Query Answering via Modal Definability with FaCT++: First Blood

S. Kikot[1], D. Tsarkov[2], M. Zakharyaschev[1], and E. Zolin[3]

[1] Department of Computer Science and Information Systems,
Birkbeck, University of London, U.K.
{kikot,michael}@dcs.bbk.ac.uk
[2] School of Computer Science, University of Manchester, U.K.
tsarkov@cs.man.ac.uk
[3] Faculty of Mathematics and Mechanics,
Moscow State University, Russia
ezolin@gmail.com

Abstract. We use results on modal definability of first-order formulas to reduce the problem of answering conjunctive queries over knowledge bases (of any expressivity) to checking inconsistency of those knowledge bases extended with a number of $\mathcal{ALCI}$ concept assertions. This reduction has been shown to work for conjunctive queries without cycles that only involve bound variables. In this paper, we present an optimised algorithm for the reduction, its implementation using FaCT++ as an underlying reasoner and the results of first experiments.

1 Introduction

Conjunctive query (CQ) answering over description logic (DL) ontologies has recently become one of the hottest topics in the DL community. The DLs considered range from

– the *DL-Lite* family and *OWL 2 QL*, which support first-order rewritability (and so belong to the class AC^0 for data complexity) and can be used with standard relational database systems [2, 20, 1, 13], to
– Horn DLs encompassing the $\mathcal{EL}$ family, *OWL 2 EL*, Horn-$\mathcal{SROIQ}$ and Horn-$\mathcal{SHOIQ}$, which support DataLog rewritability (and so are P-complete for data complexity) and can be used with DataLog engines [21, 14, 19, 5], and further to
– very expressive DLs which are coNP-hard for data complexity and require some special techniques for answering CQs; see [10, 15, 18, 7, 4, 3] and references therein for details.

In this paper, we present yet another approach to answering CQs over DL ontologies. It is based on a transformation of a given CQ $\boldsymbol{q}(x_1, \dots, x_n)$ to $\mathcal{ALCI}$-concepts $C_1, \dots, C_n$ such that, for any knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (in any DL language) and any n-tuple $\boldsymbol{a} = (a_1, \dots, a_n)$ of individuals from the ABox $\mathcal{A}$, we have $\mathcal{K} \models \boldsymbol{q}(\boldsymbol{a})$ iff $\mathcal{K} \cup \{a_1 \colon C_1, \dots, a_n \colon C_n\}$ is inconsistent. This transformation of CQs, which we call *concept-rewriting* (or *c-rewriting* for short), is based on the approach to modal definability of first-order formulas developed in [23, 24, 12]. It works (at least) for connected CQs that have no cycles with only bound variables. (Note that c-rewriting is based solely on the shape of a CQ and hence works even for cyclic CQs with non-simple roles in cycles, which cannot be handled by the standard query answering techniques [6, 15].) Thus, in theory, answering c-rewritable CQs can be delegated to any standard DL reasoner.

In practice, however, there are a few major obstacles to implementing the c-rewriting approach using a DL reasoner as a black box. The principal one is that checking consistency of a KB with a large ABox may be expensive, especially if we have to test all possible tuples of individuals in order to compute all answers to a given CQ. Even if a reasoner supports incremental ABoxes [9], the approach is still impractical as the algorithms for processing additions and (especially) removals of ABox assertions are quite involved and complicated.

Our solution in this paper is to integrate the c-rewriting approach into the DL reasoner FaCT++. During the consistency check, FaCT++ materialises the ABox as a completion graph. For every answer candidate, the concepts involved in the c-rewriting of the query are added, in the constructed completion graph, to the labels of the corresponding individuals, and then the tableau algorithm is used to check consistency of the updated completion graph. Thus, we do not check consistency from scratch, but rather use the already computed completion graph as a starting point. To cope with numerous answer candidates, we propose three optimisation techniques that drastically improve performance, as our first experimental results demonstrate.

2 Concept-Rewritable Conjunctive Queries

By a *knowledge base* (KB) in this paper we understand a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ formulated in the language of any DL, for example, $\mathcal{SROIQ}$ underlying *OWL 2*. We use $\mathsf{ind}(\mathcal{A})$ to denote the set of individual names that occur in the ABox $\mathcal{A}$.

As usual, a *conjunctive query* (CQ) $\boldsymbol{q}(\boldsymbol{x})$ is a formula $\exists \boldsymbol{u}\, \varphi(\boldsymbol{x}, \boldsymbol{u})$, where φ is a conjunction of *atoms* of the form $A_i(z_1)$ or $R_j(z_1, z_2)$ with $z_k \in \boldsymbol{x} \cup \boldsymbol{u}$ (without loss of generality, we assume that CQs do not contain individual names). The free variables $\boldsymbol{x}$ in $\boldsymbol{q}(\boldsymbol{x})$ are called *answer variables.* We typically use $\boldsymbol{x}$ and $\boldsymbol{y}$ to denote tuples of answer variables. A tuple $\boldsymbol{a} \subseteq \mathsf{ind}(\mathcal{A})$ is a *certain answer* to $\boldsymbol{q}(\boldsymbol{x})$ over a KB $\mathcal{K}$ if $\mathcal{I} \models \boldsymbol{q}(\boldsymbol{a})$ for all $\mathcal{I} \models \mathcal{K}$; in this case we write $\mathcal{K} \models \boldsymbol{q}(\boldsymbol{a})$. Where convenient, we regard a CQ as the set of its atoms. In this paper, we consider only CQs with at least one answer variable.

Definition 1. Given a CQ $\boldsymbol{q}(\boldsymbol{x})$ with $\boldsymbol{x} = (x_1, \dots, x_n)$, a *concept rewriting* (or a *c-rewriting*) *of* $\boldsymbol{q}(\boldsymbol{x})$ *in a DL* $\mathcal{L}$ is an n-tuple $(x_1 \colon C_1, \dots, x_n \colon C_n)$ with $\mathcal{L}$-concepts C_i such that, for any KB[1] $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and any n-tuple $\boldsymbol{a} = (a_1, \dots, a_n)$ of individuals in $\mathcal{A}$, we have $\mathcal{K} \models \boldsymbol{q}(\boldsymbol{a})$ iff the KB $\mathcal{K} \cup \{a_1 \colon C_1, \dots, a_n \colon C_n\}$ is

[1] In any DL or even first-order language; see the discussion after Definition 4 in [23].

inconsistent. We say that a CQ $\boldsymbol{q}(\boldsymbol{x})$ is *concept-rewritable* (or *c-rewritable*) in $\mathcal{L}$ if it has a c-rewriting in $\mathcal{L}$. The *size* of a c-rewriting is defined as the sum of the sizes of all C_i, $1 \leq i \leq n$. Note that a c-rewriting of a CQ may (and usually does) contain concept or role names that do not occur in the CQ. We assume that such names are *fresh* in the sense that they are taken from a special alphabet that is *not* used in any DL knowledge bases $\mathcal{K}$ mentioned above.

