Implementing Matching in $\mathcal{ALE}$—First Results

Sebastian Brandt*
Theoretical Computer Science,
TU Dresden, Germany
email: brandt@tcs.inf.tu-dresden.de

Abstract

Matching problems in Description Logics are theoretically well understood, with a variety of algorithms available for different DLs. Nevertheless, still no implementation of a general matching algorithm exists. The present paper presents an implementation of an existing matching algorithm for the DL $\mathcal{ALE}$ and shows first results on benchmarks w.r.t. randomly generated matching problems. The observed computation times show that the implementation performs well even on relatively large matching problems.

1 Motivation

Matching in Description Logics (DLs) has been first introduced by Borgida and McGuinness in the context of the CLASSIC system [9] as a means to filter out irrelevant aspects of large concept descriptions. It has also been mentioned that matching (as well as unification) can be used either to find redundancies in or to integrate knowledge bases [7, 10]. More recently, matching has been proposed to perform queries on knowledge bases, an application particularly interesting in combination with other non-standard inferences [11].

A matching problem (modulo equivalence) consists of a concept description C and a concept *pattern* D, i.e., a concept description with variables. Matching D against C means finding a substitution of variables in D by concept descriptions such that C is equivalent to the instantiated concept pattern D.

Matching algorithms have been developed for the DLs $\mathcal{ALN}$, $\mathcal{ALE}$, and their respective sublanguages [4, 3]. For $\mathcal{ALN}$ and its sublangages, algorithms could even be found for an extension of matching problems, namely matching under side conditions [1]. However, there exists no *implementation* of an algorithm providing matching in DLs as an explicit inference service. In the present paper, we present an implementation of an $\mathcal{ALE}$-matching algorithm as introduced in [3]. It has also been shown in the relevant paper that the algorithm is in EXPSPACE. As with other non-standard inferences, the question arises whether or not the actual run-time behavior of an implemented algorithm is as adverse as the theoretical upper bound suggests.

To cast light on this question, we have performed benchmarks w.r.t. randomly generated matching problems. As we shall see, in our case moderate optimization

*Supported by the DFG under grant BA 1122/4-3

strategies suffice to observe practicable run-times. The remainder of the present paper is structured as follows: after introducing relevant basic notions and definitions the existing $\mathcal{ALE}$-matching algorithm is discussed in Section 3. In Section 4 the ideas underlying our implementation will be presented while Section 5 shows the results of our benchmarks.

2 Preliminaries

Concept descriptions are inductively defined with the help of a set of *constructors*, starting with a set N_C of *concept names* and a set N_R of *role names*. For the sake of simplicity, we assume N_R to be the singleton $\{r\}$. However, all definitions and results can easily be generalized to arbitrary sets of role names. In this work, we consider the DL $\mathcal{ALE}$ which allows for the top concept ($\top$), bottom concept ($\bot$), conjunction ($C \sqcap D$), existential restrictions ($\exists r.C$), and value restrictions ($\forall r.C$). The semantics of $\mathcal{ALE}$-concept descriptions is defined in the usual model-theoretic way. For every concept description C the $\top$-*normal form* $C^\top$ of C is obtained by exhaustive application of the transformation rule $\forall r.\top \rightarrow \top$ to C.

In preparation to the following section we also need to introduce *concept patterns*. These are defined w.r.t. a finite set N_X of *concept variables* distinct from N_C. Concept patterns are an extension of concept descriptions in the sense that they allow for primitive concepts $A \in N_C$ *and* concept variables $X \in N_X$ as atomic constructors. The only restriction is that primitive negation may not be applied to concept variables. For every concept pattern D, a $\top$-*pattern of* D is obtained by syntactically replacing some variables in D by the top-concept $\top$.

One of the most important traditional inference services provided by DL systems is computing the subsumption hierarchy of concept descriptions. The concept description C is *subsumed* by the description D ($C \sqsubseteq D$) iff $C^\mathcal{I} \subseteq D^\mathcal{I}$ holds for all interpretations $\mathcal{I}$. The concept descriptions C and D are *equivalent* ($C \equiv D$) iff they subsume each other. Subsumption of $\mathcal{ALE}$-concept descriptions has been characterized by means of homomorphisms between so-called description trees [6] which are defined as follows.

