SPARQL Query Containment under Schema

Melisachew Chekol, Jérôme Euzenat, Pierre Genevès, Nabil Layaïda

Noname manuscript No.
(will be inserted by the editor)

SPARQL Query Containment under Schema

Melisachew Wudage Chekol · Jérôme Euzenat · Pierre Genevès · Nabil Layaïda

Received: date / Accepted: date

Abstract Query containment is defined as the problem of determining if the result of a query is included in the result of another query for any dataset. It has major applications in query optimization and knowledge base verification. To date, testing query containment has been performed using different techniques: containment mapping, canonical databases, automata theory techniques and through a reduction to the validity problem in logic. Here, we use the latter technique to test containment of SPARQL queries using an expressive modal logic called μ-calculus. For that purpose, we define an RDF graph encoding as a transition system which preserves its characteristics. In addition, queries and schema axioms are encoded as μ-calculus formulae. Thereby, query containment can be reduced to testing validity in the logic.

We identify various fragments of SPARQL and description logic schema languages for which containment is decidable. Additionally, we provide theoretically and experimentally proven procedures to check containment of these decidable fragments. Finally, we propose a benchmark for containment solvers which is used to test and compare the current state-of-the-art containment solvers.

Keywords SPARQL · Query Containment · μ-calculus

M. W. Chekol
Data and Web Science Group, University of Mannheim, Mannheim, Germany
E-mail: mel@informatik.uni-mannheim.de

J. Euzenat · P. Genevès · N. Layaïda
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, F-38000 Grenoble France
E-mail: jerome.euzenat@inria.fr, nabil.layaida@inria.fr, pierre.geneves@cnrs.fr

1 Introduction

Containment, equivalence and satisfiability problems are well studied for relational database query languages. Their study started with a pioneering paper from Chandra and Merlin [16], which studied optimization and containment of conjunctive queries. In general, query containment is the problem of checking whether the result of one query is included in the result of another one for any given dataset. Also in [16], it is proved that the union of conjunctive query containment and equivalence problems are NP-complete. These problems, containment, equivalence, and satisfiability, are collectively called static analysis of queries.

Equivalence and satisfiability problems can be derived from containment, thus, we mainly concentrate on the containment problem. In relational databases, union of conjunctive query containment has been studied using containment mappings also known as graph homomorphism and canonical databases. It is known that, for (union of) conjunctive queries, query answering and containment are equivalent problems since query containment can be reduced to query answering [16]. Unfortunately, to apply these techniques to a query language equipped with ontologies and regular expressions is not entirely possible. This is the reason why, for semistructured data query languages (referred as regular path queries), automata theoretic notions are often employed to address containment and other problems [11]. In addition to using automata, containment has been addressed by reduction to the satisfiability test. In this direction, queries are translated into formulas in a particular logic that supports the query languages features and then the overall problem is reduced to the satisfiability test. Several works exist that developed and used this technique [11,27,13] that also inspired this work. Specifically, the study in [27] has developed a tree-logic from the alternation-free fragment of the μ-calculus and applied it to encode XPath queries and perform static analysis tasks. Their attempt has been successful and put to practice. Their implementation allows one to perform containment, satisfiability and equivalence of XPath queries. The study is extended to other languages namely XQuery, CSS and JSON.

The importance of the containment problem goes beyond the field of databases. It has attracted attention from the description logic community. In this regard, many of the works concentrated on the problem of conjunctive query answering as containment follows from it. All these works have sound theoretical and mathematical foundations, but they fail to provide an implementation (or experimentation results) of their approaches.

Deviating from conjunctive queries in the database and description logic worlds, are regular path queries (RPQs): query languages used to query arbitrary length paths in graph databases of semistructured data. Like conjunctive queries, they have been used and studied widely. They are different from conjunctive queries in that, they allow recursion by using regular expression patterns. The problem of containment has been addressed for several extensions of these languages: CRPQs, P2RPQs, and ECRPQs [25,11,7]. One prominent language used in querying semi-structured data is XPath. This language has

been studied over 2 decades. In [27], the static analysis of XPath queries has been investigated using a graph logic and by providing an implementation which has been put to practice.

A comparison of SPARQL and relational algebra has already been made, to figure out important similarities and use the results of studies for relational algebra. Consequently, the results from [44] and [4] taken together imply that SPARQL has exactly the same expressive power as relational algebra. From early results on query containment in relational algebra and first-order logic, one can infer that containment in relational algebra is undecidable [1]. Therefore, containment of SPARQL queries is also undecidable [18].

Consequently, in this work, we study the containment problem for various fragments of SPARQL under expressive schema languages to retain decidability. Recently, the study of SPARQL query containment has gained momentum, notably in [21, 22, 38, 23, 43]. The literature in [21] and [22] address containment under the ρdf [41] entailment regime and $\mathcal{SHI}$ schema axioms respectively and establish a double exponential upper bound complexity. Additionally, in [38] the containment and optimization of OPTIONAL queries is investigated while providing a Π_2^P-complete complexity for containment. Importantly, a benchmark of containment solvers is described in [23]. The implementations of containment solvers allow the extension of query and ontology languages. Thus, we can benefit from these implementations. Overall, the contribution of this paper is the following:

- We provide an encoding of RDF graphs as transition systems that can be used for determining the containment of SPARQL queries.
- We propose a technique to determine the containment of union of conjunctive SPARQL queries under $\mathcal{ALCH}$ schema axioms. To do so, we encode queries and schema axioms as μ-calculus formulas, thereby reducing containment to unsatisfiability. We prove the soundness and completeness of our approach. We show that the containment problem can be determined in a double exponential amount of time.
- Implicitly, we address the containment of PSPARQL queries. The complexity of determining containment of path SPARQL queries (also under $\mathcal{ALCH}$ axioms) is double exponential. This complexity is an upper bound for the problem.
- We present a first benchmark for statically analyzing semantic web queries. We design test suites and compare the performance and correctness of containment solvers.

Outline The rest of the article is organized as follows. In Section 2 we present the foundations of the semantic web and modal logic, followed by a presentation of a novel procedure to translate RDF graphs into transition systems in Section 3. In Section 4, we show how to encode schema axioms and queries as μ-calculus formulas and then, consequently, reduce query containment to unsatisfiability in the logic. In Section 5, we present a containment benchmark for SPARQL queries. We conclude this article with a broad survey of related works in Section 6 and a summary of the results in Section 7.

2 Preliminaries

We assume basic familiarity with the syntax and semantics of RDF(S) and $\mathcal{ALCH}$ on the level of the RDF Primer [39,31,32,5]. We present a very minimal introduction to RDF(S), $\mathcal{ALCH}$, SPARQL, and the μ-calculus.

2.1 RDF(S)

RDF is a formalism used to express information structured as graphs on the Web. We present a compact formalization of RDF [31]. Let I, B, and L be three disjoint infinite sets denoting the set of IRIs (identifying a resource), blank nodes (denoting an unidentified resource) and literals (a character string or some other type of data) respectively. We abbreviate any union of these sets as, for instance, IBL = I ∪ B ∪ L. A triple of the form $(s, p, o) \in \text{IB} \times \text{I} \times \text{IBL}$ is called an *RDF triple*, where s is the *subject*, p is the *predicate*, and o is the *object* of the triple. Each triple can be thought of as an edge between the subject and the object labelled by the predicate, hence a set of RDF triples is often referred to as an *RDF graph*. RDF has a model theoretic semantics [31].

Example 1 (RDF Graph) G is an RDF graph (all identifiers correspond to URIs and _:b is a blank node):

$$G = \{(john, childOf, mary), (childOf, \texttt{sp}, ancestor), (\_\texttt{:b}, hasFather, john), \\ (ancestor, \texttt{dom}, Person), (ancestor, \texttt{range}, Person)\}$$

RDF Schema (RDFS) may be considered as a simple ontology language expressing subsumption relations between classes or properties [31]. Technically, this is an RDF vocabulary used for expressing axioms constraining the interpretation of graphs. The RDFS vocabulary (Voc(RDFS)) and its semantics are given in [31] through rules that allow to deduce or infer new triples using RDF Schema triples. RDFS is a lightweight schema language which is not highly expressive due to the lack of important constructs such as negation. In the following, we introduce an expressive description logic called $\mathcal{ALCH}$.

2.2 $\mathcal{ALCH}$

$\mathcal{ALCH}$ is an extension of the description logic $\mathcal{ALC}$. Its syntax and semantics are presented in Table 1.

Syntax In $\mathcal{ALCH}$ concepts and roles are formed according to the syntax presented in Table 1, where R denotes an atomic role or its inverse, A represents an atomic concept, C denotes a complex concept, o refers to a nominal, and n is a non-negative integer. An $\mathcal{ALCH}$ TBox is a finite set of axioms consisting of *concept inclusions* $C_1 \sqsubseteq C_2$ and *role inclusion* $R \sqsubseteq S$ axioms.

Example 2 Consider the following $\mathcal{ALCH}$ TBox axioms that model a university domain.

$$\begin{aligned} \text{Chair} &\equiv \text{Person} \sqcap \exists \text{headOf.Department} \\ \text{Student} &\sqsubseteq \text{Person} \sqcap \forall \text{takesCourse.Course} \\ \text{Professor} &\equiv \text{Person} \sqcap \exists \text{givesCourse.Course} \\ \exists \text{headOf.}\top &\sqsubseteq \text{Professor} \end{aligned}$$

The first axiom states that every department chair is a person and is head of a certain department. Likewise, similar explanations can be given to the other axioms.

Semantics An interpretation, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, consists of a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that assigns to each object name o an element $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, to each atomic concept A a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ of the domain, and to each atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ over the domain. The role and concept constructs can be interpreted in $\mathcal{I}$ as depicted in Tables 1.

Table 1 Syntax and semantics of $\mathcal{ALCH}$.

Construct Name	Syntax	Semantics	
top concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	$\mathcal{ALC}$
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	
atomic role	R	$R^{\mathcal{I}} \sqsubseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
exists restriction	$\exists R.C$	$\{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$	
value restriction	$\forall R.C$	$\{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$	
concept hierarchy	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$	
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$

An interpretation $\mathcal{I}$ *satisfies* an inclusion $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it satisfies an equality $C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$. If $\mathcal{T}$ is a set of axioms, then $\mathcal{I}$ satisfies $\mathcal{T}$ iff $\mathcal{I}$ satisfies each element of $\mathcal{T}$. If $\mathcal{I}$ satisfies an axiom (resp. a set of axioms), then we say that it is a *model* of this axiom (resp. set of axioms).

An $\mathcal{ALCH}$ ABox contains a finite set of concept $C(o_1)$ and role $R(o_1, o_2)$ assertions, where o_1 and o_2 are individual names. The former asserts that o_1 belongs to (the interpretation of) C, formally, $o_1^{\mathcal{I}} \subseteq C^{\mathcal{I}}$. The later asserts that o_1 is related by the role R with o_2, formally, $(o_1^{\mathcal{I}}, o_2^{\mathcal{I}}) \subseteq R^{\mathcal{I}}$. We introduce next SPARQL as a query language for RDF graphs and $\mathcal{ALCH}$ ontologies.

2.3 Fragments of SPARQL

SPARQL [45] is a W3C recommended query language for RDF. Since its creation it has been source of research from various directions, mostly in terms of

extending the language: adding paths (regular expression patterns), negation, querying modulo schema, subqueries and the likes.

Querying RDF graphs with SPARQL amounts to matching graph patterns that are given as sets of triples of subjects, predicates and objects. These triples, $t \in \text{IBV} \times \text{IV} \times \text{IBLV}$ with V a set of variables disjoint from IBL, also known as triple patterns, are usually connected to form graphs by means of joins expressed using several occurrences of the same variable. It allows variables to be bound to components in the queried graph. In addition, operators akin to relational joins, unions, left outer joins, selections, and projections can be combined to build more expressive queries. Queries are formed from query patterns which in turn are defined inductively from *triple patterns*.

In the following, we introduce PSPARQL (for Path SPARQL) that extends SPARQL with regular expression patterns. The only difference between the syntax of SPARQL and PSPARQL is on triple patterns. Triple patterns in PSPARQL contain regular expressions in property positions instead of only IRIs or variables , i.e., tuple $t \in \text{IBV} \times \text{eV} \times \text{IBLV}$, with V a set of variables disjoint from IBL (IRIs, Blank nodes and Literals – are used to identify values such as strings, integers and dates.), and e is a regular path expression. Triple patterns (t) grouped together using the AND operator are named *basic graph patterns* (BGP). Basic graph patterns (b) grouped together using the UNION operator form *query patterns* (q).

Definition 1 A query pattern q is inductively defined as:

$$\begin{aligned} e &::= \epsilon \mid U \mid e_1/e_2 \mid e_1 \shortmid e_2 \mid e^+ \mid e^* \\ t &::= \text{IBV} \times \text{eV} \times \text{IBLV} \\ b &::= t \mid b_1 \text{ AND } b_2 \\ q &::= b \mid q_1 \text{ UNION } q_2 \end{aligned}$$

A SPARQL SELECT query is formed according to the following syntax:

$$\text{SELECT } W \text{ FROM } G \text{ WHERE } \{q\}.$$

The FROM clause identifies the queried RDF graph G on which the query will be evaluated, WHERE contains a query pattern q that the query answers should satisfy and SELECT singles out the answer variables $W \in V$ from the query pattern. For this work, we consider only AND and UNION SPARQL queries.

The *arity* of a query is the number of variables which appear in the SELECT clause of that query.