Example 1. Here are some c-rewritings in $\mathcal{ALC}$ (with B a fresh concept name):

- $(x\colon \neg A)$ is a c-rewriting of the CQ $q(x) = A(x)$;
- $(x\colon B \sqcap \forall R.\neg B)$ is a c-rewriting of $\boldsymbol{q}(x) = R(x,x)$;
- $(x\colon \forall R.\neg B, y\colon B)$ is a c-rewriting of $\boldsymbol{q}(x,y) = R(x,y)$;
- $(x\colon \forall R.B, y\colon \forall R.\neg B)$ is a c-rewriting of $\boldsymbol{q}(x,y) = \exists u\,(R(x,u) \land R(y,u))$.

We only show this for $\boldsymbol{q}(x) = R(x,x)$. Suppose $\mathcal{K} = (\mathcal{T},\mathcal{A})$. We need to show that, for every $a \in \mathsf{ind}(\mathcal{A})$, we have $\mathcal{K} \models R(a,a)$ iff $\mathcal{K} \cup \{a\colon B \sqcap \forall R.\neg B\}$ is inconsistent. The implication $\Rightarrow$ is trivial. For the converse, suppose $\mathcal{K} \not\models R(a,a)$ and consider a model $\mathcal{I}$ of $\mathcal{K}$ where $\mathcal{I} \not\models R(a,a)$. Since B is a fresh concept name, we can set $B^{\mathcal{I}} := \{a^{\mathcal{I}}\}$. Then, clearly, we have $\mathcal{I} \models \mathcal{K} \cup \{a\colon B \sqcap \forall R.\neg B\}$.

Example 2. On the other hand, the query $\boldsymbol{q}(x) = \exists u\,(R(x,u) \land R(u,u))$ is not c-rewritable in $\mathcal{ALC}$. Indeed, suppose there exists an $\mathcal{ALC}$-concept C such that $\mathcal{K} \models \boldsymbol{q}(a)$ iff $\mathcal{K} \cup \{a\colon C\}$ is inconsistent, for every KB $\mathcal{K} = (\mathcal{T},\mathcal{A})$ and every $a \in \mathsf{ind}(\mathcal{A})$. Consider the KBs $\mathcal{K} = (\mathcal{T},\mathcal{A})$ and $\mathcal{K}' = (\mathcal{T},\mathcal{A}')$ in the DL $\mathcal{S}$, where

$$\begin{aligned}
\mathcal{T} &= \{\, A \sqsubseteq \forall R.\neg A,\ \top \sqsubseteq \exists R.\top,\ \mathsf{Trans}(R)\,\},\\
\mathcal{A} &= \{\, A(a)\,\},\\
\mathcal{A}' &= \{\, A(a),\ R(a,b),\ R(b,b)\,\}.
\end{aligned}$$

Clearly, $\mathcal{K} \not\models \boldsymbol{q}(a)$, and so $\mathcal{K} \cup \{a\colon C\}$ is consistent. On the other hand, we have $\mathcal{K}' \models \boldsymbol{q}(a)$, and so $\mathcal{K}' \cup \{a\colon C\}$ is inconsistent. By the finite model property of $\mathcal{S}$ (see, e.g., [16, Corollary 4.3]), $\mathcal{K} \cup \{a\colon C\}$ has a finite model $\mathcal{I}$. Since every element in $\mathcal{I}$ has an R-successor, $a^{\mathcal{I}}$ has an R-successor w such that $(w,w) \in R^{\mathcal{I}}$. Moreover, $w \neq a^{\mathcal{I}}$ due to the first axiom in $\mathcal{T}$. Now, by taking $b^{\mathcal{I}} := w$, we obtain a model of $\mathcal{K}' \cup \{a\colon C\}$, contrary to our assumption. (We do not know whether the same result can be proved using $\mathcal{ALC}$ KBs.)

Two sufficient conditions of c-rewritability of CQs in $\mathcal{ALC}$ and $\mathcal{ALCI}$ can be obtained as a consequence of the modal definability results from [12]. To formulate these conditions, we need a few more definitions.

A CQ $\boldsymbol{q}(\boldsymbol{x})$ is called *connected* if its primal graph (whose vertices are the variables in $\boldsymbol{q}$ and edges are the pairs (z,v) such that $R(z,v) \in \boldsymbol{q}$, for some role name R) is connected. We say that $\boldsymbol{q}(\boldsymbol{x})$ is *internally cycle-free* if its primal graph has no cycles that consist of bound variables. Thus, all CQs from Example 1 are internally cycle-free, while that in Example 2 is not, as $R(u,u)$ is a cycle through the bound variable u. Finally, we call $\boldsymbol{q}(\boldsymbol{x})$ *internally reachable* if each bound variable u in $\boldsymbol{q}(\boldsymbol{x})$ is reachable via a directed path from some free variable;

more precisely, if there is a path $S_1(z_0, z_1), S_2(z_1, z_2), \ldots, S_n(z_{n-1}, z_n)$ such that $z_0 \in \boldsymbol{x}$, $u = z_n$ and each S_i is a role name in the given DL. For example, the CQ $\boldsymbol{q}(x) = \exists u\, R(x, u)$ is internally reachable, while $\boldsymbol{q}'(x) = \exists u\, R(u, x)$ is not.

Theorem 1 (cf. [12], Theorem 7.8). (*i*) *Every connected and internally cycle-free CQ* $\boldsymbol{q}(\boldsymbol{x})$ *has a c-rewriting in* $\mathcal{ALCI}$ *of size* $O(|\boldsymbol{q}|)$.

