Benchmarking Ontology-Based Query Rewriting Systems

Martha Imprialou\textsuperscript{1}, Giorgos Stoilos\textsuperscript{2} and Bernardo Cuenca Grau\textsuperscript{1}

\textsuperscript{1}Department of Computer Science, University of Oxford, Oxford, UK
\texttt{martha.imperialou@balliol.ox.ac.uk}
\texttt{bernardo.cuenca.grau@cs.ox.ac.uk}

\textsuperscript{2}School of Electrical and Computer Engineering, National Technical University of Athens, Greece
\texttt{gstoil@image.ece.ntua.gr}

Abstract

Query rewriting is a prominent reasoning technique in ontology-based data access applications. A wide variety of query rewriting algorithms have been proposed in recent years and implemented in highly optimised reasoning systems. Query rewriting systems are complex software programs; even if based on provably correct algorithms, sophisticated optimisations make the systems more complex and errors become more likely to happen. In this paper, we present an algorithm that, given an ontology as input, synthetically generates “relevant” test queries. Intuitively, each of these queries can be used to verify whether the system correctly performs a certain set of “inferences”, each of which can be traced back to axioms in the input ontology. Furthermore, we present techniques that allow us to determine whether a system is unsound and/or incomplete for a given test query and ontology. Our evaluation shows that most publicly available query rewriting systems are unsound and/or incomplete, even on commonly used benchmark ontologies; more importantly, our techniques revealed the precise causes of their correctness issues and the systems were then corrected based on our feedback. Finally, since our evaluation is based on a larger set of test queries than existing benchmarks, which are based on hand-crafted queries, it also provides a better understanding of the scalability behaviour of each system.

Introduction

An important application of ontologies is data access, where an ontology provides the knowledge describing the meaning of application’s data as well as the vocabulary used to formulate user queries. In this setting, query languages are often based on conjunctive queries (CQs), and the answers to queries reflect both the application’s data and the knowledge captured by the ontology (Calvanese et al. 2007; Poggi et al. 2008; Glimm et al. 2007; Lutz, Toman, and Wolter 2009; Ortiz, Calvanese, and Eiter 2006).

The need for efficient query answering in ontology-based data access applications has motivated the development of (families of) lightweight ontology languages, such as DL-Lite (Calvanese et al. 2007) (which underpins the OWL 2 QL profile (Motik et al. 2009)), DLP (which underpins the OWL 2 RL profile), as well as (fragments of) datalog\textsuperscript{±} (Cali et al. 2010). These languages are tailored such that query answering becomes tractable in data complexity (and even in logarithmic space in the case of the logics underlying OWL 2 QL).

In this setting, query rewriting is a common reasoning technique of choice. Intuitively, a rewriting of a query \( Q \) w.r.t. an ontology \( T \) is another query (typically a union of CQs or a datalog query) that captures the information in \( T \) relevant for answering \( Q \) w.r.t. \( T \) and arbitrary data. Due to the growing interest in ontology-based data access, many rewriting algorithms have been proposed in recent years and implemented in (both commercial and non-commercial) reasoning systems. Examples of such systems include QuOnto (Acciari et al. 2005), Requiem (Pérez-Urbina, Horrocks, and Motik 2009), Presto (Rosati and Almatelli 2010), Nyaya (Gottlob, Orsi, and Pieris 2011), Rapid (Chortaras, Trivela, and Stamou 2011), and IQAROS.\textsuperscript{1}

Despite being based on provably correct algorithms, sophisticated optimisations needed in practice make these query rewriting systems rather complex and error-prone software programs. Consequently, both system and application developers would greatly benefit from practical and reliable benchmarking approaches to query rewriting.

To the best of our knowledge, little work has been done so far towards the systematic benchmarking of ontology-based query rewriting systems, and developers have relied mostly on “hand-crafted” ontologies and queries for evaluating their systems and comparing them to others. An example of a popular benchmark for query rewriting, which was first used in (Pérez-Urbina, Horrocks, and Motik 2009), consists of nine ontologies and five hand-crafted queries for each of them; systems are evaluated regarding the size of the computed rewritings and the rewriting computation time.

Existing benchmarks, however, have several important limitations. First, they are ad hoc: test queries are manually generated for each specific ontology using the developers’ personal knowledge about the ontologies in the benchmark. Second, given a test query \( Q \) and ontology \( T \), it is currently not possible to automatically check whether the relevant systems are sound (i.e., they compute only actual query answers to \( Q \) w.r.t. \( T \) and arbitrary data) and complete (i.e., they compute all answers to \( Q \) w.r.t. \( T \) and arbitrary data).

In this paper, we present a novel approach for addressing

\textsuperscript{1}http://code.google.com/p/iquaros/
these important limitations. First, we present an algorithm that synthetically generates test queries for an ontology \( T \). Our algorithm is generic, in the sense that it is applicable to a wide range of ontologies and, in particular, to (Horn) ontologies that can be captured by existential rules (Baget et al. 2011; Calì et al. 2010). Furthermore, our algorithm is fully automatic and hence does not require experts’ involvement. Most importantly, each generated query is relevant to \( T \), in the sense that it can be used to verify whether a system correctly performs a certain set of “inferences”, each of which can be traced back to axioms in \( T \). Second, we present techniques that allow us to determine whether a given system is unsound and/or incomplete for given \( Q \) and \( T \); these techniques are easy to implement and in many cases help detect errors with marginal manual effort.

