Using lazy unfolding for the computation of least common subsumers*

Anni-Yasmin Turhan* and Ralf Molitor**

*RWTH Aachen
LuFG Theoretical Computer Science
52074 Aachen, Germany
turhan@cs.rwth-aachen.de

**Swiss Life
IT Research and Development Group
8022 Zürich, Switzerland
ralf.molitor@swisslife.ch

Abstract

For description logics with existential restrictions, the size of the least common subsumer (lcs) of concept descriptions may grow exponentially in the size of the concept descriptions. To reduce the size of the output descriptions and the run-time of the lcs algorithm we present an optimized algorithm for computing the lcs in $\mathcal{ALE}$ using lazy unfolding. A first evaluation of the performance of the naive algorithm in comparison to the performance of the algorithm using lazy unfolding indicates a performance gain for both concept sizes as well as run-times.

1 Motivation

In our application of chemical process engineering we support the construction of description logic (DL) knowledge bases in a bottom-up fashion: instead of directly defining a new concept, the knowledge engineer introduces several typical examples as objects (individuals), which are then automatically generalized into a concept description by the system. This description is then offered to the knowledge engineer as a possible candidate for a definition of a concept.

The task of automatically computing such a concept description can be achieved by using two non-standard inferences: (1) computing the most specific concept (msc) of each of the introduced objects, and (2) computing the least common subsumer (lcs) of these concepts. The *most specific concept* of an object o (the *least common subsumer* of concept descriptions $C_1, \ldots, C_n$) is the most specific concept description C expressible in the given DL language that has o as an instance (that subsumes $C_1, \ldots, C_n$). Algorithms for computing the msc are given in [1; 9].

Another application for the lcs is the *structuring of DL knowledge bases*. DL knowledge bases encountered in applications often have a rather flat concept hierarchy. Deeper hierarchies are more convenient for browsing the knowledge base, and make searching more efficient. Given a concept C with direct sub-concepts $\{C_1, \ldots, C_n\}$, one could use least common subsumer of selected subsets to insert additional layers between C and its sub-concepts $C_1, \ldots, C_n$. Computing the hierarchy of all least common subsumer of subsets of $\{C_1, \ldots, C_n\}$ the system can support the knowledge engineer in choosing an appropriate concept description.

The algorithm for computing the lcs of a set of $\mathcal{ALE}$-concept descriptions as developed in [6] works on concept descriptions unfolded w.r.t. the TBox. In the worst case this algorithm may yield descriptions exponentially large in the size of the unfolded input descriptions (also

*This work was partially supported by the Deutsche Forschungsgemeinschaft, DFG Project BA 1122/4-1.

discussed in [6]). This inherent complexity is an obstacle for practical applications of the lcs. To employ the lcs in practical applications there are several requirements to fulfill. Besides correctness, the runtime of a lcs computation should be low and furthermore the size of the output concept descriptions must be small in order to be readable and comprehensible by a human reader. The latter is important for most non-standard inferences (e.g. msc), because in contrast to standard inferences (such as subsumption, which are decision procedures) they return concept descriptions. These descriptions are then examined and assessed whether they should be added to the knowledge base by the knowledge engineer. To meet all these requirements we devise an optimized algorithm which computes the lcs, but avoids complete unfolding by lazy unfolding. *Lazy unfolding* is a standard optimization technique for terminological reasoning [5; 7]. It unfolds (sub-)concept descriptions only if examination of that part of the concept description is necessary. Thus, using lazy unfolding for the computation of the lcs should not only result in better average case run-times, moreover it should yield smaller concept descriptions.

A well known approach to reduce the concept size of a result concept descriptions is to compute (minimal) rewritings of the lcs concept description in a subsequent step. Intuitively, a *minimal rewriting* of a concept description C is the concept description with the smallest concept size equivalent to C w.r.t. the underlying TBox. A heuristic algorithm for computing small (but not always minimal) rewritings is given in [2].

