columns, at the expense of longer ones, while the translation algorithm presented below is circumventing

The first step of the normalization process is identifying whether the left and right hand sides (henceforth, LHS and RHS) of the axiom are conjunctions or disjunctions.

After this identification, the normalization algorithm portrayed in figure 6 is performed over each $A_i \in T$ and $\neg\alpha$. It is complemented by the algorithms for normalizing the left and right hand sides of axioms, given in figures 9 and 10. The alert reader, however, would have noticed that the procedures **`Normalize-LHS`** and **`Normalize-RHS`** are dual: the axioms sides are dual, conjunctions in one are disjunctions in the other and the new axioms differ only in the subsumption direction.

Therefore, for implementation, we have the option of using a single routine that serves to both cases. Such routine is displayed in figure 11.

```
Normalize-Ontology (Ontology O);
For each axiom A ∈ O  Normalize-Axiom (A,O);
Normalize-Axiom (Axiom A, Ontology O);
If A not in normal form then
     If LHS(A) ∉ N_C U S_C {not a concept or pure conjunction}
       Normalize-LHS (A, O);
     If RHS(A) ∉ N_C U S_D {not a concept or pure disjunction}
       Normalize-RHS (A, O);
```

Figure 8. Normalization algorithm.

Both algorithms are more efficient in terms of memory than the one with the simpler normalization. While in the former, the number of new axioms grows linearly to the axioms' length, the two new ones are linear to the number of disjunctions in the axioms' LHS and conjunctions on its RHS.

```
Normalize-LHS (Axiom A, ontology O);
If LHS(A) ∈ S_C U S_npC (a conjunction) then
     For each D ⊆ LHS(A) | D ∈ S_D U S_npD (disjunction)
       LHS(A) ← (LHS(A) – D) ⊓ N, N ∉ O
       If D ∈ S_npD (non-pure) then
              O ← O U Normalize-LHS({D ⊑ N},O);
       Else   O ← O U {D ⊑ N};
     For each nC ⊆ LHS(A) | nC ∈ S_npC (non-pure conjunction)
       Find the first impurity I ⊆ nC | I ∈ S_I
       nc^σ ← nC{I/N}, N ∉ O
       LHS(A) ← (LHS(A) – nC) ⊓ nc^σ
       O ← O U Normalize-LHS({I ⊑ N},O}
Else {LHS(A) ∈ S_D U S_npD (disjunction)}
     O ← O – A;
     For each X ⊆ LHS(A)|X ∈ N_C U S_C
              (atomic concept or pure conjunction)
       If RHS(A) ∉ N_C U S_C  (not a concept or pure conjunction) then
              O ← O U Normalize-RHS({X ⊑ RHS(A)}, O};
       Else   O ← O U {X ⊑ RHS(A)};
     For each expression E ⊆ LHS(A)| E ∉ S_C
                    (not a pure conjunction)
       If E ⊑ RHS(A) is not in normal form then
              O ← O U Normalize-Axiom({E ⊑ RHS(A)}, O};
```

```
Else    O ← O U E ⊑ RHS(A);
```

Figure 9. Normalization algorithm for the left hand side of the axiom.

Normalize-RHS (Axiom A, ontology O);

If RHS(A) $\in$ S_D U S_{npD} (a disjunction) then

  For each C $\subseteq$ RHS(A) | D $\in$ S_C U S_{npC} (conjunction)

    RHS(A) $\leftarrow$ (RHS(A) – C) $\sqcup$ N, N $\notin$ O

    If C $\in$ S_{npC} (non-pure) then

      O $\leftarrow$ O U Normalize-RHS({N $\sqsubseteq$ C},O);

    Else O $\leftarrow$ O U {N $\sqsubseteq$ C};

  For each nD $\subseteq$ RHS(A) | nD $\in$ S_{npD} (non-pure disjunction)

    Find the first impurity I $\subseteq$ nD | I $\in$ S_I

    nd^{σ} $\leftarrow$ nD{I/N}, N $\notin$ O

    LHS(A) $\leftarrow$ (LHS(A) – nD) $\sqcup$ nd^{σ}

    O $\leftarrow$ O U Normalize({N $\sqsubseteq$ I},O}

Else {RHS(A) $\in$ S_C U S_{npC} (conjunction)}

  O $\leftarrow$ O – A;

  For each X $\subseteq$ RHS(A) | X $\in$ N_C U S_D
      (atomic concept or pure disjunction)

    If LHS(A) $\notin$ N_C U S_D (not a concept or pure disjunction) then

      O $\leftarrow$ O U Normalize-LHS({LHS(A) $\sqsubseteq$ X)}, O};

    Else O $\leftarrow$ O U {LHS(A) $\sqsubseteq$ X};

  For each expression E $\subseteq$ RHS(A) | E $\notin$ S_D
      (not a pure disjunction)

    If LHS(A) $\sqsubseteq$ E is not in normal form

      O $\leftarrow$ O U Normalize-Axiom({LHS(A) $\sqsubseteq$ E)}, O};

    Else O $\leftarrow$ O U LHS(A) $\sqsubseteq$ E;

Figure 10. Normalization algorithm for the right hand side of the axiom.

```
Normalize (Boolean Side, Axiom A, ontology O);

Boolean LHS=true;
        RHS=false;

{Settings}

If Side=LHS then

     PartOfAxiom ← LHS(A)
     RemainingPartOfAxiom ← RHS(A)
     Format ← conjunction
     OppositeFormat ← disjunction
     Subsumes ← ⊑

Else {Side=LHS}

     PartOfAxiom ← RHS(A)
     RemainingPartOfAxiom ← RHS(A)
     Format ← disjunction
     OppositeFormat ← conjunction
     Subsumes ← ⊒

{Normalization}

If PartOfAxiom is in Format then

     For each pure or non-pure D ∈ PartOfAxiom in OppositeFormat

       PartOfAxiom ← Format((PartOfAxiom – D), N), N ∉ O
       If D is non-pure then
              O ← O U Normalize(D Subsumes N},O);
       Else   O ← O U {D Subsumes N};

     For each non-pure Format nC in PartOfAxiom
       Find the first impurity I ∈ nC
       nC^σ ← nC{I/N}, N ∉ O
       PartOfAxiom ← Format(PartOfAxiom – nC), nC^σ )
       O ← O U Normalize({I Subsumes N},O}

Else if PartOfAxiom is in OppositeFormat then
     O ← O – A;

     For each X, atomic concept or pure Format in the
              OppositeFormat ∈ PartOfAxiom

       O ← O U ({X Subsumes RemainingPartOfAxiom}, O};

     For each expression E, that is not in pure Format
              in the OppositeFormat ∈ PartOfAxiom

       O ← O U Normalize-Axiom({E Subsumes RemainingPartOfAxiom}, O};
```

Figure 11. Single procedure for normalizing an axiom's left and right sides.

4.2.2 Translation Rules for the more efficient normalization

With all the axioms in the negated DNF given in the previous section, it is easy to map them both to FOL and to the matricial form, by simply applying the rules given in table 4.

Table 4. Translation rules to map ALC into FOL positive DNNF and matrices.