Definition 1 *An $\mathcal{ALE}$-description tree is a tree of the form $\mathcal{G} = (N, E, n_0, \ell)$ where*

1. *N is a finite set of nodes;*
2. *$E \subseteq N \times \{\exists, \forall\} \times N_R \times N$ is a finite set of edges each labeled with a quantor and a role name;*
3. *n_0 is the root node of $\mathcal{G}$;*
4. *ℓ is a labeling function with $\ell(n) \subseteq \{\bot\} \cup N_C \cup \{\neg A \mid A \in N_C\} \cup N_X$ for all $n \in N$.*

Description trees correspond to syntax trees of concept descriptions (or concept patterns). It is therefore easy to see that concept descriptions can be translated into description trees and back (See [5] for a formal translation). By *tree*(C) we denote the description tree of the concept description (or concept pattern) C while *con*($\mathcal{G}$) denotes the concept description obtained from the tree $\mathcal{G}$. For every node n in the description tree *tree*(C) of C we denote by $C|_n$ the subdescription obtained by translating the subtree of *tree*(C) induced by n back into a concept description.

Definition 2 *A mapping $\varphi\colon N_H \to N_G$ from an $\mathcal{ALE}$-description tree $\mathcal{H} := (N_H, E_H, m_0, \ell_H)$ to an $\mathcal{ALE}$-description tree $\mathcal{G} := (N_G, E_G, n_0, \ell_G)$ is called* homomorphism *if and only if the following conditions hold:*

1. *$\varphi(m_0) = n_0$;*
2. *for all nodes $n \in N_H$ it holds that $\ell_H(n) \setminus N_X \subseteq \ell_G(\varphi(n))$ or $\bot \in \ell_G(\varphi(n))$;*
3. *For all edges $(n\, Qr\, m) \in E_H$, either $(\varphi(n)\, Qr\, \varphi(m)) \in E_G$, or $\varphi(n) = \varphi(m)$ and $\bot \in \ell_G(\varphi(n))$.*

It has been shown in [6] that $C \sqsubseteq D$ for two concept descriptions C and D iff there exists a homomorphism φ from $tree(D^\top)$ onto $tree(C)$. Note, however, that the above definition includes homomorphisms from a description tree representing a concept pattern onto one representing a concept description.

For the $\mathcal{ALE}$-matching algorithm we also need to introduce the least common subsumer of $\mathcal{ALE}$-concept descriptions.

Definition 3 (lcs) *Given $\mathcal{ALE}$-concept descriptions $C_1, \ldots, C_n$, the $\mathcal{ALE}$-concept description C is the* least common subsumer *(lcs) of $C_1, \ldots, C_n$ ($C = lcs\{C_1, \ldots, C_n\}$ for short) iff (i) $C_i \sqsubseteq C$ for all $1 \leq i \leq n$, and (ii) C is the least concept description with this property, i.e., if C' satisfies $C_i \sqsubseteq C'$ for all $1 \leq i \leq n$, then $C \sqsubseteq C'$.*

It has been shown in [6] that in the DL $\mathcal{ALE}$ the lcs of two or more concept descriptions always exists and is uniquely determined up to equivalence. Moreover, it can be computed in exponential time.

3 Matching in $\mathcal{ALE}$

In order to define matching problems we first need to introduce substitutions on concept patterns. A substitution σ is a mapping from N_X into the set of all $\mathcal{ALE}$-concept descriptions. Substitutions are extended to concept patterns by induction on the structure of the pattern, thus modifying only the occurrences of variables in the pattern. The notion of subsumption is extended to substitutions in the following way. A substitution σ is subsumed by a substitution τ ($\sigma \sqsubseteq \tau$) iff $\sigma(X) \sqsubseteq \tau(X)$ for all $X \in N_X$. With these preliminaries we can define matching problems.