In the mentioned applications of the lcs it is necessary to avoid the computation of *all* (in worst case exponentially many) least common subsumers of all possible subsets of a selected set of concept descriptions. To filter out some of the subsets we employ the *attribute exploration algorithm* [3], which computes a concept lattice isomorphic to the subsumption hierarchy of all least common subsumers of all subsets of $\{C_1, \ldots, C_n\}$. Only least common subsumers from that concept lattice are computed to obtain candidates for new concepts to be added to the knowledge base.

The rest of the paper is structured as follows. In the next section we outline the basic lcs algorithm and discuss the lcs algorithm using lazy-unfolding for the description logic $\mathcal{ALE}$. Then we give some details about our implementations of both algorithms and present an evaluation of them w.r.t. to run-time and concept size.

2 Computing the lcs for $\mathcal{ALE}$

The description logic $\mathcal{ALE}$ provides conjunction, existential restriction ($\exists\, r.\, C$), value restrictions ($\forall\, r.\, C$) and atomic negation in concept descriptions. Only unfoldable TBoxes are appropriate for the lcs, so the set of concept definitions occurring in a TBox have to be acyclic and concept names may appear only once on the left-hand side of a definition. The set of all roles appearing in the TBox is denoted N_R and the set of all primitive concepts (i.e., concepts without a definition in the TBox) is denoted N_C.

Definition 1 (Least Common Subsumer) *Let $C_1, \ldots, C_n$ be concept descriptions in a DL $\mathcal{L}$. The $\mathcal{L}$-concept description C is a* least common subsumer (lcs) *of $C_1, \ldots, C_n$ in $\mathcal{L}$ iff*

1. *$C_i \sqsubseteq C$ for all $1 \leq i \leq n$, and*
2. *C is the least concept description with this property, i.e., if D is a concept description satisfying $C_i \sqsubseteq D$ for all $1 \leq i \leq n$, then $C \sqsubseteq D$.*

The lcs does always exist for $\mathcal{ALE}$-concept descriptions. If a lcs exists for a DL $\mathcal{L}$, then it is unique up to equivalence. This justifies to refer to *the* lcs of $C_1, \ldots, C_n$ in $\mathcal{L}$.

Before turning to the lcs algorithm we need the notion of concept size and role-level. The *concept size* $|C|$ of a concept description C is increased by 1 for each occurrence of a role name or a concept name in C (with $|\top| = |\bot| = 0$). A *role-level* of a (sub-)concept description

refers to the number of exists or value restrictions it is nested in. If a concept description has role-level 0 it is on *top-level*. For example the concept description $A \sqcap (\exists\, r.\, (\forall\, r.\, (B \sqcap C)))$ has concept size 5 and the role-level of sub-concept description $(B \sqcap C)$ in this description is 2, while A is on top-level.

We outline the lcs algorithm as given in [6]. It operates on description trees a representation of concept descriptions which are unfolded w.r.t. the TBox. The basic lcs algorithm is given as a binary operation, since the n-ary lcs can be reduced to the binary operation. So if a lcs of n (with $2 \leq n$) concept descriptions is computed, $n-1$ successive lcs computations are made. The basic lcs algorithm consists of the following steps:

1. unfold all input concept descriptions
2. normalize all unfolded descriptions
3. compute the product of the normalized concept descriptions.

Thus, in order to compute the lcs of a set of concept descriptions $C_1, \ldots, C_n$ defined in a TBox $\mathcal{T}$, the concept descriptions are completely unfolded w.r.t. $\mathcal{T}$. To unfold a concept description completely all defined concepts appearing in the concept description are recursively replaced by their definitions in the TBox, until only primitive concepts remain in the description. The process of unfolding a description may already cause an exponential blow-up of the concept description, see [11]. In the second step of the lcs algorithm the $\mathcal{ALE}$-normal form is computed. To this end the $\mathcal{ALE}$-normalization rules remove concept descriptions equivalent to $\top$, replace inconsistent concept descriptions by $\bot$, join value restrictions, and propagate value restrictions into existential restrictions on all role-levels. This last step of the normalization is yet another source of an exponential blow-up (see [6; 4] for details). The third step computes the product of two description trees, where each tree corresponds to a concept description. Each of the three steps of the lcs algorithm traverses the whole structure of the concept description recursively. We illustrate this basic algorithm and its drawbacks by means of the following example.

