Using Prolog unification to solve non-standard reasoning problems in Description Logics

Simona Colucci, Francesco M. Donini

DISUCOM, Università della Tuscia, Viterbo, Italy

Abstract. We present a Logic Programming prototype implementation working as proof-of-concept for a unified strategy proposed in our past research to solve several non-standard reasoning problems in Description Logics (DLs), denoted by *Constructive Reasoning*. In order to proof both the problem-independence and the logic-independence of the adopted approach, the prototype is focused on the solution of three different problems — namely Least Common Subsumer, Concept Abduction and Concept Difference — and two different, though simple, DLs, *i.e.*, $\mathcal{EL}$ and $\mathcal{ALN}$. Accordingly to the implemented strategy, problems are formalized as conjunction of both subsumption and non-subsumption statements, causing the whole prototype to rely on a Prolog program solving subsumption. The program is built around a predicate, which on the one hand checks for the existence of subsumption relations between ground elements, providing boolean answers, and on the other hand, if inverted, exploits Prolog built-in unification to enumerate variable values making subsumption true between concept terms containing concept variables.

1 Introduction

The power of knowledge lays in its ability to enhance the production of unknown information, through management strategies whose significance increases with the level of novelty introduced by provided results.

In past knowledge management literature, in fact, interest has been given to the proposal of special purpose inferences allowing for exploiting as much as possible the informative content achieved through knowledge representation effort. To this aim, several non-standard reasoning services have been proposed and continue to be investigated to cope with different representation or inference needs. The most relevant services we may cite are explanation [1], interpolation [2], concept abduction [3], concept contraction [4], concept unification [5], concept difference [6], concept similarity [7], concept rewriting [8], least common subsumer [9], most specific concept [10], knowledge base completion [11], forgetting or uniform interpolation [12].

We notice that the crucial role of non-standard reasoning in the process of capturing unexpected sources of information has been stressed also in research fields apparently far from knowledge representation [13].

Moreover, recent Description Logics (DLs) literature has shown interest for easily tractable, even though not very expressive, sub-languages, like $\mathcal{EL}$ ([14, 15, 16]).

In our past research [17] we proposed an integrated approach and solving strategy for dealing with several different non-standard inferences. The framework, presented

as independent of the DL adopted for knowledge representation, takes a *constructive reasoning*[1] perspective on problem solving: most inferences are in the form "Find one or more concept(s) C such that $\{$sentence involving C $\}$" and the proposed framework aims at building such C.

In order to show the feasibility of such an integrated constructive reasoning approach, we here present a prototype implementation in Logic Programming solving Least Common Subsumer, Concept Difference and Concept Abduction in the simple DLs $\mathcal{EL}$, $\mathcal{ALN}$.

Though still inefficient at this stage, the prototype works as proof-of-concept for the integrated solution framework. It exploits the property of our approach according to which most non-standard reasoning problems may be formalized as conjunction of both subsumption and non-subsumption statements and therefore relies on a Prolog program solving subsumption, built around a main predicate called either *subs_el* or *subs_aln*, depending on the adopted $\mathcal{DL}$. In particular, we show how to invert the *subs* predicate (either *subs_el* or *subs_aln*), so that not only it can check for the existence of subsumption relations between ground elements, providing boolean answers, but it can also exploit Prolog built-in unification to enumerate variable values making subsumption true between concept terms containing concept variables. The approach takes a generate-and-test strategy.

In the next section, we recall some preliminary notions of the formalism and reasoning services we adopt. In Section 3 we shortly recall how to model the three problems in the integrated framework. Then,we describe the architecture of the prototype implementing the solving strategy in Section 4, before delving into details of subsumption program, on which the whole prototype relies, in Section 5. We show how to query the presented prototype in Section 6, and, finally, close the paper with discussions and future work.

2 Basic Description Logics

In the following, to make this paper self-contained, we briefly recall the formalism we adopt for knowledge representation; the reader interested in further details may consult a reference book [19]. DLs are a family of formalisms and reasoning services widely employed for knowledge representation in a decidable fragment of First Order Logic.

The alphabet of each DL is therefore made up by unary and binary predicates, denoted as **Concepts Names** and **Role Names**, respectively. Each DL allows for a different set of constructors for describing concepts. The set of constructors allowed by a DL characterizes it in terms of expressiveness and reasoning complexity: the more a DL is expressive, the harder is inferring new knowledge on its descriptions [19, Ch.3].

More complex concepts inclusions and definitions may be modeled for the formal representation of the domain of interest, the intensional knowledge which takes the name **TBox** in DL systems. By the way, in this paper we admit an empty **TBox**, given that it has been shown that the presence of a **TBox** affects the termination of some of the services we implement even for very simple DLs [20].

[1] We notice that, although the lexical similarity of denotation, our approach is not related to *Constructive Description Logics* [18].

The semantic of concept descriptions is conveyed through an **Interpretation** $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non–empty set denoting the domain of $\mathcal{I}$ and $\cdot^{\mathcal{I}}$ is an interpretation function such that: i) $\cdot^{\mathcal{I}}$ maps each concept name A in a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$; ii) $\cdot^{\mathcal{I}}$ maps each role name R in a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.

DL constructors adopted in the paper are shown in Table 2. The most important

Table 1. DLs set of conctructors

Constructor Name	**Syntax**	$\mathcal{EL}$	$\mathcal{ALN}$
top-concept	$\top$	x	x
bottom-concept	$\bot$		x
atomic negation	$\neg A$		x
conjunction	$C \sqcap D$	x	x
value restriction	$\forall r.C$		x
existential restriction	$\exists r.C$	x	$\exists r.\top$
at-least restriction	$(\geq m\, r)$		x
at-most restriction	$(\leq m\, r)$		x

reasoning service in DL checks for specificity hierarchies, by determining whether a concept description is more specific than another one or, formally, if there is a *subsumption* relation between them.