Definition 4 *Let C be an $\mathcal{ALE}$-concept description and D be an $\mathcal{ALE}$-concept pattern. Then, $C \equiv^? D$ is a $\mathcal{ALE}$-matching problem. A substitution σ is a* matcher *iff $C \equiv \sigma(D)$. A set S of matchers to $C \equiv^? D$ is called* s-complete *iff for every matcher τ to $C \equiv^? D$ there exists an element $\sigma \in S$ with $\sigma \sqsubseteq \tau$.*

In general a solvable matching problem has several matchers. One way to restrict the attention to 'interesting' sets matchers is to compute s-complete sets of matchers as defined above. Figure 1 shows the relevant $\mathcal{ALE}$-matching algorithm originally presented in [2, 3]. It has been shown that it in fact computes s-complete sets of matchers, that the number of returned matchers is at most exponential, and that each matcher is of size at most exponential in the size of the matching problem.

In [3] it is also shown that the matching algorithm is in EXPSPACE. It is still open how 'tight' this upper bound is, and especially, if sets of s-complete matchers can also be computed in EXPTIME—currently the best lower bound for this computation problem.

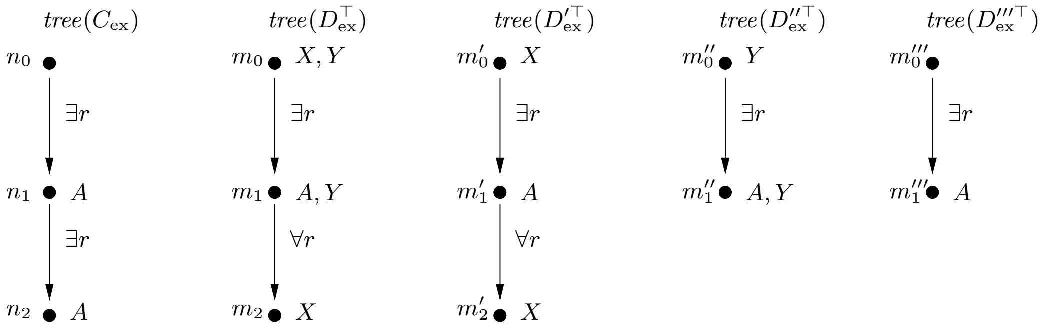
Input: $\mathcal{ALE}$-matching problem $C \equiv^? D$
Output: s-complete set $\mathcal{C}$ of matchers for $C \equiv^? D$

$\mathcal{C} := \emptyset$
For all $\top$-patterns D' of D do
  For all homomorphisms φ from $\mathcal{H} := \mathit{tree}(D'^{\top})$ into $\mathit{tree}(C)$
    Define σ by $\sigma(X) := \begin{cases} \mathit{lcs}\{C|_{\varphi(m)} \mid m \in N_{\mathcal{H}}, X \in \ell_{\mathcal{H}}(m)\} & \text{if } X \in \mathit{var}(D') \\ \top & \text{otherwise} \end{cases}$
    If $C \sqsupseteq \sigma(D)$ then $\mathcal{C} := \mathcal{C} \cup \{\sigma\}$

Figure 1: The $\mathcal{ALE}$-Matching Algorithm

Example 5 Let $N_C := \{A\}$ and $N_R := \{r\}$. Consider the matching problem $C_{\text{ex}} \equiv^? D_{\text{ex}}$ with $C_{\text{ex}} := \exists r.(A \sqcap \exists r.A)$ and $D_{\text{ex}} := X \sqcap Y \sqcap \exists r.(A \sqcap Y \sqcap \forall r.X)$. The relevant description trees are shown below:

In order to apply the matching algorithm shown in Figure 1 we have to start by computing all $\top$-patterns D'_{ex} of D_{ex}. Apart from D_{ex} itself, these are $Y \sqcap \exists r.(A \sqcap Y \sqcap \forall r.X) =: D'_{\text{ex}}$, $X \sqcap \exists r.(A \sqcap \forall r.\top) =: D''_{\text{ex}}$, and $\exists r.(A \sqcap \forall r.\top) =: D'''_{\text{ex}}$ The next step is to compute the respective $\top$-normal forms. It is easy to see that the $\top$-normal form of D_{ex} and D'_{ex} is equivalent to the original concepts. For D''_{ex} and D'''_{ex}, however, the value restriction $\forall r.\top$ is removed. The description trees of the relevant normalized concepts are shown below.