Example 2 (lcs) *Given the following TBox $\mathcal{T}$ with*

$$\begin{aligned}\mathcal{T} = \{\ & C_1 := A_1 \sqcap \exists\, r.\, D_1, && C_2 := A_2 \sqcap \exists\, r.\, D_1,\\ & D_1 := (\forall\, s.\, B_1) \sqcap (\exists\, s.\, D_2) \sqcap (\exists\, s.\, D_3),\\ & D_2 := B_2 \sqcap B_3, && D_3 := B_3 \sqcap B_4\ \}\end{aligned}$$

we compute the lcs of C_1 and C_2. After the first and second step – unfolding and the application of the $\mathcal{ALE}$-normalization rules – we have

$$\begin{aligned}C_i = A_i \sqcap (\exists\, r.\, (\forall\, s.\, & B_1 \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_2 \sqcap B_3) \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_3 \sqcap B_4))\end{aligned}$$

for $i \in \{1, 2\}$. Then in the third step the algorithm determines the concept names appearing on the top-level of the lcs – in this case none ($\{A_1\} \cap \{A_2\} = \emptyset$). Then the algorithm makes a recursive call to compute the lcs of the r-successors of C_1 and C_2, which in turn makes recursive calls for the pair of value restrictions and for all four pairs of exists restrictions for the s-successors. The algorithm yields

$$\begin{aligned}\mathsf{LCS}(C_1, C_2) = \exists\, r.\, (\forall\, s.\, & B_1 \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_2 \sqcap B_3) \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_3) \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_3) \sqcap\\ & \exists\, s.\, (B_1 \sqcap B_3 \sqcap B_4))\end{aligned} \qquad \textit{with } |\mathsf{LCS}(C_1, C_2)| = 17.$$

So, the result is computed by three recursive traversals of the input descriptions and its size results from the actually unnecessary unfolding of D_1. Even if the redundant second and third existential restriction are eliminated from the result, the concept size of the returned lcs concept description is still quite big in comparison to the equivalent and obviously smaller lcs concept description $\exists\, r.\, D_1$.

In general the size of the lcs of two $\mathcal{ALE}$-concept descriptions may be exponential in the size of the (unfolded) input descriptions in the worst case, see [6]. To reduce the size of the lcs concept descriptions in the average case when computing the lcs w.r.t. a non-empty TBox, we now introduce an algorithm for the lcs using lazy unfolding.

2.1 Computing the lcs using lazy unfolding

The idea of lazy unfolding is to replace a part of a description only if examination of that part is necessary. Lazy unfolding unfolds concept names appearing on top-level of a concept description. It leaves defined concept names on deeper role-levels unchanged. In the following let C be an $\mathcal{ALE}$-concept description and let r be a role name.

Definition 3 ($\forall$-Normal form) *An $\mathcal{ALE}$-concept description C is in $\forall$-normalform iff every conjunction in C contains at most one value restriction for each role name $r \in N_R$.*

The algorithm for computing the lcs with lazy unfolding as shown in Figure 1 is based on the $\forall$-normalform and the following sets:

- $prim(C)$ ($\neg prim(C)$) denotes the set of all (negated) primitive names occurring on the top-level of C.
- $val_r(C)$ denotes the concept description occurring in the unique value restriction on the top-level of the $\forall$-normal form of C, where $val_r(C) := \top$ if there is no such value restriction.
- $exr_r(C)$ denotes the set $\{C_1, \ldots, C_n\}$ of concept descriptions occurring in existential restrictions of the form $\exists r.C_i$ on the top-level of C.