Axiom type	FOL Positive DNNF mapping	Matrix
$C \sqsubseteq \exists r.\hat{C}$, where $\hat{C} = \bigcap_{i=1}^{n} A_i$, with $A_i \in S_C$ (pure conjunction)	$(C(x) \wedge \neg r(x,f(x))) \vee (C(x) \wedge \neg A_1(f(x)))$ v... v $(C(x) \wedge \neg A_n(f(x)))$	$\begin{bmatrix} C & C & \cdots & C \\ \neg r & \neg A_1 & \cdots & \neg A_n \end{bmatrix}$
$\forall r.\check{D} \sqsubseteq C$, where $\check{D} = \bigcup_{j=1}^{m} A'_j$, with $A'_j \in S_D$ (pure disjunction)	$(\neg r(x,f(x)) \wedge \neg C(x)) \vee (\neg A'_1(f(x)) \wedge \neg C(x))$ v...v $(\neg A'_m(f(x)) \wedge \neg C(x))$	$\begin{bmatrix} \neg r & A'_1 & \cdots & A'_m \\ \neg C & \neg C & \cdots & \neg C \end{bmatrix}$
$\hat{C} \sqsubseteq \check{D}$, where $\hat{C} = \bigcap_{i=1}^{n} A_i$, $\check{D} = \bigcup_{j=1}^{m} A'_j$, $A_i \in S_C$ (pure conjunction), $A'_j \in S_D$ (pure disjunction)	$A_1(x) \wedge ... \wedge A_n(x) \wedge \neg A'_1(x) \wedge ... \wedge \neg A'_m(x)$	$\begin{bmatrix} A_1 \\ \vdots \\ A_n \\ \neg A'_1 \\ \vdots \\ \neg A'_n \end{bmatrix}$

Besides the lack of variables, another new notational feature was introduced. Given that there is recursivity in all cases over the A and A', $A_i \in S_C$ and $A'_j \in S_D$, then A_i being pure conjunctions, can themselves generate further predicates in the same column. A' and A'_j, are pure disjunctions, consequently in all cases they can generate new columns. Example 3 shows this last case.

Table 5 brings the mapping treatment of such recursive sub-cases of existential and universal restrictions, when they occur inside any of the three normal forms. Note, however, that position constraints apply here: existential restrictions occur only on the left side of axioms (like in $\exists r.C \sqsubseteq E$), while universal take place on the right.

Example 3 (More efficient normalization and its translation to FOL).
Let's normalize and translate the same Example 2 to DNF FOL:

PizzaMargherita ≡ *Pizza* ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓
¬∃*hasTopping.* ¬(*Tomate* ⊔ *Muzzarella*)

Table 5. Recursive subcases of the last case.

Axiom type	FOL DNF Negated mapping	DNF Negated Matrix mapping	Direct Matrix mapping
A_i is an existencial restriction: …⊓ ∃*r.A* ⊓...	... ∧ *r(x,y)*∧ *A(y)*∧ ...	$\begin{bmatrix} \vdots \\ r(x,y) \\ A(y) \\ \vdots \end{bmatrix}$	$\left[\begin{array}{c} \vdots \\ r \\ \hline A \\ \vdots \end{array}\right]$
A'_j is an universal restriction: …⊔ ∀*r.A'* ⊔…	…∧ *r(x,y)* ∧ ¬*A'(y)* ∧ …	$\begin{bmatrix} \vdots \\ r(x,y) \\ \neg A'(y) \\ \vdots \end{bmatrix}$	$\left[\begin{array}{c} \vdots \\ r \\ \hline \neg A' \\ \vdots \end{array}\right]$

The steps of eliminating equivalence and solving inner negations are the same, thus yielding

((PizzaMargherita ⊑ *Pizza* ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓
∀*hasTopping.* (*Tomate* ⊔ *Muzzarella))*

⊓

(Pizza ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓
∀*hasTopping.* (*Tomate* ⊔ *Muzzarella)* ⊑ *PizzaMargherita))*

Normalizing the first axiom (step by step):

PizzaMargherita ⊑ *Pizza* ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓
∀*hasTopping.* (*Tomate* ⊔ *Muzzarella))*

The LHS is only a concept name, therefore it does not need to be normalized. The RHS, however, is a conjunction. The original axiom is dropped from the ontology (step `O ← O - A` from **`Normalize-RHS`**), and new axioms are created for each atomic concept or pure disjunction from the RHS:

PizzaMargherita ⊑ *Pizza*

PizzaMargherita ⊑ ∃*hasTopping.Tomate*

PizzaMargherita ⊑ ∃*hasTopping.Muzzarella*

PizzaMargherita ⊑ ∀*hasTopping.* (*Tomate* ⊔ *Muzzarella))*

All axioms are already in normal form. The first and fourth are in normal form c), like , $A_1 \sqcap \ldots \sqcap A_n \sqsubseteq A'_1 \sqcup \ldots \sqcup A'_m$, while the others are in normal form a), such as $A \sqsubseteq \exists r.C$.

Translating to matricial DNF using the rules of Table 3:

PMargh	*PMargh*	*PMargh*	*PMargh*	*PMargh*	*PMargh*
¬*Pizza*	¬*hasTopp*	¬*Tomate*	¬*hasTopp*	¬*Muzza*	*hasTopp*
					¬*Tomate*
					¬*Muzza*

Figure 10. Translating an axiom to matricial DNF using the more efficient method.

Normalizing the second axiom (step by step):

Pizza ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓
∀*hasTopping.* (*Tomate* ⊔ *Muzzarella)* ⊑ *PizzaMargherita*

Now it is the RHS that is only a concept name, not needing any normalization. The LHS is a conjunction, so, at first, only disjunctions and impurities in conjunctions require new axioms. There are no impurities in conjunctions, and only one disjunction is replaced by a new concept name then. A new axiom is created linking this new concept with the original pure disjunction:

Pizza ⊓ ∃*hasTopping.Tomate* ⊓ ∃*hasTopping.Muzzarella* ⊓ *N* ⊑ *PizzaMargherita*
∀*hasTopping.* (*Tomate* ⊔ *Muzzarella)* ⊑ *N*

The last axiom need not be normalized. Its LHS is pure a disjunction with RHS being a concept name.

Translating to matricial DNF using the rules of Table 3:

The translation could be seen on figure 11, while figure 12 contains the matrix resulting from the merge of the two matrices, meaning the two subsumptions of an equivalence. Comparing the size of the figures 5 and 12, one can witness the gain of memory when the more efficient method is used. Figure 13 displays the usual form of an equivalence axiom represented as a DNF matrix using the method.

This representation form is has at least three advantages over the most optimized representation which uses no new concepts. The first resides in the process of the translation itself, which is much more complex, because it would require operations to deal directly with the impurities' nestings. Although it can bring some advantages, such issue is beyond the purposes of that work will be put aside for further investigation. The second advantage is memory usage; the same equivalence axiom represented without any new concepts may contain many redundant (long) rows;

indeed the number of new introduced symbols (and also the number of axioms) grows linearly with the number of impurities. It is a substantial gain compared with other normalizations (the one presented here and another used for distributed resolution [Schlicht & Stuckenschmidt 2010]), whose number of new axioms grows linearly with the axioms' length. Finally, a third advantage resides on the inference processing itself. Such redundancies would cause unnecessary repetitions of searching connections, which is loss of time even if there are indices to help finding them.