Example 3 (SPARQ queries) Consider the following queries $q(?x)$ on the graph of Example 1:

```
SELECT ?x WHERE {  ?x  type Person .  }
```

The semantics of SPARQL queries is provided based on a mapping function that maps variables into IRIs in the queried graph. Two BGP mappings μ_1

and μ_2 are said to be *compatible* if $\forall x \in dom(\mu_1) \cap dom(\mu_2), \mu_1(x) = \mu_2(x)$. If μ_1 and μ_2 are compatible, then $\mu_1 \cup \mu_2$ is also a mapping obtained by extending μ_1 according to μ_2 on all the variables in $dom(\mu_2) \setminus dom(\mu_2)$. Given two sets of mappings M_1 and M_2, the *union* of M_1 and M_2 is defined as: $M_1 \cup M_2 = \{\mu \mid \mu \in M_1 \text{ or } M_2\}$. The evaluation $[\![q]\!]_G$ of a query pattern q over an RDF graph G is defined as:

$$[\![q_1 \text{ AND } q_2]\!]_G = [\![q_1]\!]_G \bowtie [\![q_2]\!]_G$$
$$[\![q_1 \text{ UNION } q_2]\!]_G = [\![q_1]\!]_G \cup [\![q_2]\!]_G$$
$$[\![q(\overrightarrow{w})]\!]_G = \pi_{\overrightarrow{w}}([\![q]\!]_G)$$

The projection operator $\pi_{\overrightarrow{w}}$ selects only those part of the mappings relevant to variables in $\overrightarrow{w}$. For detailed discussions we refer the reader to [28]. Beyond these particular examples, the goal of query containment is to determine whether this holds for any graph.
The evaluation of SPARQL queries is proved to be PSPACE-complete [42,28]. However, the evaluation problem is NP-complete for the fragment containing only AND and UNION query patterns [42,3,28]. To determine containment, we encode SPARQL queries as μ-calculus formulas. So, next we present a brief introduction about this logic.

2.4 μ-calculus

The μ-calculus is a logic obtained by adding fixpoint operators to ordinary modal logics, or the Hennessy-Milner logic [37]. The result is a very expressive logic, sufficient to subsume many other temporal logics such as CTL and CTL* [10]. The modal μ-calculus is easy to model-check, and so makes a good 'back-end' logic for tools. In this paper, we mainly use the μ-calculus with nominals and converse modalities. In the following, we present its syntax and semantics.

The syntax of the μ-calculus is composed of countable sets of *atomic propositions* and *nominals* AP, a set of *variables* Var, a set of *programs and their respective converses* Prog for navigating in graphs. A μ-calculus formula, φ, can be defined inductively as follows:

$$\varphi ::= \top \mid \bot \mid q \mid X \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X \varphi \mid \nu X \varphi$$

where $q \in AP, X \in Var$ and $a \in Prog$ is a transition program or its converse $\bar{a}$. The greatest and least fixpoint operators (ν and μ) respectively introduce general and finite recursion in graphs [37]. A *sentence* is a formula with no free variable, i.e., each variable in the formula appears within the scope of μ or ν. Besides, we use the following syntactic sugar:

$$\bot = \neg\top \qquad \varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$$
$$[a]\varphi = \neg\langle a \rangle \neg\varphi \qquad \nu X.\varphi(X) = \neg\mu X.\neg\varphi({}^{\neg X}/_{X})$$
$$\varphi \Rightarrow \psi = \neg\varphi \vee \psi \qquad \varphi \Leftrightarrow \psi = (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$$

The semantics of the μ-calculus is given in terms of a transition system and a valuation function.

The transition system is a tuple $K = (S, R, L)$, where S is a non-empty set of nodes, $R : Prog \to 2^{S \times S}$ is the transition function, and $L : AP \to 2^S$ assigns to each atomic proposition or nominal the set of nodes where it holds. It is such that $L(p)$ is a *singleton* for each nominal p. For converse programs, R can be extended as $R(\bar{a}) = \{(s', s) \mid (s, s') \in R(a)\}$.

The valuation function $V : \text{Var} \to 2^S$ maps each variable into a set of nodes. For a valuation V, a variable X, and a set of nodes $S' \subseteq S$, $V[X/S']$ is the valuation that is obtained from V by assigning S' to X.

The semantics of a formula with regard to a transition system K and a valuation function V is represented by $[\![\varphi]\!]^K_V \subseteq S$. The semantics of basic μ-calculus formulae is defined as follows:

$$
\begin{aligned}
[\![\top]\!]^K_V &= S \\
[\![\bot]\!]^K_V &= \emptyset \\
[\![q]\!]^K_V &= L(q), q \in AP,\ L(q) \text{ is singleton if } q \text{ is a nominal} \\
[\![X]\!]^K_V &= V(X), X \in Var \\
[\![\neg\varphi]\!]^K_V &= S \backslash [\![\varphi]\!]^K_V \\
[\![\varphi \wedge \psi]\!]^K_V &= [\![\varphi]\!]^K_V \cap [\![\psi]\!]^K_V \\
[\![\varphi \vee \psi]\!]^K_V &= [\![\varphi]\!]^K_V \cup [\![\psi]\!]^K_V \\
[\![\langle a \rangle \varphi]\!]^K_V &= \{s \in S \mid \exists s' \in S.(s, s') \in R(a)\ \wedge s' \in [\![\varphi]\!]^K_V\} \\
[\![[a]\varphi]\!]^K_V &= \{s \in S \mid \forall s' \in S.(s, s') \in R(a) \Rightarrow s' \in [\![\varphi]\!]^K_V\} \\
[\![\mu X \varphi]\!]^K_V &= \bigcap \{S' \subseteq S \mid [\![\varphi]\!]^K_{V[X/S']} \subseteq S'\} \\
[\![\nu X \varphi]\!]^K_V &= \bigcup \{S' \subseteq S \mid S' \subseteq [\![\varphi]\!]^K_{V[X/S']}\}
\end{aligned}
$$

Note that the evaluation of sentences is independent of valuations and hence we define the following. For a sentence φ, a transition system $K = (S, R, L)$, and $s \in S$, we denote $K, s \models \varphi$ if and only if $s \in [\![\varphi]\!]^K$, henceforth K is considered as a *model* of φ. In other words, K is considered as a model of ϕ if there exists an $s \in S$ such that $K, s \models \phi$. If a sentence has a model, then it is called *satisfiable*.

To study SPARQL query containment, only a specific subset of the μ-calculus presented above, namely the *alternation-free* modal μ-calculus with nominals and converse [50], is of interest. A μ-calculus formula φ is alternation-free if whenever $\mu X.\varphi_1$ (respectively $\nu X.\varphi_1$) is a subformula of φ and $\nu Y.\varphi_2$ (respectively $\mu Y.\varphi_2$) is a subformula of φ_1 then X does not occur freely in φ_2. For instance, $\nu X.(\mu Y.(\langle s \rangle Y \wedge a) \vee \langle p \rangle X)$ is alternation-free but $\nu X.(\mu Y.(\langle s \rangle Y \wedge X) \vee a)$ is not since X, bound by ν, appears freely in the scope of μY. RDF graphs can be encoded as transition systems.

3 Encoding RDF graphs as Transition Systems

In order to reduce query containment in SPARQL to unsatisfiability in the μ-calculus, we translate RDF graphs into transition systems and SPARQL queries and schema axioms into μ-calculus formulae. An additional benefit of using a μ-calculus encoding is to take advantage of fixpoints and modalities for encoding recursion. They allow to deal with property hierarchies of SPARQL queries [3] or queries modulo RDF Schema.

We now show how to translate RDF graphs into labelled transition systems. First of all, translating RDF graphs into transition systems is necessary in order to restrict the models of the μ-calculus formula obtained from the translation of queries. Additionally, if RDF graphs can be translated into transition systems, then model checking can be used to evaluate SPARQL queries. In this regard, there are already some results in the literature [40] where it is possible to extend and encode SPARQL queries in a logic and use model checking to evaluate the result of the query. There are several ways of encoding RDF graphs as transition systems. For instance, consider the following:

– for each triple $(s, p, o) \in G$, s and o become nodes of the transition system and p is a transition program, there is an edge $\langle s, o \rangle$ where transition from node s to o and vice versa can be traversed using program p and its converse $\bar{p}$ respectively. While this approach is simple and intuitive, it does not work in the general case, i.e., in an RDF graph predicates or properties can also be nodes in an RDF graph as shown in Figure 1, thus, p cannot be a transition program.

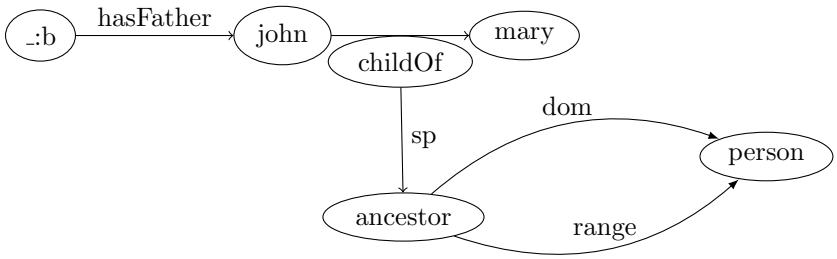


Fig. 1 An RDF graph where a predicate appears as a node

– for each triple $(s, p, o) \in G$, s, p, and o become atomic propositions that are true in the states n_s, n_p, and n_o respectively of a transition system, there are are edges $\langle n_s, n_p \rangle$ and $\langle n_p, n_o \rangle$ that are accessible through transition programs 1 and 2 respectively. As the above approach, this translation procedure does not work in the general case when encoding RDF schema graphs. For example, consider an RDF graph that contains the triple (subPropertyOf, subPropertyOf, subPropertyOf).

– the last approach considers encoding RDF graphs as bipartite graphs, i.e., for each $t = (s, p, o) \in G$, it introduces two types of nodes in the transition system: one node for each triple n_t and another node for each element of the triple n_s, n_p, and n_o where atomic propositions s, p and o are set to be true on each node respectively. Additionally, there are edges $\langle n_s, n_t\rangle$, $\langle n_t, n_p\rangle$, and $\langle n_t, n_o\rangle$ in the transitions system that are accessible through programs s, p, o and their converses respectively. The idea of representing RDF triples as other types of graphs (for instance, hypergraphs) was first introduced in [6], in fact, this translation coincides with the notion of reification of n-ary relations [13] that is one edge from the triple node to subject, predicate, and object nodes of the triple in this case. This approach overcomes the limitations of the other two approaches. Thus, in the following, we discuss in detail how this technique works.

3.1 Encoding of RDF graphs

An RDF graph is encoded as a transition system in which nodes correspond to RDF entities and RDF triples. Edges relate entities to the triples they occur in. Different programs are used for distinguishing the subject, predicate, and object. Expressing predicates as nodes, instead of atomic programs, makes it possible to deal with full RDF expressiveness in which a predicate may also be the subject or object of an RDF statement.

Definition 2 (Transition system associated to an RDF graph) Given an RDF graph, $G \subseteq UB \times U \times UBL$, the transition system associated to G, $\sigma(G) = (S, R, \mathsf{L})$ over $AP = \mathrm{U_G B_G L_G} \cup \{s', s''\}$, is $\sigma(G)$ such that:

– $S = S' \cup S''$ with S' and S'' the smallest sets such that $\forall u \in U_G, \exists n^u \in S'$, $\forall b \in B_G, \exists n^b \in S'$, $\forall l \in L_G, \exists n^l \in S'$, and $\forall t \in G, \exists n^t \in S''$,
– $\forall t = (s, p, o) \in G$, $\langle n^s, n^t\rangle \in R(s)$, $\langle n^t, n^p\rangle \in R(p)$, and $\langle n^t, n^o\rangle \in R(o)$,
– $\mathsf{L} : AP \to 2^S$; $\forall u \in U_G, \mathsf{L}(u) = \{n^u\}$, $\forall b \in B_G, \mathsf{L}(b) = S'$, $\mathsf{L}(s') = S'$, $\forall l \in L_G, \mathsf{L}(l) = \{n^l\}$ and $\mathsf{L}(s'') = S''$,
– $\forall n^t, n^{t'} \in S''$, $\langle n^t, n^{t'}\rangle \in R(d)$.

The program d is introduced to render each triple accessible to the others and thus facilitate the encoding of queries. The function σ associates what we call a *restricted transition system* to any RDF graph. Formally, we say that a transition system K is a *restricted transition system* iff there exists an RDF graph G such that $K = \sigma(G)$.

A restricted transition system is thus a bipartite graph composed of two sets of nodes: S', those corresponding to RDF entities, and S'', those corresponding to RDF triples. For example, Figure 2 shows the restricted transition system associated with the graph of Example 1.

Given that the logic chosen to determine containment is the μ-calculus with nominals (lacking functionality or number restriction), one cannot impose that each triple node is connected to exactly one node for each of the three triple-components (subject, predicate, and object). However, we can impose a lighter

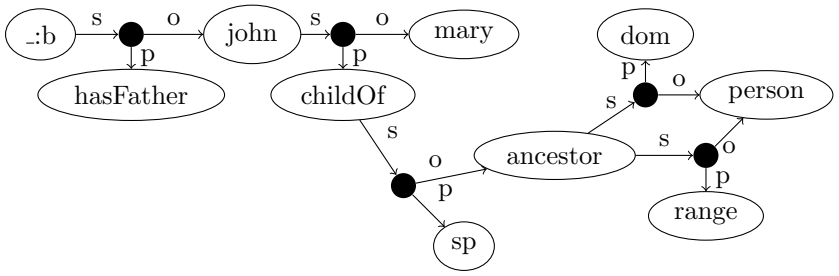


Fig. 2 Transition system encoding the RDF graph of Example 1. Nodes in S'' are black anonymous nodes; nodes in S' are the other nodes (d-transitions are not displayed).

restriction to achieve this by taking advantage of the technique introduced in [26]. Since it is not possible to ensure that there is only one successor, then we restrict all the successors to bear the same constraints. They, thus, become interchangeable (bisimulation). To do this, we introduce a rewriting function f such that all occurrences of $\langle a\rangle\varphi$ (existential formulas) are replaced by $\langle a\rangle\top\wedge[a]\varphi$. As such, f is inductively defined on the structure of a μ-calculus formula as follows:

$$
\begin{aligned}
f(\top) &= \top \\
f(q) &= q \quad q \in AP \cup Nom & f(\langle a\rangle\varphi) &= \langle a\rangle\top \wedge [a]f(\varphi) \quad a \in \{\bar{s}, p, o\} \\
f(X) &= X \quad X \in Var & f(\langle a\rangle\varphi) &= \langle a\rangle f(\varphi) \quad a \in \{d, s, \bar{p}, \bar{o}\} \\
f(\neg\varphi) &= \neg f(\varphi) & f([a]\varphi) &= [a]f(\varphi) \quad a \in \text{Prog} \\
f(\varphi \wedge \psi) &= f(\varphi) \wedge f(\psi) & f(\mu X.\varphi) &= \mu X.f(\varphi) \\
f(\varphi \vee \psi) &= f(\varphi) \vee f(\psi) & f(\nu X.\varphi) &= \nu X.f(\varphi)
\end{aligned}
$$

Thus, when checking for query containment, we assume that the formulas are rewritten using function f. Along with that, we also consider the following restrictions:

- The set of programs is fixed: $Prog = \{s, p, o, d, \bar{s}, \bar{p}, \bar{o}, \bar{d}\}$.
- A model must be a restricted transition system.