Definition 1 (Subsumption). *Given two concept descriptions C and D in a DL $\mathcal{L}$, we say that D subsumes C if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. We write $C \sqsubseteq D$.*

The two, simple, DLs adopted in this paper are such that subsumption of concepts can be computed by so-called structural subsumption algorithms, i.e., algorithms that compare the syntactic structure of (possibly normalized) concept descriptions. Algorithms for structural subsumption in $\mathcal{ALN}$ have been proposed in the literature [21]. Differently from the approaches proposed for $\mathcal{EL}$ so far [22], we adopt a structural subsumption algorithm, which, although quite inefficient, allows for easily inverting subsumption in our Prolog implementation.

3 Background Framework

The approach presented in our past research [17] models each of the problems at hand as Optimal Solution Problem, whose definition exploits specific second order formulas, written as conjunction of concept subsumptions and non-subsumptions, in the following form:

$$\Gamma = (C_1 \sqsubseteq D_1) \wedge \cdots \wedge (C_\ell \sqsubseteq D_\ell) \wedge (C_{\ell+1} \not\sqsubseteq D_{\ell+1}) \wedge \cdots \wedge (C_m \not\sqsubseteq D_m) \quad (1)$$

In Formula (1), $C_1, \ldots, C_m$,$D_1, \ldots, D_m \in \mathcal{DL}$ denote concept terms containing concept variables $X_0, X_1, \ldots, X_n$. We say that Γ is satisfiable in $\mathcal{DL}$ iff there exists a substitution $\sigma = X_0 \rightarrow E_0, ...X_n \rightarrow E_n$, where E_1, E_n, are concept terms in $\mathcal{DL}$, such that $\sigma(\Gamma)$ is true (*i.e.*, , each subsumption and non-subsumption statement in (1) is true). If Γ is satisfiable in $\mathcal{DL}$ then $\mathcal{E}$ is called a **solution** for Γ and the set of solutions for Γ is defined as:

$$SOL(\Gamma) = \{\mathcal{E} = \langle E_0, \ldots, E_n \rangle \mid \mathcal{E} \text{ is a solution for } \Gamma\}$$

Definition 2 (OSP). *An Optimal Solution Problem (OSP)* ***P*** *is a pair $\langle \Gamma, \prec \rangle$, where Γ is a formula of the form (1) and $\prec$ is a preorder over $SOL(\Gamma)$. A solution to* ***P*** *is a concept tuple $\mathcal{E}$ such that both $\mathcal{E} \in SOL(\Gamma)$ and there is no other $\mathcal{E}' \in SOL(\Gamma)$ with $\mathcal{E}' \prec \mathcal{E}$.*

3.1 Non-standard Services in DLs as OSPs

In the following, we recall how to model the three investigated problems as OSP. Aiming at a fixpoint computation for solving each of the problems below, a greatest element (*i.e.*, a least preferred one) w.r.t. $\prec$ is provided, which could be used to start the iteration of an inflationary operator.

Least Common Subsumer

Definition 3. *[23] Let C_1 and C_2 be two concepts. The Least Common Subsumer (LCS) of C_1, C_2 is the least element w.r.t. $\sqsubseteq$ of the set of concepts which are Common Subsumers of C_1, C_2 and is unique up to equivalence.*

Common subsumers of C_1, C_2 satisfy the formula of the form (1):

$$\Gamma_{LCS} = (C_1 \sqsubseteq X) \wedge (C_2 \sqsubseteq X)$$

Then, the LCS problem can be expressed by the OSP $LCS = \langle \Gamma_{LCS}, \sqsubset \rangle$. We note that $\top$ is always a solution of Γ_{LCS} which is a greatest element w.r.t. $\sqsubset$.

Concept Difference Following the algebraic approaches adopted in classical information retrieval, Concept Difference [6] was introduced as a way to measure concept similarity.

Definition 4. *[6] Let C and D be two concepts such that $C \sqsubseteq D$. The Concept Difference $C - D$ is defined by $max_{\sqsubseteq}\{B \in \mathcal{DL}$ such that $D \sqcap B \equiv C\}$.*

We can define the following formula of the form (1):

$$\Gamma_{DIFF} = (C \sqsubseteq (D \sqcap X)) \wedge ((D \sqcap X) \sqsubseteq C)$$

Such a definition causes Concept Difference to be modeled as the OSP $DIFF = \langle \Gamma_{DIFF}, \sqsupset \rangle$. We recall that, in spite of its name, a Concept Difference problem may have several solutions [6]. Note that a greatest solution for Γ_{DIFF} w.r.t. $\sqsupset$ is C itself.

Concept Abduction Concept Abduction is a straight adaptation of Propositional Abduction.

Definition 5. *[3] Let C, D, be two concepts in $\mathcal{DL}$, both C and D satisfiable. A* Concept Abduction Problem *(CAP) is finding a concept $H \in \mathcal{DL}$ such that $C \sqcap H \not\sqsubseteq \bot$, and $C \sqcap H \sqsubseteq D$.*

Every solution H of a CAP satisfies the formula

$$\Gamma_{ABD} = (C \sqcap X \not\sqsubseteq \bot) \wedge (C \sqcap X \sqsubseteq D)$$

The preference relation for evaluating solutions is subsumption-maximality, since less specific solutions should be preferred because they hypothesize the least. According to the proposed framework, we can model Subsumption-maximal Concept Abduction as $ABD = \langle \Gamma_{ABD}, \sqsupset \rangle$. Note that a greatest—*i.e.*, most specific—solution of ABD w.r.t. $\sqsupset$ is D, if $C \sqcap D$ is a satisfiable concept (if it is not, then ABD has no solution at all [3, Prop.1]).