The algorithm as given in Figure 1 uses the function $\mathsf{lazy-unfold}()$, which unfolds the top-level of its input concept description w.r.t. the $\mathcal{ALE}$-TBox $\mathcal{T}$. The $\mathsf{LCS}_{|u}$ algorithm first tests on each recursion, if an input concept is equivalent to $\top$ or $\bot$, in this case the lcs concept can immediately be returned. Next, the top-level of the input concepts is unfolded and the auxiliary sets and concept descriptions are computed. The returned lcs concept description is a conjunction of four components:

1. a conjunction of all positive concept names appearing on top-level C' and D',
2. a conjunction of all negative concept names appearing on top-level C' and D',
3. a conjunction of value restrictions derived from recursive $\mathsf{LCS}_{|u}$ calls for each role which has a value restriction on top-level of C' and D',
4. a conjunction of existential restrictions derived from recursive $\mathsf{LCS}_{|u}$ calls. Where a call is made for each pair of existential restrictions (in combination with the appropriate value restrictions) for all roles appearing on top-level of C' and D'.

In contrast to the three independent recursions in the basic algorithm the $\mathsf{LCS}_{|u}$ algorithm traverses the structure of the concept description recursively only once. The three steps of the basic algorithm are now interwoven on each role-level, e.g., the propagation of the

Input: Two $\mathcal{ALE}$-concept descriptions C, D and an $\mathcal{ALE}$-TBox $\mathcal{T}$

Algorithm: $\mathsf{LCS}_{\mathsf{lu}}(C, D)$

if $C \sqsubseteq_{\mathcal{T}} D$ **then** $\mathsf{LCS}_{\mathsf{lu}}(C, D) = D$
if $D \sqsubseteq_{\mathcal{T}} C$ **then** $\mathsf{LCS}_{\mathsf{lu}}(C, D) = C$
else

$C' = \mathsf{lazy-unfold}(C, \mathcal{T})$,
$prim(C') = \{P_1, \ldots, P_n\}$,
$\neg prim(C') = \{Q_1, \ldots, Q_n\}$,
for all $r \in N_R$:
$val_r(C') = C_0$,
$exr_r(C') = \{C_1, \ldots, C_n\}$,
end for

$D' = \mathsf{lazy-unfold}(D, \mathcal{T})$,
$prim(D') = \{P_1, \ldots, P_n\}$,
$\neg prim(D') = \{Q_1, \ldots, Q_n\}$,
for all $r \in N_R$:
$val_r(D') = D_0$,
$exr_r(D') = \{D_1, \ldots, D_n\}$,
end for

$$\mathsf{LCS}_{\mathsf{lu}}(C, D) = \Big(\bigsqcap_{P \in prim(C') \cap prim(D')} P \quad \sqcap \bigsqcap_{P \in \neg prim(C') \cap \neg prim(D')} \neg P \quad \sqcap \bigsqcap_{r \in N_R} \big(\forall r.\, \mathsf{LCS}_{\mathsf{lu}}\, (val_r(C'), val_r(D')) \big) \quad \sqcap \bigsqcap_{r \in N_R} \Big(\bigsqcap_{\substack{C_i \in exr_r(C') \\ D_j \in exr_r(D')}} \exists r.\, \mathsf{LCS}_{\mathsf{lu}}\, (C_i \sqcap val_r(C'),\ D_j \sqcap val_r(D')) \Big) \Big)$$

Figure 1: The lcs algorithm $\mathsf{LCS}_{\mathsf{lu}}$ for $\mathcal{ALE}$ using lazy unfolding.

value restrictions is here realized role-level-wise by including the conjunct $val_r(C')$ ($val_r(D')$ respectively) in the recursive calls for the existential restrictions.