Our evaluation shows that most publicly available query rewriting systems are unsound and/or incomplete, even on commonly used benchmark ontologies; most importantly, our techniques revealed the precise causes of their unsoundness and incompleteness issues, and as a result some of the evaluated systems were actually corrected by their developers based on our feedback. Finally, since our collection of tests queries is more “exhaustive” than those in existing benchmarks, our evaluation provides further insights into the scalability behaviour of each system.

**Preliminaries**

We use standard notions of first-order constants, (free and bound) variables, function symbols, terms, substitutions, predicates, atoms, (ground) formulae, and sentences. A fact is a ground atom, and an instance is a finite set of facts. For \( \phi \) a formula, with \( \phi(\vec{x}) \) we denote that \( \vec{x} \) are the free variables of \( \phi \), while for \( \sigma \) a substitution, \( \phi[\sigma] \) is the result of applying \( \sigma \) to \( \phi \). Satisfiability and entailment are defined as usual.

**Ontologies**

We use description logics in the wider framework of first order logic, and identify a DL TBox \( T \) with a finite set of first order sentences. Specialised DL syntax will sometimes be used in examples, and we assume that the reader is familiar with the basics of such syntax (Baader et al. 2002).

**Existential Rules**

An existential rule (Baget et al. 2011; Calì et al. 2010) is a sentence of the form

\[
\forall \vec{x}. \forall \vec{z}. [\phi(\vec{x}, \vec{z}) \rightarrow \exists \vec{y}. \psi(\vec{x}, \vec{y})]
\]

(1)

where \( \phi(\vec{x}, \vec{z}) \) and \( \psi(\vec{x}, \vec{y}) \) are conjunctions of function-free atoms and \( \vec{x} \) and \( \vec{z} \) are pair-wise disjoint. Function \( \phi \) is the body, function \( \psi \) is the head, and universal quantifiers are often omitted. Note that, by definition, existential rules are safe—that is, all variables in \( \vec{x} \) occur both in the body and the head. If \( \vec{y} \) is empty, the rule is datalog. Finally, the instantiation of a datalog rule \( r \) w.r.t. a substitution \( \sigma \) mapping all variables in \( r \) to constants is the instance \( I_\sigma \) consisting all facts \( B\sigma \) with \( B \) a body atom in \( r \). If \( \sigma \) is an injective mapping, then \( I_\sigma \) is an injective instantiation of \( r \).

Many popular DLs (e.g., those underlying the OWL 2 profiles (Motik et al. 2009)) can be captured by existential rules.

**Queries**

A query \( Q \) is a finite set of sentences containing a distinct query predicate \( Q \). A tuple of constants \( \vec{a} \) is an answer to \( Q \) w.r.t. TBox \( T \) and instance \( I \) if the arity of \( \vec{a} \) agrees with the arity of \( Q \) and \( T \cup I \cup Q \models Q(\vec{a}) \). We denote with \( \text{cert}(Q, T \cup I) \) the answers to \( Q \) w.r.t. \( T \cup I \). A query \( Q \) is a union of conjunctive queries (UCQ) if it is a set of datalog rules containing \( Q \) in the head but not in the body. In this case, we sometimes abuse notation and use \( Q \) to denote the head atom, rather than the head predicate. A UCQ is a conjunctive query (CQ) if it has exactly one rule.

**Query Rewriting**

Intuitively, a rewriting of \( Q \) w.r.t. \( T \) is another query that captures all the information from \( T \) relevant for answering \( Q \) over an arbitrary instance \( I \) (Calvanese et al. 2007; P`erez-Urbina, Motik, and Horrocks 2010; Gottlob, Orsi, and Pieris 2011). UCQs and datalog are common target languages for query rewriting; furthermore, virtually all TBoxes to which existing rewriting techniques are applicable can be captured by existential rules.

**Definition 1.** A datalog rewriting of a CQ \( Q \) w.r.t. \( T \) is a tuple \( \langle R_D, R_Q \rangle \) with \( R_D \) a set of datalog rules not mentioning \( Q \) and \( R_Q \) a UCQ with query predicate \( Q \) whose body atoms mention only predicates from \( T \), and where for each \( I \) using only predicates from \( T \) we have

\[
\text{cert}(Q, T \cup I) = \text{cert}(R_Q, R_D \cup I).
\]

The rewriting \( \langle R_D, R_Q \rangle \) is a UCQ rewriting if \( R_D = \emptyset \).

**The Chase**

Given a set of existential rules \( R \) and an instance \( I \), CQ answering over \( R \cup I \) can be characterised using the chase: a technique that computes a (possibly infinite) set chase \( \text{chase}(R \cup I) \) of facts implied by \( R \) and \( I \) in a forward-chaining manner; the result is a universal model over which the CQ can then be evaluated. Although many chase procedures have been proposed in the literature (Fagin et al. 2005; Deutsch, Nash, and Remmel 2008; Marnette 2009), all variants of the chase satisfy the following basic properties:

- \( \text{cert}(Q, R \cup I) = \text{cert}(Q, \text{chase}(R \cup I)) \) for all \( Q, R, I \).
- For each ground fact \( \alpha \) mentioning only constants and functions from \( R \cup I, R \cup I \models \alpha \) if \( \alpha \in \text{chase}(R \cup I) \).