3.2 Optimality by Fixpoint

Optimal solutions w.r.t. a preorder might be reached by iterating an inflationary operator. We now specialize the definition of inflationary operators and fixpoints to our setting.

Definition 6 (Inflationary operators and fixpoints). *Given an OSP $\mathbf{P} = \langle \Gamma, \prec \rangle$, we say that the operator $b_{\mathbf{P}} : SOL(\Gamma) \rightarrow SOL(\Gamma)$ (for **b**etter) is* inflationary *if for every $\mathcal{E} \in SOL(\Gamma)$, it holds that $b_{\mathbf{P}}(\mathcal{E}) \prec \mathcal{E}$ if $\mathcal{E}$ is not a least element of $\prec$, $b_{\mathbf{P}}(\mathcal{E}) = \mathcal{E}$ otherwise. In the latter case, we say that $\mathcal{E}$ is a fixpoint of $b_{\mathbf{P}}$.*

Intuitively, $b_{\mathbf{P}}(\mathcal{E})$ is a solution better than $\mathcal{E}$ w.r.t. $\prec$, if such a solution exists, otherwise a fixpoint has been reached, and such a fixpoint is a solution to $\mathbf{P}$. Being $b_{\mathbf{P}}$ inflationary, a fixpoint is always reached by the following induction: starting from a solution $\mathcal{E}$, let

$$\mathcal{E}_0 = \mathcal{E}$$
$$\mathcal{E}_{i+1} = b_{\mathbf{P}}(\mathcal{E}_i) \text{ for } i = 0, 1, 2, \ldots$$

Then, there exists a limit ordinal λ such that $\mathcal{E}_\lambda$ is a fixpoint of $b_{\mathbf{P}}$. For each of the previous non-standard reasoning services, we highlighted a greatest solution $\mathcal{E} \in SOL(\Gamma)$ which this iteration can start from. Obviously, when $\prec$ is well-founded (in particular, when $SOL(\Gamma)$ is finite) the fixpoint is reached in a finite number of steps. However, also when after n iterations $\mathcal{E}_n$ is not a fixpoint, $\mathcal{E}_n$ can be considered as an approximation of an optimal solution, since $\mathcal{E}_{i+1} \prec \mathcal{E}_i$ for every $i = 0, \ldots, n$.

We stress the fact that we are *not* proving here that every instance of Formula (1) can be solved by this method. For instance, deciding whether a formula of the form (1) is satisfiable is an open problem for $\mathcal{ALN}$, to the best of our knowledge. In this paper we address particular cases of (1), corresponding to known non-standard inferences, for which a solution is always known to exist.

It is interesting to observe that such particular cases are similar to *matching* problems [24], in that variables appear only on one side of each subsumption and non-subsumption statement.

4 Prototype Architecture

In the following, we present a prototype Logic Programming system implementing the approach to non-standard inference above recalled [17]. The system has been developed exploiting the integrated environment provided by SWI-Prolog[2] (Multi-threaded, 32 bits, Version 5.6.64) and follows the modular architecture depicted in Figure 1.

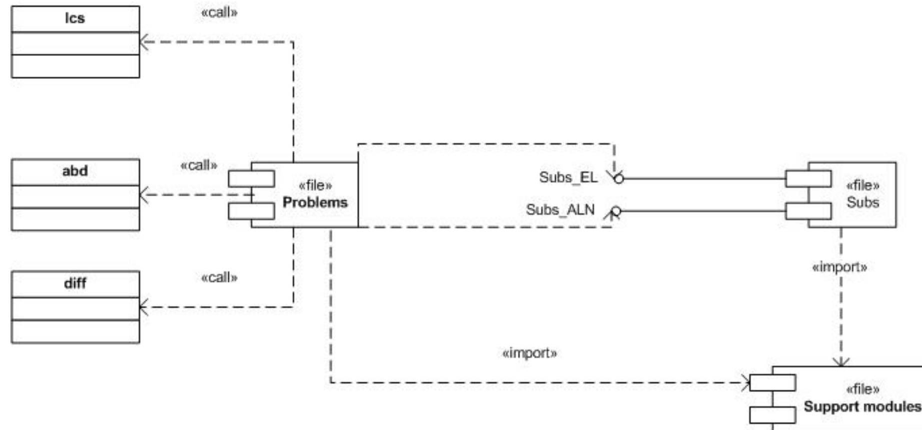

Fig. 1. Prototype architecture

Among possible languages for prototyping, we believe that Prolog combines in a unique fashion three distinguishable advantages:

1. its built-in unification easily parses concept terms, so the implementation of a parsing module can be skipped over
2. the double nature of programs-as-predicates allows us to isolate Subsumption both as a predicate and as a DL-specific module that can be changed according to the adopted Description Logic, while keeping fixed the incremental computation, and
3. the input/output duality of arguments in a predicate lets us *invert* Subsumption by (carefully) building a program that *decides* Subsumption; the tradeoff between direct implementation and generate-and-test inefficiency tends towards the former in a prototype.