Because of the universal r-edge in $\mathit{tree}(D^{\top}_{\text{ex}})$ and $\mathit{tree}(D'^{\top}_{\text{ex}})$ which is missing in $\mathit{tree}(C_{\text{ex}})$ it is easy to see that no homomorphism exists from $\mathit{tree}(D^{\top}_{\text{ex}})$ or $\mathit{tree}(D'^{\top}_{\text{ex}})$ onto $\mathit{tree}(C_{\text{ex}})$. However, by mapping m''_0 onto n_0 and m''_1 onto n_1 we find a homomorphism φ from $\mathit{tree}(D''^{\top}_{\text{ex}})$ onto $\mathit{tree}(C_{\text{ex}})$. Hence, the next step is to construct a substitution σ according to the definition in Figure 1. Since X is no element of $\mathit{var}(D''_{\text{ex}})$ we obtain $\sigma(X) = \top$. Moreover, we find that Y occurs in m''_0 and m''_1. Hence, we have to compute the lcs of $C_{\text{ex}}|_{\varphi(m''_0)}$ and $C_{\text{ex}}|_{\varphi(m''_1)}$. Since $\varphi(m''_0) = n_0$ and $\varphi(m''_1) = n_1$ this means to compute the lcs of C_{ex} and $\exists r.A$. Thus, we obtain $\sigma(Y) = \exists r.A$. In the next step of the algorithm we find that $\sigma(D) = \exists r.A \sqcap \exists r.(A \sqcap \exists r.A)$ which is subsumed by the input concept C_{ex}. Thus, σ is added to the list $\mathcal{C}$ of solutions.

For the $\top$-pattern D'''_{ex} it is easy to see that the only homomorphism φ from $\mathit{tree}(D'''^{\top}_{\text{ex}})$ onto $\mathit{tree}(C_{\text{ex}})$ also maps m'''_0 onto n_0 and m'''_1 onto m_1. However, since $D'''^{\top}_{\text{ex}}$ contains no variables, we immediately obtain the substitution $\sigma' = \{X \mapsto \top, Y \mapsto \top\}$.

In this case, however, the final subsumption test does not hold, i.e., $Cex \not\sqsupseteq \sigma'(D)$.

As a result, $\sigma = \{X \mapsto \top, Y \mapsto \exists r.A\}$ is returned as the only matcher for the matching problem $C_{\text{ex}} \equiv^? D_{\text{ex}}$.

4 Implementation

Considering the matching algorithm in Figure 1 we can identify three major tasks to be solved by an implementation. Firstly, all $\top$-patterns D' of the input pattern D must be generated; secondly, all homomorphisms φ from $tree(D')$ onto $tree(C)$ must be found; and thirdly, for every variable X we must compute the lcs of all subconcepts $C|_{\varphi(m)}$ for which X occurs at position m in $tree(D'^{\top})$.

The first task regards only the input concept pattern and requires only some simple syntactical replacements. Even the computation of the $\top$-normal form $D'^{\top}$ of a $\top$-pattern D' can be done in a straightforward way in polynomial time. As (even optimized) implementations of the lcs algorithm for $\mathcal{ALE}$ exist [8] the third task is simple as soon as D' and φ are determined. The final subsumption test $C \sqsupseteq \sigma(D)$ can also be carried out by a standard reasoner, such as FaCT [13] or Racer [12].

The crucial task, however, is the second one. An obvious approach to constructing homomorphisms between two description trees is the usual top-down strategy known from lcs algorithms. Starting at the root nodes of the source and the destination tree in question, one could test for all pairs of edges respecting Condition 3 whether or not a homomorphism exists between the subtrees induced by the endpoints of these edges. Recursively descending in such a way, all homomorphisms between source and destination tree could be computed. The problem with this approach is that subproblems may be solved several times over—for instance if two homomorphisms are equal w.r.t. some subtrees of the original description tree.