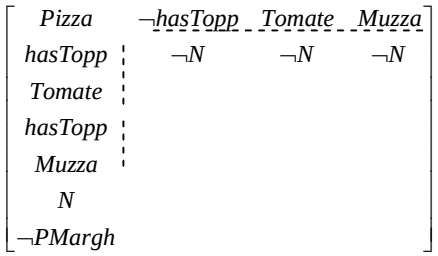

Figure 11. Translating an axiom to matricial DNF using the more efficient method.

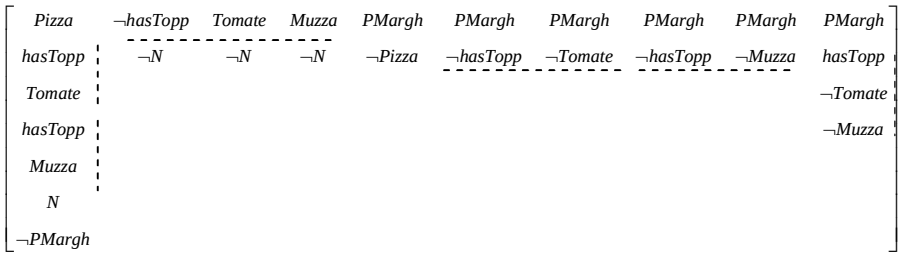

Figure 12. Final DNF matrix of an equivalence axiom using the more efficient method.

Example 4 (Positive normal form, skolemization, clause, matrix). The query

$$\left.\begin{array}{l}\mathsf{Animal} \sqcap \exists \mathsf{hasPart.Bone} \sqsubseteq \mathsf{Vertebrate} \\ \mathsf{Bird} \quad \mathsf{Animal} \sqcap \exists \mathsf{hasPart.Bone} \sqcap \exists \mathsf{hasPart.Feather}\end{array}\right\} \models \mathsf{Bird} \sqsubseteq \mathsf{Vertebrate}$$

reads in FOL as $(\forall w(Animal(w) \wedge (\exists z\ (hasPart(w,z) \wedge Bone(z)) \rightarrow Vertebrate(w))) \wedge (\forall x(Bird(x) \rightarrow Animal(x) \wedge \exists y(hasPart(x,y) \wedge Bone(y)) \wedge \exists v(hasPart(x,v) \wedge Feather(v))) \rightarrow \forall t(Bird(t) \rightarrow Vertebrate(t)))$

where the variables *y* and *t* were respectively skolemized by the function *f(x)* and the constant *c*. In positive matricial clausal form, the formula is represented by

{{*Bird(x)* ,¬*Animal(x)*}, {*Bird(x)* ,¬*hasPart(x,f(x))*}, {*Bird(x)* ,¬*Bone(f(x))*}, {*Bird(x)* ,¬*hasPart(x,g(x))*}, {*Bird(x)* ,¬*Feather(g(x))*}, {*Animal(w), hasPart(w,z), Bone(z),* ¬*Vertebrate(w)*}, {¬*Bird(c)*}, {*Vertebrate(c)*)}}. Figure 14 shows it as a matrix.

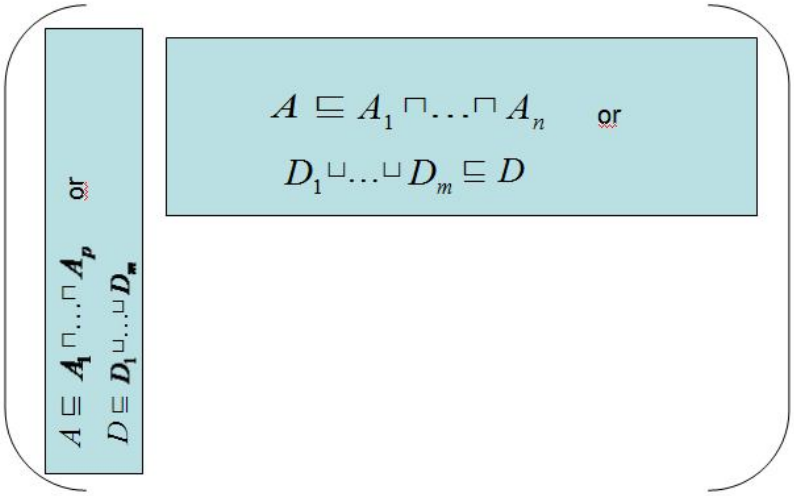

Figure 13. Usual form of an DNF matrix of an equivalence axiom represented as a DNF matrix using the more efficient method.

Bird	*Bird*	*Bird*	*Bird*	*Bird*	*Animal*	$\neg Bird(c)$	*Vertebrate*(*c*)
$\neg$*Animal*	$\neg$*hasPart*	$\neg$*Bone*	$\neg$*hasPart*	$\neg$*Feather*	*hasPart*		
					Bone		
					$\neg$*Vertebrate*		

Figure 14. An *ALC* formula and query in disjunctive clausal form represented as a matrix.

4.3 ABox representation

Regardless of the normalization scheme adopted, atomic assertions will be represented outside the matrices. This causes at least two benefits: it reduces the size of the matrices and enables the use of powerful indexing mechanisms for retrieving assertions, such as relational databases. The use of deductive databases can even bring additional benefits for reducing the reasoning strain on the *ALC* CM; nevertheless, such improvements will not be looked into for the moment. Note, however that the last example portrayed in figure 14 is an exception; the instances appear only because they stem from the query consequent Bird $\sqsubseteq$ Vertebrate ( ).

Having described the representation in detail, we explain our *ALC* connection method to reason over it in the next section.

3 The Connection Method

The connection method (CM) was created by W. Bibel in the 70s, and earned good reputation in the field known as automated theorem proving for around three decades. The reasons for this recognition are manifold:

- The method was the first easy-to understand and -implement method to deal with formulae and theorems directly, in opposition to the classical refutation methods like resolution and tableaux [Bibel 87];
- It was well formalized and supportive material was extensively published (books, articles, etc, for instance, [Bibel 87, 93]), are still under use nowadays in some universities to teach automated theorem proving;
- Any reductions, compressions, optimizations and improvements that fit to resolution, can also be properly applied to the connection method [Bibel 87 4.4, 93];
- Efficient implementations are available, including probably the smallest lean provers ever coded, leanCoP [Otten & Bibel 2003].

Such distinctive features attracted our attention in order to propose a connection method especially tailored to infer over the so popular Semantic Web ontologies, written in description logic [Baader et al 2003]. In the present work, we are proposing a connection inference method for the DL *ALC*, mainly because this formalism is the basis of many other DLs.

In order to describe the connection method as a formal inference system, we will on the following describe very briefly some necessary concepts.

Definition 10 (Path, connection, unifier, substitution). A *path* is a set of literals from a matrix in which every clause (or column) contributes with one literal. A *connection* is a pair of complementary literals from different clauses, like $\{L_1^c, \neg L_1^d\}$, where $\sigma(L_1)$ (or $\sigma(L_2)$) is the most general unifier (mgu) between predicates L_1 and $\neg L_2$. σ is the set of *substitutions*, which are mappings from variables to terms. All occurrences of the variable contained in a substitution would be replaced by the term indicated in σ. ■

Observe that each path of a formula represented as a matrix is a disjunction, therefore a verbose and non-optimized way to convert a DNF matrix into CNF (conjunctive normal form) is building a conjunction of all the possible paths (which are disjunctions).