The system design has been focused on proving main distinguishing features of our approach: the generality and the independence of the adopted DL (within a given subset) of non-standard inferences solving strategy. In particular, the prototype here presented is devoted to the solution of three different reasoning services, namely Least Common Subsumer, Concept Difference and Concept Abduction, in $\mathcal{EL}$ and $\mathcal{ALN}$.[3]

[2] http://www.swi-prolog.org/

[3] An executable version of the prototype is available at the following address: http://dl.dropbox.com/u/28260263/CILC2012exe.rar.

Coherently with the strategy introduced so far, the prototype searches for solutions for the system of the OSP modeling the non-standard inference need at hand. It is easy to notice that, therefore, the whole approach relies on the logic rules formalizing structural subsumption, which is at the basis of each formula to be solved.

The crucial role of subsumption affects the system architecture in Figure 1, whose main components are described below:

- **Subs** is the central component, which implements a recursive algorithm solving subsumption between concept descriptions; such a module is designed to provide one interface for each DL adopted to model the problem: the current prototype allows for solving subsumption in $\mathcal{EL}$ and $\mathcal{ALN}$.
- **Problems** is the component implementing OSP solving algorithms: the current prototype allows for solving Least Common Subsumer (**lcs**), Concept Difference (**diff**) and Concept Abduction (**abd**), but **Problems** may be extended to include further services. It is noteworthy how, depending on the DL adopted to model the problem, a different subsumption interface, either **subs_EL** or **subs_ALN**, is invoked.
- **Support Modules** includes clauses supporting the performance of subsumption and inferences included in Problems, but related to sorts of information processing outside the core solving algorithms, such as special purpose lists manipulation and concepts normalization.

5 Inverting Subsumption

In order to show the prototype implementing the solving strategy detailed so far, we refer to Least Common Subsumer computation, solved by the Prolog code fragment in the following, excerpted from **Problems** module. The formalization of Concept Difference and Concept Abduction problems is given in Appendix to enhance paper readability.

```
1  :-use_module('subs_el').
2  :-use_module('subs_aln').
3  :-use_module('support_modules').
4  :-use_module('normalization').

5  problem(lcs,C, D, Result, DL):- lcs(C, D, Result, DL).
6  problem(abd, C, D, Result, DL):- abd(C, D, Result, DL).
7  problem(diff, C, D, Result, DL):- diff(C, D, Result, DL).

8  lcs(C1, C2, LN, DL) :-
9     manage_concept(C1, C1N, DL),
10    manage_concept(C2, C2N, DL),
11    find_lcs(C1N, C2N, [top], L, DL),
12    normalization_top(L, LN).

13 find_lcs(C1, C2, L1, L3, DL) :-
14   decorate(L1, L2),
15   better_lcs(C1, C2, L1,L2, DL),  !,
16   find_lcs(C1, C2, L2, L3, DL).
17 find_lcs(C1, C2, L1, L1, _).

18 decorate(C, C0):- list(C, CL), select(some(R, D), CL, Rest),
19     decorate(D, DL),     append(Rest, [some(R, DL)], C0).
20 decorate(C, C0):-  list(C, CL), append(CL, [X0], C0).

21 better_lcs(C1, C2, L1, L2, el):-
22   subs_el(C1, L2),
23   subs_el(C2, L2),
```

```
24   not(subs_el(L1,  L2)).

25 better_lcs(C1, C2, L1, L2, aln):-
26    computeMaxAtLeast(C1,Max3),
27    computeMaxAtLeast(C2,Max4),
28    MaxL is max(Max3,Max4),
29    computeMaxAtMost(C1,Max1),
30    computeMaxAtMost(C2,Max2),
31    MaxM is max(Max1,Max2),
32    subs_aln(C1, L2, MaxL, MaxM),
33    subs_aln(C2, L2, MaxL, MaxM),
34    not(subs_aln(L1,  L2, MaxL, MaxM)).
```

We shortly recall that the shared strategy we proposed relies on the solution of an **Optimal Solution Problem** in which we search for solutions which are optimal w.r.t. a given preorder, by incrementally trying to *find* solutions which are *better* than the one at hand, till the best one is reached.

In order to compute the Least Common Subsumer LN of two concepts, C_1 and C_2 in a DL (see line 8), we need to incrementally construct a concept which subsumes both C_1 and C_2, and is optimal w.r.t. subsumption minimality (in fact, LN must be the most specific common subsumer of C_1 and C_2). To this aim, we start considering the trivial, subsumption maximal, solution, $L_1 = \top$ (line 11) and recursively try to *find* (lines 13–16) *better* common subsumers L_2 (line 15), by solving the system reported hereafter: $\{C_1 \sqsubseteq L_2; C_2 \sqsubseteq L_2; L_1 \not\sqsubseteq L_2\}$ (lines 22–24 or 32–34, depending on the adopted $\mathcal{DL}$). When no common subsumer L_n such that $L_{n-1} \not\sqsubseteq L_n$ exists, L_{n-1} is returned as best (Least) Common Subsumer (line 17).

The incremental construction of candidate better common subsumers $L2$ exploits a predicate, namely *decorate*, which makes the common subsumer at hand $L1$ more specific by appending fresh variables to it (lines 18–20). In fact, given that $L1$ is represented as a Prolog list, the predicate *append* performs a sort of conjunction of the appended variable to $L1$.