**Query Rewriting Systems**

Many optimised query rewriting systems for various ontology languages have been developed in recent years (Calvanese et al. 2007; Rosati and Almatteli 2010; Chortaras, Trivela, and Stamou 2011; Gottlob, Orsi, and Pieris 2011). To abstract from the implementation details, and focus only on the properties of the system that we want to test for, we introduce an abstract notion of a query rewriting system.

**Definition 2.** A query rewriting system \( \text{rew} \) for a DL \( L \) is a computable function that, for each TBox \( T \in L \) and each CQ \( Q \) with head predicate \( Q \) computes in a finite number of steps a tuple \( \text{rew}(Q, T) = \langle R_D, R_Q \rangle \) where \( R_D \) is a set of datalog rules not containing \( Q \) and \( R_Q \) is a UCQ with head predicate \( Q \) whose body atoms mention only predicates from \( T \). The system \( \text{rew} \) is \( (Q, T) \)-sound if \( \text{cert}(R_Q, R_D \cup I) \subseteq \text{cert}(Q, T \cup I) \) for every instance \( I \) containing only predicates from \( T \). It is \( (Q, T) \)-complete if \( \text{cert}(Q, T \cup I) \subseteq \text{cert}(R_Q, R_D \cup I) \) for every instance \( I \) containing only predicates from \( T \).
Most query rewriting systems are based on algorithms whose behaviour on input $\mathcal{T}$ and $Q$ can be characterised by the application of the following two steps:

1. **Pre-processing**, where the input TBox $\mathcal{T}$ is transformed into a set $\mathcal{R}_\mathcal{T}$ of existential rules.

2. **Query rewriting**, where a calculus (often a variant of resolution) is used to derive $\mathcal{R}_D$ and $\mathcal{R}_Q$ from $\mathcal{R}_\mathcal{T}$ and $Q$.

Consequently, two main sources of implementation errors for rewriting systems can be identified, namely those caused by either an incorrect pre-processing, or an incorrect implementation of the calculus.

**Example 3.** Let $\mathcal{T}$ consist of the following DL axiom:

$$\text{Student} \equiv \text{Person} \cap \exists \text{enrolled.Course} \quad (2)$$

TBox $\mathcal{T}$ can be translated into the following set $\mathcal{R}$ of equivalent existential rules:

$$\text{Student}(x) \rightarrow \text{Person}(x) \quad (3)$$

$$\text{Person}(x) \land \text{enrolled}(x, y) \land \text{Course}(y) \rightarrow \text{Student}(x) \quad (4)$$

$$\text{Student}(x) \rightarrow \exists y. (\text{enrolled}(x, y) \land \text{Course}(y)) \quad (5)$$

Some rewriting systems that would accept $\mathcal{T}$ as a valid input do not handle qualified existential restrictions (i.e., concepts such as $\exists$enrolled.Course). If pre-processing $\mathcal{T}$ naively, one such system $\text{rew}_1 \equiv \text{ignore}$ the conjunction involving sub-concept $\exists$enrolled.Course and translate $\mathcal{T}$ into $\mathcal{R}_\mathcal{T}$ consisting only of rule (3) and rule $\text{Person}(x) \rightarrow \text{Student}(x)$.

Such system would be unsound for $\mathcal{T}$: given query $\mathcal{Q}_1 = \text{Student}(x) \rightarrow Q(x)$, it would compute the UCQ rewriting $\langle \emptyset, \{ Q_1, Q'_1 \} \rangle$, with $Q'_1 = \text{Person}(x) \rightarrow Q(x)$, which for instance $I_1 = \{ \text{Person}(c) \}$ leads to the incorrect answer $c$. Consider now a system $\text{rew}_2$ that can handle qualified existential restrictions, but which, due to an implementation error, translates $\mathcal{T}$ into $\mathcal{R}_\mathcal{T}$ consisting of rules (3), (4), and rule $\text{Student}(x) \rightarrow \exists y. (\text{enrolled}(x, y))$, thus leaving out conjunction $\text{Course}(y)$ in rule (5). Such system is incomplete for $\mathcal{T}$, as witnessed by $\mathcal{Q}_2 = \text{enrolled}(x, y) \land \text{Course}(y) \rightarrow Q(x)$ and $I_2 = \{ \text{Student}(d) \}$.

As described later on in our evaluation section, the aforementioned normalisation issues are related to actual errors we detected in the Rapid system (Chortaras, Trivedi, and Stamou 2011) using our approach, and which were not revealed by existing benchmarks.

Finally, as already mentioned, state of the art implementations of rewriting calculi are highly optimised. As described later on, our approach also allowed us to detect several correctness issues in implemented optimisations; these include errors in ordering criteria for query atoms in IQAROS, and issues with subsumption optimisations in Requiem (Pérez-Urbina, Horrocks, and Motik 2009), among others.

Example 3 suggests that implementation errors in both pre-processing and rewriting steps can be effectively “uncovered” by a suitable choice of test queries and subsequent analysis of the systems’ outputs for such relevant queries. In what follows, we present an algorithm for computing test queries that are relevant to a TBox. We then present techniques for checking systems’ correctness for a test query.

**Algorithm 1 QueryGeneration($\mathcal{R}, bound, \Sigma$)**

input: Existential rules $\mathcal{R}$; integer bound.