(*ii*) *Every connected, internally cycle-free, and internally reachable CQ* $\boldsymbol{q}(\boldsymbol{x})$ *has a c-rewriting in* $\mathcal{ALC}$ *of size* $O(|\boldsymbol{q}|)$.

Below we present an algorithm that runs in quadratic time and, given a connected CQ $\boldsymbol{q}(\boldsymbol{x})$, checks whether it is internally cycle-free and, if it is, constructs a c-rewriting of $\boldsymbol{q}(\boldsymbol{x})$ in $\mathcal{ALCI}$ of size $O(|\boldsymbol{q}|)$. Moreover, if $\boldsymbol{q}(\boldsymbol{x})$ is internally reachable, the resulting c-rewriting will be in $\mathcal{ALC}$.

Note that the c-rewriting algorithm from [12] can only be applied to CQs without *concept* (unary) atoms, and in order to deal with arbitrary CQs, it was suggested to eliminate such atoms at the price of introducing fresh role names. In contrast, the c-rewriting algorithm presented below is directly applicable to queries with concept atoms. These atoms are treated uniformly with fresh concept names introduced by the algorithm (see A and B_2 in Examples 6 and 7 below). However, one can see that the output of the algorithm below coincides with that of the algorithm from [12] after eliminating the fresh role names in the latter in accordance with their definitions. Thus, the new algorithm is correct.

3 C-Rewriting Algorithm

Suppose we are given a connected CQ $\boldsymbol{q}(\boldsymbol{x})$ (in this section, we treat every CQ as the set of its atoms). The algorithm proceeds in four stages.

First, it 'unravels' $\boldsymbol{q}(\boldsymbol{x})$ to a CQ $\boldsymbol{q}'(\boldsymbol{y})$ that has no cycles containing at least one free variable. While doing this, it keeps the information about the substitution $\sigma\colon \boldsymbol{y} \to \boldsymbol{x}$ of free variables required to transform $\boldsymbol{q}'(\boldsymbol{y})$ back to $\boldsymbol{q}(\boldsymbol{x})$. Secondly, the algorithm checks whether the CQ $\boldsymbol{q}'(\boldsymbol{y})$ is tree-shaped, in which case it represents $\boldsymbol{q}'(\boldsymbol{y})$ by means of an $\mathcal{ELIO}$-concept C with nominals. Thirdly, the algorithm translates C into a c-rewriting $(y_1\colon D_1, \ldots, y_m\colon D_m)$ of $\boldsymbol{q}'(\boldsymbol{y})$. Finally, it identifies variables in $\boldsymbol{y}$ in accordance with the substitution σ and returns the resulting c-rewriting $(x_1\colon C_1, \ldots, x_n\colon C_n)$ of $\boldsymbol{q}(\boldsymbol{x})$ in $\mathcal{ALCI}$ (or even in $\mathcal{ALC}$, if $\boldsymbol{q}(\boldsymbol{x})$ is internally reachable).

Example 3. Consider again the CQ $\boldsymbol{q}(x) = R(x, x)$.

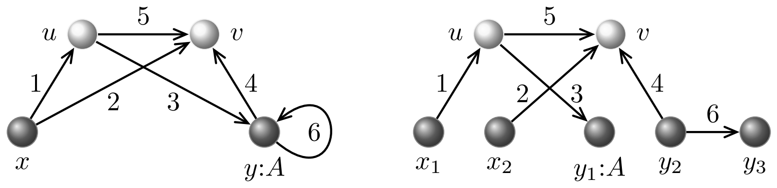
Stage 1: We unravel $\boldsymbol{q}(x)$ into the CQ $\boldsymbol{q}'(x, y) = R(x, y)$ with the substitution $\sigma(x) = x$ and $\sigma(y) = x$.
Stage 2: We represent $\boldsymbol{q}'(x, y)$ by the $\mathcal{ELIO}$-assertion $x\colon (\{x\} \sqcap \exists R.\{y\})$.
Stage 3: We construct the c-rewriting $(x\colon \neg\exists R.B,\ y\colon B)$ of $\boldsymbol{q}'(x, y)$ in $\mathcal{ALC}$, where B is a fresh concept name.
Stage 4: We merge x and y and take the conjunction of their concepts, which gives us the c-rewriting $x\colon (B \sqcap \neg\exists R.B)$ of the initial query $\boldsymbol{q}(x)$ in $\mathcal{ALC}$. Note that it is equivalent to the c-rewriting we gave in Example 1.

Now we proceed to a detailed description of all the stages of the algorithm.

Stage 1: Unravelling the CQ

In this stage, the input is a connected CQ $\boldsymbol{q}(\boldsymbol{x})$. The output is a connected CQ $\boldsymbol{q}'(\boldsymbol{y})$, with $|\boldsymbol{y}| \geq |\boldsymbol{x}|$, that has no cycles containing any free variables, together with a substitution $\sigma\colon \boldsymbol{y} \to \boldsymbol{x}$ such that $\sigma(\boldsymbol{q}'(\boldsymbol{y})) = \boldsymbol{q}(\boldsymbol{x})$. In particular, if $\boldsymbol{q}$ is internally cycle-free, then $\boldsymbol{q}'$ is a tree-shaped CQ (i.e., its primal graph is a tree).

Example 4. For example, the CQ $\boldsymbol{q}(x, y)$ on the left in the figure below is unravelled into the tree-shaped CQ $\boldsymbol{q}'(x_1, x_2, y_1, y_2, y_3)$ on the right:



More precisely, we transform a CQ $\boldsymbol{q}(\boldsymbol{x})$ into a CQ $\boldsymbol{q}'(\boldsymbol{y})$ as follows:

1. Initially, we take the identical substitution $\sigma\colon \boldsymbol{x} \to \boldsymbol{x}$, i.e., $\sigma(x_i) = x_i$.
2. Find in the current CQ a cycle that contains at least one free variable. If no such cycle is found, we return the current CQ and halt.
3. Pick from this cycle any free variable x and any role atom containing this variable. Assume that the chosen atom has the form $R(x, z)$, where R is a role name (and possibly $z = x$). (If the chosen atom has the form $R(z, x)$ the subsequent operations are analogous.)
4. Take a fresh free variable y, remove the atom $R(x, z)$ from the CQ and add the atom $R(y, z)$. Extend the substitution σ by $\sigma(y) := x$. Go to Step 2.