Example 5 (Path, connection, unifier, substitution).
The (different) paths of our example matrix (see figure 2) are 12. Their orderly enumeration is exemplified below:

{Bird(x), ¬hasPart(x,f(x)), ¬Bone(f(x)), Animal(w) ,¬Bird(c), Vertebrate(c)},
{Bird(x), ¬hasPart(x,f(x)), ¬Bone(f(x)), hasPart(w,z),¬Bird(c), Vertebrate(c)},
{Bird(x), ¬hasPart(x,f(x)), ¬Bone(f(x)), Bone(z),¬Bird(c), Vertebrate(c)},

{Bird(x), ¬hasPart(x,f(x)), ¬Bone(f(x)), ¬Vertebrate(w),¬Bird(c), Vertebrate(c)},
{ ¬Animal(w), Bird(x), ¬Bone(f(x)), Animal(w) ,¬Bird(c), Vertebrate(c)},
…
…
{ ¬Animal(w), ¬hasPart(x,f(x)), ¬Bone(f(x)), ¬Vertebrate(w) ,¬Bird(c), Vertebrate(c)}

or, in CNF (read as a logical sentence) as

(Bird(x) ∨¬hasPart(x,f(x)) ∨ ¬Bone(f(x)) ∨ Animal(w) ∨¬Bird(c) ∨ Vertebrate(c)) ∧
(Bird(x) ∨ ¬hasPart(x,f(x)) ∨ ¬Bone(f(x)) ∨ hasPart(w,z)∨ ¬Bird(c) ∨Vertebrate(c))∧
(Bird(x) ∨ ¬hasPart(x,f(x)) ∨ ¬Bone(f(x)) ∨ Bone(z) ∨ ¬Bird(c) ∨ Vertebrate(c)) ∧
(Bird(x) ∨ ¬hasPart(x,f(x)) ∨ ¬Bone(f(x)) ∨ ¬Vertebrate(w)∨¬Bird(c)∨Vertebrate(c))
∧ (¬Animal(w) ∨ Bird(x) ∨ ¬Bone(f(x)) ∨ Animal(w) ∨¬Bird(c) ∨ Vertebrate(c)) ∧
…
…
(¬Animal(w) ∨ ¬hasPart(x,f(x)) ∨ ¬Bone(f(x)) ∨ ¬Vertebrate(w) ∨¬Bird(c) ∨ Vertebrate(t))

The first path can be seen on figure 15.

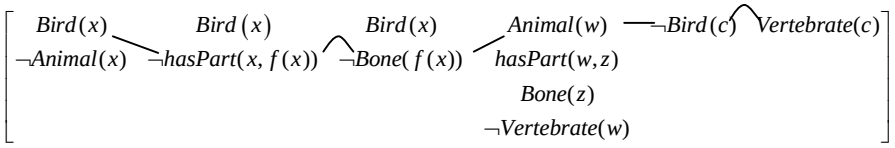

Figure 15. A path in a matrix.

Some connections from the matrix are *{Bird(x), ¬Bird(c)}, {¬hasPart(x,f(x)), hasPart(w,z)},* and *σ={x/c, w/c, z/f(c)}* is the whole matrix' mgu.

When there is a connection in a path, this path is true, since it has the form $P_\sigma \vee \neg Q_\sigma \vee \ldots$, like {*Bird(c), ¬Bird(c)*} contained in the path above, after the mgu $\sigma = \{x/c\}$ is applied. Therefore, a formula is valid when every path contains a connection. As a result, the aim of the connection method is simply to systematically find a connection in each path, being one of the major proof tasks to find a unifier that encompasses the whole matrix. A single connection indeed can eliminate many paths, e,g, the connection {*Vertebrate(w), ¬Vertebrate(c)*}, $\sigma = \{w/c\}$ solves the paths:

{Bird(x), ¬hasPart(x,f(x)), ¬Bone(f(x)), ¬Vertebrate(w),¬Bird(c), Vertebrate(c)},
{¬Animal(w), Bird(x), ¬Bone(f(x)), ¬Vertebrate(w),¬Bird(c), Vertebrate(c)} and
{¬Animal(w), ¬hasPart(x,f(x)), ¬Bone(f(x)), ¬Vertebrate(w),¬Bird(c), Vertebrate(c)}.

That leads us to the following corollary:

Corollary 1 (Validity, active path, set of concepts). An *ALC* formula represented as a matrix is *valid* when every path contains a connection $\{L, \neg L\}$ provided

that $\sigma(L_1) = \sigma($. This is due to the fact that a connection represents the tautology I - in DNF. As a result, the connection method aims at finding a connection in each path, together with a unifier for the whole matrix. During the proof, the current path is called *active path* and denoted by The *set of concepts* of a variable or instance *x* during a proof is defined by $\tau(x) \stackrel{def}{=} \{C | C(x) \in$ [Tishkovsky 2007, 2008]. ■

5 A Connection Method for *ALC*

Any method that is adapted for DL should take into account the following aspects on the design of a new method:

- For the representation:
    - Simplify the representation to a 'variableless' form, if possible
    - Identify the typical structures formed by the constructs' representation of the specific DL;
    - Characterize the representations of A- and T-Boxes;
- For the reasoning:
    - Adapt the original method to cope with the new structures;
    - Propose optimizations which reduce the problem and assure soundness, completeness and/or termination;
    - Try to prove that a set of inference services as largest as possible can be dealt by the system;

We already accomplished the representation requirements with the normalizations, particularly the last one. In the following, we describe an adapted connection calculus that reasons over *ALC* ontologies.

Definition 11 (*ALC* connection sequent calculus). Figure 16 brings the rules in sequent style of the *ALC* connection calculus, adapted from [Otten 2010].

The rules must be applied bottom-up. The matrix *M* representing $KB \models$ is put in the bottom of the *Start rule*, and a clause $C_1 \in$ is chosen as the first clause to ensure that takes part in the proof. Then, the three last rules are applied. The key rule for the calculus is the *Extension rule*, once it finds the connections of the form { $E(x), \neg E($}, ({*E* ,-} in the notation used here) or {*r(x,y), r(z,w)*}and the proper unifier σ at the same time. In the first case σ={*x/y*} and in the second σ={*x/z, y/w*}. If an active path contains a connection and the set of literals *C* is empty, the subgoal is proved, and the *Axiom rule* is applied.