1: $I_0 := \emptyset$ and root := $\emptyset$
2: for all $r \in \mathcal{R}$ do
3: \[ I'_r := \text{injective instantiation of } r \text{ to fresh const. } \vec{a} \]
4: \[ I_0 := I_0 \cup I'_r \]
5: root := root $\cup \{ \vec{a} \}$
6: end for
7: $I_c := I_0$
8: repeat
9: \[ I_c := I_c \cup \text{applyChaseRule}(\mathcal{R}, I_c) \]
10: until terminate($I_c, bound$)
11: $I := I_c \setminus I_0$ and $\mathcal{CQ} := \emptyset$
12: for all $\vec{a} \in \text{root}$ do
13: \[ \text{for all } I' \in \text{paths}(\vec{a}, I, \Sigma) \text{ do} \]
14: \[ \text{Let } \sigma \text{ map each constant in } I' \text{ to fresh variable} \]
15: \[ \mathcal{CQ} := \mathcal{CQ} \cup \{ \bigwedge_{P(\vec{b}) \in I'} P(\sigma(\vec{b})) \rightarrow Q(\sigma(\vec{a})) \} \]
16: end for
17: end for
18: return $\mathcal{CQ}$

**Test Query Generation**

Virtually all TBoxes to which state of the art rewriting systems are applicable can be captured by existential rules containing only unary and binary predicates. Thus, we only consider such TBoxes $\mathcal{T}$ in this section. Translation of $\mathcal{T}$ into existential rules $\mathcal{R}$ can be performed using well-known structural transformations, which might introduce “fresh” symbols $\Sigma$ (e.g., see (Motik, Shearer, and Horrocks 2009)).

Algorithm 1 can then be used to compute test queries from $\mathcal{R}$. The algorithm directly exploits the chase technique, and works in three main stages:

1. **Chase initialisation**, via instantiation of body atoms in $\mathcal{R}$.
2. **Chase expansion**, up to a pre-defined bound.
3. **Query generation**, by “navigating” the expanded chase while replacing ground terms by fresh variables.

**Chase initialisation** For each rule $r \in \mathcal{R}$, Algorithm 1 instantiates all body atoms using fresh constants, which are marked as *root constants*, thus constructing an instance $I_0$ that “triggers” the application of each rule in $\mathcal{R}$ (lines 2–6). For the rules in Example 3 we obtain the following initial instance, where root $= \{ a, (b, c) \}$:

\[ I_0 = \{ \text{Student}(a), \text{Person}(b), \text{enrolled}(b, c), \text{Course}(c) \} \]

**Chase expansion** A chase procedure is then used to materialise new implied facts from existing ones by forward chaining application of rules in $\mathcal{R}$ (lines 8–10). Intuitively, when initialised with $I_0$, the properties of the chase ensure that each rule application represents an inference that is relevant to derive some answer to some query $\mathcal{Q}$ w.r.t. some input instance (and for the given, fixed, TBox $\mathcal{T}$).

\[ \text{Note that rules (3) and (5) share the same body, so it suffices to instantiate one of them.} \]
Since the chase might not terminate, the algorithm accepts an integer bound (e.g., maximum number of generated facts), which can be used as a termination condition. Alternatively, one can use any of the available acyclicity conditions for existential rules (e.g., (Fagin et al. 2005; Marnette 2009)), which analyse the information flow between the rules to ensure that no cyclic generation of fresh terms occurs. If \( \mathcal{R} \) satisfies any such condition, the chase terminates, and one can dispense with the input bound. In our example, chase expansion would lead to the following instance \( I_c \), where \( a_1 \) and \( b_1 \) are fresh constants generated by the application of rule (5).

\[
I_c = I_0 \cup \{ \text{Person}(a), \text{enrolled}(a, a_1), \text{Course}(a_1) \\
\text{Student}(b), \text{enrolled}(b, b_1), \text{Course}(b_1) \}
\]

**Query generation** Instance \( I = I_c \setminus I_0 \) contains all facts that have been derived from \( I_0 \) during chase expansion. The properties of the chase ensure that \( I \) is forest-shaped. More precisely, since \( I \) contains only unary and binary predicates, we can define \( G(I) \) as the graph whose nodes are the constants in \( I \) and which contains an (undirected) edge between \( c \) and \( d \) iff \( c \) and \( d \) occur together in a fact from \( I \); then \( G(I) \) consists of a set of disconnected trees each of which is rooted at an individual from root.

Algorithm 1 generates queries by traversing \( G(I) \) (lines 12–17) along “paths”. More precisely, paths \((\vec{a}, I, \Sigma)\) (see line 13) is the set of all instances \( \vec{I}' \subseteq I \) not containing predicates from \( \Sigma \) and s.t. \( G(\vec{I}') \) is a path in \( G(I) \) involving a constant from \( \vec{a} \). Each such \( \vec{I}' \) is then transformed into a CQ by mapping each constant to a fresh variable s.t. root constants are mapped to answer variables in the head of the query. Clearly, all queries mentioned in Example 3 would be generated by Algorithm 1 for input rules (3)–(5).

Note that, in the worst-case, Algorithm 1 might generate exponentially many queries w.r.t. the size of the input. As shown in our evaluation, however, the number of generated queries for commonly-used benchmark ontologies is relatively modest, and they can be computed in just a few seconds. Finally, note that relevant test queries could also be generated from \( I \) by considering instances \( \vec{I}' \subseteq I \) whose graph is a sub-tree of \( G(I) \), rather than simply a path; however, on the one hand, this results in a blowup in the number of queries in practice and, on the other hand, our evaluation suggests that path queries suffice for uncovering many errors in systems and for analysing and comparing their behaviour.