Stage 2: Rolling-up a tree-shaped CQ into an $\mathcal{ELIO}$-concept

In this stage, the input is a connected CQ $\boldsymbol{q}'(\boldsymbol{y})$ that has no cycles involving any free variables. If the CQ contains a cycle (which consists of bound variables only), then an error message is returned (since this means that the initial query $\boldsymbol{q}(\boldsymbol{x})$ is not internally cycle-free). Otherwise, the CQ is tree-shaped and the output is an $\mathcal{ELIO}$-concept C with nominals $\boldsymbol{y}$, denoted $C(\boldsymbol{y})$, that 'represents' the query $\boldsymbol{q}'(\boldsymbol{y})$ in the following sense: for any interpretation $\mathcal{I}$ and any tuple $\boldsymbol{e}$ of its elements with $|\boldsymbol{y}| = |\boldsymbol{e}|$ and e_1 being the first component of $\boldsymbol{e}$,

$$\mathcal{I} \models \boldsymbol{q}'(\boldsymbol{e}) \quad \Longleftrightarrow \quad \mathcal{I} \models e_1 \colon C(\boldsymbol{e}).$$

The algorithm given below performs traversal of the primal graph of the CQ in a depth-first manner, starting from some answer variable and marking all 'visited' variables. If it encounters an already marked variable then the graph has a cycle. Otherwise, the traversal is recorded as an $\mathcal{ELIO}$-concept. All this is achieved by assigning $C(\boldsymbol{y}) = \textsc{Build\_ELIO\_Concept}(\boldsymbol{q}', \emptyset, \texttt{NULL}, y_1)$.

Example 5. The CQ $\boldsymbol{q}'(x_1, x_2, y_1, y_2, y_3)$ from Example 4 is transformed by this algorithm into the pair $(x_1 \colon C)$, where

$$C = \{x_1\} \sqcap \exists R_1.\bigl[\exists R_3.(\{y_1\} \sqcap A) \sqcap \exists R_5.\bigl(\exists R_2^-.\{x_2\} \sqcap \exists R_4^-.(\{y_2\} \sqcap \exists R_6.\{y_3\})\bigr)\bigr].$$

Algorithm 1 Rolling-up a tree-shaped CQ into an $\mathcal{ELIO}$-concept (here R is either a role name or its inverse and $\textsc{InverseOf}(R(u,v)) = R^-(v,u)$).

```
function Build_ELIO_Concept(Query $q$, Set $Visited$, atom $at$, var $v$)
    Concept $C \leftarrow \top$
    if $v$ is free in $q$ then
        $C \leftarrow \{v\}$
    for all $A(v) \in q$ do
        $C \leftarrow C \sqcap A$
    $Visited \leftarrow Visited \cup \{v\}$
    for all $R(v,w) \in q : R(v,w) \neq$ InverseOf($at$) do
        if $w \in Visited$ then
            throw 'Cycle through only bound variables found'
        $C \leftarrow C \sqcap \exists R.$Build_ELIO_Concept($q, Visited, R(v,w), w$)
    return $C$
```

As shown in [12], this concept $C = C(x_1, x_2, y_1, y_2, y_3)$ is related to the query $\boldsymbol{q}'(x_1, x_2, y_1, y_2, y_3)$ and to the original query $\boldsymbol{q}(x, y)$ in the following way: for any KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and any individuals $a, b \in \mathsf{ind}(\mathcal{A})$, we have $\mathcal{K} \models \boldsymbol{q}(a, b)$ iff $\mathcal{K} \models \boldsymbol{q}'(a, a, b, b, b)$ iff $\mathcal{K} \models a\colon C(a, a, b, b, b)$.

Remark 1. If we are happy to deal with $\mathcal{ELIO}$-concepts (for instance, in case our KB is already formulated in an extension of $\mathcal{ELIO}$), then we can terminate the algorithm now, for the above $\mathcal{ELIO}$-concept can already be used for finding answers to $\boldsymbol{q}(\boldsymbol{x})$. For example, to check whether a given pair of individuals (a, b) is a certain answer to the CQ $\boldsymbol{q}(x, y)$ from Example 5, we can simply check whether the KB $\mathcal{K} \cup \{a\colon \neg C(a, a, b, b, b)\}$ is inconsistent.

However, usually the addition of inverse roles and nominals to a DL makes reasoning harder. So, our aim below is to avoid these constructors and produce concepts that have neither nominals nor (if possible) inverse roles.

Stage 3: Converting $\mathcal{ELIO}$-concepts to c-rewritings

As an input, we are given an $\mathcal{ELIO}$-concept C with nominals $\boldsymbol{y} = (y_1, \ldots, y_m)$. For an output, we construct an m-tuple of $\mathcal{ALCI}$-concepts $(D_1, \ldots, D_m)$ such that $(y_1\colon D_1, \ldots, y_m\colon D_m)$ is a c-rewriting of $\boldsymbol{q}'(\boldsymbol{y})$ in $\mathcal{ALCI}$. Moreover, as shown in [12], if the initial query $\boldsymbol{q}(\boldsymbol{x})$ is internally reachable, the algorithm below constructs a c-rewriting in $\mathcal{ALC}$.

To describe the algorithm, we require a few definitions. Denote by $d(\{x\}, C)$ the *role-name-depth* of an *occurrence* of a nominal $\{x\}$ in C, which is the number of constructors of the form $\exists R$, where R is a role name (and not an inverse role), under which this occurrence of $\{x\}$ lies in C. Formally, it is defined by induction:

- $d(\{x\}, \{x\}) = 0$;
- $d(\{x\}, C \sqcap D) = d(\{x\}, D \sqcap C) = d(\{x\}, C)$, if this occurrence of $\{x\}$ is in C;
- $d(\{x\}, \exists R.C) = d(\{x\}, C) + 1$;
- $d(\{x\}, \exists R^-.C) = d(\{x\}, C)$.