We notice that, even though different clauses are needed to check if $L2$ is *better* than $L1$ in $\mathcal{EL}$ (lines 21–24) and $\mathcal{ALN}$ (lines 25–34), such a distinction is only due to efficiency reasons: *subs_aln* needs two parameters more than *subs_el* and the adoption of a logic-independent unique *better_lcs* would force *subs_el* to work less efficiently with such two parameters, even though instantiated to anonymous variables. By the way, the reader can notice that the solving strategy underlying *better* is shared by both characterizations.

We observe also that all predicates invoked but not listed in the previously reported excerpt belong to one of the imported modules. In particular, *subs_el* and *sub_aln* modules provide the related logic-dependent subsumption programs, listed in Section 5.1. The other imported modules, *i.e.*, *support_modules* and *normalization*, include clauses crucial for the problem solution, but outside the core solving algorithms.

5.1 Subsumption

Both in $\mathcal{EL}$ and in $\mathcal{ALN}$, the subsumption algorithm takes as input concept descriptions written as conjunctions, formalized as Prolog lists. Given two concept descriptions $C1$ and $C2$ in a DL $\mathcal{DL}$, in order to prove whether $C2$ subsumes $C1$ (formally $C1 \sqsubseteq C2$), the algorithm recursively searches, for each member of the list related to $C2$, at least one

subsumed member in the list representing $C1$. In other words, the whole subsumption check mechanism reverts to a one-one comparison between list members (or, more appropriately, conjuncts).

With ground lists, the proposed subsumption predicate just returns boolean answers showing check results. Nevertheless, we notice that conjuncts in input concept descriptions may also include concept variables: when lists are not ground, subsumption is inverted to exhibit possible variables substitutions making subsumption between list members true. The mechanism exploits Prolog built-in unification.

The predicate making sumbsumption inversion possible is *my_member* (see calls in lines 15 and 18 in $\mathcal{EL}$, for example), whose code is included in *support_modules* and shown in the following:

```
1 mymember(X, L):-
2    ground(L), !,
3    member(X, L).

4 mymember(X, [X]).
5 mymember(X, [X|L]).
6 mymember(X, [_|L]) :-
7    mymember(X, L).
```

The reader can notice that *my_member* manages both the case of ground (lines 1–3) and non-ground (lines 4–7) lists. The former case reverts to the use of built-in predicate *member*, which lists all members of the ground list at hand. The latter case allows for avoiding premature variables unification in membership check.

As hinted before, the overall mechanism solving subsumption is shared by both implementations and is built on one-to-one comparison of list members, whose membership is checked with *my_member* predicate; such members may therefore be either ground elements or variables.

Clauses comparing single list members (lines 20–38 in $\mathcal{EL}$ and 1–32 in $\mathcal{ALN}$) exploit syntactical features of the DL at hand to either check subsumption between ground elements or unify variables to values making subsumption true.

In the following, we provide the complete Prolog code for the implementation of subsumption in $\mathcal{EL}$, to make the reader aware of the overall list comparison mechanism, shared also by the corresponding implementation in $\mathcal{ALN}$. For the sake of synthesis, we therefore provide only clauses related to the one-to-one comparison in the case of $\mathcal{ALN}$.

Subsumption in $\mathcal{EL}$ The Prolog program solving subsumption in $\mathcal{EL}$ is shown hereafter. We recall that the implemented algorithm exploits a structural characterization of subsumption, which makes the above mentioned inversion easy, despite the inherent inefficiency introduced.

We notice that, when input concepts include concept variables (or, according to the alphabet of our Prolog code, are not ground), the inversion could generate the same solution and unify a variable to the same value more than once. In order to prevent such an inefficiency and redundancy source, we manage a list, which we call BL (acronym for *Black List*), of variables substitutions already produced by *subs_el* clauses. Clauses in lines 20–29 succeed, in fact, when both structural subsumption holds and the involved

variable unification has not been adopted before (see line 22 and line 29). Whenever a *subsoneone* clause succeeds, the concept associated to the variable value causing the success is added to the list BL, which thus represents our progressive solution. In particular, in line 27, it is prevented the success with variable values producing concepts subsuming BL, which are useless w.r.t. the construction of our target solution.

```
1  subs_el(C1, C2):-
2      manage_concept(C1, C1L, el),
3      manage_concept(C2, C2L, el), !,
4      subsmanymany(C1L, C2L, []).

5  subsmanymany(L1, [], _) :-
6      not(ground(L1)) -> putTop(L1); true.
7  subsmanymany(L1, [C|L2], BL):-
8      ground(C) -> (
9       subsonemanyground(L1, C),
10      subsmanymany(L1,  L2, BL));
11          (subsonemany(L1, C,  BL,  BLF),
12           subsmanymany(L1, L2, BLF)).

13 subsonemany(L,A, BL, BLF):-
14     not(member(top, BL)),
15     mymember(B, L),
16     subsoneone(B, A, BL, BLF).

17 subsonemanyground(L,A):-
18     mymember(B, L),
19     subsoneoneground(B, A).

20 subsoneone(A, A, BL, BLF):-
21     literal(A),
22     not(member(A, BL)),
23     append([A], BL, BLF).
24 subsoneone(some(R,C1), some(R, C2N),  BL, BLF):-
25       subs_el(C1, C2),
26       normalization_top(C2, C2N),
27       not(subs_el(BL,some(R,C2N))),
28       append([some(R,C2N)],BL,BLF) .
29 subsoneone(Any, top,   [], [top]).

30 subsoneoneground(A, A):- literal(A).
31 subsoneoneground(some(R,C1), some(R, C2)):-
32       subs_el(C1, C2).
33 subsoneoneground(_, top).

34 putTop(L):-
35      term_variables(L,Vars),
36      allTop(Vars).

37 allTop(Vars) :-
38      Vars = [] -> true; (Vars = [top| Rest], allTop(Rest)).
```