**Testing Correctness**

Algorithm 1 provides a flexible way for generating relevant test queries. In this section, we present practical techniques for testing soundness and completeness of a rewriting system for a given test query and a given TBox.

**Testing Soundness**

Assume that \( \text{rew}(Q) = (\mathcal{R}_D, \mathcal{R}_Q) \). Clearly, if we have \( \mathcal{T} \cup Q \models \mathcal{R}_D \cup \mathcal{R}_Q \), the properties of first-order logic entailment ensure that each answer to \( Q \) w.r.t. \( \mathcal{R}_D \cup \mathcal{R}_Q \) and an instance \( I \) is also an answer to \( Q \) w.r.t. \( \mathcal{T} \cup I \), and hence the query rewriting system \( \text{rew} \) is \( (Q, \mathcal{T}) \)-sound.

Proposition 4 shows that \( \mathcal{T} \cup Q \models \mathcal{R}_D \cup \mathcal{R}_Q \) can be checked using any reasoner that is sound and complete for checking entailment of a fact by \( \mathcal{T} \) and an instance. For standard DLs, example such reasoners include HermiT (Motik, Shearer, and Horrocks 2009) and Pellet (Sirin et al. 2007).

**Proposition 4.** Let \( \text{rew}(Q, \mathcal{T}) = (\mathcal{R}_D, \mathcal{R}_Q) \). Then, we have that \( \mathcal{T} \cup Q \models \mathcal{R}_D \cup \mathcal{R}_Q \) iff the following conditions hold:

1. \( \mathcal{T} \cup I_r \models H \sigma \) for each \( r \in \mathcal{R}_D \), where \( H \) is the head of \( r \) and \( I_r \) an injective instantiation.
2. \( \mathcal{T} \cup Q \cup I_r \models Q \sigma \) for each \( r \in \mathcal{R}_Q \), where \( Q \) is the head of \( r \) and \( I_r \) an injective instantiation.

**Proof.** Since \( Q \) does not occur in \( \mathcal{R}_D \), we have that \( \mathcal{T} \cup Q \models \mathcal{R}_D \cup \mathcal{R}_Q \) iff \( \mathcal{T} \models \mathcal{R}_D \) and \( \mathcal{T} \cup Q \models \mathcal{R}_Q \). Furthermore, it is easy to check that a set \( F \) of sentences entails a datalog rule \( r \) with head \( H \) iff \( F \cup I_r \models H \sigma \) for \( I_r \) an injective instantiation; thus, since \( \mathcal{R}_D \) and \( \mathcal{R}_Q \) consist of datalog rules, we have \( \mathcal{T} \models \mathcal{R}_D \) iff Condition 1 holds and \( \mathcal{T} \cup Q \models \mathcal{R}_Q \) iff Condition 2 holds. □

In practice, however, checking soundness of \( \text{rew} \) using a fully-fledged DL reasoner, as suggested by Proposition 4, has the drawback that the process is subject to implementation errors of the DL reasoner. Although this problem can be alleviated by performing the required entailment tests using several DL reasoners to detect possible discrepancies, an alternative is to use a chase procedure, such as the one we used for query generation (see Algorithm 1). Such procedure does not need to be optimised, and consequently its implementation can be rather simple and transparent.

The following proposition shows how to exploit the chase for checking \( (Q, \mathcal{T}) \)-soundness of \( \text{rew} \).

**Proposition 5.** Let \( \text{rew}(Q, \mathcal{T}) = (\mathcal{R}_D, \mathcal{R}_Q) \) and let \( \mathcal{R} \) be a set of existential rules such that \( \mathcal{T} \) and \( \mathcal{R} \) are logically equivalent. The following conditions imply that \( \text{rew} \) is \( (Q, \mathcal{T}) \)-sound:

1. \( H \sigma \in \text{chase}(\mathcal{R} \cup I_r) \) for each \( r \in \mathcal{R}_D \), where \( H \) is the head of \( r \) and \( I_r \) an injective instantiation.
2. \( \vec{a} \in \text{cert}(Q, \text{chase}(\mathcal{R} \cup I_r)) \) for each \( r \in \mathcal{R}_Q \), where \( Q \) is the head of \( r \) and \( I_r \) an injective instantiation mapping the variables in \( Q \) to \( \vec{a} \).

Furthermore, if Condition 2 does not hold, then \( \text{rew} \) is not \( (Q, \mathcal{T}) \)-sound.