The last constraint can be expressed in the μ-calculus as follows:

Proposition 1 (RDF restriction on transition systems) *Let φ be a formula that can be stated over a restricted transition system, φ is satisfied by some restricted transition system if and only if $f(\varphi) \wedge \varphi_r$ is satisfied by some transition system over Prog, i.e. $\exists K_r.[\![f(\varphi)]\!]^{K_r} \neq \emptyset \iff \exists K.[\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$, where:*

$$\varphi_r = \ \nu X.\ \theta \wedge \kappa \ \wedge (\neg\langle d\rangle\top \vee \langle d\rangle X)$$

in which

$\theta = \langle\bar{s}\rangle s' \wedge \langle p\rangle s' \wedge \langle o\rangle s' \wedge \neg\langle s\rangle\top \wedge \neg\langle\bar{p}\rangle\top \wedge \neg\langle\bar{o}\rangle\top$, *and*

$\kappa = [\bar{s}]\xi \wedge [p]\xi \wedge [o]\xi$, *with*
$\xi = \neg\langle\bar{s}\rangle\top \wedge \neg\langle o\rangle\top \wedge \neg\langle p\rangle\top \wedge \neg\langle d\rangle\top \wedge \neg\langle\bar{d}\rangle\top \wedge \neg\langle s\rangle s' \wedge \neg\langle\bar{o}\rangle s' \wedge \neg\langle\bar{p}\rangle s'$

The formula φ_r ensures that θ and κ hold in every node reachable by a d edge, i.e., in every s'' node. The formula θ forces each s'' node to have a subject, predicate and object. The formula $f(\varphi)$ enforces reification (makes sure that each s'' node is connected to one subject, one predicate, and one object node). The formula κ navigates from a s'' node to every reachable s' node, and forces the latter not to be directly connected to other subject, predicate or object nodes.

Proof ($\Rightarrow$) Assume that $\exists K_r [\![f(\varphi)]\!]^{K_r} \neq \emptyset$, since φ_r is satisfied by any restricted transition system, one gets $[\![\varphi_r]\!]^{K_r} \neq \emptyset$. As a result, we have that: $\exists K_r [\![f(\varphi)]\!]^{K_r} \neq \emptyset$ and $[\![\varphi_r]\!]^{K_r} \neq \emptyset$ which imply $\exists K_r [\![f(\varphi)]\!]^{K_r} \wedge [\![\varphi_r]\!]^{K_r} \neq \emptyset$. From this, using the μ-calculus semantics, one obtains $\exists K_r [\![f(\varphi) \wedge \varphi_r]\!]^{K_r} \neq \emptyset$. Since a restricted transition system is also a transition system, $K_r \subseteq K$, it follows that $\exists K. [\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$.
($\Leftarrow$) Assume that $\exists K [\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$. We construct a restricted transition system model $K_r = (S_r = S_r^{'} \cup S_r^{''}, R_r, L_r)$ and a function $g : K_r \rightarrow K$ from $K = (S, R, L)$. Add a node n_0' to S_r with $g(n_0') = n_0$ where $f(\varphi) \wedge \varphi_r$ is satisfied in K. Suppose we have constructed a node n_r of S_r. For $j \in \{s, p, o\}$, if there is $n \in S$ with $(g(n_r), n) \in R(j)$, then pick one such n and add a node n_r' to S_r with $g(n_r') = n$. In such a construction, if there are concurrent $\bar{s}, p, o$ transitions from an $S_r^{''}$ node, we retain one transition for each modality. This is because, if such transitions are part of the model that satisfy $f(\varphi) \wedge \varphi_r$, then they will be under the influence of the constraints $f(.)$ and φ_r, and will bear these constraints. However, if they belong to K that does not satisfy the aforementioned formula, then cutting them will not affect the capacity of the model to be a model for the formula. Finally, for an atomic proposition p, $L_r(p) = \{n_r \in S_r \mid g(n_r) \in L(p)\}$.
The RDF triple structure is maintained in K_r i.e., $\langle (s, s''), (s'', p), (s'', o)\rangle$ is valid throughout the graph. If there were node pairs outside of this structure, then φ_r will not be satisfied. Throughout the graph, θ, $f(.)$ and κ ensure that for each triple node $s'' \in S_r$, there exists an incoming subject edge, an outgoing property edge, and an outgoing object edge. Hence, $[\![\varphi_r]\!]^{K_r} \neq \emptyset$.
To verify that $[\![f(\varphi)]\!]^{K_r} \neq \emptyset$, it is enough to show that $[\![f(\varphi)]\!]^{K} \neq \emptyset \Rightarrow [\![f(\varphi)]\!]^{K_r} \neq \emptyset$ by induction on the structure of $f(\varphi)$.

If a μ-calculus formula ψ appears under the scope of a least (μ) or greatest (ν) fixed point operator over all the programs $\{s, p, o, d, \bar{s}, \bar{p}, \bar{o}, \bar{d}\}$ as, $\mu X.\psi \vee \langle s\rangle X \vee \langle p\rangle X \vee \cdots$ or $\nu X.\psi \wedge \langle s\rangle X \wedge \langle p\rangle X \wedge \cdots$, then, for the sake of legibility, we denote the formulae by $lfp(X, \psi)$ and $gfp(X, \psi)$, respectively.

So far we have shown how RDF graphs can be translated into transition systems over which the μ-calculus formulas are interpreted. In the following, we propose methods that are used to encode schema axioms and queries as μ-calculus formulas. Thus, we use these encodings to reduce the containment test to the validity problem.

4 SPARQL Containment under Schema

In this section we study the containment of SPARQL queries under $\mathcal{ALCH}$ axioms. We provide theoretical foundations and establish complexity bounds of the problem.

4.1 Encoding $\mathcal{ALCH}$ Schema

We provide the μ-calculus encoding of $\mathcal{ALCH}$ axioms. This encoding is used together with query encodings to determine if any two queries are contained in each other under a given set of axioms.

Definition 3 (μ-calculus encoding of a $\mathcal{ALCH}$ schema) Given a set of axioms $c_1, c_2, ..., c_n$ of a schema $\mathcal{C}$, the μ-calculus encoding of $\mathcal{C}$ is:

$$\eta(\mathcal{C}) = \eta(c_1) \wedge \eta(c_2) \wedge ... \wedge \eta(c_n).$$

such that η translates each axiom into an equivalent formula:

$$\eta(C_1 \sqsubseteq C_2) = gfp\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big) \qquad \text{(Class subsumption)}$$
$$\eta(R_1 \sqsubseteq R_2) = gfp\big(X, R_1 \Rightarrow R_2\big) \qquad \text{(Role subsumption)}$$

using ω which recursively encodes concepts and roles:

$$\begin{aligned}
\omega(\bot) &= \bot \\
\omega(A) &= A \\
\omega(\neg C) &= \neg\omega(C) \\
\omega(C_1 \sqcap C_2) &= \omega(C_1) \wedge \omega(C_2) \\
\omega(\exists R.C) &= \langle s\rangle\big(\langle p\rangle R \wedge \langle o\rangle(\langle s\rangle\langle o\rangle\omega(C))\big) \\
\omega(\forall R.C) &= [s]\big([p]R \Rightarrow [o]([s][o]\omega(C))\big)
\end{aligned}$$

We prove the correctness of this shema axiom encoding.

Lemma 1 *Given a set of $\mathcal{ALCH}$ schema axioms $\mathcal{C}$, $\mathcal{C}$ has a model iff $\eta(\mathcal{C})$ is satisfiable.*

Proof ($\Rightarrow$) assume that there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ of $\mathcal{C}$ such that $\mathcal{I} \models \mathcal{C}$. We build a restricted transition system $K = (S, R, L)$ from $\mathcal{I}$ using the following:

- for each element of the domain $e^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, we create a node $n^e \in S'$,
- for each atomic concept A, if $a^{\mathcal{I}} \in A^{\mathcal{I}}$, then $(n^a, t) \in R(s), (t, n^{type}) \in R(p)$, $(t, n^A) \in R(o)$, $L(type), = n^{type}, L(A) = n^A$ and $L(a) = n^a$ where $t \in S''$,
- for each atomic role T, if $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in T^{\mathcal{I}}$, then $(n^x, t) \in R(s)$, $(t, n^T) \in R(p)$, and $(t, n^y) \in R(o)$ such that $n^x, n^y, n^T \in S'$, $t \in S''$, and $L(x) = n^x$, $L(T) = n^T$, $L(y) = n^y$,
- $S = S' \cup S''$

To show that $\eta(\mathcal{C})$ is satisfiable in K. We proceed inductively on the construction of the formula. As the axioms, $\{c_1, \ldots, c_n\}$, are made of role or concept inclusions or nominals, we consider the following cases:

- when $\eta(c_i) = gfp\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big)$.
  From the assumption, it is the case that $\mathcal{I} \models c_i$, alternatively, $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$. To show that $\eta(c_i)$ is satisfiable in K, we proceed on the construction of C_1 and C_2.
  1. If C_1 and C_2 are atomic concepts, then their encodings are atomic propositions C_1 and C_2 in the μ-calculus. From $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, we have that $[\![\omega(C_1)]\!]^K \subseteq [\![\omega(C_2)]\!]^K = [\![C_1]\!]^K \subseteq [\![C_2]\!]^K$. Hence, $\omega(C_1) \Rightarrow \omega(C_2)$ is satisfiable in K. Besides, the general recursion ν guarantees that the constraint is satisfied in each state of the transition system. Therefore, $\eta(c_i)$ is satisfiable.
  2. If C_1 and C_2 are complex concepts, then $K \models \eta(c_i)$ can be proved by exploiting the construction of each axiom and ω, e.g., if the axiom is $\mathcal{I} \models C \wedge D \wedge \exists R.C \sqsubseteq \forall R.D$

     $\Leftrightarrow (C \wedge D \wedge \exists R.C)^{\mathcal{I}} \subseteq (\forall R.D)^{\mathcal{I}}$
     $\Leftrightarrow C^{\mathcal{I}} \cap D^{\mathcal{I}} \cap (\exists R.C)^{\mathcal{I}} \subseteq (\forall R.D)^{\mathcal{I}}$
     $\Leftrightarrow [\![\omega(C)]\!]^K \cap [\![\omega(D)]\!]^K \cap [\![\omega(\exists R.C)]\!]^K \subseteq [\![\omega(\forall R.D)]\!]^K$ from case (1).
     $\Leftrightarrow [\![\omega(C) \wedge \omega(D) \wedge \omega(\exists R.C)]\!]^K \subseteq [\![\omega(\forall R.D)]\!]^K$ μ-calculus semantics.
     $\Leftrightarrow [\![gfp\big(X, \omega(C) \wedge \omega(D) \wedge \omega(\exists R.C) \Rightarrow \omega(\forall R.D)\big)]\!]^K$ the gfp (or ν) ensures that theimplication holds in the entire transition system.

     This extends likewise to any axiom composed of complex concept constructs.
- when $\eta(c_i) = gfp\big(X, \omega(r_1) \Rightarrow \omega(r_2)\big)$. From $r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$ we have that $\exists n^{r_1} \in L(r_1)$ implies $\exists n^{r_2} \in L(r_2)$ in K. Thus, $\exists s \in [\![\omega(r_1) \Rightarrow \omega(r_2)]\!]^K$. As K is a construction of $\mathcal{I}$, $\eta(c_i)$ is satisfiable in K.

Since K is a model of each $\eta(c_i)$, then $\eta(\mathcal{C})$ is satisfiable.
($\Leftarrow$) consider a transition system (S, R, L) which is a model of K for $\eta(\mathcal{C})$. From K, we construct an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ and show that it is a model of $\mathcal{C}$.

- $\Delta^{\mathcal{I}} = S$, $A^{\mathcal{I}} = [\![A]\!]^K$ for each atomic concept A,
- $\top^{\mathcal{I}} = [\![\top]\!]^K$, for a top concept,
- $r^{\mathcal{I}} = \{(s, s') \mid \forall t \in [\![r]\!]^K \wedge t' \in S \wedge (s, t') \in R(s) \wedge (t', t) \in R(p) \wedge (t', s') \in R(o)\}$ for each atomic role r,

Consequently, formulas such as $gfp\big(X, \omega(r_1) \Rightarrow \omega(r_2)\big)$ and $gfp\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big)$ are true in $\mathcal{I}$. The first formula expresses that there is no node in the transition system where $\omega(r_1)$ holds and $\omega(r_2)$ does not hold. This is equivalent to $\omega(r_1) \Rightarrow \omega(r_2)$ and $[\![r_1]\!]^K \subseteq [\![r_2]\!]^K$ since r_1 and r_2 are basic roles. Thus, we obtain $r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$ and $\mathcal{I} \models r_1 \sqsubseteq r_2$.

One can exploit the construction of the latter formula. Note however that, similar justifications as above can be worked out to arrive at $\mathcal{I} \models C_1 \sqsubseteq C_2$ if C_1 and C_2 are basic concepts. Nonetheless, if they are complex concepts, we proceed as below. Consider the case when $C_1 = A \sqcap B$ and $C_2 = \exists R.C$,

$$\begin{aligned}
[\![\omega(C_1) \Rightarrow \omega(C_2)]\!]^K &\Leftrightarrow [\![\omega(A \sqcap B)]\!]^K \subseteq [\![\omega(\exists R.C)]\!]^K \\
&\Leftrightarrow [\![A \wedge B]\!]^K \subseteq [\![\langle s\rangle\big(\langle p\rangle R \wedge \langle o\rangle(\langle s\rangle\langle o\rangle C)\big)]\!]^K \\
&\Leftrightarrow [\![A]\!]^K \wedge [\![B]\!]^K \subseteq \{s \mid \exists s'.s \in [\![\langle s\rangle\langle p\rangle R]\!]^K \wedge s' \in [\![\langle s\rangle\langle o\rangle C]\!]^K\} \\
&\Leftrightarrow A^{\mathcal{I}} \cap B^{\mathcal{I}} \subseteq \{s \mid \exists s'.(s,s') \in R^{\mathcal{I}} \wedge s' \in C^{\mathcal{I}}\} \\
&\Leftrightarrow (A \sqcap B)^{\mathcal{I}} \subseteq (\exists R.C)^{\mathcal{I}} \\
&\Leftrightarrow \mathcal{I} \models C_1 \sqsubseteq C_2
\end{aligned}$$

Accordingly, from $\mathcal{I} \models c_1 \wedge \cdots \wedge \mathcal{I} \models c_n$, it follows that $\mathcal{I} \models \mathcal{C}$.