Lazy unfolding does not only save time (and storage) by avoiding complete unfolding of a concept description, but it may also require lesser recursion depth for computation of the lcs. If names of concepts defined in the TBox appear in the input descriptions on the same role-level, these names may be directly used in the lcs concept description without unfolding them. In this case lazy unfolding reduces the size of the descriptions to be handled by the algorithm and thereby the size of the resulting concept description. Let us return to Example 2 to illustrate this effect.

Example 4 (lcs using lazy unfolding) *Assume we want to compute the same lcs as before w.r.t. the TBox $\mathcal{T}$ and apply the $\mathsf{LCS}_{\mathsf{lu}}(C_1, C_2)$ algorithm. In the first step none of the two conditions hold and the algorithm calls $\mathsf{lazy-unfold}(C_1, C_2)$, but there is no defined concept to replace on top-level. Then the algorithm calls $\mathsf{LCS}_{\mathsf{lu}}$ recursively for the existential restriction. This call directly yields D_1 by the comparison at the beginning of the procedure. The returned concept description is $\mathsf{LCS}_{\mathsf{lu}}(C_1, C_2) = \exists r.\, D_1$, with $|\mathsf{LCS}_{\mathsf{lu}}(C_1, C_2)| = 2$.*

So comparing Example 2 to the result obtained here, it shows that $\mathsf{LCS}_{\mathsf{lu}}$ needs less recursions with lesser recursion depth and furthermore the concept description returned by $\mathsf{LCS}_{\mathsf{lu}}$ is much smaller.

In general the application of lazy unfolding is a benefit in most cases of computing a lcs w.r.t. a TBox, but there may of course still be combinations of input concept descriptions where an exponential growth of the lcs concept description is unavoidable.

Our lcs implementations actually support the DL $\mathcal{ALE}f$, which is $\mathcal{ALE}$ extended by features (i.e., functional roles). Features may be used in existential or value restrictions. The extension to features of the lcs algorithms is straightforward. Features are treated like roles, with the exception that there may only be a single feature successor per role-level or feature-level, respectively. This is guaranteed by combining *all* existential restrictions for a feature f (together with the value restrictions for f) in one single conjunction for each input description. These two conjunctions are then used as arguments for the recursive call.

3 Implementations of the lcs

We have implemented both, the basic and the lazy unfolding lcs algorithms. Both implementations are done in Lisp and use the FaCT system [8] for the classification of the TBox and for the subsumption tests. Both implement a binary lcs function wrapped by a function that makes successive calls for the binary LCS function.

The "old lcs" is a straightforward implementation of the fundamental algorithm presented in Section 2 and discussed in [6]. The old implementation further includes some of the methods needed in our application framework mentioned in Section 1. It contains the implementation of the heuristic rewriting algorithm for computing small (but not always minimal) rewritings of $\mathcal{ALE}$-concept descriptions mentioned earlier (see [2]), which we use in our evaluation.

The "new lcs" implements the algorithm introduced in Section 2.1. It is also a straightforward implementation and does not use special encoding tricks to improve the performance. In contrast to the old lcs implementation the new lcs may be coupled to different DL reasoner.

3.1 A first evaluation of the implementations

To compare the implementations of both algorithms we use an acyclic variant of a TBox developed for our application in chemical process engineering. It contains 52 primitive concepts, 67 defined concepts, 23 roles and 20 features. It has a deep concept hierarchy, what makes it likely that least common subsumers computed for concept descriptions from this TBox will not collapse to $\top$.

The input concept descriptions we used for the evaluation are the least common subsumers of seven REACTOR concepts defined in the application TBox. To compute the lcs of all combinations of least common subsumers, we started from the hierarchy of least common subsumers as shown in Figure 2. This hierarchy is computed by the earlier mentioned attribute exploration algorithm as described in [3; 10].