**Proof.** To show the first claim in the proposition, it suffices to prove that Condition 1 (resp. Condition 2) in the proposition implies Condition 1 (resp. Condition 2) in Proposition 4. Let \( r \in \mathcal{R}_D \), and assume that \( H \sigma \in \text{chase}(\mathcal{R} \cup I_r) \) with \( I_r \) an injective instantiation; since \( H \sigma \) mentions only individuals from \( I_r \) (recall that rules are assumed to be safe), the properties of the chase (see preliminaries) ensure that \( \mathcal{R} \cup I_r \models H \sigma \); but then, since \( \mathcal{T} \models \mathcal{R} \), we have \( \mathcal{T} \cup I_r \models H \sigma \), as required. Finally, let \( r \in \mathcal{R}_Q \) and let \( \vec{a} \in \text{cert}(Q, \text{chase}(\mathcal{R} \cup I_r)) \), where \( I_r \) is an injective instantiation mapping the variables in \( Q \) to \( \vec{a} \). The properties of chase ensure that \( \vec{a} \in \text{cert}(Q, \mathcal{R} \cup I_r) \), which implies \( \mathcal{R} \cup Q \cup I_r \models Q \sigma \). Since \( \mathcal{T} \models \mathcal{R} \), we then have \( \mathcal{T} \cup Q \cup I_r \models Q \sigma \), as required.
To show proposition’s last claim, consider \( r \in R_Q \) and an injective instantiation \( I^*_{\sigma} \), where \( \bar{x} \) are the variables in \( Q \) and \( \bar{a} = \sigma(\bar{x}) \). Clearly, \( \bar{a} \in \text{cert}(R_Q, R_D \cup I^*_\sigma) \); furthermore, \( I^*_\sigma \) contains only predicates from \( T \). Next, assume \( \bar{a} \notin \text{cert}(Q, \text{chase}(R \cup I^*_\sigma)) \); then, the properties of the chase imply that \( \bar{a} \notin \text{cert}(Q, R \cup I^*_\sigma) \); finally, since \( R \models T \), we have \( \bar{a} \notin \text{cert}(Q, T \cup I^*_\sigma) \). Thus, \( \text{rew} \) is not \((Q, T)\)-sound. \( \Box \)

Note that Proposition 5 provides only a sufficient condition for \((Q, T)\)-soundness of \( \text{rew} \); indeed, it may be the case that Condition 1 fails (i.e., \( H \sigma \notin \text{chase}(R \cup I^*_\sigma) \)) for some \( r \in R_Q \), but \( \text{rew} \) is \((Q, T)\)-sound. In contrast, as shown, failure of Condition 2 for some \( r \in R_Q \) provides a counter-example for \((Q, T)\)-soundness. Thus, if \( \text{rew} \) is a UCQ (i.e., \( R_D = \emptyset \)), Proposition 5 provides both a necessary and sufficient condition for \((Q, T)\)-soundness.

**Example 6.** Consider TBox \( T \), rules \( R \), queries \( Q_1, Q'_1 \) and system \( \text{rew}_1 \) from Example 3. For \( Q_1 \) and \( T \), \( \text{rew}_1 \) computes \( \text{rew}_1(Q_1, T) = \{ \emptyset, \{ Q_1, Q'_1 \} \} \). Let \( I^*_{\sigma} = \{ \text{Person}(a) \} \) be an injective instantiation of \( Q^*_1 \) for \( \sigma = \{ x \rightarrow a \} \). Clearly, \( T \models R \) and \( a \notin \text{cert}(Q_1, \text{chase}(R \cup I^*_\sigma)) \). Thus, Proposition 5 implies that \( \text{rew}_1 \) is indeed unsound for \( Q_1 \) and \( T \). \( \Diamond \)

In practice, to ensure \((Q, T)\)-soundness of \( \text{rew} \) it suffices to check that Conditions 1 and 2 in Proposition 5 hold already for a subset of the corresponding chase, in which case chase termination does not need to be ensured. On the one hand, these conditions involve checking whether a particular fact can be found in the chase (or whether a particular answer can be derived from the chase); on the other hand, chase construction is monotonic (i.e., derived facts are never deleted), so if a fact can be found in (or an answer can be derived from) a finite subset of the chase, then the fact (or answer) is guaranteed to hold in the final chase.

To prove unsoundness, however, one needs to check failure of Condition 2, which requires the full expansion of \( \text{chase}(R \cup I^*_\sigma) \) for each relevant \( r \in R_Q \); thus, chase termination becomes an important issue. Our evaluation will show that these issues can be dealt with, and Proposition 5 can be used effectively to determine soundness and unsoundness in practice for typical benchmark ontologies.

### Testing Completeness

Our strategy for detecting sources of incompleteness is to look for and (semi)automatically explicate “disagreements” between query rewriting systems for given \( Q \) and \( T \). The following definition provides the required notions to compare systems’ outputs for completeness evaluation purposes.

**Definition 7.** Let \( \text{rew}_i(Q, T) = (R^i_D, R^i_Q) \) for \( i \in \{1, 2\} \). We say that \( \text{rew}_2 \text{ extends} \text{rew}_1 \) for \( Q \) and \( T \) if the following condition holds for each instance \( I \) mentioning only predicates from \( T \):

\[
\text{cert}(R^1_Q \cup R^1_D \cup I) \subseteq \text{cert}(R^2_Q \cup R^2_D \cup I) \quad (6)
\]

If \( \text{rew}_2 \text{ extends} \text{rew}_1 \) and the inclusion in Condition (6) is proper for some \( I \), we say that \( \text{rew}_2 \text{ strictly extends} \text{rew}_1 \).

Intuitively, if \( \text{rew}_2 \text{ strictly extends} \text{rew}_1 \), then it is “at least as complete as” \( \text{rew}_1 \) for the given \( Q \) and \( T \). Thus, when evaluating several systems in practice, if one of them extends all the others, we have strong evidence to believe that such system is \((Q, T)\)-complete. Furthermore, if a system \( \text{rew}_2 \) is \((Q, T)\)-sound and strictly extends \( \text{rew}_1 \) for \( Q \) and \( T \), we can conclude that \( \text{rew}_1 \text{ is incomplete for} Q \) and \( T \).