Besides this slight change in the *Start rule*, the major differences from the original CM sequent calculus [Otten 2010] are a new *Copy rule (Cop)*, together with a blocking mechanism. If the current literal can only connect a clause already in the active path, this clause is copied to the matrix, and the indexing function $\mu: M \rightarrow$, that assigns the number of copies of each clause, is incremented. The use of this

function avoids the ordering of individuals, as done in the original connection method.

$$Axiom\ (Ax)\ \frac{}{\{\},M,Path}$$

$$Start\ Rule\ (St)\ \frac{C_2,M,\{\}}{\varepsilon,M,\varepsilon}$$

where M *is the matrix* $KB \models \alpha$, C_2 *is a copy of* $C_1 \in \alpha$

$$Reduction\ Rule\ (Red)\ \frac{C^{\sigma},M,Path \cup \{L_2\}}{C \cup \{L_1\},M,Path \cup \{L_2\}}$$
$$with\ \sigma(L_1) = \sigma(\overline{L_2})$$

$$Extension\ Rule\ (Ext)\ \frac{C_2^{\sigma}\backslash\{L_2^{\sigma}\},M,Path \cup \{L_1\} \quad C^{\sigma},M,Path}{C \cup \{L_1\},M,Path}$$
$$with\ \ C_2\ is\ a\ copy\ of\ C_1 \in M, L_2 \in C_2, \sigma(L_1) = \sigma(\overline{L_2}),$$

$$Copy\ Rule\ (Cop)\ \frac{C \cup \{L_1\},M \cup \{C_2^{\mu}(x_{\mu})\},Path \cup \{L_2(x)\}}{C \cup \{L_1\},M,Path \cup \{L_2\}}$$
$$with\ L_1 \in C_2\ and\ \sigma(L_1) = \sigma(\overline{L_2})\ and$$
$$(x_{\mu}^{\sigma} \in N_{\sigma}\ or\ \tau(x_{\mu}^{\sigma}) \subseteq \tau(x^{\sigma}))\ (blocking\ conditions)$$

Figure 16. The *ALC* connection sequent calculus rules (adapted from [Otten 2010]).

Note that the existence of this rule was not mandatory, since the original system already disposes of a similar behavior. Nevertheless, here it serves the purposes of isolating the blocking from the *Ext* rule, as well as letting explicit the copy process. *Ext* precedes over *Cop*, so that clauses are copied only when no extension is possible.

Another important remark about the rule is that the copy is virtual, in the sense that only the index is incremented. The connection method requires only one copy of the matrix in memory, and this main advantage should not be neglected. Its use of memory can be better than tableaux in cases the proof demands many derivations, like with cyclic ontologies.

Blocking didn't occur in the original CM due to FOL semi-decidability, but it consists in a common practice in DL to guarantee termination. In order to carry out that, we have to check whether the set of concepts associated to the variable (i.e., if the new was unified) of the new literal being created by the *Cop* rule is not contained in the set of concepts of the original variable x from $L_2($ (in the rule, $\tau(x^{\iota}$ [Tishkovsky 2007, 2008].

Example 6 (*ALC* connection sequent calculus). Figure 17 brings the proof of our example from figure 14.

Figure 17. *ALC* connection sequent calculus example.

Theorem 1 (Correctness and completeness) An *ALC* formula in positive matricial form is valid iff there is a connection proof for " ", i.e. the application of the rules makes the *Axiom rule (Ax)* applicable to all leaves of the generated tree.

Proof The changes in rules do not lose the correctness and completeness of the original CM [Bibel 1987], III.3.9 and 3.10, *viz*: (i) The original *Start rule* allows any clause to start the method; (ii) blocking does not affect correctness and completeness; it only guarantees termination instead.

Theorem 2 (Termination). The *ALC* CM always terminates.

Proof If there are no cycles, then the calculus finishes in a finite number of steps because the number of choices arising from the finite number of clauses and possible connections is also finite. If , even in the presence of cycles, the system always terminates (proof at [Bibel 1987]).

For , with a cyclic *KB*, the *Copy rule* obliges the set of concepts of the newly created instance of literal not to be a subset of any of the sets of concepts of other introduced instances of the literal. Once every set of concepts is a subset of and is finite, the number of distinct sets is limited by . Therefore, the number of copies for any literal is finite and hence *ALC* CM always terminates. ■

The system can also be expressed in an easier way using matrices to build proofs. An *ALC* CM calculus' algorithm of the system was adapted from the one described in [Kreitz 2006] and [Bibel 1987], III.7.2. , and is presented on Figure 18. The algorithm *ALC* has the following changes from these originals:

- Instead of choosing a random clause and maintaining a list of alternative start clauses (ALTc, in the original algorithms), this new algorithm reflects the new *Start Rule* which requires a start clause from the consequent ( ), and need not iterate in the ALTc list.
- The introduction of blocking required the creation of the separate function **`Instantiate`** that tests whether an "artificial instance" was already created (blocking condition). Therefore, separate sets for "normal" and "artificial" instances were needed, and should manage the creation of instances during unification. This is the role of the new function **`Unify.`**

Example 7 (*ALC* connection calculus). Figure 19 deploys the proof of the query stated in example 1. In the figure, literals of the active path are in boxes and arcs denote connections. For building a proof, we first choose a clause from the consequent (*Start rule*), say, the clause { *Bird(c)*} and a literal from it (*Bird(c)*).

Step 1 connects this clause with the first matrix clause. An instance or variable - representing a fictitious individual that we are predicating about -, appears in each arc, for this connection, the instance *c*. The arrow points to literals still to be checked in the clause (¬*Animal* in Step 1), that should be checked afterwards. After step 2, the connection {¬*Animal*, *Animal*} is not enough to prove all paths stemming from the other clause, the one with literal ¬*Animal*. In order to assure that, the remaining literals from that clause, *viz hasPart*, *Bone* and *Vertebrate*, have still to be connected.

```
ALC  C(F: ALC  DL matrix): boolean (valid or invalid);
[Inicialization]
↑-list:=[];                    // list of literals to be checked - open paths
ALTc :=  []                    // extension alternative clauses
paths(1) :=  [] ; paths(2) :=  // open paths stemming from copies
p:={}; (current path) i := 1; M :=  F ; σ :={}; (unifier)
σ-List := [] ;                                  [
For all clauses c ∈ α and α ∈ M and c not completely connected yet
      Instantiate (c, x ∉ A )  // x is a new instance
      M← M-c;
         While M != null
            Choose a literal L ∈ c; c ← c-L              [paths]
            If c != {} then ↑-List ←push(c; p;M)
            p ←p U {L};
            paths (i+1) ←push (L;M; σ; p; ↑-List)       [open paths]
            If there is a clause d ∈ M and ¬L ∈ d then   [Extension clause]
               ALTc ← push (L; d';M; σ; p; ↑-List)
                       for alternative clauses d' ∈ M and ¬L ∈ d';
               M← M-d; Choose a clause e ∈ d;            [Extension amount]
               If Unify (L,d) then c←d-e;                [Extension]
                    If c = {} then
                       if ↑-List = [] then F valid       [Success]
                       Choose a (c; p;M) to pop from ↑-List;
               else    if ALTc !=[] then        [alternative extension clauses]
                          Choose the last (L; d;M; σ; p; ↑-List)
                                and pop it from ALTc;
                          Define the extension step with the choice
                                of the next extension;
                       else if paths(i) =[] then [alternative copies]
                                i++; Choose an open path (L;M; σ; p; ↑-List)
                                        and pop it from paths(i);
                                M ← M U F ;
                                Define the extension step with a choice of the
                                          extension clause
                             else F invalid [Failure]
Unify (instance x, clause c):  Boolean;
For all e ∈ d (literals on c)
      If (x,e ∈ A and x,e ∉ AA , nIA) [pure instances]
         or (x or e ∈ nIA  [x or e is in an universal restriction]
         and the other x or e either ∉ A , nIA , AA  [not instantiated yet]
         or ∉ AA  [artificial instance]
                then  Failure
      else    For all literals e ∈ c , if !Instantiate (x,e) return failure
              return Success;
Instantiate (literal L, instance x): Boolean;
If L ∈ N_C (concept) then
  If L(x)∉A (not instantiated yet) or there is no L(y)∈AA (blocking) then
     A ←A + L(x); AA ← AA + L(x); Success
  Else Failure
Else [L is a property]
  If L(x,y)∉A (not instantiated yet) or there is no L(z,w)∈AA (blocking)  then
     A ←A + L(x,y), AA ←AA + L(x,y), where y ∉ A [y is a new instance];
     If L is linked vertically [∃r.C] then Instantiate(C,y);
     Else if L is linked horizontally [∀r.C] then
             Instantiate(C,y); nIA ←nIA + L(x,y);
          Else Failure;
```

Figure 18. *ALC* connection method algorithm *ALC* CR_1^*.