Our test suite included 22 different lcs calls, starting from binary lcs calls. For each computation of these least common subsumers we measured run-times and sizes of the output concept descriptions of four settings:

1. *old LCS*: Computation of the lcs using the old lcs implementation,
2. *old LCS + Rew.*: Computation of the lcs using the old lcs implementation followed by rewriting the lcs concept description,
3. *new LCS*: Computation of the lcs using the new lcs implementation,
4. *new LCS + Rew.*: Computation of the lcs using the new lcs implementation followed by rewriting the lcs concept description.

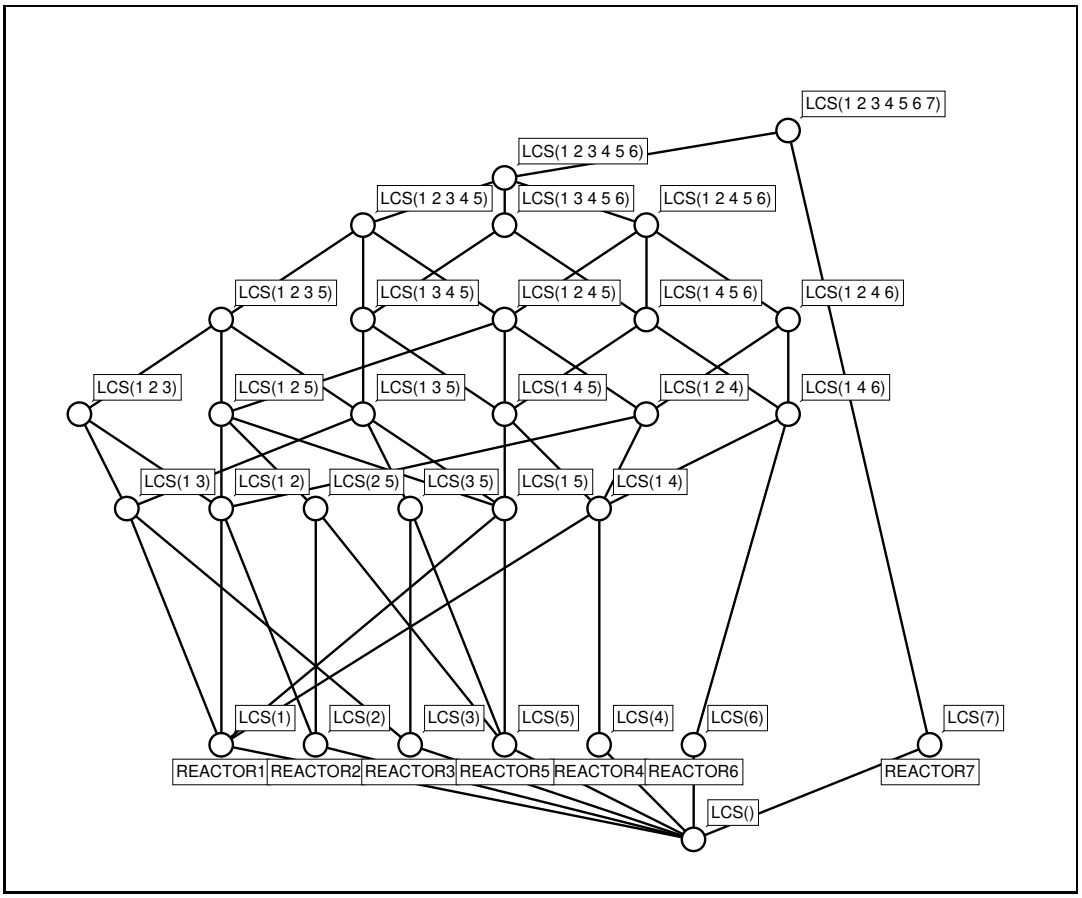

Figure 2: The hierarchy of least common subsumers of seven reactor descriptions.