The following proposition provides sufficient conditions for checking whether \( \text{rew}_2 \text{ (strictly) extends} \text{rew}_1 \); hence, when used in combination with Proposition 5, it allows us to draw conclusions about systems’ completeness.

**Proposition 8.** Let \( \text{rew}_i(Q, T) = (R^i_D, R^i_Q) \) for \( i \in \{1, 2\} \).

If both of the following conditions hold, then \( \text{rew}_2 \text{ extends} \text{rew}_1 \) for \( Q \) and \( T \):

1. \( H \sigma \in \text{chase}(R^2_D \cup I^*_\sigma) \) for each \( r \in R^1_D \) with head \( H \), where \( I^*_\sigma \) is an injective instantiation.

2. \( Q \sigma \in \text{chase}(R^2_Q \cup I^*_\sigma) \) for each \( r \in R^1_Q \) with head \( Q \), where \( I^*_\sigma \) is an injective instantiation.

Furthermore, if \( R^1_D = R^2_D = \emptyset \), \( \text{rew}_2 \text{ strictly extends} \text{rew}_1 \) for \( Q \) and \( T \).

**Proof.** Assume that \( H \sigma \in \text{chase}(R^2_D \cup I^*_\sigma) \) for each \( r \in R^1_D \); the properties of chase ensure that \( R^2_D \cup I^*_\sigma \models H \sigma \) for each \( r \in R^1_D \); but then, since \( I^*_\sigma \) is an injective instantiation of \( r \), this implies that \( R^2_D \models R^1_D \). Similarly, if Condition 2 holds, then we have \( R^2_Q \models R^1_Q \). As a result, (6) holds by the properties of entailment, as required.

Finally, assume that \( R^1_D = R^2_D = \emptyset \) and \( r \in R^2_Q \) exists such that \( Q \sigma \notin \text{chase}(R^2_Q \cup I^*_\sigma) \) for \( I^*_\sigma \) an injective instantiation. The properties of the chase then ensure that \( R^1_Q \cup I^*_{\sigma} \notin Q \sigma \); since \( R^2_Q \cup I^*_\sigma \models Q \sigma \), and \( \text{rew}_2 \text{ extends} \text{rew}_1 \), the inclusion in (6) is indeed proper. \( \Box \)

Note that \( R^1_D \) and \( R^2_Q \) consist of datalog rules; thus, the chase computations involved in Proposition 8 are guaranteed to terminate. Furthermore, if we determine that \( \text{rew}_2 \text{ strictly extends} \text{rew}_1 \), we can track the application of chase rules in the corresponding chase expansion to detect the source of the systems’ disagreement in a (semi)automatic way. As described in the evaluation section, this allowed us to detect the sources of incompleteness in practice.

**Example 9.** Consider \( Q_2, T \) and \( \text{rew}_2 \) from Example 3 and let \( \text{rew}_2(Q_2, T) = (\emptyset, R^2_Q) \). Since \( \text{rew}_2 \) computes rewritings based on \( R^2_T \) in Example 3, we have \( Q'_2 \notin R^2_Q \) for \( Q'_2 = \text{Student}(x) \rightarrow Q(x) \). Let \( \text{rew}_3 \) be a sound system s.t. \( \text{rew}_3(Q_2, T) = (\emptyset, R^3_Q \cup \{ Q'_1 \}) \). \( \text{rew}_3 \) strictly extends \( \text{rew}_2 \) since \( Q(a) \notin \text{chase}(R^3_Q \cup I^*_\sigma) \) for \( I^*_\sigma = \{ \text{Student}(a) \} \); thus, \( \text{rew}_2 \text{ is not} (Q_2, T)\)-complete. \( \Diamond \)

### Experiments

We have implemented a test query generator (see Algorithm 1) that is applicable to TBoxes in the DL $\mathcal{ELHI}$.\(^3\) We then applied our query generator to the benchmark ontologies described in (Pérez-Urbina, Horrocks, and Motik 2009).

---

\(^3\)Available at http://code.google.com/p/sygenia/
For all test ontologies, except for A, AX, and S, the chase naturally terminated in just a few seconds. Instead of fixing an a priori integer bound for the “cyclic” ontologies (as required by Algorithm 1), our implementation annotates each skolem constant generated by the chase with the rules with existentials in the head that have been applied so far to generate them. If such a rule \( r \) is applicable to facts involving a skolem constant, but \( r \) has been used before in order to generate this particular skolem constant, then we detect a “cycle”; what we bound is the number of such detected cycles. Table 1 summarises our results. We can observe that test queries for all ontologies can be generated rather efficiently and the number of them is also relatively small.