Thus, in Example 5, nominal occurrences have the following role-name-depths:

$$C = \underset{0}{\{x_1\}} \sqcap \exists R_1.\big[\exists R_3.(\underset{2}{\{y_1\}} \sqcap A) \sqcap \exists R_5.\big(\exists R_2^-.\underset{2}{\{x_2\}} \sqcap \exists R_4^-.(\underset{2}{\{y_2\}} \sqcap \exists R_6.\underset{3}{\{y_3\}})\big)\big].$$

Simple $\mathcal{ELIO}$-concepts are defined by induction (note 'or' in the last item):

- each nominal $\{x_i\}$ is a simple concept;
- if C is simple, then so is $\exists R^-.C$;
- if C ***or*** D is simple, then so is $C \sqcap D$.

Typical examples of simple concepts are $\{x\}$, $\exists R^-.\{x\}$, $\exists R^-.(\{x\} \sqcap \exists R.\{y\})$. Simple concepts were introduced in [11] for technical reasons: they correspond to minimal valuations for antecedents of generalised Sahlqvist formulas [8]. We need this notion to ensure that the resulting rewriting is in $\mathcal{ALC}$ for internally reachable CQs. Examples of non-simple concepts are $\exists R.\{x\}$ and $\exists R^-.\exists R.\{x\}$.

Now, the algorithm takes an $\mathcal{ELIO}$-concept C with nominals $\boldsymbol{y} = (y_1, \dots, y_m)$. Note that each nominal from $\boldsymbol{y}$ has exactly one occurrence in C. The algorithm produces an ordered set of pairs `Res` $= (y_1 \colon D_1, \dots, y_m \colon D_m)$, where D_j are $\mathcal{ALCI}$-concepts. Initially, `Res` is set to empty. Then we proceed as follows.

1. The concept C has the form $C = \{y_1\} \sqcap E$. If E has no nominals, then add $(y_1 \colon \neg E)$ to `Res` and halt.
2. Otherwise, pick a nominal $\{y\}$ in E with maximal $d(\{y\}, E)$.
3. Find in E the maximal (w.r.t. the subconcept relation) simple subconcept H containing $\{y\}$ and no other nominals. To this end, start the search with $\{y\}$ (which is simple) and go upwards in the syntactic tree of C until the concept under consideration either is non-simple or contains another nominal.
4. Replace this occurrence of H in C with a fresh concept name B and denote the result by C'.
5. Starting with the sequent $H \Rightarrow B$, apply the following rules repeatedly (the omitted rule with $F' \sqcap F$ is symmetrical):

$$\textbf{(r1)} \quad \frac{F \sqcap F' \Rightarrow G}{F \Rightarrow \neg F' \sqcup G} \text{ (where } \{y\} \text{ is in } F) \qquad\qquad \textbf{(r2)} \quad \frac{\exists R^-.F \Rightarrow G}{F \Rightarrow \forall R.G}$$

   until a sequent of the form $\{y\} \Rightarrow D$ is obtained, for some $\mathcal{ALCI}$-concept D.
6. Add the pair $(y \colon D)$ to `Res`.
7. Go to Step 1 with the new concept $C := C'$.

Note that since the concept H is simple and contains only one nominal, the transformation of Step 5 is always possible and even deterministic. As shown in [12, Theorem 7.9], if the initial CQ $\boldsymbol{q}(\boldsymbol{x})$ was internally cycle-free, then the above algorithm returns a c-rewriting of $\boldsymbol{q}'(\boldsymbol{y})$ in $\mathcal{ALCI}$; if additionally the initial CQ $\boldsymbol{q}(\boldsymbol{x})$ was internally reachable, then D and E above are always $\mathcal{ALC}$-concepts, so in this case the algorithm produces a c-rewriting of $\boldsymbol{q}'(\boldsymbol{y})$ in $\mathcal{ALC}$.

Example 6. Let C be as in Example 5. We show how Stage 3 works.

Iteration 1: The deepest nominal in C is $\{y_3\}$. Since $\exists R_6.\{y_3\}$ is not simple, we take $H = \{y_3\}$ and replace it with a fresh concept name B_1. No rules from Step 4 were needed. So we add the pair $\boxed{y_3 \colon B_1}$ to `Res` and obtain

$$C = \{x_1\} \sqcap \exists R_1.\big[\exists R_3.(\{y_1\} \sqcap A) \sqcap \exists R_5.\big(\exists R_2^-.\{x_2\} \sqcap \exists R_4^-.(\{y_2\} \sqcap \exists R_6.\boldsymbol{B_1})\big)\big].$$

Iteration 2: The nominals $\{y_1\}$, $\{x_2\}$ and $\{y_2\}$ are equally deep, so any of them will do. Let us pick $\{y_2\}$. Then $H = \exists R_4^-.(\{y_2\} \sqcap \exists R_6.B_1)$, since it is simple, while $\exists R_2^-.\{x_2\} \sqcap \exists R_4^-.(\{y_2\} \sqcap \exists R_6.B_1)$ contains a different nominal $\{x_2\}$. We introduce a fresh concept name B_2 and transform the sequent $H \Rightarrow B_2$ as follows:

$$\begin{array}{rcrl}
\exists R_4^-.(\{y_2\} \sqcap \exists R_6.B_1) & \Rightarrow & B_2 & \text{(initial sequent)} \\
\{y_2\} \sqcap \exists R_6.B_1 & \Rightarrow & \forall R_4.B_2 & \text{by } \mathbf{(r2)} \\
\{y_2\} & \Rightarrow & \neg\exists R_6.B_1 \sqcup \forall R_4.B_2 & \text{by } \mathbf{(r1)}
\end{array}$$

So we add the pair $\boxed{y_2 \colon \neg\exists R_6.B_1 \sqcup \forall R_4.B_2}$ to `Res` and obtain:

$$C = \{x_1\} \sqcap \exists R_1.\big(\exists R_3.(\{y_1\} \sqcap A) \sqcap \exists R_5.(\exists R_2^-.\{x_2\} \sqcap \boldsymbol{B_2})\big).$$

Iteration 3: Now we pick the nominal $\{x_2\}$, so $H = (\exists R_2^-.\{x_2\} \sqcap B_2)$. Replace it in C with a fresh concept name B_3 and transform the sequent $H \Rightarrow B_3$:

$$\begin{array}{rcrl}
\exists R_2^-.\{x_2\} \sqcap B_2 & \Rightarrow & B_3 & \text{(initial sequent)} \\
\exists R_2^-.\{x_2\} & \Rightarrow & \neg B_2 \sqcup B_3 & \text{by } \mathbf{(r1)} \\
\{x_2\} & \Rightarrow & \forall R_2.(\neg B_2 \sqcup B_3) & \text{by } \mathbf{(r2)}
\end{array}$$