4.2 Query Containment

We provide procedures to translate unions of conjunctive SPARQL queries, i.e., SPARQL queries that only use UNION and AND operators, into μ-calculus formulas.

4.2.1 Encoding Queries as μ-calculus Formulae

For any query $q(\overrightarrow{w})$, we call the variables in $\overrightarrow{w}$ *distinguished* or answer variables. Furthermore, we denote the *non-distinguished* or existential variables in q by $ndvar(q)$, the URIs/constants by $uris(q)$, and the distinguished variables by $dvar(q)$. When encoding $q \sqsubseteq q'$, we call q left-hand side query and q' right-hand side query.

Queries are translated into μ-calculus formulas. The principle of the translation is that each triple pattern is associated with a sub-formula stating the existence of the triple somewhere in the graph. Hence, they are quantified by μ so as to put them out of the context of a state. In this translation, variables are replaced by nominals or some formula that are satisfied when they are at the corresponding position in such triple relations. A function called $\mathcal{A}$ is used to encode queries inductively on the structure of query patterns. AND and UNION are translated into boolean connectives $\wedge$ and $\vee$ respectively.

Encoding left-hand side query The encoding of the left hand side query is done by freezing all the terms in the query. In other words, each term in the query becomes a nominal in the μ-calculus. Further, $\mathcal{A}$ computes recursively a μ-calculus formula. Consequently, the encoding of the SPARQL query q is $\mathcal{A}(q)$ such that:

$$\begin{aligned}
\mathcal{A}\big((x,e,z)\big) &= \mathit{lfp}\big(X, \langle\bar{s}\rangle x \wedge \mathcal{R}(e,z)\big) \\
\mathcal{A}\big((x,e^*,z)\big) &= (x \Leftrightarrow z) \vee \mathit{lfp}\big(X, \langle\bar{s}\rangle x \wedge \mathcal{R}(e^+,z)\big)
\end{aligned}$$

$$\mathcal{A}(q_1 \mathsf{AND} q_2) = \mathcal{A}(q_1) \wedge \mathcal{A}(q_2)$$
$$\mathcal{A}(q_1 \mathsf{UNION} q_2) = \mathcal{A}(q_1) \vee \mathcal{A}(q_2)$$

Regular expression patterns that appear in the query are encoded using the function $\mathcal{R}$. This function takes two arguments (the predicate which is a regular expression pattern and the object of a triple).

$$\begin{aligned}
\mathcal{R}(uri, y) &= \langle p \rangle uri \ \wedge \langle o \rangle y \\
\mathcal{R}(x, y) &= \langle p \rangle x \wedge \langle o \rangle y \\
\mathcal{R}(e \shortmid e', y) &= (\mathcal{R}(e, y) \vee \mathcal{R}(e', y)) \\
\mathcal{R}(e \cdot e', y) &= \mathcal{R}(e, \langle s \rangle \mathcal{R}(e', y)) \\
\mathcal{R}(e^+, y) &= \mu X. \mathcal{R}(e, y) \vee \mathcal{R}(e, \langle s \rangle X) \\
\mathcal{R}(e^*, y) &= \mathcal{R}(e^+, y) \vee \langle \bar{s} \rangle y
\end{aligned}$$

*Example 4 (Encoding Kleene star *)* Consider the encoding of the following query:

$$\begin{aligned}
q\{x, y\} &= (x, z, y)(z, sp^*, ancestor) \\
\mathcal{A}(q) &= lfp(X, \langle \bar{s} \rangle x \wedge \langle p \rangle z \wedge \langle o \rangle y) \ \wedge (z \Leftrightarrow ancestor \\
&\quad \vee lfp\big(X, \mu Y. \mathcal{R}(sp, ancestor) \vee \mathcal{R}(sp, \langle s \rangle Y\big) \\
&= lfp(X, \langle \bar{s} \rangle x \wedge \langle p \rangle z \wedge \langle o \rangle y) \ \wedge (z \Leftrightarrow ancestor \\
&\quad \vee lfp\big(X, \mu Y. \langle p \rangle sp \wedge \langle o \rangle ancestor \vee \langle p \rangle sp \wedge \langle s \rangle \langle o \rangle Y\big))
\end{aligned}$$

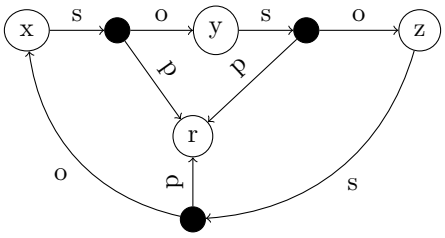
Cyclic queries In order to encode the right-hand side query, we need to define the notion of cyclic queries with respect to transition system graphs.

Definition 4 (Cyclic Query) A SPARQL query is referred to as cyclic if a transition graph induced from the query patterns is cyclic. The transition graph[1] is constructed in the same way as the transition system of Definition 2.

Note that the only difference in the transition system, obtained from SPARQL and PSPARQL queries to determine cyclicity, is that: for SPARQL queries, nodes of the transition system are either variables or constants (URIs), whereas in the case of PSPARQL, nodes can also be regular expression patterns. This is a sufficient condidtion to determine cyclicness among non-distinguished variables in PSPARQL queries.

Example 5 (Cyclic query) Consider the following cyclic query: $q() = (x, r, y)$ $\mathsf{AND}(y, r, z)\mathsf{AND}(z, r, x)$ a transition graph obtained from the triple patterns is shown below:

[1] The transition graph is similar to the tuple-graph used in [13] to detect the dependency among variables.



We can identify various features from this example:

- *cyclicity*: the query contains a cycle,
- *distinguished variable-free*: the query does not contain any distinguished variables,
- *constant-free*: the query does not contain any constant.

We refer to such cycles as *constant and distinguished variable-free cycles*, which are formally defined below.

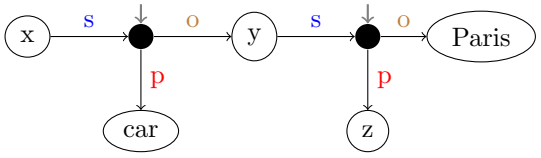
Definition 5 A *constant and distinguished variable-free cyclic (CDFC)* component of a query is a component of the query graph that contains: *no constants*, *no distinguished variables*, and a *cycle*.

Definition 6 (Purely cyclic SPARQL queries) A SPARQL$_{pc}$ query is a SPARQL query that contains a CDFC component.

As a consequence of Definition 5, we can identify two classes of queries: queries that contain a CDFC component (which we refer to as *purely cyclic* SPARQL queries, in short, SPARQL$_{pc}$) and queries without a CDFC (we refer to them as just SPARQL). The reasons for such distinction are: (1) according to the experimental results in [23], such queries are rarely used in real-world scenarios, and finally (2) to give separate encodings due to formula size. In other words, when encoding $q \sqsubseteq q'$, the size of the formula corresponding to q' is $|ndvar(q)|^{|terms(q)|}$. However, this size reduces reasonably when done differently as discussed below. (3) SPARQL$_{pc}$ queries can easily be detected with worst case time complexity $\mathcal{O}(|V| + |E|)$ where V is the vertex set and E is the edge set of the transition graph obtained from the query,

Encoding right-hand side SPARQL query The encoding of the right-hand side query q' is different from that of the left due to the non-distinguished variables that appear in cycles in the query. Before we show how multiply appearing non-distinguished variables are encoded, we present motivating examples.

Example 6 Consider the query $q(x) = (x, car, y)\mathsf{AND}(y, z, Paris)$, it has one distinguished variable x and two non-distinguished variables y and z. z can be encoded as $\top$ whereas to encode y, we produce its encodings using the nominals corresponding to the encodings of the distinguished variables and constants in q. We obtain two possible subformulas for y, $\{y \rightarrow \langle \bar{o} \rangle \langle p \rangle car, y \rightarrow \langle s \rangle \langle o \rangle Paris\}$, using these formulas we can construct the encoding for q. A restricted transition system that corresponds to the query q is shown below:



When encoding the right-hand side query, the distinguished variables and constants are encoded as nominals whereas the non-distinguished variables ndvar(q') are encoded as follows:

- If a non-distinguished variable x occurs only once in q', x is encoded as $\top$.
- If a non-distinguished variable appears multiple times in q', then we produce a set of mappings $m = \{m_1, \ldots, m_n\}$ such that each m_i contains formula assignments to the non-distinguished variables of the query. m is computed in as follows:
  - we denote the union of the set of distinguished variables and constants of q' by $\mathbf{X}$, i.e., $\mathbf{X} = uris(q') \cup dvar(q')$,
  - for any triple $t = (s, p, o)$, functions f_s, f_p, and f_o return the subject, predicate, and object of t respectively:

$$f_s((s,p,o)) = s \qquad f_p((s,p,o)) = p \qquad f_o((s,p,o)) = o$$

  - for each multiply occurring non-distinguished variable x_l, given that $\{x_1, \ldots, x_k\} \in ndvar(q')$, assign it one of the triple patterns $t_j \in q'$ where it appears in, i.e., x_l appears in the triple pattern t_j, from that we obtain m_i's as:

$$m_i = \bigcup_{l=1}^{k} \{x_l \mapsto \alpha(x_l, t_j) \mid x_l \in t_j\}$$

$$\alpha(x,t) = \begin{cases} \varphi(s', f_p(t)) & \text{if } x = f_s(t) \text{ and } f_p(t) \in \mathbf{X} \\ \langle s \rangle \langle o \rangle f_o(t) & \text{if } x = f_s(t), f_o(t) \in \mathbf{X} \text{ and } f_p(t) \notin \mathbf{X} \\ \langle \bar{p} \rangle \langle \bar{s} \rangle f_s(t) & \text{if } x = f_p(t) \text{ and } f_s(t) \in \mathbf{X} \\ \langle \bar{o} \rangle \langle o \rangle f_o(t) & \text{if } x = f_p(t) \text{ and } f_o(t) \in \mathbf{X} \\ \varphi(o', f_p(t)) & \text{if } x = f_o(t) \text{ and } f_p(t) \in \mathbf{X} \\ \langle \bar{o} \rangle \langle \bar{s} \rangle f_s(t) & \text{if } x = f_o(t), f_s(t) \in \mathbf{X} \text{ and } f_p(t) \notin \mathbf{X} \end{cases}$$

  where in the function φ, s' and o' denote subject and object of the triple pattern t_j respectively. φ is defined as:

$$\begin{aligned} \varphi(s,a) &= \langle s \rangle \langle p \rangle a & \varphi(o,a) &= \langle \bar{o} \rangle \langle p \rangle a \\ \varphi(s, a \cdot b) &= \varphi(s,a) & \varphi(o, a.b) &= \varphi(o,b) \\ \varphi(s, a \intercal b) &= \big(\varphi(s,a) \vee \varphi(s,b)\big) & \varphi(o, a \intercal b) &= \big(\varphi(o,a) \vee \varphi(o,b)\big) \\ \varphi(s, a^+) &= \varphi(s,a) & \varphi(o, a^+) &= \varphi(o,a) \\ \varphi(s, a^*) &= \varphi(s,a) & \varphi(o, a^*) &= \varphi(o,a) \end{aligned}$$

– finally, the function $\mathcal{A}$ works inductively on the query structure using m to generate the formula. As for the left-hand side query, $\mathcal{R}$ is used to produce the encodings of regular expressions. $\mathcal{A}$ is the same as before except that it uses the new m and $\mathcal{R}$ functions.

$$\begin{aligned}
\mathcal{A}(q, m) &= \bigvee_{i=1}^{|m|} \mathcal{A}(q, m_i) \\
\mathcal{A}((x, y, z), m) &= \mathit{lfp}\big(X, \langle \bar{s} \rangle d(m, x) \wedge \mathcal{R}(d(m, y), d(m, z))\big) \\
\mathcal{A}(q_1 \mathsf{AND} q_2, m) &= \mathcal{A}(q_1, m) \wedge \mathcal{A}(q_2, m) \\
\mathcal{A}(q_1 \mathsf{UNION} q_2, m) &= \mathcal{A}(q_1, m) \vee \mathcal{A}(q_2, m) \\
d(m, x) &= \begin{cases} \varphi & \text{if } (x \mapsto \varphi) \in m \\ \top & \text{if } \mathit{unique}(x) \\ x & \text{otherwise} \end{cases}
\end{aligned}$$

where d is a function that fetches a subformula corresponding to the encoding of a variable given a mapping m and that variable as an input.

The disjuncts in the encoding guarantee that possible sets of substitutions m capture the intended semantics of a cyclic query that contains distinguished variables and constants in its cyclic component.