The second and the fourth setting use the same rewriting implementation of the heuristic algorithm. In contrast to our application framework, where the computation of lcs with n input concept descriptions uses the resulting description of the lcs with n-1 input concept descriptions, all least common subsumers are computed from scratch. To obtain representative run-times we did run each LCS in each setting 100 times.

The results for the average concept size is displayed in Figure 3. Note the use of a logarithmic scale. The measured values indicate that a lcs using lazy unfolding returns concept descriptions that are about an order of magnitude smaller than the concept descriptions returned by the basic algorithm. Rewriting the lcs concept descriptions is resulting in terms that are two (one) orders of magnitude smaller than the lcs concept description returned by the lcs (lcs using lazy unfolding). Comparing the concept sizes yielded by both settings including rewriting shows that starting from a smaller concept description does not yield a smaller rewritten concept by the heuristic algorithm.

The absolute values for the concept sizes show that rewriting is still necessary, even if lazy unfolding is employed to compute the concept description of the lcs. The average concept description obtained from the new LCS has a concept size of about 100. Therefore these concept descriptions are still too big to be comprehensible for a human reader. In our application framework, a knowledge engineer has to chose an appropriate description from a set of lcs concept descriptions to be added to the terminology. Therefore, rewriting remains necessary

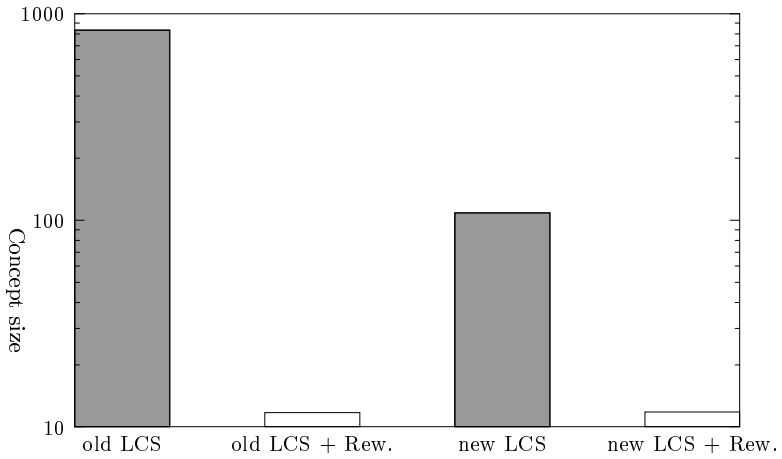

Figure 3: Average concept sizes obtained from the four settings.

as an additional step for this application.

The run-times in Figure 4 show the added run-times of computing the lcs (grey rectangles) and of rewriting the obtained lcs concept description (white rectangles). The comparison of run-times for the lcs implementations indicate a speed-up of factor 3.5. Thus, the application of lazy unfolding provides only a moderate speed-up for the lcs alone. However, in combination with rewriting a bigger enhancement is obtained. The run-time for rewriting a lcs concept description is shorter for the smaller descriptions returned from the new lcs implementation. In case of our application the run-time is roughly halved.

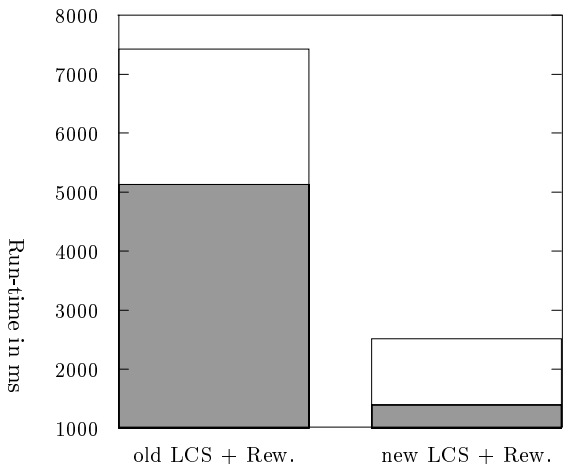

Figure 4: Average run-times needed by the different settings.