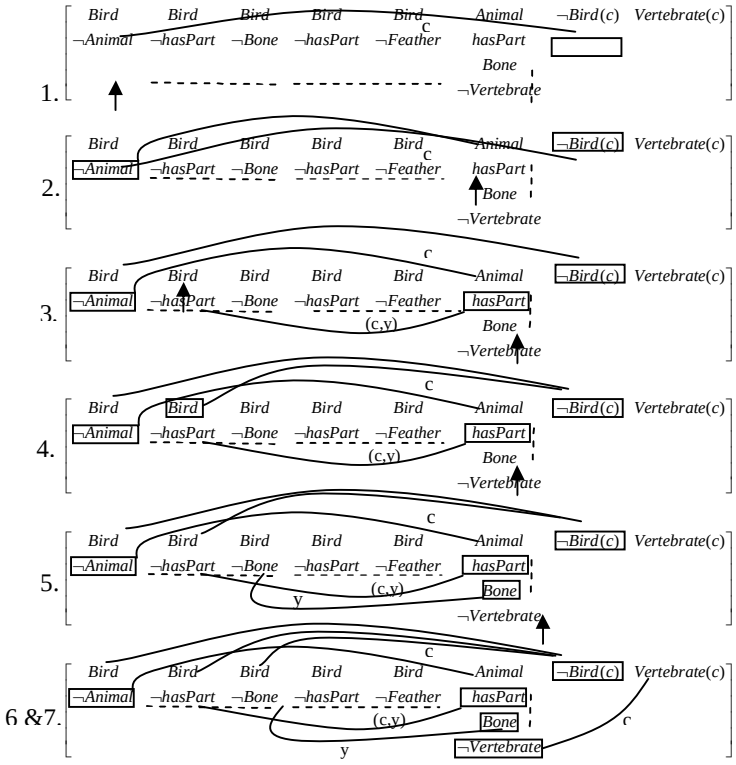

Figure 19. A connection proof example in matricial form.

Then, in step 3, when we connect *hasPart*, we are not talking about instance *c* anymore, but about a relation between it and another variable or fictitious individual, say *y* (indicated by *(c,y)*).

Until that moment, we were only applying the *Extension rule*. However, in step 4, we use the *Reduction rule*, triggered by its two enabling conditions: (i) there is a connection for the current literal already in the proof; and (ii) unification can take place. Unification would not be possible if we were referring to different individuals or skolemized functions (in *ALC*, equality among individuals is not necessary).

A small note on unification is necessary here, because it brings a small trick to the calculus. Since horizontal dashlines represent universal restrictions ( ), the qualifier concept (*C*, represented as - in the matrix) correspond to a skolemized concept (say *C(f(c))*). Therefore, it can only be unified with variables, but not with concrete individuals or other skolemized qualifier concepts.

Resuming to the example, a connection for the literal *Bone* was needed. That time we are referring to the variable (or fictitious individual) *y*, for the reason that it stands for a part of the individual *c* (*hasPart(c,y)*).

In case the system is able to summon the query, the processing finishes when all paths are exhausted and have their connections found. In case a proof cannot be entailed, the system would have tried all available options of connections, unifiers and clause copies, having backtracked to the available options in case of failure.

In the next subsection we show that the connection method fits to all the DL standard inference services and reduce them both to subsumption and validity.

5.1 Inference services and their reductions to subsumption and validility

According to [Baader et al 2003], there are four standard inference services for any DL. We enlist and explain them in the following:

Subsumption If, in every model of a TBox $\mathcal{T}$, the interpretations C^I and D^I from the concepts C and D respectively are so that $C^I \subseteq D^I$, then we say that $\mathcal{T} \models C \sqsubseteq D$, or, in other words, that C is *subsumed* by D.

Unsatisfiability A concept C is deemed *unsatisfiable* w.r.t. $\mathcal{T}$, if no non-empty model of $\mathcal{T}$ can be generated for it.

Equivalence If, in every model of $\mathcal{T}$, the interpretations C^I and D^I from the concepts C and D are so that $C^I = D^I$, then we say that $\mathcal{T} \models C \equiv D$, or, in other words, that C and D are *equivalent.*

Disjointness If, in every model of $\mathcal{T}$, the interpretations C^I and D^I from the concepts sC and D are so that $C^I \cap D^I = \varnothing$, then we say that C and D are *disjoint*.

All these inference services are available for our $\mathcal{ALC}$ connection method. As occurs with traditional tableaux methods, these services can be achieved in our $\mathcal{ALC}$ CM either by reduction to subsumption or to unsatisfiability. Before showing show this occurs, we first show how to prove unsatisfabiliaty regardless of the reduction adopted.

Proposition 1. (Unsatisfiability) Proving an arbitrary concept C *unsatisfiable* in any direct proof method is equivalent to proving $\mathcal{T} \models \top \sqsubseteq$.

Proof Proving a class inconsistent stands for proving $\mathcal{T} \models C \sqsubseteq$ Since $A \sqsubseteq$ is equivalent to $\neg B \sqsubseteq$ -, then $\mathcal{T} \models C \sqsubseteq$ can be proved by $\mathcal{T} \models \top \sqsubseteq$ -

Example 8. (Unsatisfiability). If we consider concepts Man and Woman disjoint (or vice-versa) the axiom Human Man Woman is inconsistent, as could be seen in the matrix of figure. Note that the matrix is $\mathcal{T}$ with a last column for Human.

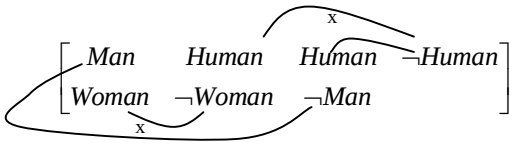

Figure 20. Inconsistency proof of the concept Human, if Human Man Woman.

We can then present the reductions.

Proposition 2. (Reduction to subsumption) For concepts *C* and *D* and instances *C(a)* and *r(a,b)* over individuals a and b, we have that:

i. *C* and *D* are equivalent ⇔ *C* is subsumed by *D* and *C* is subsumed by *D*, as usual;
ii. *C* and *D* are disjoint ⇔ either *C* is subsumed by ¬*D* or *D* is subsumed by ¬*C*.
Since we don't need to use the top and bottom concepts ⊤ and ⊥, we can prove disjointness either by proving that $\mathcal{T} \models C \sqsubseteq \neg D$ or $\mathcal{T} \models D \sqsubseteq \neg C$.
iii. *C* is unsatisfiable ⇔ $\mathcal{T} \models \neg C$.