Example 7 (Encoding queries) Consider the encoding of $q \sqsubseteq q'$ where

$$q(x, z) = \ (x, (c \shortmid d) \cdot (a \shortmid b), z) \quad q'(x, z) = \ (x, c \shortmid d, y) \mathsf{AND} (y, a \shortmid b, z)$$

– The encoding of q is obtained by freezing the query and recursively constructing the formula using $\mathcal{A}$.

$$\begin{aligned}
\mathcal{A}(q) &= \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \mathcal{R}\big((c \shortmid d) \cdot (a \shortmid b), z\big)\big) \\
&= \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge (\langle p \rangle c \vee \langle p \rangle d) \wedge \langle o \rangle \langle s \rangle ((\langle p \rangle a \vee \langle p \rangle b) \wedge \langle o \rangle z)\big)
\end{aligned}$$

– The encoding of q' is as follows:
  – the constants and distinguished variables are encoded as nominals,
  – $y \in \mathit{var}(q')$ is encoded as $\varphi(o, (c \shortmid d))$, since y is an object of the triple $(x, (c \shortmid d), y)$. Hence, $m_1 = \{y \mapsto (\langle \bar{o} \rangle \langle p \rangle c \vee \langle \bar{o} \rangle \langle p \rangle d)\}$. On the other hand, y can also be encoded as $\varphi(s, (a \shortmid b))$, since y is a subject of the triple $(y, a \shortmid b, z)$. Thus, we get $m_2 = \{y \mapsto (\langle s \rangle \langle p \rangle a \vee \langle s \rangle \langle p \rangle b)\}$.
  – finally, we use $\mathcal{A}$ to encode q' recursively:

$$\begin{aligned}
\mathcal{A}(q', m) &= \bigvee_{i=1}^{|m|} \mathcal{A}(q', m_i) = \mathcal{A}(q', m_1) \vee \mathcal{A}(q', m_2) \\
&= \big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge (\langle p \rangle c \vee \langle p \rangle d) \wedge \langle o \rangle (\langle \bar{o} \rangle \langle p \rangle c \vee \langle \bar{o} \rangle \langle p \rangle d)\big) \\
&\quad \wedge \mathit{lfp}\big(Y, \langle \bar{s} \rangle (\langle \bar{o} \rangle \langle p \rangle c \vee \langle \bar{o} \rangle \langle p \rangle d) \wedge (\langle p \rangle a \vee \langle p \rangle b) \wedge \langle o \rangle z\big)\big) \\
&\quad \vee \\
&\quad \big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge (\langle p \rangle c \vee \langle p \rangle d) \wedge \langle o \rangle (\langle s \rangle \langle p \rangle a \vee \langle s \rangle \langle p \rangle b)\big) \\
&\quad \wedge \mathit{lfp}\big(Y, \langle \bar{s} \rangle (\langle s \rangle \langle p \rangle a \vee \langle s \rangle \langle p \rangle b) \wedge (\langle p \rangle a \vee \langle p \rangle b) \wedge \langle o \rangle z\big)\big)
\end{aligned}$$

Encoding right-hand side SPARQL$_{\text{pc}}$ Queries To encode the containment problem when the right-hand side query is a SPARQL$_{pc}$ query, we use a set of mappings of its non-distinguished variables into the frozen terms of the left-hand side query. That is, each non-distinguished variable is substituted with every term (constant, and distinguished and non-distinguished variables) of the left-hand side query. Since regular expression patterns cannot appear in subject and object positions of a triple pattern, we have to define a mapping which takes this into account.

Definition 7 (Generate Mappings) Given a query q and a SPARQL$_{pc}$ query q', a set of mappings m for the encoding $q \sqsubseteq q'$ can be produced as follows:

- non-distinguished variables occurring only once are encoded as $\top$.
- constants (i.e., URIs) are encoded as nominals.
- a non-distinguished variable occurs multiple times in q', then we produce a set of mappings $m = \{m_1, \ldots, m_n\}$ such that each m_i contains formula assignments to the non-distinguished variables of the query.

$$\begin{aligned} term(q) &= const(q) \cup dvar(q) \cup ndvar(q) \cup regex(q) \\ subj(q), pred(q), obj(q) &= \text{all subjects (resp. predicates and objects)} \\ &\quad \text{of the triple patterns in } q. \end{aligned}$$

$$m = \bigtimes_{i=0}^{k} \bigcup_{r=0}^{n} \{x_i \mapsto \begin{cases} l_r \in term(q) \text{ if } x_i \in pred(q) \setminus subj(q) \cup obj(q) \\ l_r \in term(q) \setminus regex(q) \text{ otherwise} \end{cases} \}$$

Once the mappings m are generated, the encoding of SPARQL$_{pc}$ queries are generated in the same way as that of non-SPARQL$_{pc}$ queries, i.e., using the function $\mathcal{A}$.

The maximum size of the encoding of the right-hand side SPARQL$_{pc}$ query is $|term(q)|^{|ndvar(q')|}$ where $|term(q)|$ is the size of q (in terms of the number of constants and variables) and $|ndvar(q)|$ is the number of multiply appearing non-distinguished variables in q'.

Example 8 (Encoding SPARQL$_{\text{pc}}$ queries) Consider the following SPARQL$_{\text{pc}}$ queries.

$$q() = (x, r, x) \qquad q'() = (w_1, w_2, w_3)\mathsf{AND}(w_3, w_2, w_1)$$

Let us produce the encodings of the containment tests: $q \sqsubseteq q'$ and $q' \sqsubseteq q$. Firstly, we derive the mappings for $q \sqsubseteq q'$ by assigning the frozen terms of q to the non-distinguished variables of q' in all possible ways:

$$\begin{aligned} m_1 &= \{w_1 \mapsto x, w_2 \mapsto x, w_3 \mapsto x\} & m_5 &= \{w_1 \mapsto x, w_2 \mapsto r, w_3 \mapsto x\} \\ m_2 &= \{w_1 \mapsto r, w_2 \mapsto x, w_3 \mapsto x\} & m_6 &= \{w_1 \mapsto x, w_2 \mapsto x, w_3 \mapsto r\} \\ m_3 &= \{w_1 \mapsto r, w_2 \mapsto r, w_3 \mapsto x\} & m_7 &= \{w_1 \mapsto r, w_2 \mapsto x, w_3 \mapsto r\} \\ m_4 &= \{w_1 \mapsto r, w_2 \mapsto r, w_3 \mapsto r\} & m_8 &= \{w_1 \mapsto x, w_2 \mapsto r, w_3 \mapsto r\} \end{aligned}$$

Since there are 2 terms in q, i.e., $|q| = 2$, and 3 ndvars in q', i.e., $|ndvar(q')| = 3$, we have $|q|^{|ndvar(q')|}$ different mappings. $q \sqsubseteq q'$ into $\mathcal{A}(q) \wedge \neg\mathcal{A}(q', m)$ where

$$\begin{aligned}
\mathcal{A}(q) &= \mathcal{A}\big((x, r, x)\big) \\
\mathcal{A}(q', m) =& \mathcal{A}\big((w_1, w_2, w_3)\mathsf{AND}(w_3, w_2, w_1), m_1\big) \vee \mathcal{A}\big((w_1, w_2, w_3)\mathsf{AND} \\
& (w_3, w_2, w_1), m_2\big) \\
& \vee \cdots \vee \mathcal{A}\big((w_1, w_2, w_3)\mathsf{AND}(w_3, w_2, w_1), m_8\big) \\
=& \mathcal{A}\big((x, x, x)\mathsf{AND}(x, x, x)\big) \vee \mathcal{A}\big((r, x, x)\mathsf{AND}(x, x, r)\big) \\
& \vee \cdots \vee \mathcal{A}\big((x, x, r)\mathsf{AND}(r, x, x)\big)
\end{aligned}$$

On the other hand, the mappings for the encoding of $q' \sqsubseteq q$ are the following:

$m_1 = \{x \mapsto w_1, r \mapsto \top\}, m_2 = \{x \mapsto w_2, r \mapsto \top\},$ and $m_3 = \{x \mapsto w_3, r \mapsto \top\}$.

Accordingly, $q' \sqsubseteq q \longrightarrow \mathcal{A}(q') \wedge \neg\mathcal{A}(q, m)$ becomes:

$$\begin{aligned}
\mathcal{A}(q') &= \mathcal{A}\big((w_1, w_2, w_3)\mathsf{AND}(w_3, w_2, w_1)\big) \\
\mathcal{A}(q', m) &= \mathcal{A}\big((x, r, x), m_1\big) \vee \mathcal{A}\big((x, r, x), m_2\big) \vee \mathcal{A}\big((x, r, x), m_3\big) \\
&= \mathcal{A}\big((w_1, \top, w_1)\big) \vee \mathcal{A}\big((w_2, \top, w_2)\big) \vee \mathcal{A}\big((w_3, \top, w_3)\big)
\end{aligned}$$

From the above encodings, it is possible to check that $q \sqsubseteq q'$ but $q' \not\sqsubseteq q$.

4.3 Reducing Containment to Unsatisfiability

Once correct encodings of queries have been produced, the next step requires reducing containment to the validity problem in the μ-calculus. Intuitively, $q \sqsubseteq q'$ is reduced to the validity test of $\mathcal{A}(q) \wedge \neg\mathcal{A}(q', m) \wedge \varphi_r$. In doing so, we prove the soundness and completeness of the reduction. To give an overview of the overall procedure, we start of with an example.

Example 9 (Containment test) We show the containment of the following queries: select all descendants and ancestors (q) whose names are "john" and (q') who share the same name.
$q\{x, z\} = (x, name, \text{“}john''\text{)}\mathsf{AND}(x, ancestor^*, z)\mathsf{AND}(z, name, \text{“}john''\text{)}$
$q'\{x, z\} = (x, name, y)\mathsf{AND}(x, ancestor^*, z)\mathsf{AND}(z, name, y)$
We proceed by first obtaining their encodings. Consider the encoding of $q \sqsubseteq q'$, we encode triple patterns using θ and $m = \{y \mapsto \langle\bar{o}\rangle name\}$.

$$\begin{aligned}
\mathcal{A}(q) &= \mathit{lfp}\big(X, \theta(x, name, \text{“}john''\text{)}\big) \wedge \mathit{lfp}\big(X, \theta(x, ancestor^*, z)\big) \wedge \\
&\quad \mathit{lfp}\big(X, \theta(z, name, \text{“}john''\text{)}\big) \\
\neg\mathcal{A}(q', m) &= \mathit{gfp}\big(X, \neg\theta(x, name, \langle\bar{o}\rangle name)\big) \vee \mathit{gfp}\big(X, \neg\theta(x, ancestor^*, z)\big) \vee \\
&\quad \mathit{gfp}\big(X, \neg\theta(z, name, \langle\bar{o}\rangle name)\big)
\end{aligned}$$

The formula $\mathcal{A}(q) \wedge \neg\mathcal{A}(q', m) \wedge \varphi_r$ is unsatisfiable because $\mathcal{A}(q)$ requires its model to satisfy the encoding of each triple pattern somewhere in the

transition system. On the contrary, the formula $\neg\mathcal{A}(q', m)$ requests this model to satisfy the negation of the encoding of the triples in the entire transition system. Hence, this leads to a contradiction and no such model exists for the formula. Therefore, $q \sqsubseteq q'$. On the other hand, it can be verified similarly that $q' \not\sqsubseteq q$.

Lemma 2 *Given a SPARQL query $q(\overrightarrow{w})$, there exists an RDF graph G such that $[\![q(\overrightarrow{w})]\!]_G \neq \emptyset$.*

Proof (*Sketch*) From any query it is possible to build a homomorphic graph by collecting all triples connected by AND and only those at the left of UNION (replacing variables by blank nodes). If triples contain regular expression patterns (a.k.a. path patterns), then we use a function g to inductively transform them into triple patterns as shown below:

$$
\begin{aligned}
g((x,y,z)) &= (x,y,z)\\
g((x,e,z)) &= (x,e,z)\\
g((x,e_1 \cdot e_2,z)) &= g((x,e_1,y))\mathsf{AND}g((y,e_2,z)) \quad \text{where } y \text{ is a fresh variable}\\
g((x,e_1 \shortmid e_2,z)) &= g((x,e_1,y))\mathsf{UNION}g((y,e_2,z)) \quad \text{where } y \text{ is a fresh variable}\\
g((x,e^+,z)) &= g((x,e,y_1))\mathsf{AND}g((y_1,e,y_2))\mathsf{AND}\cdots\mathsf{AND}g((y_i,e,z))\\
&\quad \text{such that each } y_i \text{ is a fresh variable where } 1 \leq i \leq n\\
&\quad \text{and } n \geq 1. \text{ If } n = 1, \text{ then } y_i = z\\
g((x,e^*,z)) &= g((x,\epsilon,z))\mathsf{UNION}g((x,e^+,z))
\end{aligned}
$$

The graph obtained in such a way is consistent as all RDF graphs are [31], thus, this graph satisfies the query q.

Next, we prove the correctness of the encoding procedure:

Lemma 3 *Let q be a SPARQL query, for every restricted transition system K whose associated RDF graph is G, we have that $[\![q]\!]_G \neq \emptyset$ iff $[\![\mathcal{A}(q,m) \wedge \varphi_r]\!]^K \neq \emptyset$.*

Proof ($\Rightarrow$) $[\![q]\!]_G \neq \emptyset$ implies that G is at least a canonical instance of q and can be produced in the same way as in Lemma 2. Consequently, since G is an instance of q, G is a model of q. Now, we construct a transition system $\sigma(G) = (S, R, L)$ in the same way as is done in Definition 2. To prove that $\sigma(G)$ is a model of $\mathcal{A}(q,m) \wedge \varphi_r$, we consider three cases:

(i) when q is cycle-free,
(ii) when q contains a cyclic component among its non-distinguished variables which are connected to a distinguished variable or constant, and
(iii) when q is purely cyclic (SPARQL$_{pc}$ query)

First, (i) when q is cycle-free, then the encoding of the non-distinguished variables with $\top$ suffices to justify that $\sigma(G)$ is a model of its encoding. Since $\top$ gets instantiated (in all possible ways) with the constants (and frozen variables) appearing in the left-hand side query.