Subsumption in $\mathcal{ALN}$

```
1 subsoneone(bottom, _ , _, _).
2 subsoneone(_, top, _, _).
3 subsoneone(A, A, _, _):- literal(A).
4 subsoneone(atleast(N,R), atleast(M, R), _, _):-
5     integer(N),
6      integer(M),
7      >=(N, M).
8 subsoneone(atleast(N,R), atleast(M, R),_, _):-
9      var(M),
10      geqpositive(N,M).
11 subsoneone(atleast(N,R), atleast(M, R), MaxL, _):-
```

```
12      var(N),
13      integer(M),
14      integer(MaxL),
15      leqBounded(M,N,MaxL).
16 subsoneone(atmost(N,R), atmost(M, R),_,  _):-
17      integer(N),
18      integer(M),
19      !,
20      =<(N,M).
21 subsoneone(atmost(N,R), atmost(M, R),_,  MaxM):-
22      var(M),
23      integer(MaxM),
24      leqBounded(N, M, MaxM).
25 subsoneone(atmost(N,R), atmost(M, R),_,  _):-
26      var(N),
27      geqpositive(M,N).
28 subsoneone(_, all(R, top), MaxL, MaxM):-
29      nonvar(R).
30 subsoneone(all(R,C1), all(R, C2), MaxL, MaxM):-
31  subsoneone(C1, C2, MaxL, MaxM).
32 subsoneone(atmost(0, R), all(R, C), _, _).
```

6 Querying the Prototype

In order to show our prototype working mode, we refer to the examples in the following, related to the three computational problems and the two DLs investigated in the paper:

1. $L = LCS(C_1, C_2)$, DL = $\mathcal{EL}$
   $C_1 = \exists R.(A \sqcap B) \sqcap \exists R.(C \sqcap D)$;
   $C_2 = \exists R.(A \sqcap C) \sqcap \exists R.(B \sqcap D)$
2. $L = LCS(C_1, C_2)$, DL = $\mathcal{EL}$
   $C_1 = \exists R.P \sqcap \exists R.(\exists R.Q)$;
   $C_2 = \exists R.(P \sqcap Q \sqcap \exists R.P \sqcap \exists R.Q)$
3. $L = LCS(C_1, C_2)$, DL = $\mathcal{ALN}$
   $C_1 = (\geqslant 3\, G) \sqcap (\leqslant 7\, S) \sqcap \forall R.(\leqslant 2\, M)$;
   $C_2 = (\geqslant 4\, G) \sqcap (\leqslant 3\, S) \sqcap \forall R.U$
4. $L = LCS(C_1, C_2)$, DL = $\mathcal{ALN}$
   $C_1 = (\geqslant 3\, G) \sqcap \forall R.(\leqslant 2\, M)$;
   $C_2 = (\leqslant 1\, G) \sqcap \forall R.(\leqslant 3\, M)$
5. $L = DIFF(C_1, C_2)$, DL = $\mathcal{EL}$
   $C_1 = A \sqcap B \sqcap \exists R.(C \sqcap D \sqcap \exists S.(H \sqcap J))$;
   $C_2 = A \sqcap B \sqcap \exists R.(\exists S.H)$
6. $L = DIFF(C_1, C_2)$, DL = $\mathcal{ALN}$
   $C_1 = A \sqcap \forall R.(B \sqcap (\leqslant 4\, S)) \sqcap (\leqslant 0\, T)$;
   $C_2 = A \sqcap \forall R.(\leqslant 4\, S) \sqcap \forall T.(D \sqcap \forall U.E \sqcap (\geqslant 2\, V))$
7. $L = DIFF(C_1, C_2)$, DL = $\mathcal{ALN}$
   $C_1 = \forall R.\bot$;
   $C_2 = \forall R.(\neg P \sqcap P_1)$
8. $L = ABD(C_1, C_2)$, DL = $\mathcal{EL}$
   $C_1 = \exists R.(\exists S.H)$;
   $C_2 = A \sqcap B \sqcap \exists R.(C \sqcap D \sqcap \exists S.(H \sqcap J))$
9. $L = ABD(C_1, C_2)$, DL = $\mathcal{ALN}$
   $C_1 = (\geqslant 2\, R) \sqcap \forall R.\neg A \sqcap B, \sqcap C$; $C_2 = B \sqcap (\geqslant 3\, R)$

10. $L = ABD(C_1, C_2)$, DL = $\mathcal{ALN}$
$C_1 = (\geqslant 3\, G) \sqcap \forall R.(\leqslant 2\, M)$; $C_2 = \bot$

In Table 2 we show the Prolog formalization and related results for the queries corresponding to the problems before. We notice that, when problems admit multiple solutions, like in the case of Concept Abduction and Concept Difference, the system stops searching for solutions when the first one is retrieved.