We will show in the following that, in practical terms, this proof depends upon a subsumption axiom. Given that in *ALC*, all assertions $a \mid a \in N_O$, either concept or role assertions, are not referenced in the TBox $\mathcal{T}$ ($a \notin \mathcal{T}$), we need an axiom of the form $E \sqsubseteq C$, where *E* is any *ALC* expression, in order to connect ¬*C*. As a result, we will then have to prove this axiom ($E \sqsubseteq C$), or more specifically, to connect *E* properly.

Proposition 3. (Reduction to satisfiability) For concepts *C* and *D* and instances *C(a)* and *r(a,b)* over individuals *a* and *b*, we have that:

i. *C* is subsumed by *D* ⇔ ¬*C* ⊔ *D* is a tautology.
ii. *C* and *D* are equivalent ⇔ ¬*C* ⊔ *D* and *C* ⊔ ¬*D* are both tautologies.
iii. *C* and *D* are disjoint ⇔ ¬*C* ⊔ ¬*D* is a tautology

For the *ALC* CM, reducing to subsumption is much more natural, so we have chosen this reduction as the procedure to be adopted for this proof system. Having discussed how the system works, in the next section we describe how it deals with cycles and blocking.

5.2 Cycles, blocking

A relevant ability for a DL reasoner is dealing with cyclic ontologies. *ALC* CM not only can cope with the task (thanks to the *Copy rule*), but also saves memory during the process, due to needing only one copy of the matrix in memory [Bibel 87, 93].

During the proof search of an assertional query or sub-query (i.e., one which refers to assertions), connections to instances of the ABox should be privileged, rather than to variables, otherwise blocking could happen when connections are still available. An example on cycling shows how *ALC* CM works in this case.

Example 9 (Cycles). If the TBox hasSon.(Dr ⊔ DrAncestor) DrAncestor

receives the query KB DrAncestor(ZePadre), under the ABox:

hasSon (ZePadre,Moises), hasSon (Moises,Luiz), hasSon (Luiz,Fred), Dr (Fred),

then the proof is represented by Figures 21a and 21b.

Figure 21a brings an explicit copy of a clause which participates in the proof twice. This presentation is surely more user-friendly than the one from Figure 21b. Figure 6b uses indices to denote how the only copy was used with the different individuals and instantiations. Let us explain this last figure in detail.

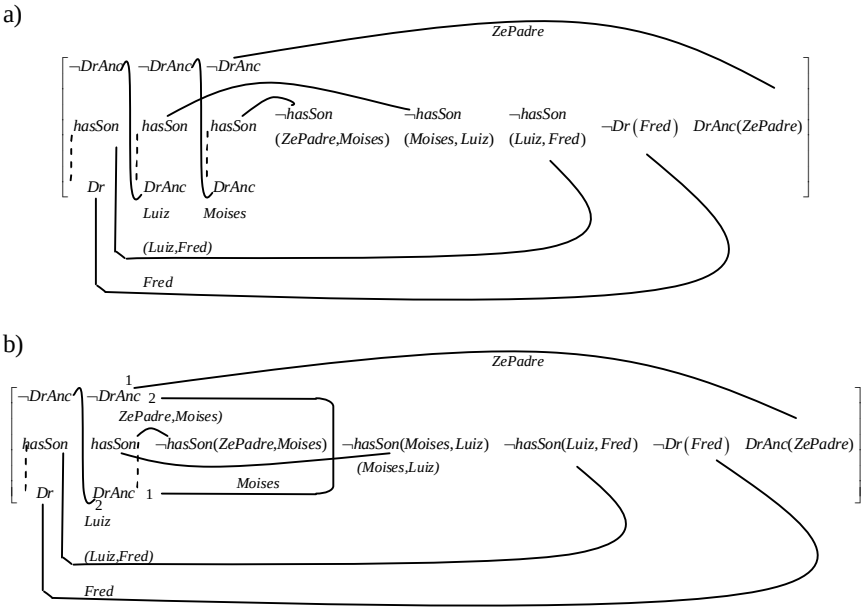

Figure 21. Proof representations of a cyclic axiom, with (a) explicit and (b) implicit copies.

First, a connection with the consequent (DrAncestor (ZePadre)) was settled. The connected clause was unified with instance ZePadre, generating two literals to be connected: hasSon (ZePadre, x) and DrAncestor(x). The first was connected to hasSon (ZePadre, Moises), and since σ(x) = Moises, then DrAncestor (Moises) needs to be proved. After that, instead of copying the clause, a connection to a literal in the same column was set. The index representing to denote virtual copies was incremented when this connection was created; that is indicated by the numbers 1 and 2 in the extremes of the connection. This practice already exists in the CM, under the name *implicit amplification* [Bibel 83]; we adopt it here with exactly the same notation.

The last connection generated literals hasSon (Moises, y) and DrAncestor (y), which were unified with ABox instances, with σ(y) = Luiz. Next, already with =2, DrAncestor (Luiz), was connected to the first column. The remaining literals hasSon (Luiz, z) and Dr(z) were then connected to instances hasSon (Luiz, Fred) and Dr(Fred), closing all possible paths and consequently proving the query.

Next, we look into the functioning of blocking, since it is based on copies.

Example 10 (Blocking). Suppose the last example contained no ABox. The system does not summon the query, as can be seen in Figure 22.

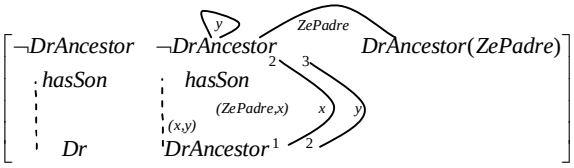

Figure 22. A blocking representation example.

At first, the connection {DrAncestor (ZePadre), DrAncestor(z)} was found with σ(z) = ZePadre. Next, we had literals hasSon(ZePadre, x) and DrAncestor(x) yet to be proved. Assuming that the variable x plays the role of a fictitious individual, then we had to prove that, if x exists, then DrAncestor(x) is also true. Thus, we set to 2 and made a cyclical connection in the same clause, so leaving the literals hasSon(x,y) and DrAncestor(y) yet to be proved.

Next, the same connection is repeated, creating another virtual copy of the clause, with set to 3. However, y could not be accepted as a fictitious individual this time, once the blocking conditions were met: (i) y is a variable, not a concrete instance (y ∉ N_I; (ii) $\pi(y) \sqsubset \pi$ (both equal to {DrAncestor}), and (ii) $\mu(C_y$ $\mu($. Thus, y was blocked and the query was not entailed.

Therefore, during proof search for an assertional query (i.e., one which refers to assertions) or sub-query, connections to the ABox must be privileged, rather than to variables. This is justified by the possible presence of blocking as shown.

5.3 Open World Semantics

Finally, a last issue which is worth discussing is open world reasoning. We noticed that *ALC* CM allows for the usage suggested in [Baader et al 2003] for solving queries that require this type of reasoning.

Example 11 (Open World Semantics). Does the ABox A_{OE} example (from [Baader et al 2003])

hasChild(Jocasta, Oedipus), hasChild(Jocasta, Polyneikes),
hasChild(Oedipus, Polyneikes), hasChild(Polyneikes, Thersandros),
Patricide(Oedipus), ¬Patricide(Thersandros),

entails the following query:

$A_{OE} \models (\exists hasChild.(Patricide \sqcup \exists\ hasChild.\neg Patricide))(Jocasta)$?