To overcome this problem, we have chosen a dynamic-programming strategy to compose homomorphisms in a bottom-up fashion, thereby storing and re-using sets of admissible destination nodes for every source node. As a consequence, only polynomially many subproblems have to be solved for the computation of one homomorphism. The dynamic-programming approach, however, suggests a more sophisticated data structures for the representation of description trees. It proved expedient not to choose an algebraic data structure (as used in the lcs implementations), but to represent a description tree by a set of arrays indexed either by the nodes of the tree, by the role names occurring in the edge labels, or by the occurring variable names. As a result, all aspects important for the computation of homomorphisms can be retrieved instantly.

In our implementation, the composition of a homomorphism is done in two steps. In the first step—the actual bottom-up computation—a set of admissible destination nodes is computed for every node of the source description tree. The results are then used in the second one to compute the actual homomorphisms.

The crucial part in the first step is to determine whether or not a certain node is an admissible destination node. This part is shown in further detail in Figure 2. The idea is to test for stricter conditions than Definition 1 suggests in order to detect pairs of nodes which cannot be part of a homomorphism as soon as possible. For instance, according to Definition 1, a leaf labeled with $\bot$ is always an admissible destination

Input: description trees $\mathcal{G}_s =: (N_s, E_s, n_{s0}, \ell_s), \mathcal{G}_d =: (N_d, E_d, n_{d0}, \ell_d)$,
$n_s \in N_s, n_d \in N_d$

Output: n_d admissible destination of n_s? True iff:

- $\perp \in \ell_d(n_d)$ and
  - $depth(n_s) > depth(n_d)$ or
  - $depth(n_s) = depth(n_d)$ and either $n_s = n_{s0}$ and $n_s = n_{s0}$ or both nodes are successors w.r.t. the same quantor and role name
- $\perp \notin \ell_d(n_d)$
  - $depth(n_s) = depth(n_d)$
  - for every successor n'_s of n_s there exists at least one successor n'_d of n_d as admissible destination for n'_s.

Figure 2: Test for admissible destination nodes

node. However, if its depth exceeds that of the source node then every mapping containing this pair at some node on the path from the root to the source node violates Condition 4. Note that in the second case shown in Figure 2 the test for the successor n'_s only ends in a recursive call if n'_s has never been considered beforehand. Note also that the dynamic programming strategy implies that no backtracking is necessary.

In comparison to the theoretical algorithm the implemented one contains some mentionable optimizations:

- *Preprocessing*
  The input concept pattern and concept description is simplified to keep the relevant description trees as small as possible.
- *Necessary conditions*
  Let $\top(D)$ and $\perp(D)$ denote the concept obtained from the pattern D by replacing all variables in D by $\top$ and $\perp$, respectively. If $C \not\sqsubseteq \top(D)$ or $\perp(D) \not\sqsubseteq C$ then the matching problem $C \equiv^? D$ has no solution.
- $\top$*-patterns*
  In many cases it is not necessary to generate all top-patterns D' of D. This is only promising when replacing variables by $\top$ leads to a removal of subterms in the $\top$-normal form $D'^\top$ and hence to a removal of edges in the relevant description tree $tree(D'^\top)$. Moreover, if one $\top$-pattern $D'^\top$ does admit of a homomorphism then any specialization of D' does also, leading only to a solution not minimal w.r.t. $\sqsubseteq$.

In the following section shows some first performance tests for the implemented algorithms with the optimizations discussed above.

5 Benchmarks

An obvious approach to benchmarking our implementation of $\mathcal{ALE}$-matching is to test the performance on randomly generated matching problems. Nevertheless, if C and D are generated independently of each other then it is unlikely that a matcher for

$C \equiv^? D$ exists. In particular, in the second optimization (necessary conditions) is likely to solve such matching problems without even invoking the actual matching algorithm.

To overcome this difficulty, we randomly generate a concept C and then construct a concept pattern D from C by randomly replacing subconcepts of C by variables. Note that matching problems obtained in this way are not necessarily solvable because of multiple occurrences of variables. As a simple example, consider $C := \exists r.A \sqcap \forall r.B$ and $D := \exists r.X \sqcap \forall r.X$. The matching problem $C \equiv^? D$ has no solution. As a consequence, the second optimization is not reflected in the results.