Table 2. Prolog queries

Query	Formalization	Result
1	*problem(lcs, [some(r,[a,b]), some(r,[c,d])], [some(r,[a,c]), some(r,[b,d])], L, el)*	*L = [some(r,[a]), some(r,[b]), some(r,[c]), some(r,[d])]* $L = \exists R.A \sqcap \exists R.B \sqcap \exists R.C \sqcap \exists R.D$
2	*problem(lcs,[some(r,p), some(r,some(r, q))], [some(r, [p,q,some(r,p), some(r,q)])], L, el)*	*L = [some(r, [p]), some(r, [some(r, [q])])]* $L = \exists R.P \sqcap \exists R.(\exists R.Q)$
3	*problem(lcs, [atleast(3,g), atmost(7,s), all(r,atmost(2,m))], [atleast(4,g), atmost(3,s), all(r,u)], L, aln)*	*L = [atleast(3, g), atmost(7, s), all(r, top)]* $L = (\geq 3\, G) \sqcap (\leq 7\, S) \sqcap \forall R.\top$
4	*problem(lcs, [atleast(3,g), all(r,atmost(2,m))], [atmost(1,g), all(r,atmost(3,m))], L, aln)*	*L = [all(r, top), all(r, atmost(3, m))]* $L = \forall R.\top \sqcap \forall R.(\leq 3\, M)$
5	*problem(diff, [a,b, some(r, [c,d, some(s,[h,j])])], [a,b, some(r,[some(s, [h])])], L, el)*	*L = [some(r, [c, d, some(s, [h, j])])]* $L = \exists R.(C \sqcap D \sqcap \exists S.(H \sqcap J))$
6	*problem(diff, [a, all(r, [b, atmost(4, s)]), atmost(0, t)],[a, all(r, atmost(4, s)), all(t, [d,all(u, e), atleast(2,v)])], L , aln)*	*L = [atmost(0, t), all(r, b)]* $L = (\leq 0\, T) \sqcap \forall R.B$
7	*problem(diff, [all(r, bottom)], [all(r, [neg(p), p1])], L, aln)*	*L = [all(r, neg(p1))]* $L = \forall R.\neg P_1$
8	*problem(abd, [some(r,[some(s, [h])])], [a,b, some(r, [c,d, some(s,[h,j])])], L, el)*	*L = [a,b, some(r, [c, d, some(s, [h, j])])]* $L = A \sqcap B \sqcap \exists R.(C \sqcap D \sqcap \exists S.(H \sqcap J))$
9	*problem(abd, [atleast(2,r),all(r,neg(a)), b, c],[b, atleast(3, r)],L , aln)*	*L = [atleast(3, r)]* $L = (\geq 3\, R)$
10	*problem(abd, [atleast(3,g), all(r,atmost(2,m))], [bottom], L, aln)*	*L = [atmost(2, g)]* $L = (\leqslant 2\, G)$

7 Discussion and Future Work

Motivated by the need to unify as much as possible the process of solving non-standard reasoning problems, we proposed a general framework dealing with several inferences according to a logic-independent strategy, to be further specialized to cope with the DL adopted to model the problem at hand.

The paper presents a modular Logic Programming prototype system demonstrating the feasibility of the proposed strategy for Least Common Subsumer, Concept Difference and Concept Abduction computation in $\mathcal{EL}$ and $\mathcal{ALN}$.

The extension of the approach to different DL sublanguages, and the implementation, for each investigated DL, of further non-standard reasoning services in the prototype is part of our future work, together with the improvement of system efficiency. In particular, by investigating on the queries execution, we noticed that the main source of complexity is due to the adoption of a generate-and-test strategy: all solutions need to be produced before discarding the not relevant ones. We therefore will investigate on how the program efficiency can be increased by designing a concurrent goal solution strategy, exploiting meta-programming.

Of course, the approach presented in this paper has some theoretical limitations. Namely, the use of an invertible logic program for Structural Subsumption limits this approach to DLs for which Structural Subsumption is complete. For more expressive DLs, the fixpoint mechanism could still be exploited, but implementing a different program inverting Subsumption. One possibility would be to implement the higher-order tableaux method presented by Colucci et al. [17]. However, it is known that such a general calculus does not always terminate for DLs whose expressiveness is equal or above $\mathcal{SHI}$ [25], and for less expressive DLs its termination is still to be assessed.

References

[1] McGuinness, D.L., Borgida, A.: Explaining subsumption in description logics. In: Proc. of IJCAI'95. (1995) 816–821

[2] Schlobach, S.: Explaining subsumption by optimal interpolation. In: Proc. of JELIA'2004. (2004) 413–425

[3] Di Noia, T., Di Sciascio, E., Donini, F.M., Mongiello, M.: Abductive matchmaking using description logics. In: Proc. of IJCAI 2003. (2003) 337–342

[4] Di Noia, T., Di Sciascio, E., Donini, F.M.: Semantic matchmaking as non-monotonic reasoning: A description logic approach. J. of Artificial Intell. Res. **29** (2007) 269–307

[5] Baader, F., Narendran, P.: Unification of concept terms in description logics. J. of Symbolic Computation **31** (2001) 277–305

[6] Teege, G.: Making the difference: A subtraction operation for description logics. In: Proc. of KR'94. (1994) 540–550

[7] Borgida, A., Walsh, T., Hirsh, H.: Towards measuring similarity in description logics. In: Proc. of DL 2005. (2005)

[8] Baader, F., Küsters, R., Molitor, R.: Rewriting concepts using terminologies. In: Proc. of KR 2000. (2000) 297–308

[9] Baader, F., Sertkaya, B., Turhan, A.Y.: Computing the least common subsumer w.r.t. a background terminology. J. of Appl. Log. **5**(3) (2007) 392–420