Open world inferences demands careful analysis of which sets of models could arise from the missing information. In this simple example, there are two possible sets of models: the ones in which Polyneikes is a patricide (in which thus *Patricide(Polyneikes)* should be an assertion) and the other in which he is not a patricide (and then ¬*Patricide(Polyneikes)* is an assertion). Proving both cases would draw the entailment problem above stated. We could represent them in a same matrix, but for the sake of clarity, we do it in separate ones:

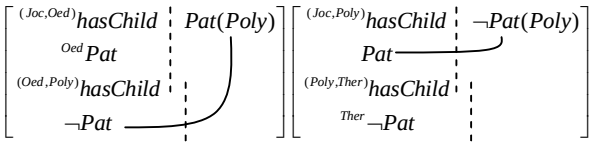

Figure 23. Open world proof example, in which the two sets of models (represented by the assertion *Patricide(Polyneikes)* and its negation) are successfully tried out.

In the next section, we discuss *ALC* CM in the light of knowledge representation, potential advantages on reasoning and implementation details to be incorporated.

6 Discussion

While planning the *ALC* CM, i.e. adapting the original CM to the DL *ALC*, we designed its notation trying to avoid any ambiguities to arise from the fact that variables do not take part of the representation – that were the reasons for the dashlines replacing the quantifiers. We plan to the change this notation even further, since atomic assertions are better stored outside the matrices, causing at least two benefits: matrix size can be dramatically shrunk and the possibility of relying on powerful indexing mechanisms for retrieving assertions, such as relational databases. The use of deductive databases can bring additional profits for reducing the reasoning strain on the *ALC* CM. However, even keeping only TBoxes, matrices can be sparse, causing a memory waste. A simple solution is storing matrices as arrays of arrays.

Still concerning representation, although we have proposed a normal form which includes less introduced artificial concepts, thus being more efficient than others [Baader et al 2005, 2005a, Schlicht 2010], an even compacter normal form is possible, if sub-matrices come into play. By doing so, many columns could be saved, as portrayed in Figure 24. In this example, the matrix has four less predicates than the original one (from Example 2). Moreover, such representation would require less reasoning overhead for the backtracking, since one connection on a shared column part serves many clauses.

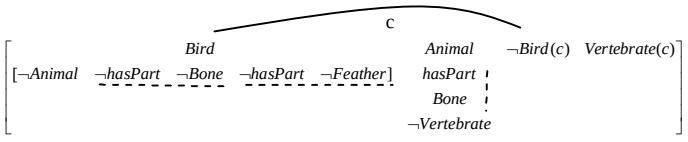

Figure 24. An *ALC* formula and query represented as a matrix, with a shared connection.

As for the reasoning, the first fact to be taken into account is the strong similarity between CM and tableaux [Bibel 93]. Therefore, reductions and optimizations designed over the latter are very likely to be incorporated to the former. Additionally, a performance at least comparable to DL tableaux is expected, given the fact that CM itself usually displays a good performance (it already won a CASC competition once [Moser et al 97]). Of course, a deeper investigation on the cases in which could be advantageous to rely on tableaux or the CM is on our research agenda.

Another aspect regarding reasoning resides in *ALC* CM specific reductions. For instance, to infer over two equivalence axioms, that usually occupy many columns, an *ALC* CM reasoner will need only one of the two subsumption axioms of each, thus reducing both matrix size and proof search space.

6 Conclusions and Future Work

We have formalized a connection method to take on the DL *ALC*, by adapting the CM sequent rules from [Otten 2010] and introducing some notational improvements, the key one. being the representation without variables. Of course, we plan to continue this work in many research directions, such as implementations, other DLs, Semantic Web, etc.

First, we intend to extend the work presented here to more complex description logic languages in a near future. Particularly, formalizations and implementations for the DLs *EL*, *EL++*, *SHIQ* and *SROIQ* will be practically useful for Semantic Web and for some other biomedical applications that we are involved in.

Last but not least, lean implementations written in Prolog, in the flavor of leanCop [Otten & Bibel 93], that demand small memory space, can serve applications that are constrained in memory, such as stream reasoning in mobile applications, for instance. They are also in our research agenda.

Acknowledgments

The first author thanks the Brazilian research agency “Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (Capes)” for the partial support of the work, under Project number 317/09.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (Eds.): The Description Logic Handbook. Cambridge University Press, 2003.
2. Baader, F., Brandt, S., Lutz, C. Pushing the EL Envelope. In Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence IJCAI-05, Edinburgh, UK, 2005.
3. Baader, F., Brandt, S., Lutz, C. Pushing the EL Envelope. LTCS-Report 05-01, Chair for Automata Theory, Inst. for Theoretical Computer Science, TU Dresden, Germany, 2005.

4. Bibel, W. Automated theorem proving. Vieweg Verlag, Wiesbaden, 1987.
5. Bibel, W. Deduction: Automated Logic. Academic Press, London, 1993.
6. Bibel, W. Matings in Matrices. Communications of the ACM. Vol. 26 Issue 11, Nov. 1983.
7. Ghilardi, S., Lutz, C., Wolter, F. Did I damage my ontology: A Case of Conservative Extensions of Description Logics. Proceedings of the Tenth International Conference of Principles of Knowledge Representation and Reasoning 2006 (KR'06), AAAI Press, 2006
8. Kreitz, C. Die Konnektionsmethode (Prädikatenlogik). Class notes of course on Inference Methods, http://www.cs.uni-potsdam.de/ti/lehre/06-Inferenzmethoden/slides/slides-05-anim.pdf 2006-2007 [Accessed 10 Dec 2010].
9. Moser, M., Ibens, O., Letz, R., Steinbach, J., Goller, C., Schumann, J., Mayr, K. SETHEO and e-SETHEO - the CADE-13 systems. J. of Automated Reasoning, 18(2):237{246, 1997.
10. Motik, B., Shearer, R., and Horrocks, I. Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research, 36:165–228, 2009.
11. Otten, J. Restricting backtracking in connection calculi. AI Comm., 23(2-3): 159-182 2010.
12. Otten, J., Bibel, W. leanCoP: Lean Connection-Based Theorem Proving. Journal of Symbolic Computation, Volume 36, pages 139-161. Elsevier Science, 2003.
13. Patel-Schneider, P., Hayes, P., Horrocks, I.: OWL - Web Ontology Language Semantics and Abstract Syntax W3C Recommendation. www.w3.org/tr/2004/rec-owl-semantics-20040210/ , 2004. [Accessed 10 Jan 2010].
14. Schlicht, A. Stuckenschmidt, H. Peer-to-peer Reasoning for Interlinked Ontologies. Int. Journal of Semantic Computing, Special Issue on Web Scale Reasoning, 2010
15. Schmidt, R., Tishkovsky, D. Analysis of Blocking Mechanisms for Description Logics. In Proceedings of the Workshop on Automated Reasoning, 2007.
16. Tishkovsky, D. Blocking Mechanisms in Description Logics, A General Approach. www.uivt.cas.cz/semweb/download.php?file=08-Tishkovsky-Logic-Blocking&type=pdf Prague, 2008. [Accessed 10 Dec 2010].
17. Watt, D. Programming Language Concepts and Paradigms. Prentice Hall. 1993.