4 Conclusion and Future Work

In general the performance of the lcs algorithm using lazy unfolding depends very much on the structure of the TBox in use. If the concept descriptions in the concept definitions of the TBox do not make use of defined concepts, lazy unfolding will not be able to increase the performance of the lcs significantly.

The first evaluation of the lcs implementations in our application framework indicates that using lazy unfolding can be a substantial gain for the size of the resulting concept descriptions. On the other hand our results also indicate that it is still necessary to perform rewriting after computing the lcs in order to obtain readable concept descriptions. With lazy unfolding in use we expect the run-times to decrease for the rewriting step.

We plan to implement further optimizations of the lcs. To speed-up a single call of the binary lcs function it may be advantageous to minimize the exr_r list and the conjunction val_r w.r.t. subsumption to avoid redundant recursive calls. So before starting a recursive call all elements that are subsuming another element of the list are removed. This method would make heavy use of the underlying DL reasoner and it has to be evaluated if the overhead from the subsumption tests does not outweigh the benefits.

To compute the lcs of many input concept descriptions several successive calls to the binary lcs function are made. In order to speed-up the computation for the whole set of input descriptions (not only one run of the binary lcs procedure), a sorting for the input descriptions seems to be a good idea. If a "general" lcs concept description is computed by the first few calls of the binary lcs function, it is likely that the general lcs subsumes the remaining input terms. As a consequence the remaining calls of the binary lcs function will yield the result already after the subsumption test at the beginning of the lcs algorithm without any recursive calls at all. Therefore sorting the list of input descriptions according their "generality" may be a useful preprocessing step. Another method to further optimize the lcs implementation is of course a caching strategy for already computed least common subsumers, but this also remains to be future work.

References

[1] F. Baader and R. Küsters. Computing the least common subsumer and the most specific concept in the presence of cyclic $\mathcal{ALN}$-concept descriptions. In O. Herzog and A. Günter, editors, *KI-98*, volume 1504 of *Lecture Notes in Computer Science*, pages 129–140, Bremen, Germany, 1998. Springer-Verlag.

[2] F. Baader, R. Küsters, and R. Molitor. Rewriting concepts using terminologies. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR-00)*, pages 297–308, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[3] F. Baader and R. Molitor. Building and structuring description logic knowledge bases using least common subsumers and concept analysis. In B. Ganter and G. Mineau, editors, *ICCS-00*, volume 1867 of *Lecture Notes in Artificial Intelligence*, pages 290–303. SV, 2000.

[4] F. Baader and A.-Y. Turhan. TBoxes do not yield a compact representation of least common subsumers. In *DL-2001*, 2001. To appear.

[5] Franz Baader, Enrico Franconi, Bernhard Hollunder, Bernhard Nebel, and Hans-Jürgen Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.

[6] Franz Baader, Ralf Küsters, and Ralf Molitor. Computing least common subsumer in description logics with existential restrictions. In T. Dean, editor, *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*, pages 96–101, Stockholm, Sweden, 1999. Morgan Kaufmann, Los Altos.

[7] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics.* PhD thesis, University of Manchester, 1997.

[8] Ian Horrocks. Optimisation techniques for expressive description logics. Technical Report UMCS-97-2-1, University of Manchester, Department of Computer Science, 1997.

[9] Ralf Küsters and R. Molitor. Approximating most specifc concepts in description logics with existential restrictions. In *Proc. of the 24th German Annual Conf. on Artificial Intelligence (KI'01)*, 2001. to appear.

[10] Ralf Molitor. *Unterstützung der Modellierung verfahrenstechnischer Prozesse durch Nicht-Standardinferenzen in Beschreibungslogiken.* PhD thesis, Department of Computer Science, RWTH Aachen, Germany, 2000. In German.

[11] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence Journal*, 43:235–249, 1990.