So we add $\boxed{x_2 \colon \forall R_2.(\neg B_2 \sqcup B_3)}$ to `Res` and obtain the concept

$$C = \{x_1\} \sqcap \exists R_1.(\exists R_3.(\{y_1\} \sqcap A) \sqcap \exists R_5.\boldsymbol{B_3}).$$

Iteration 4: Pick the nominal $\{y_1\}$, set $H = (\{y_1\} \sqcap A)$, introduce a fresh concept name B_4, add $\boxed{y_1 \colon (\neg A \sqcup B_4)}$ to `Res` and obtain the concept

$$C = \{x_1\} \sqcap \exists R_1.(\exists R_3.\boldsymbol{B_4} \sqcap \exists R_5.B_3).$$

Iteration 5: We notice that $C = \{x_1\} \sqcap E$, where the concept E has no nominals. So we add $\boxed{x_1 \colon \neg\exists R_1.(\exists R_3.B_4 \sqcap \exists R_5.B_3)}$ to `Res` and halt.

The resulting set `Res` consists of all the framed pairs.

Stage 4: Merging nominals

In the set `Res` of pairs $(y_1 \colon D_1, \ldots, y_m \colon D_m)$, we replace the variables $\boldsymbol{y}$ with $\boldsymbol{x}$ in accordance with the substitution σ defined in Stage 1. Thus we obtain a new set `Res′`. Finally, for every variable $x_i \in \boldsymbol{x}$, we set C_i to be the conjunction of all concepts that are assigned in `Res′` to the variable x_i. As shown in [12], the resulting tuple $(x_1 \colon C_1, \ldots, x_n \colon C_n)$ is a c-rewriting of the initial CQ $\boldsymbol{q}(\boldsymbol{x})$.

Example 7. For the set `Res` constructed in Example 6, Stage 4 yields the following c-rewriting in $\mathcal{ALC}$ of the CQ $\boldsymbol{q}(x,y)$ from Example 4:

$$\big(x\colon \neg\exists R_1.(\exists R_3.B_4 \sqcap \exists R_5.B_3) \ \sqcap\ \forall R_2.(\neg B_2 \sqcup B_3),\\ y\colon B_1 \ \sqcap\ (\neg A \sqcup B_4) \ \sqcap\ (\neg\exists R_6.B_1 \sqcup \forall R_4.B_2)\big).$$

4 Implementation, Optimisations and Experiments

We have implemented the c-rewriting approach to answering CQs over ontologies in the DL reasoner FaCT++ [22]. FaCT++ is a tableau-based reasoner, which means that in order to check whether a given KB $\mathcal{K}$ is consistent, it constructs a so-called *completion graph* whose nodes include all the individuals from $\mathcal{K}$ labelled with concept expressions.

Given a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and a CQ $\boldsymbol{q}(\boldsymbol{x})$, FaCT++ first tests $\mathcal{K}$ for consistency and keeps in memory the 'deterministic' part $\mathfrak{D}$ of its completion graph obtained by applying deterministic tableau rules (which are always applied before non-deterministic ones in FaCT++). This will save a lot of reasoning time in later stages. Next, it checks whether $\boldsymbol{q}(\boldsymbol{x})$ satisfies the conditions of Theorem 1 and, if this is the case, constructs its c-rewriting $(x_1\colon C_1, \dots, x_n\colon C_n)$. Now, for every n-tuple of $\boldsymbol{a} = (a_1, \dots, a_n)$ of individuals from $\mathcal{A}$, it adds the concept C_i to the label of a_i, for $1 \le i \le n$, in the completion graph $\mathfrak{D}$ and runs the tableau algorithm to check whether the resulting tableau is closed. If it is, which means that the KB $\mathcal{K} \cup \{a_1\colon C_1, \dots, a_n\colon C_n\}$ is inconsistent, the tuple $\boldsymbol{a}$ is returned as a certain answer to $\boldsymbol{q}$ over $\mathcal{K}$.

The benefit of such an implementation over the 'reasoner as a black box' approach is that every time KB consistency checking starts from $\mathfrak{D}$ rather than from scratch. As the size of a typical CQ (and hence of its c-rewriting) is negligibly smaller than the size of an ABox, only a small part of the completion graph requires rebuilding. However, iterating over all tuples of individuals from $\mathcal{A}$ is still a very time consuming task. Below, we describe a few optimisations that have been implemented to reduce the search space.

4.1 Optimisations

We call an n-tuple of concepts $(E_1, \dots, E_n)$ an *upper bound* for a CQ $\boldsymbol{q}(\boldsymbol{x})$ if $\mathcal{K} \models \boldsymbol{q}(\boldsymbol{a})$ implies $\mathcal{K} \models a_i\colon E_i$, $1 \le i \le n$, for any KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and any n-tuple $\boldsymbol{a}$ of individuals from $\mathcal{A}$. The concept E_i is called an *upper bound* for x_i in $\boldsymbol{q}(\boldsymbol{x})$. If $(E_1, \dots, E_n)$ is an upper bound for $\boldsymbol{q}(\boldsymbol{x})$, then we can clearly restrict the search for answers to $\boldsymbol{q}(\boldsymbol{x})$ in $\mathcal{K}$ to the tuples $\boldsymbol{a}$ for which $\mathcal{K} \models a_i\colon E_i$, $1 \le i \le n$. For example, if $\boldsymbol{q}(\boldsymbol{x})$ contains atoms $A_i(x_i)$, $1 \le i \le n$, then $(A_1, \dots, A_n)$ is an upper bound for $\boldsymbol{q}(\boldsymbol{x})$. If $\boldsymbol{q}(\boldsymbol{x})$ contains an atom $R(x_1, z)$, then $(\exists R.\top, \top, \dots, \top)$ is another upper bound for $\boldsymbol{q}(\boldsymbol{x})$.

We are now in a position to describe our optimisations.