Our benchmarks were taken on a standard PC with one 1.7GHz Pentium-4 processor and 512MB of memory. A total of 1200 matching problems (in 10 groups, using different parameters for the random generation) was examined. Taking overall averages, the concept description C had an average size of 518 with a maximum of 992, and the concept pattern D had size 185 with a maximum of 772. The matching algorithm on average took 1.2 seconds to solve the problem, the observed maximum was 58.2 seconds.

6 Conclusion

In the present paper we have presented first experiences with an implementation of the $\mathcal{ALE}$-matching algorithm as proposed by Baader and Küsters [3]. The algorithm is based on a tree representation of the involved concept description and concept pattern. The main problem for the implementation is posed by that step of the algorithm in which all homomorphisms between the relevant description trees must be generated. Here we have chosen a dynamic programming approach which avoids solving identical subproblems several times. In addition to that, the implementation includes some straightforward optimizations aimed at identifying cases which have no solution as soon as possible.

The benchmarks have shown that despite the high theoretical upper bound currently known for the $\mathcal{ALE}$-matching algorithm the implementation performs well even on relatively large randomly generated concepts.

Obviously, our next step is to confirm our findings by further testing. Firstly, a greater variety of randomly generated matching problems could be considered. Secondly, if available, matching problems resulting from realistic applications might give further insight into the practical benefit of our implementation.

In case the current implementation performs well under the above circumstances, the next step could be an extension to matching under side conditions.

References

[1] F. Baader, S. Brandt, and R. Küsters. Matching under side conditions in description logics. In *Proc. of IJCAI'01*, pages 213–218, Seattle, Washington, 2001. Morgan Kaufmann.

[2] F. Baader and R. Küsters. Matching in Description Logics with Existential Restrictions. In *Proc. of DL 1999*, number 22 in CEUR-WS, Sweden, 1999.

[3] F. Baader and R. Küsters. Matching in description logics with existential restrictions. In *Proc. of KR2000*, pages 261–272, Breckenridge, CO, 2000. Morgan Kaufmann Publishers.

[4] F. Baader, R. Küsters, A. Borgida, and D. McGuinness. Matching in Description Logics. *Journal of Logic and Computation*, 9(3):411–447, 1999.

[5] F. Baader, R. Küsters, and R. Molitor. Computing least common subsumers in description logics with existential restrictions. LTCS-Report LTCS-98-09, LuFG Theoretical Computer Science, RWTH Aachen, Germany, 1998. See http://www-lti.informatik.rwth-aachen.de/Forschung/Papers.html.

[6] F. Baader, R. Küsters, and R. Molitor. Computing Least Common Subsumers in Description Logics with Existential Restrictions. In *Proc. of IJCAI'99*, pages 96–101, Stockholm, Sweden, 1999. Morgan Kaufmann Publishers.

[7] F. Baader and P. Narendran. Unification of concept terms in description logics. In *Proc. of ECAI-1998*, pages 331–335, Brighton, UK, 1998. John Wiley & Sons Ltd.

[8] F. Baader and A.-Y. Turhan. On the problem of computing small representations of least common subsumers. In *Proc. of KI 2002*, Lecture Notes in Artificial Intelligence, Aachen, Germany, 2002. Springer–Verlag.

[9] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A Structural Data Model for Objects. In *Proc. of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 58–67. ACM Press, 1989.

[10] A. Borgida and R. Küsters. What's not in a name: Some Properties of a Purely Structural Approach to Integrating Large DL Knowledge Bases. In *Proc. of DL2000*, number 33 in CEUR-WS, Aachen, Germany, 2000. RWTH Aachen.

[11] S. Brandt and A.-Y. Turhan. Using non-standard inferences in description logics—what does it buy me? In *Proc. of KIDLWS'01*, number 44 in CEUR-WS, Vienna, Austria, September 2001. RWTH Aachen.

[12] Volker Haarslev and Ralf Möller. RACER system description. *Lecture Notes in Computer Science*, 2083:701–??, 2001.

[13] I. Horrocks. The FaCT system. In *Proc. of Tableaux'98*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 307–312, Berlin, 1998. Springer-Verlag.