[10] Baader, F.: Least common subsumers and most specific concepts in a description logic with existential restrictions and terminological cycles. In: Proc. of IJCAI 2003. (2003) 319–324
[11] Baader, F., Sertkaya, B.: Usability issues in description logic knowledge base completion. In: ICFCA-2009. (2009) 1–21
[12] Konev, B., Walther, D., Wolter, F.: Forgetting and uniform interpolation in large-scale description logic terminologies. In: Proc. of IJCAI 2009. (2009)
[13] Lecue, F., Kotoulas, S., Aonghusa, P.M.: Capturing the pulse of cities: A robust stream data reasoning approach. Position paper, IBM Research, Smarter Cities Technology Centre, Dublin, Ireland (2011)
[14] Nikitina, N.: Uniform interpolation in general $\mathcal{EL}$ terminologies. Techreport, Institut AIFB, KIT, Karlsruhe (Mai 2011)
[15] Baader, F., Morawska, B.: Unification in the description logic $\mathcal{EL}$. Logical Methods in Computer Science **6**(3) (2010)
[16] Kazakov, Y., Krötzsch, M., Simancik, F.: Concurrent classification of $\mathcal{EL}$ ontologies. In: International Semantic Web Conference (1). (2011) 305–320
[17] Colucci, S., Di Noia, T., Di Sciascio, E., Donini, F.M., Ragone, A.: A unified framework for non-standard reasoning services in description logics. In: Proc. of ECAI 2010
[18] de Paiva, V.: Constructive description logics: what, why and how. In: Context Representation and Reasoning. (2006)
[19] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook – 2nd edition. Cambridge Univ. Press (2007)
[20] Baader, F., Turhan, A.Y.: Tboxes do not yield a compact representation of least common subsumers. In: Proc. of DL 2001. (2001)
[21] Borgida, A., Patel Schneider, P.F.: A semantics and complete algorithm for subsumption in the CLASSIC description logic. J. of Artificial Intell. Res. **1** (1994) 277–308
[22] Baader, F., Küsters, R., Molitor, R.: Computing least common subsumers in description logics with existential restrictions. In Dean, T., ed.: Proc. of IJCAI'99, Morgan Kaufmann (1999) 96–101
[23] Cohen, W., Borgida, A., Hirsh, H.: Computing least common subsumers in description logics. In: Proc. of AAAI'92, AAAI Press
[24] Baader, F., Küsters, R., Borgida, A., McGuinness, D.L.: Matching in description logics. J. of Log. and Comp. **9**(3) (1999) 411–447
[25] Wolter, F., Zakharyaschev, M.: Undecidability of the unification and admissibility problems for modal and description logics. ACM Trans. on Computational Logic **9**(4) (2008)

Appendix: Prolog Code for Optimal Solution Problems

Concept Abduction

```
abd(C, D, H, DL) :-
   manage_concept(C, CM, DL),
   manage_concept(D, DM, DL),
   find_abd(CM, DM, DM, H, DL), !.

find_abd(C, D, H1, H3, DL):-
    better_abd(C, D, H1, H2, DL), !,
    find_abd(C, D, H2, H3, DL).
find_abd(C, D, H, H, DL).

better_abd( C, D, H1, H2, el):-
    abd1(C, D, H2, el),
    subs_el(H1, H2),!,
    not(subs_el(H2, H1)).
```

```
better_abd( C, D, H1, H2, aln):-
    computeMaxAtLeast(C,Max3),
    computeMaxAtLeast(D,Max4),
    MaxL is max(Max3,Max4),
    computeMaxAtMost(C,Max1),
    computeMaxAtMost(D,Max2),
    MaxM is max(Max1,Max2),
    abd1(C, D, H2, MaxL, MaxM, aln),
    subs_aln(H1, H2, MaxL, MaxM),!,
    not(subs_aln(H2, H1,  MaxL, MaxM)).

abd1(C, D, H, el):-
   append(C, H, L),
    subs_el(L, D).
abd1(C, D, H, MaxL, MaxM, aln):-
    append(C, H, L),
    subs_aln(L, D, MaxL, MaxM).
```

Concept Difference

```
diff(C, D, X, aln):-
     manage_concept(C, CN, aln),
     manage_concept(D, DN, aln),
    computeMaxAtLeast(C,Max3),
    computeMaxAtLeast(D,Max4),
    MaxL is max(Max3,Max4),
    computeMaxAtMost(C,Max1),
    computeMaxAtMost(D,Max2),
    MaxM is max(Max1,Max2),
    subs_aln(CN, DN, MaxL, MaxM),
    find_diff(CN, DN, CN, X, aln), !.
diff(C, D, X, el):-
     subs_el(C, D),
    find_diff(C, D, C, X, el), !.
diff(C, D, X, DL):- fail.

find_diff(C, D, X1, X3, DL):-
    better_diff(C, D, X1, X2, DL),!,
    find_diff(C, D, X2, X3, DL).
find_diff(C, D, X, X, DL).

better_diff(C, D, X1, X2, aln):-
     computeMaxAtLeast(C,Max3),
     computeMaxAtLeast(D,Max4),
     MaxL is max(Max3,Max4),
     computeMaxAtMost(C,Max1),
     computeMaxAtMost(D,Max2),
     MaxM is max(Max1,Max2),
     diff1(C, D, X2, MaxL, MaxM, aln),
     subs_aln(X1, X2,MaxL, MaxM),
     not(subs_aln(X2, X1, MaxL, MaxM)).
better_diff(C, D, X1, X2, el):-
     diff1(C, D, X2, el),
     subs_el(X1, X2), !,
     not(subs_el(X2, X1)).

diff1(C, D, X, MaxL, MaxM, aln):-
    append(D, X, L),
    subs_aln(L, C, MaxL, MaxM),
    subs_aln(C, L, MaxL, MaxM).
diff1(C, D, X, el):-
    append(D, X, L),
    subs_el(L, C), !,
    subs_el(C, L).
```