Rolling-up elimination. Suppose that the $\mathcal{ELIO}$-concept C built in Stage 2 has a subexpression of the form[2] $\{y_j\} \sqcap F$, where F is an $\mathcal{ELI}$-concept (without nominals). Then clearly F can be taken as an upper bound for y_j in $\boldsymbol{q}'(\boldsymbol{y})$ and, consequently, for the variable $\sigma(y_j)$ in $\boldsymbol{q}(\boldsymbol{x})$. Moreover, it is now safe to remove F from the $\mathcal{ELIO}$-concept C, that is, replace $\{y_j\} \sqcap F$ with $\{y_j\}$ in C, which simplifies the subsequent reasoning.

Query approximation. Here we build a stronger upper bound, but do not alter the $\mathcal{ELIO}$-concept C. Recall that, in Stage 2, we constructed an $\mathcal{ELIO}$-concept, say H_1, by traversing the graph of $\boldsymbol{q}'(\boldsymbol{y})$ starting from y_1. Let us construct $\mathcal{ELIO}$-concepts H_j, for $1 \le j \le m$, similarly but starting from y_j. Now we replace in H_j every nominal $\{y\}$ with the conjunction $\bigsqcap\{A \mid A(x_i) \in \boldsymbol{q}\}$, where $x_i := \sigma(y)$ and the empty conjunction is understood as $\top$. It is not hard to see that the resulting tuple of $\mathcal{ELI}$-concepts $(F_1, \dots, F_m)$ forms an upper bound for $\boldsymbol{q}'(\boldsymbol{y})$. An upper bound $(E_1, \dots, E_n)$ for the original CQ $\boldsymbol{q}(\boldsymbol{x})$ can be obtained by taking $E_i := \bigsqcap\{F_j \mid \sigma(y_j) = x_i\}$.

Avoiding search. Suppose that $(x\colon B, x_1\colon C_1, \dots, x_n\colon C_n)$ is a c-rewriting of a CQ $\boldsymbol{q}(x, x_1, \dots, x_n)$, where B is a *fresh* concept name. Fix any tuple $\boldsymbol{a} = (a_1, \dots, a_n)$ of individuals from $\mathcal{A}$. In order to find all answers to $\boldsymbol{q}$ of the form $(b, \boldsymbol{a})$, we would normally run, for every individual b from $\mathcal{K}$, the tableau algorithm on the KB $\mathcal{K} \cup \{b\colon B, a_1\colon C_1, \dots, a_n\colon C_n\}$ to check whether it is inconsistent. We can optimise this procedure as follows. Initially, the set of candidates for b is taken to be $G := \mathsf{ind}(\mathcal{A})$. We run the tableau algorithm only once on the KB $\mathcal{K} \cup \{a_1\colon C_1, \dots, a_n\colon C_n\}$ and, for every complete clash-free tableau for it, we remove from G all individuals b whose labels in M do not include $\neg B$. After that[3] we return the tuples $(b, \boldsymbol{a})$, for all $b \in G$, as answers. A similar optimisation is applied to a c-rewriting of the form $(x\colon \neg B, x_1\colon C_1, \dots, x_n\colon C_n)$.

4.2 Evaluation

We tested our implementation of query answering via c-rewriting with FaCT++ using the testbed described in [17]. It contains an extension of the LUBM ontology, an ABox generator and 11 CQs. For our tests, we generated an ABox for 1 university; it contains 17,138 individuals and 76,039 assertions. The tests were performed on a Mac OS X laptop with 3.06 GHz processor and 8 Gb of RAM. On this machine, the consistency check for the whole KB takes 2.3 seconds, while each of the subsequent checks for answer candidates takes about 0.2 ms.

The results of experiments are shown in the table below, which gives the number of tuples to be checked in order to answer the 11 CQs from [17]. We ran four series of tests: the first one without optimisations, the second one with the

[2] We assume that conjunctions are stored as sets of conjuncts, so $\{y_j\}$ and F are not necessarily adjacent conjuncts, but rather 'belong' to the same conjunction.

[3] For better performance, the implemented algorithm is even more involved: for instance, it uses the information whether $\neg B$ was added to the label of a node deterministically or non-deterministically. We omit details because of the space limit.

rolling-up optimisation, the third with both rolling-up and query approximation, and the fourth series with all three optimisations.

method	q1	q2	q3	q4	q5	q6	req1	req2	req3	req4	req5
no optim.	0.3G	0.3G	17k	0.3G	17k	0.3G	17k	0.3G	5T	0.3G	17k
rolling-up	4.2M	2.8M	536	8	6k	0.9M	0	13.5M	6.8G	10M	8k
r+q	4M	2.8M	507	8	3k	0	0	0.8M	2.2G	16k	0
r+q+a	4M	2.8M	507	1	3k	0	0	536	2.2G	1k	0

The first row in the table is clear: as the ABox contains about 17k individuals, the number of tuples to be checked is $17{,}138^n$, where n is the number of free variables in the CQ. The 'rolling-up' row already shows a significant reduction in the number of answer candidates. An interesting bit is the CQ

$$\texttt{req1}(x) = \exists y, z\,(\mathit{worksFor}(x, y) \land \mathit{affiliatedOrganizationOf}(y, z)),$$

which is c-rewritten to $(x\colon \forall \mathit{worksFor}.\forall \mathit{affiliatedOrganizationOf}.\bot)$, and there are no existential restrictions or property assertions in the KB that involve the role *affiliated Organization Of*. As a result, the set of individuals to choose from is empty. The 'r+q' row shows that in some cases (e.g., `q6` and `req5`) the answer set becomes empty as there are no instances of the *upper bound* concepts. Note also a significant reduction in the number of answer candidates for `req4`. The 'r+q+a' row shows that 3 out of 11 CQs are of the form suitable for the avoiding search optimisation. For example, the CQ

$$\texttt{req2}(x, y) = (\mathit{Person}(x) \land \mathit{teacherOf}(x, y) \land \mathit{Course}(y))$$

leads to the rolled-up c-rewriting $(x\colon \forall \mathit{teacherOf}.\neg B, y\colon B)$ with B a fresh concept name. By avoiding search for y, the number of checks is reduced by 3 orders of magnitude, as there are only 1600 instances of the concept *Course* in the KB.