Second, (ii) when q contains a cyclic component that is a connected to a non-distinguished variable or constant, in this case, its encoding is $\bigvee_{i=1}^{|m|} \mathcal{A}(q, m_i)$. Using the nominals that correspond to the encodings of the distinguished variables and constants of q, one can successfully create a formula that can encode multiply occurring non-distinguished variables. Henceforth, creating a formula that is satisfiable in cyclic models. Further, the disjuncts in the encoding guarantee that possible set of substitutions in m capture the intended semantics of a query that has a cyclic component which involves a distinguished variable or a constant. One can verify that $\sigma(G)$ is a model of the disjuncts $\mathcal{A}(q, m_i)$, this is because nominals encoding the constants and distinguished variables are true in $\sigma(G)$ as they exist in G. Further, since the formulas corresponding to the encoding of the non-distinguished variables are obtained using the constants or distinguished variables, they are also true in $\sigma(G)$. Therefore, $\mathcal{A}(q, m)$ is satisfiable in $\sigma(G)$. To elaborate, if l is either x or y or z of the triple $(x, y, z) \in q$,

- for l either a distinguished variable or constant, l is satisfiable in $\sigma(G)$ since $[\![l]\!]^{\sigma(G)} \in L(l)$,
- for l a uniquely appearing non-distinguished variable, l is true in $\sigma(G)$ since its encoding $\top$ is true everywhere in the transition system,
- a multiply occurring non-distinguished variable l is true in $\sigma(G)$ since $\exists t \in S'.t \in L(c) \wedge t \in [\![c]\!]^{\sigma(G)}$ and the encoding of l is $\langle a\rangle\langle b\rangle c$, where c is a nominal encoding a constant or distinguished variable of the triple (x, y, z) and $a, b \in \{s, p, o, \bar{s}, \bar{p}, \bar{o}\}$.
- for $l = e$ a regular expression, its encoding $\mathcal{R}(e, z)$ is satisfiable in $\sigma(G)$. This can be worked out inductively on the construction of the formula. Alternatively, since triple patterns that contain regular expressions can be transformed into a query without them and the latter case has been shown above.

Thus, since $\sigma(G)$ is a restricted transition system, we obtain $[\![\mathcal{A}(q, m)\varphi_r]\!]_G \neq \emptyset$.

Third, (iii) when q is purely cyclic ($\Leftarrow$) Assume that $[\![\mathcal{A}(q, m) \wedge \varphi_r]\!]^K \neq \emptyset$. We now create an RDF graph G from K as follows:

- if $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, y_i, z_i) \in q$ if $s_1 \in L(x_i) \wedge s_2 \in L(y_i) \wedge s_3 \in L(z_i)$, then $(x_i, y_i, z_i) \in G$. This case holds if x_i, y_i and z_i are either distinguished variables or constants. Note here that if x_i or y_i or z_i appear in another triple $t_j = (x_j, y_j, z_j) \in q$, then the equivalent item in t_j is replaced with the value of the corresponding entry in t_i.
- if $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, y_i, z_i) \in q$ if $s_1 \in L(x_i) \wedge s_2 \in L(y_i)$, then $(x_i, y_i, c_i) \in G$ where c_i is a fresh constant. This case holds if z_i is a non-distinguished variable. Similarly, the case when x_i or y_i or both are variables can be worked out.
- if $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, e_i, z_i) \in q$ if $s_1 \in [\![\langle s\rangle\langle p\rangle e_i]\!]^K \in L(e_i) \wedge s_2 \in$

$L(e_i) \wedge s_3 \in [\![\langle \bar{o} \rangle \langle p \rangle e_i]\!]^K$, then $(c_i, e_i, d_i) \in G$ where c_i and d_i are fresh constants. This case holds if x_i and y_i are multiply occurring non-distinguished variables. Similarly, all the other cases can be worked out.

Since G is a technical construction obtained from a restricted transition system associated to q, it holds that $[\![q]\!]_G \neq \emptyset$.

In the following, for the sake of legibility, we denote $\eta(\mathcal{C}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q', m) \wedge \varphi_r$ by $\Phi(\mathcal{C}, q, q')$.

Theorem 1 (Soundness) *Given SPARQL queries $q(\overrightarrow{w})$, $q'(\overrightarrow{w})$, and a set of $\mathcal{ALCH}$ axioms $\mathcal{C}$, if $\Phi(\mathcal{C}, q, q')$ is unsatisfiable, then $q(\overrightarrow{w}) \sqsubseteq^{\mathcal{C}} q'(\overrightarrow{w})$.*

Proof We show the contrapositive. If $q \not\sqsubseteq^{\mathcal{C}} q'$, then $\Phi(\mathcal{C}, q, q')$ is satisfiable. One can verify that every model G of $\mathcal{C}$ in which there is at least one tuple satisfying q but not q' can be turned into a transition system model for $\Phi(\mathcal{C}, q, q')$. To do so, consider a graph G that satisfies schema axioms $\mathcal{C}$. Assume also that there is a tuple $\overrightarrow{a} \in [\![q]\!]_G$ and $\overrightarrow{a} \notin [\![q']\!]_G$. Let us construct a transition system K from G. From Lemma 1, we obtain that $[\![\eta(\mathcal{C})]\!]^K \neq \emptyset$. Further, since K is a restricted transition system (Definition 2), $[\![\varphi_r]\!]^K \neq \emptyset$. At this point, it remains to verify that $[\![\mathcal{A}(q)]\!]^K \neq \emptyset$ and $[\![\mathcal{A}(q', m)]\!]^K = \emptyset$.

Let us construct the formulas $\mathcal{A}(q)$ and $\mathcal{A}(q', m)$ by first skolemizing the distinguished variables using the answer tuple $\overrightarrow{a}$. Consequently, from Lemma 3 one obtains, $[\![\mathcal{A}(q)]\!]^K \neq \emptyset$. However, $[\![\mathcal{A}(q', m)]\!]^K = \emptyset$, this is because the nominals in the formula corresponding to the constants and non-distinguished variables are not satisfied in K. This implies that $[\![\neg\mathcal{A}(q', m)]\!]^K \neq \emptyset$. This is justified by the fact that if a formula φ is satisfiable in a restricted transition system, then $[\![\varphi]\!]^K = S$ thus $[\![\neg\varphi]\!]^K = \emptyset$. So far we have $[\![\eta(\mathcal{C})]\!]^K \neq \emptyset$ and $[\![\varphi_r]\!]^K \neq \emptyset$ and $[\![\mathcal{A}(q)]\!]^K \neq \emptyset$ and $[\![\neg\mathcal{A}(q', m)]\!]^K \neq \emptyset$. Without loss of generality, $[\![\Phi(\mathcal{C}, q, q')]\!]^K \neq \emptyset$. Therefore, $\Phi(\mathcal{C}, q, q')$ is satisfiable.

Theorem 2 (Completeness) *Given SPARQL queries $q(\overrightarrow{w})$, $q'(\overrightarrow{w})$, and a set of $\mathcal{ALCH}$ axioms $\mathcal{C}$, if $\Phi(\mathcal{C}, q, q')$ is satisfiable, then $q\{\overrightarrow{w}\} \not\sqsubseteq^{\mathcal{C}} q'\{\overrightarrow{w}\}$.*

Proof If $\Phi(\mathcal{C}, q, q')$ is satisfiable, then $\exists K.[\![\Phi(\mathcal{C}, q, q')]\!]^K \neq \emptyset$. Consequently, K is a restricted transition system due to $[\![\varphi_r]\!]^K \neq \emptyset$ (Proposition 1). Using $K = (S' \cup S'', R, L)$ we construct a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{C}$ such that $q \not\sqsubseteq q'$ holds:

- $\Delta^{\mathcal{I}} = S'$, $A^{\mathcal{I}} = [\![A]\!]^K$ for each atomic concept A,
- $\top^{\mathcal{I}} = [\![\top]\!]^K$, for a top concept,
- $r^{\mathcal{I}} = \{(s, s') \mid \forall t \in [\![r]\!]^K \wedge t' \in S'' \wedge (s, t') \in R(s) \wedge (t', t) \in R(p) \wedge (t', s') \in R(o)\}$ for each atomic role r,
- for each constant c in q and q', $c^{\mathcal{I}} = [\![c]\!]^K$,
- for each distinguished and non-distinguished variable v in q, $v^{\mathcal{I}} = [\![v]\!]^K$, and
- for each distinguished variable v in q', $v^{\mathcal{I}} = [\![v]\!]^K$.

Lemma 1 may be used to check that $\mathcal{I}$ is a model of $\mathcal{C}$. Thus, it remains to show that $[\![q]\!]_{\mathcal{I}} \not\subseteq [\![q']\!]_{\mathcal{I}}$. From our assumption, one anticipates the following:

$$\begin{aligned}[\![\mathcal{A}(q) \wedge \neg\mathcal{A}(q')]\!]^K \neq \emptyset &\Rightarrow [\![\mathcal{A}(q)]\!]^K \neq \emptyset \text{ and } [\![\neg\mathcal{A}(q', m)]\!]^K \neq \emptyset \\ &\Rightarrow [\![\mathcal{A}(q)]\!]^K \neq \emptyset \text{ and } [\![\mathcal{A}(q', m)]\!]^K = \emptyset\end{aligned}$$

Note here that, if a formula φ is satisfiable in a restricted transition system K_r, then $[\![\varphi]\!]^{K_r} = S$. We use a function f to construct an RDF graph G from the interpretation $\mathcal{I}$. f uses assertions in $\mathcal{I}$ to form triples:

$$\begin{aligned}f(a \in A^{\mathcal{I}}) &= (a, \texttt{type}, A) \in G \\ f((a,b) \in r^{\mathcal{I}}) &= (a, r, b) \in G \\ f((a,b) \in (r^-)^{\mathcal{I}}) &= (b, r, a) \in G \\ f((x,y,z)) &= (x,y,z) \in G,\ \forall (x,y,z) \in q\end{aligned}$$

As a consequence, $[\![q]\!]_G \neq \emptyset$ and $[\![q']\!]_G = \emptyset$ because G contains all those triples that satisfy q and not q'. Therefore, we get $[\![q]\!]_G \not\subseteq [\![q']\!]_G$. Fundamentally, there are two issues to be addressed (i) when q' is not cyclic and (ii) when q' contains a cycle. (i) if there are no cycles in q', then replacing non-distinguished variables with $\top$ suffices (Proof of Lemma 3). On the other hand, (ii) can be dealt with nominals, i.e., since cycles can be expressed by a formula of a μ-calculus extended with nominals and inverse, cyclic queries can be encoded by such a formula. Hence, the constraints expressed by $\neg\mathcal{A}(q', m)$ are satisfied in a transition system containing cycles

4.3.1 Complexity

Due to duplication in the encoding of the right-hand side query q', the size of $|\mathcal{A}(q', m)|$ is exponential in terms of the non-distinguished variables that appear in cycles in the query. Therefore, we obtain a 2ExpTime upper bound for containment. The problem is solvable in ExpTime if there is no cycle in the right-hand side query. In this case, this complexity is a lower bound due to the complexity of satisfiability in the μ-calculus.

Proposition 2 *SPARQL query containment under $\mathcal{ALCH}$ schema can be solved in a double exponential amount of time.*

Proof This proposition is a consequence of Theorem 1 and 2. The size of the encoding of the containment problem is exponential due to cycles in the right-hand side query. In the μ-calculus, the satisfiability test of a formula can be performed in exponential time. Consequently, containment of SPARQL queries can be performed in a double exponential amount of time.

Next we design a benchmark for testing the containment of SPARQL queries.

5 Containment Benchmark

SPARQL Query containment has recently been investigated theoretically and some query containment solvers are available. Yet, there were no benchmark to compare these systems and foster their improvement. In order to experimentally assess implementation strengths and limitations, we provide a SPARQL containment test benchmark. It has been designed with respect to both the capabilities of existing solvers and the study of typical queries. Some solvers support optional constructs and cycles, while other solvers support projection, union of conjunctive queries and RDF Schemas. The study of query demographics on DBPedia logs shows that the vast majority of queries are acyclic and a significant part of them uses UNION or projection. We thus test available solvers on their domain of applicability on three different benchmark suites. These experiments show that (i) tested solutions are overall functionally correct, (ii) in spite of its complexity, SPARQL query containment is practicable for acyclic queries, (iii) state-of-the-art solvers are at an early stage both in terms of capability and implementation.

5.1 Query containment solvers

We briefly present three state-of-the-art query containment solvers used in the experiments. Our goal is to characterize their capabilities in order to design appropriate benchmarks. Thus, we also analyze actual queries used on the semantic web.

5.1.1 SPARQL-Algebra

SPARQL-Algebra is an implementation of SPARQL query subsumption and equivalence based on the theoretical results in [38]. This implementation supports AND and OPTIONAL queries with no projection.

5.1.2 AFMU

AFMU (Alternation Free two-way μ-calculus) [51] is a satisfiability solver for the alternation-free fragment of the μ-calculus [37]. It is a prototype implementation which determines the satisfiability of a μ-calculus formula by producing a yes-or-no answer. To turn it into a query containment solver, it is necessary to turn the problem into a μ-calculus satisfiability problem.

5.1.3 TreeSolver

The XML tree logic solver *TreeSolver*[2] performs static analysis of XPath queries which comprise containment, equivalence and satisfiability. To perform these tasks, the solver translates XPath queries into μ-calculus formulas

[2] http://tyrex.inria.fr/websolver/

Table 2 Comparison of features supported by current systems.

System	projection	UCQ	optional	blanks	cycles	RDFS
SPARQL-Algebra			√		√	
AFMU	√	√		√		√
TreeSolver	√	√		√		√

and then tests the unsatisfiability of the formula. Unlike AFMU, the unsatisfiability test is performed in time of $2^{\mathcal{O}(n)}$ whereas it is $2^{\mathcal{O}(n \log n)}$ for AFMU, such that n is the size of the formula.

Using TreeSolver follows the same procedure as using AFMU with a slightly different encoding. Indeed, because TreeSolver is restricted to tree-shaped models, we use a specific encoding of query formulas.

A summary of the features supported by these query containment solvers is presented in Table 2.

Out of the three systems, SPARQL-Algebra is self contained whereas the other two are μ-calculus satisfiability solvers that need an intermediate query translation into formulas to determine containment.

Part of the query structures can be transformed into concept expressions in description logics and submitted to satisfiability (or subsumption) tests as well. So, in principle, query containment solvers based on description logic reasoners could be designed. However, we do not know any such solver.

5.2 State of the query landscape

To assess the type of queries actually used, we analyzed DBpedia query logs[3]. We report on two log sets because there is a lot of variation between them. 2 905 035 queries from the logs were syntactically correct (90%). The results are given in Table 3. We tested the cyclicness of queries and found out that more than 90% of these queries are acyclic (94% of the small sample and 99% on the total). This justifies designing and evaluating acyclic queries. Projection is used in 11% of the large log and 22% of the smaller one, but such figures more than double if only "SELECT *" queries are counted as projection-free queries.

OPTIONAL are used in around 30% of queries, whereas UNION is used in 18% of those in the full log and 43% in the large one. Union of conjunctive queries with optional are 15 to 30% of the logs. This make them operators to be supported in query containment.