<table>
<thead>
<tr>
<th>System</th>
<th>Ontology</th>
<th>#Queries</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requiem</td>
<td>AX</td>
<td>8</td>
<td>incomplete</td>
</tr>
<tr>
<td>CGLLR</td>
<td>AX</td>
<td>8</td>
<td>incomplete</td>
</tr>
<tr>
<td>Rapid</td>
<td>U</td>
<td>36</td>
<td>unsound</td>
</tr>
<tr>
<td></td>
<td>U</td>
<td>18</td>
<td>incomplete</td>
</tr>
<tr>
<td>IQAROS</td>
<td>S</td>
<td>25</td>
<td>incomplete</td>
</tr>
<tr>
<td>Nyaya</td>
<td>S</td>
<td>82</td>
<td>incomplete</td>
</tr>
<tr>
<td></td>
<td>U, UX</td>
<td>23</td>
<td>incomplete</td>
</tr>
</tbody>
</table>

Table 2: Statistics about unsoundness and incompleteness

All systems, except for Rapid, were reported sound for all ontologies and queries using the results in Proposition 5. Rapid was reported unsound for ontology U; since Rapid compute a UCQ rewriting for ontology U and all test queries, we could prove unsoundness and find suitable “witness” instances by again exploiting Proposition 5. Further inspection of the problematic queries revealed that unsoundness was due to a normalisation error similar to the one described in Example 3. The error was reported to the developers of Rapid and fixed in a subsequent release of the tool.

As shown in Table 2, all evaluated systems were found incomplete for some test ontologies and queries. In particular, we were able to exploit the results in Proposition 8 to detect discrepancies in the UCQ rewritings computed by each system; a more detailed inspection of the problematic queries and witness instances confirmed that discrepancies were indeed due to incompleteness issues.

Rapid’s incompleteness was again due to normalisation issues similar to those described in Example 3. Incompleteness in both CGLLR and Requiem was due to an implementation bug in query subsumption and redundancy elimination optimisations. Incompleteness in IQAROS was due to implementation bugs in ordering criteria for query atoms. These errors were again reported to systems’ developers and fixed based on our feedback.

**Performance Evaluation**

Performance was evaluated by using the systems’ latest versions; in the case of CGLLR, Requiem, Rapid, and IQAROS, the latest versions include fixes for all the correctness issues we identified in our experiments.

Table 3: Sum of rewriting times (seconds) for all test queries

<table>
<thead>
<tr>
<th>System</th>
<th>CG</th>
<th>Req</th>
<th>Rap</th>
<th>IQAROS</th>
<th>Nyaya</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>65.4</td>
<td>1.9</td>
<td>1.6</td>
<td>0.4</td>
<td>662.2</td>
</tr>
<tr>
<td>AX</td>
<td>5,680.9</td>
<td>10,442.2</td>
<td>586.8</td>
<td>64.2</td>
<td>37.5</td>
</tr>
<tr>
<td>P5</td>
<td>163.0</td>
<td>9.4</td>
<td>0.2</td>
<td>1.1</td>
<td>0.4</td>
</tr>
<tr>
<td>P5X</td>
<td>310.0</td>
<td>78.7</td>
<td>4.8</td>
<td>4.0</td>
<td>5h</td>
</tr>
<tr>
<td>U</td>
<td>0.7</td>
<td>0.6</td>
<td>0.1</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>UX</td>
<td>0.3</td>
<td>0.6</td>
<td>0.06</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>2.3</td>
<td>3.4</td>
<td>0.7</td>
<td>0.8</td>
<td>5.7</td>
</tr>
</tbody>
</table>

Table 2 presents, for each ontology and each system, the total time that the system took to compute a UCQ rewriting for all test queries. We can observe that Rapid and IQAROS were significantly faster than the other systems. Moreover, Nyaya could not finish rewriting the test queries for ontologies AX and P5X after 5 hours.

Finally, we have evaluated systems’ redundancy elimination mechanisms by measuring the size of the computed UCQ rewritings (i.e., the number of CQs they contain). Table 4 presents, for each ontology and each system, the sum of the sizes of all UCQ rewritings computed for all the test queries. We can observe that CGLLR presents the highest degrees of redundancy in its output, whereas Rapid and IQAROS computed the most succinct rewritings.

---

4http://www.cs.ox.ac.uk/projects/requiem/C.zip
5http://www.cs.ox.ac.uk/projects/requiem/home.html
6http://www.image.ece.ntua.gr/~achort/rapid.zip
7http://code.google.com/p/iqaros/
8http://mais.dia.uniroma3.it/Nyaya/Home.html
9We do not present results for ontology P1 since it was trivial (under 50 milliseconds) for all systems.
10See Table 1 for total number of test queries for each ontology.
Related Work and Conclusions

To the best of our knowledge all benchmarks for ontology-based systems rely on hand-crafted test queries (Pérez-Urbina, Motik, and Horrocks 2010; Guo, Pan, and Heflin 2005; Ma et al. 2006). The work that is closest to ours is the formal study of incompleteness of ontology reasoners (Stoilos, Cuenca Grau, and Horrocks 2010; Cuenca Grau and Stoilos 2011), where the authors describe techniques for synthetic query generation; however, these were limited to very lightweight DLs (EL and DL-Lite) under very strict acyclicity conditions, and the queries were used to evaluate incomplete materialisation-based reasoners, such as Jena, Sesame, and OWLIM.

In this paper, we presented a test query generation algorithm that can be applied to Horn ontologies. We also presented correctness evaluation techniques which have already been proved useful to uncover serious implementation errors in state of the art rewriting systems. These were not revealed by existing benchmarks and hence our techniques have already proved valuable to systems’ developers.

Acknowledgements

This work was supported by the EU FP7 project SEALS and by the EPSRC projects ConDOR, ExODA, and LogMap. B. Cuenca Grau is supported by a Royal Society University Research Fellowship. Giorgos Stoilos is supported by a Marie Curie FP7-Reintegration-Grants within European Union’s Seventh Framework Programme (FP7/2007-2013).

References