5 Conclusions

In this paper, we have presented a novel c-rewriting algorithm that reduces the problem of answering conjunctive queries over ontologies to knowledge base consistency checking. The CQs allowed by the algorithm must contain no cycles that involve only existentially quantified variables (note that all the 11 CQs from [17] satisfy this condition). The allowed KBs are formulated in any decidable DL. We integrated the c-rewriting algorithm into the DL reasoner FaCT++, together with a number of optimisations. Our first experiments show that the c-rewriting approach with FaCT++ works reasonably well for CQs with one answer variable: it takes under one second to find all answers to such CQs in the experiments described above. Answering CQs with two variables may take up to 15 minutes. The experiments also demonstrate that even simple optimisations can drastically reduce the number of answer candidates. Apart from implementing and fine-tuning new optimisation techniques, we plan to characterise those CQs that are not c-rewritable ([12] characterises *modal definability*, not c-rewritability of CQs). Another interesting question is whether c-rewritings with number restrictions (i.e., in $\mathcal{ALCQ}$ or $\mathcal{ALCIQ}$) can extend the class of c-rewritable CQs.

References

1. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The *DL-Lite* family and relations. J. of Artificial Intelligence Research 36, 1–69 (2009)
2. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. J. of Automated Reasoning 39(3), 385–429 (2007)
3. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: View-based query answering in description logics: Semantics and complexity. J. of Computer and System Sciences 78(1), 26–46 (2012)
4. Eiter, T., Ortiz, M., Simkus, M.: Conjunctive query answering in the description logic $\mathcal{SH}$ using knots. J. of Computer and System Sciences 78(1), 47–85 (2012)
5. Eiter, T., Ortiz, M., Simkus, M., Tran, T.K., Xiao, G.: Query rewriting for Horn-$\mathcal{SHIQ}$ plus rules. In: Hoffmann, J., Selman, B. (eds.) Proc. of the 26th Nat. Conf. on Artificial Intelligence (AAAI 2010). pp. 726–733. AAAI Press (2012)
6. Glimm, B., Kazakov, Y., Lutz, C.: Status $\mathcal{QIO}$: An update. In: Rosati, R., Rudolph, S., Zakharyaschev, M. (eds.) Proc. of the 24th Int. Workshop on Description Logics (DL 2011). CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)
7. Glimm, B., Lutz, C., Horrocks, I., Sattler, U.: Conjunctive query answering for the description logic $\mathcal{SHIQ}$. J. of Artificial Intelligence Research 31, 157–204 (2008)
8. Goranko, V., Vakarelov, D.: Elementary canonical formulae: extending Sahlqvist's theorem. Annals of Pure and Applied Logic 141(1–2), 180–217 (2006)
9. Halaschek-Wiener, C., Parsia, B., Sirin, E., Kalyanpur, A.: Description logic reasoning for dynamic ABoxes. In: Parsia, B., Sattler, U., Toman, D. (eds.) Proc. of the 19th Int. Workshop on Description Logics (DL 2006). CEUR Workshop Proceedings, vol. 189. CEUR-WS.org (2006)
10. Hustadt, U., Motik, B., Sattler, U.: Reasoning in description logics by a reduction to disjunctive Datalog. J. of Automated Reasoning 39(3), 351–384 (2007)
11. Kikot, S.: An extension of Kracht's theorem to generalized Sahlqvist formulas. Journal of Applied Non-Classical Logics 19(2), 227–251 (2009)
12. Kikot, S., Zolin, E.: Modal definability of first-order formulas with free variables and query answering. Journal of Applied Logic 11(2), 190–216 (2013)
13. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in *DL-Lite*. In: Lin, F., Sattler, U., Truszczynski, M. (eds.) Proc. of the 12th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2010). pp. 247–257. AAAI Press (2010)
14. Lutz, C., Toman, D., Wolter, F.: Conjunctive query answering in the description logic $\mathcal{EL}$ using a relational database system. In: Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI 2009). pp. 2070–2075. AAAI Press (2009)
15. Lutz, C.: The complexity of conjunctive query answering in expressive description logics. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Proc. of the 4th Int. Joint Conf. on Automated Reasoning (IJCAR 2008). LNAI, vol. 5195, pp. 179–193. Springer (2008)
16. Lutz, C., Areces, C., Horrocks, I., Sattler, U.: Keys, nominals, and concrete domains. J. of Artificial Intelligence Research 23, 667–726 (2004)
17. Lutz, C., Seylan, I., Toman, D., Wolter, F.: The combined approach to OBDA: taming role hierarchies using filters. In: Fokoue, A., Liebig, T., Goodman, E., Weaver, J., Urbani, J., Mizell, D. (eds.) Joint Workshop on Scalable and High-Performance Semantic Web Systems (SSWS+HPCSW 2012). pp. 16–31. CEUR-WS.org (2012)

18. Ortiz, M., Calvanese, D., Eiter, T.: Data complexity of query answering in expressive description logics via tableaux. J. of Automated Reasoning 41(1), 61–98 (2008)
19. Ortiz, M., Rudolph, S., Simkus, M.: Query answering in the Horn fragments of the description logics $\mathcal{SHOIQ}$ and $\mathcal{SROIQ}$. In: Walsh, T. (ed.) Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI 2011). pp. 1039–1044. AAAI Press (2011)
20. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. J. on Data Semantics X, 133–173 (2008)
21. Rosati, R.: On conjunctive query answering in $\mathcal{EL}$. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Turhan, A.Y., Tessaris, S. (eds.) Proc. of the 20th Int. Workshop on Description Logics (DL 2007). CEUR Workshop Proceedings, vol. 250. CEUR-WS.org (2007)
22. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Furbach, U., Shankar, N. (eds.) Proc. of the 3rd Int. Joint Conf. on Automated Reasoning (IJCAR 2006). Lecture Notes in Artificial Intelligence, vol. 4130, pp. 292–297. Springer (2006)
23. Zolin, E.: Query answering based on modal correspondence theory. In: Proc. of the 4th "Methods for modalities" Workshop (M4M-4). pp. 21–37. m4m.loria.fr (2005)
24. Zolin, E.: Modal logic applied to query answering and the case for variable modalities. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Turhan, A.Y., Tessaris, S. (eds.) Proc. of the 20th Int. Workshop on Description Logics (DL 2007). CEUR Workshop Proceedings, vol. 250. CEUR-WS.org (2007)