On the contrary, on this huge set of queries, none contain CDFC components. This support the choice of solvers, in addition to the computation cost of query containment in this case, to optimize for the other type of queries.

[3] DBpedia 3.5.1 logs (`ftp://download.openlinksw.com/support/dbpedia/`) contain 3 210 368 queries between 30/04/2010 and 20/07/2010 and 378 530 queries of 13/07/2010 only.

Table 3 Query characteristics of the full DBPedia logs.

	projection			no projection		
operator	tree	dag	cycle	tree	dag	cycle
none	175 220	562	1	1 534 150	1 761	1 748
union	9	26 625	547	24	29 629	1 166
opt	2 052	685	0	311 608	722	1
filter	7 912	711	6	264 821	340	1
un-opt	0	306	0	0	12 659	1
opt-filt	7 991	779	0	4 933	52 401	0
filt-un	2	183	0	23 802	12 286	0
un-opt-filt	0	102 765	0	0	302 657	23 969

5.3 Benchmark design

We first present the design of containment benchmark suites. Each test suite is made of elementary test cases asking for the containment of one query into another. We then introduce the principles and software used for evaluating containment solvers. The benchmark and software is available on-line at `http://sparql-qc-bench.inrialpes.fr/`.

5.3.1 Structure of the benchmark

There are three qualitative dimensions along which tests can be designed: the type of graph pattern connectors (AND, UNION, MINUS, Projection, OPTIONAL, FILTER etc.), the type of ontology: (no schema, RDFS, SHI, OWL, etc.) and the query structure (tree, DAG, cyclic). In addition to these dimensions, quantitative measures are: the number of triple patterns, the number of variables, the number of triple patterns involving more than one variable (Tjoins), and the size of the ontology.

We designed test suites of homogeneous qualitative dimensions selected with respect to the capacity of the current state-of-the-art solvers. The benchmark contains three test suites:

- **C**onjunctive **Q**ueries with **No Proj**ection (CQNoProj)
- **U**nion of **C**onjunctive **Q**ueries with **Proj**ection (UCQProj)
- **U**nion of **C**onjunctive **Q**ueries under **RDFS** reasoning (UCQrdfs)

The test suites are designed to model increasing expected difficulty by using more constructors. We did not provide tests of cyclic queries since only one solver is currently able to deal with them. However, this would be a natural addition to these tests.

Each test suite contains tests of different quantitative measures. Most of them are used for conformance testing, i.e., testing that solvers return the

Table 4 The CQNoProj testsuite. In the AND column, figures correspond to the number of AND in the left-hand side query of the test. Vars is the number of variables in each queries and Tjoin the number of triples in which occurs at least two variables.

Test case	Problem	AND	Vars	Tjoin
nop1	Q1a ⊑ Q1b	1	1	0
nop2	Q1b ⊒ Q1a	0		0
nop3	Q2a ⊑ Q2b	5	3	3
nop4	Q2b ⊒ Q2a	5		3
nop5	Q3a ⊑ Q3b	2	2	2
nop6	Q3b ⊒ Q3a	1		1
nop7	Q4c ⊑ Q4b	5	3	2
nop8	Q4b ⊒ Q4c	3		2
nop9	Q6a ⊒ Q6b	2	3	1
nop10	Q6b ⊒ Q6a	2		1
nop11	Q6a ⊑ Q6c	2	3	1
nop12	Q6c ⊒ Q6a	0		1
nop13	Q6b ⊑ Q6c	2	3	1
nop14	Q6c ⊒ Q6b	0		1
nop15	Q7a ⊑ Q7b	9	10	9
nop16	Q7b ⊒ Q7a	10		9
nop17	Q8a ⊑ Q8b	3	4	3
nop18	Q8b ⊒ Q8a	2		3
nop19	Q9a ⊒ Q9b	4	3	2
nop20	Q9b ⊒ Q9a	4		2

correct answer, but we also identify some stress tests trying to evaluate solvers at or beyond their limits. We discuss these test suites below.

5.3.2 CQNoProj

This test suite is designed for containment of basic graph patterns. It contains conjunctive queries with no projection. We have identified 20 different test cases (nop1–nop20), each one testing containment between two queries. All the cases in this setting are shown in Table 4, along with the number of connectives and variables in the queries. The more difficult test used for stress testing are nop3, nop4, nop15, and nop16. The two former ones have a larger number of conjunction (and of Tjoin), while the two latter ones have an even larger number of conjunctions and variables. We have selected Tjoins (triples having two variables) as a measure of difficulty because simple triple joins may be compiled efficiently as tuples.

5.3.3 UCQProj

This test suite is made of 28 test cases, each comprising two acyclic union of conjunctive queries with projection. In fact, 14 tests contain projection only, 6 tests contains union only and 2 tests contains both (see Table 5.3.3). The test cases differ in the number of distinguished variables (*Dvars*) and connectives (conjunction or union). Particular stress tests are p3, p4 (without union nor projection), p15, p16, p23, and p24.

5.3.4 Benchmarking software architecture

For testing containment solvers, we designed an experimental setup which comprises several software components. This setup is illustrated in Figure 3. It simply considers a containment checker as a software module taking as input two SPARQL queries (q and q'), eventually an RDF Schema ($\mathcal{S}$), and returning true or false depending if q' is entailed by q (under the constraints of $\mathcal{S}$).

Table 5 The UCQProj test suite.

Test case	Problem	AND	UNION	Dvars	Vars	Tjoin	Test case	Problem	AND	UNION	Dvars	Vars	Tjoin
p1	Q11a $\sqsubseteq$ Q11b	1	0	1	1	0	p15	Q17a $\sqsubseteq$ Q17b	9	0	10	10	9
p2	Q11b $\sqsubseteq$ Q11a	0	0		1	0	p16	Q17b $\sqsubseteq$ Q17a	10	0		11	10
p3	Q12a $\sqsubseteq$ Q12b	5	0	3	3	3	p17	Q18a $\sqsubseteq$ Q18b	3	0	4	4	3
p4	Q12b $\sqsubseteq$ Q12a	5	0		3	3	p18	Q18b $\sqsubseteq$ Q18a	2	0		4	3
p5	Q13a $\sqsubseteq$ Q13b	2	0	2	2	2	p19	Q19a $\sqsubseteq$ Q19b	4	0	2	3	2
p6	Q13b $\sqsubseteq$ Q13a	1	0		2	1	p20	Q19b $\sqsubseteq$ Q19a	4	0		3	2
p7	Q14c $\sqsubseteq$ Q14b	3	0	1	3	2	p21	Q19c $\sqsubseteq$ Q19b	4	0	2	4	3
p8	Q14b $\sqsubseteq$ Q14c	5	0		3	2*	p22	Q19b $\sqsubseteq$ Q19c	4	0		3	2
p9	Q15a $\sqsubseteq$ Q15b	0	0	2	3	1	p23	Q20a $\sqsubseteq$ Q20b	2	7	10	10	9
p10	Q15b $\sqsubseteq$ Q15a	0	0		2	1	p24	Q20b $\sqsubseteq$ Q20a	8	1		10	9
p11	Q16a $\sqsubseteq$ Q16b	2	0	1	3	1	p25	Q21a $\sqsubseteq$ Q21b	6	2	2	4-6	5
p12	Q16b $\sqsubseteq$ Q16a	2	0		3	1	p26	Q21b $\sqsubseteq$ Q21a	8	0		6	5
p13	Q16a $\sqsubseteq$ Q16c	2	0	1	3	1	p27	Q22a $\sqsubseteq$ Q22b	3	1	2	2	2
p14	Q16c $\sqsubseteq$ Q16a	0	0		3	1	p28	Q22b $\sqsubseteq$ Q22a	3	1		2	2

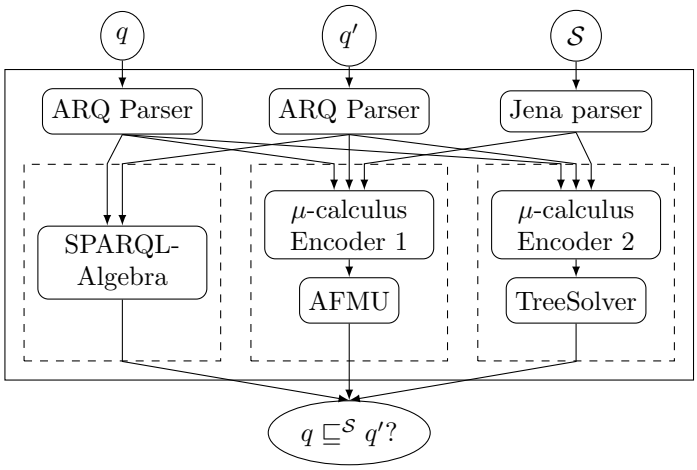


Fig. 3 Experimental setup for testing query containment. The tester (plain rectangle) parses queries and schemas and passes them to a solver wrapper (dashed rectangle).

This has been provided as a Java interface using Jena to express queries and RDF Schema. We have developed three wrappers implementing this interface for the three tested systems. Other systems may be wrapped in the same interface and tested in the same conditions. This platform may also be used for providing non regression tests for containment solvers.

Tests proceeds by providing test cases to the interface, timing the execution of the containment test around this common interface call. So timing occurs at the frontier of the dashed box of Figure 3, i.e., after query and schema parsing. This advantages SPARQL-Algebra, because it works directly on the ARQ representation, whereas the two other solvers have first to translate the ARQ

representation into a μ-calculus formula which is then parsed and transformed in each solver's internal representation.

5.4 Experiments

We evaluated the three identified query containment solvers with the three test suites. Rather than a definitive assessment of these solvers, our goal is to give first insights into the state-of-the-art and highlight deficiencies of engines based on the benchmark outcome. None of these systems is sharply optimized. However, their behavior is sufficient for highlighting test difficulty.

We run the experiments with an ordinary laptop computer running Mac OS X (specifically a 2.7 Ghz MacBook Pro with 16GB RAM).

The solvers were not genuinely reentrant. Hence, each test case has been run in a separate process after that the first case of each suite has been run as a warm up. All solvers are Java programs. The Java virtual machines were run with maximum heap size of 2024MB and a timeout at 20s (20000ms). Raising memory size to 1GB and timeout to 40s does not change timeout results. The μ-calculus solvers take advantage of a native BDD library. Using the native implementation doubles the speed of these solvers, however, it also brings large initialization time (in spite of warm-up set up).

Reported figures are the average of 5 runs (we run the tests 7 times and ruled out each time the best and worst performance).

5.4.1 CQNoProj Results

On conjunctive queries without projection, the SPARQL Algebra implementation is at least 10 times faster than the μ-calculus implementations (Figure 4). This comes as no surprise, since the latter are exponential time solvers whereas the former is a polynomial time solver.

AFMU times out on stress tests (nop3, nop4, nop15 and nop16). This happens whenever containment is determined between queries that contain more than 10 joins, such as in test cases nop15 and nop16. TreeSolver is able to deal with such cases albeit at the price of long response times. Overall, TreeSolver outperforms AFMU.

The fact that SPARQL-Algebra does not suffer from these sets, shows that the encoding of the μ-calculus solvers can be improved for such practical cases.

SPARQL-Algebra responded incorrectly, in test case nop7 (cf. Table 4), when blank nodes are used in the queries. It is not expected to deal with blank nodes. The other solvers are able to take them into account.

5.4.2 UCQProj Results

On the UCQProj test suite (see Section 5.3.3), we compared the two systems able to deal with UNION: TreeSolver and AFMU. Figure 5 shows that the performances of AFMU and TreeSolvers are roughly comparable with the notable

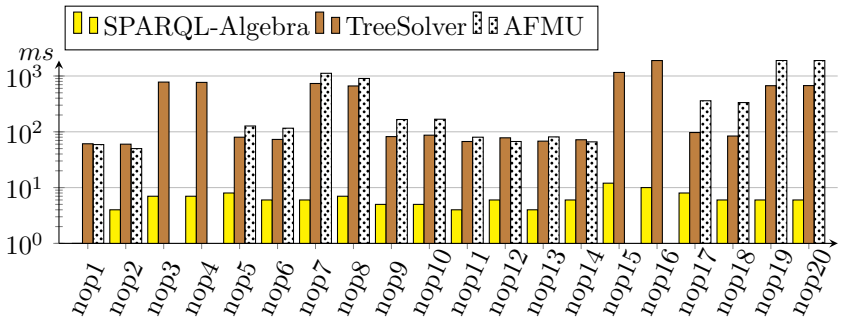


Fig. 4 Results for the CQNoProj test suite (logarithmic scale).

exception that TreeSolver answers for cases where AFMU fails. Specifically, TreeSolver times out only on test p24, while AFMU cannot deal with all stress tests: p3-p4, p15-p16, p23-p26. For this test suite, the necessary run time tends to be far longer as it often ends up in filling the available heap. For some of these tests (p15-16), performances could certainly be improved by adopting a better encoding of triples.

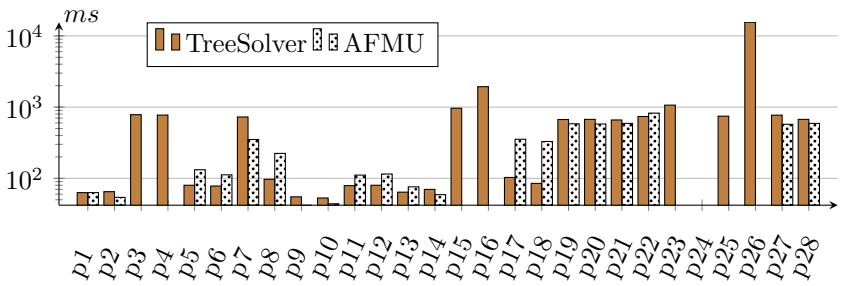


Fig. 5 Results for the UCQProj test suite (logarithmic scale).

5.4.3 Discussion

In summary, all solvers under all experimental settings responded positively i.e., they all determined containment correctly under their stated application limits (we tested this independently). However, from these experiments, a lot remains to be done in order to alleviate the shortcomings of the current systems.

SPARQL-Algebra is faster on its domain of application. The advantages of this solver compared to the others are that it supports subsumption of OPTIONAL query patterns and also cyclic CQs. However, blank nodes are not supported.

AFMU is able to determine containment of acyclic UCQs under ontological axioms. For queries of reasonable size, the solver determined their containment correctly. The problem is that when queries have a larger size, e.g., more than 8 joins, the solver saturates memory. This is shown for test cases nop15 and nop16 (Figure 4) as well as for test cases p15 and p16 (Figure 5). However, the implementation of this solver is not optimal: the authors have documented improvements. Moreover, determining containment of general UCQs (beyond the acyclic ones) will require extending the solver.

TreeSolver has globally similar limitations as AFMU: no support for cyclic queries and difficulty with queries of large size (as we can expect from worst case complexity), such as nop16. However, TreeSolver globally outperforms AFMU: TreeSolver is most often much faster, and, moreover capable of successfully dealing with more tests. This can probably be explained by the fact that the TreeSolver's algorithm [27] is based on a least-fixpoint computation, whereas AFMU's algorithm is based on a greatest-fixpoint computation [51]. By nature, AFMU's algorithm starts from all possibilities and repeatedly removes inconsistencies until a fixpoint is reached. In contrast, TreeSolver's algorithm basically starts from the emptyset and repeatedly tries to prove new relevant branches until it finds a fully proved model. A major consequence is that AFMU is required to compute a whole fixpoint each time before concluding about the existence (or inexistence) of a model. The situation is very different with TreeSolver, that can conclude as soon as it finds a (fully proved) satisfying model, without necessarily having to build a fixpoint completely before concluding about satisfiable formulas.

Determining the type of queries to compare (cyclic, disjunctive, with blank nodes, with projections, etc.) is easy. Hence, it is possible to build a system assembling these solvers and providing the best possible performances for each case.

6 Related Work

Query optimization has been the subject of an important research effort for many types of query languages, with the common goal of speeding up query processing. The works found in [49,29,47] considered the problem of SPARQL query optimization. So, this paper can be used to prove the correctness of query rewriting techniques. In the following we briefly review works that previously established closely related results for related query languages.

An early formalization of RDF(S) graphs has been presented in [30], in which the complexity of query evaluation and containment is also studied. The authors investigate a datalog-style, rule-based query language for RDF(S) graphs. In particular, they establish the NP-completeness of query containment over simple RDF graphs, this result is also published in the RDF semantics document [31]. The query language is rather simple compared to SPARQL and no constraints were assumed for the problem. [48] provides algorithms for the containment and minimization of RDF(S) query patterns utilizing concept and property hierarchies for the query language RQL (RDF Query Language). The NP-completeness is established for query containment concerning conjunctive and union of conjunctive queries. All of the results below and the ones in this paper were obtained under the assumption that queries are evaluated under set semantics. This means that the database relations given as inputs to queries are sets and that queries return sets as answers. In real database systems, however, queries are usually evaluated under bag semantics, not set semantics: input database relations may be bags (multisets), and queries may return bags as answers. The same holds for SPARQL in which duplicate tuples are not eliminated unless explicitly specified in the syntax using the SELECT DISTINCT construct. This is because SPARQL, instead of being defined according to its logical semantics, is defined as graph manipulation. However, most of the studies on query containment use set semantics rather than bag semantics. In particular, containment becomes undecidable (even for) union of conjunctive queries under bag semantics [33]. Thus, this study relies on the set semantics of SPARQL to obtain decidability results.

The work in [29], investigated static analysis of SPARQL queries that are embedded in a Java program. It checks the correctness of the syntaxes of RDF data, SPARQL queries, and SPARUL update queries. Beyond this, their system called SWOBE (Semantic Web Objects Database Programming Language), detects if a query has a non-empty result set. Most recently, static analysis and optimization of OPTIONAL graph patterns is studied in [38,43]. Pichler et all carried out a comprehensive complexity analysis of containment and equivalence for several fragments of SPARQL with OPTIONAL. Both papers concentrate on complexity results and it is unclear whether these complexity bound can be leveraged in practical terms in a usable implementation. However, the containment problem under Schema has not been considered in either of the two articles. See Table 6 for a summary of the result reported in [43], where *wd* denotes the well designed fragment and $\{\cup\}, \{\pi\}$ denote the UNION and SELECT extensions, respectively. Besides works that focus on querying

Table 6 The Containment complexity for well-designed SPARQL fragments

$Q_1 \setminus Q_2$	wd	$wd + \{\cup\}$	$wd + \{\pi\}$	$wd + \{\cup, \pi\}$
wd	NP-complete	Π_2^P-complete	undecidable	undecidable
$wd + \{\cup\}$	NP-complete	Π_2^P-complete	undecidable	undecidable
$wd + \{\pi\}$	NP-complete	Π_2^P-complete	undecidable	undecidable
$wd + \{\cup, \pi\}$	NP-complete	Π_2^P-complete	undecidable	undecidable

RDF graphs, in the following, we explore the relations and containment problem between SPARQL and query languages from other domains.

SPARQL vs. Relational Algebra It has been shown that SPARQL is equally expressive as relational algebra (RA) [4]. It is easy to see that relational algebra with SPJUD (Selection, Projection, Join, Union and Difference) [1] operators is equivalent to that of SPARQL with SELECT, AND, UNION, OPTIONAL and FILTER. The algebraic operators that are defined in SPARQL resemble the algebraic operators defined in relational algebra; in particular, AND is mapped to the algebraic join, FILTER is mapped to the algebraic selection operator, UNION is mapped to the union operator, OPTIONAL is mapped to the left outer join (which allows for the optional padding of information), and SELECT is mapped to the projection operator. As opposed to the operators in relational algebra, which are defined on top of relations with fixed schema, the algebraic SPARQL operators are defined over so called mapping sets, obtained when evaluating triple patterns. In contrast to the fixed schema in relational algebra, the "schema" of mappings in SPARQL algebra is loose in the sense that such mappings may bind an arbitrary set of variables. This means that in the general case we cannot give guarantees about which variables are bound or unbound in mappings that are obtained during query evaluation.

Studies on the translation of SPARQL into relational algebra and SQL [24,17] indicate a close connection between SPARQL and relational algebra in terms of expressiveness. In [44], a translation of SPARQL queries into a datalog fragment (non-recursive datalog with negation) that is known to be equally expressive as relational algebra was presented. This translation makes the close connection between SPARQL and rule-based languages explicit and shows that RA is at least as expressive as SPARQL. Tackling the opposite direction, it was recently shown in [4] that SPARQL is relationally complete, by providing a translation of the above-mentioned datalog fragment into SPARQL. As argued in [4], the results from [44] and [4] taken together imply that SPARQL has the same expressive power as relational algebra. From early results on query containment in relational algebra and first-order logic, one can infer that containment in relational algebra is undecidable. Therefore, containment of SPARQL queries is also undecidable. Hence, in this paper, we considered various fragments of SPARQL to study containment.

Semistructured data In line with CQs in databases and description logic, regular path query languages are used to query arbitrary length paths in graph databases — in semi structured data. Like CQs, they have been used and studied widely. They are different from CQs in that, they allow recursion by using regular expression patterns. The problem of containment has been addressed for extensions of this language. In this regard, a prominent language used in semi-structured data is XPath. This language has been studied extensively over the last decade. These studies range from extending or reducing to static analysis. Static analysis of XPath queries has been studied in [27], encompassing containment, equivalence, coverage, and satisfiability of XPath queries. In fact, [27] inspired the approach to study these problems using a graph logic and provide a working implementation.

Other notable results come from the study of Regular Path Queries (RPQs). RPQs are extremely useful for expressing complex navigations in a graph. In particular, union and transitive closure are crucial when we do not have a complete knowledge of the structure of the knowledge base as is the case for RDF graphs. Containment of (two-way) regular path queries (2RPQs) have been studied extensively [11,12,7]. These languages are used to query graph databases and containment has been shown to be PSPACE-complete, this complexity bound jumps to EXPTIME-hard under the presence of functionality constraints. On the other hand, the containment of conjunctive 2RPQs is EXPSPACE-complete, this bound jumps to 2EXPTIME when considered under expressive description logic (DL) constraints [14]. However, it is exponential if the query on the right hand side has a tree structure [13]. Further, paths are included in the new version of SPARQL. Query evaluation and containment of this language has recently been studied in [21,18,36].

Containment under constraints Query containment has also been studied under different kinds of constraints. Results in this setting include, decidability of conjunctive query containment under functional and inclusion dependencies [34]. [2] proved the decidability of this problem under functional and multi-valued dependencies. Further, decidability and undecidability results are proved in [13] for non-recursive Datalog queries under expressive description logic constraints. Moreover, undecidability is proved in [15] for recursive queries under inclusion dependencies.

Another inspirational work comes from [13] in which query containment under description logic constraints is studied based on an encoding in propositional dynamic logic with converse (CPDL). Subsequently, in [13], a double exponential upper bound is proved for containment of UCQs under $\mathcal{DLR}$ schema axioms. Our work is similar in spirit, in the sense that, we apply encoding of the containment problem in the μ-calculus to take advantage of its superior expressive power and a number of available implementations. Thereby, opening a way for extensions of the query and schema languages (for instance OWL-DL [19]). Besides, the two query languages are different since SPARQL allows for predicates to be used as subject or object of other triple patterns and can be in the scope of a variable. This is not directly allowed in $\mathcal{DLR}$ UCQs. Our

encoding of RDF graphs and SPARQL queries preserves this capability. On the other hand, CQ containment is studied extensively for DL-Lite and $\mathcal{EL}$ family of languages as well as for DLs $\mathcal{ALC}, \mathcal{ALCI}$ and $\mathcal{ALCF}$. Interestingly, they allow each query its own ontology rather than assuming a single ontology for both queries, which they claim is relevant in applications to versioning and modularity. Comparatively, our query language is more expressive (because it has union, paths and variables in predicate positions) and the schema language we consider differs (i.e., RDFS and $\mathcal{ALCH}$). Although, containment under RDFS entailment regime can be translated into query containment with a separate schema for each query. This can be done by considering the schema part of the queries as a schema. We leave out the detail as a future work.

Lastly, the most closely related study is found in [22] where containment of union of conjunctive SPARQL queries under $\mathcal{SHI}$ schema axioms is investigated. They apply similar techniques as the ones used here however the schema language studied here is a different one as well as the query language which includes paths. Moreover, in this work, we provide a different encoding for cyclic SPARQL queries and point out clearly their contribution towards a further jump in complexity.

Implementation and evaluation Recently, static analysis and optimization of SPARQL queries has attracted widespread attention, notably [20,38,21,22,38, 23] for static analysis and [49,29,47,38,35,43,36] for optimization. However, to the best of our knowledge, there are only two implementations: (1) *SPARQL-Algebra* supports only conjunction and OPTIONAL queries with no projection (containment of basic and optional graph patterns) [38], and (2) *SPARQL containment benchmark* supports UCQ SPARQL containment under RDF schema axioms [23].

Finally, various SPARQL query evaluation performance benchmarks have been proposed [8,9,46], but only one SPARQL query containment benchmark to our knowledge [23]. Consequently in this work, concerning experimental tests, we rerun the benchmarks of [23] showing an improvement on the TreeSolver.

7 Conclusion

We now summarize the main contributions of this work, and propose further research directions. Having well-behaved computational and model theoretic properties and implementations that have been put to practice, μ-calculus has been chosen for the task of static analysis of SPARQL queries. μ-calculus formulas are interpreted over transition systems. SPARQL queries are evaluated over RDF graphs, these graphs can be transformed to other types of graphs: hypergrahs, bipartite graphs, transition systems and others. Thus, given a graph logic, RDF graphs can be translated into transition systems and SPARQL queries into μ-calculus formulas, consequently the formulas can be interpreted over transition systems.

Containment of queries can be reduced to satisfiability test by encoding the set inclusion as implication and the queries as formulas. In order to do this, we proposed to translate queries and $\mathcal{ALCH}$ axioms into μ-calculus formulas. The principle of the translation is based on reification where each triple is represented by a node, connected to the subject, predicate and object elements of the query. The encoding of the right-hand side query is different from that of the left due to the non-distinguished variables that appear in cycles in the query. In fact, this is the reason for the high complexity bounds in containment problems in general. Finally, the soundness and completeness of the reduction is proved and a double exponential upper bound complexity is established for the problem.

In order to complement the theoretical results, we have carried out experiments. We proposed a compliance benchmark for containment, equivalence and satisfiability of semantic web queries. The benchmark is used to test the current-state-of-the-art tools. A comparison of these tools based on running times is discussed. Furthermore, the benchmark is designed to assess the containment solvers that are available presently, with room for extension.

It is beyond the scope of this paper to address the containment problem for every possible fragment of SPARQL (resp. PSPARQL) and schema language (for instance, the $\mathcal{SH}$ family). This work lays the theoretical and experimental foundations, provides simply extendible and implementable methods, for instance, the containment benchmark is designed with respect to the current-state-of-the-art, so it needs to evolve side-by-side with the containment solvers.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases, vol. 8. Addison-Wesley (1995)
2. Aho, A.V., Sagiv, Y., Ullman, J.D.: Equivalences among relational expressions. SIAM Journal on Computing **8**(2), 218–246 (1979)
3. Alkhateeb, F., Baget, J.F., Euzenat, J.: Extending SPARQL with regular expression patterns (for querying RDF). Web Semantics: Science, Services and Agents on the World Wide Web **7**(2), 57–73 (2009)
4. Angles, R., Gutierrez, C.: The expressive power of SPARQL. In: Proc. ISWC, pp. 114–129. Springer (2008)
5. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2007). ISBN 9780511717383
6. Baget, J.F.: RDF entailment as a graph homomorphism. The Semantic Web–ISWC 2005 pp. 82–96 (2005)
7. Barcelo, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. p. 31. ACM (2012)
8. Bizer, C., Schultz, A.: Benchmarking the performance of storage systems that expose SPARQL endpoints. In: Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS) (2008)
9. Bizer, C., Schultz, A.: The Berlin SPARQL benchmark. International Journal on Semantic Web and Information Systems **5**(2), 1–24 (2009)
10. Blackburn, P., van Benthem, J.F., Wolter, F.: Handbook of Modal Logic, vol. 3. Elsevier (2007)
11. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Containment of Conjunctive Regular Path Queries with Inverse. In: Proc. 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000), pp. 176–185 (2000)
12. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Reasoning on regular path queries. SIGMOD Record **32**(4), 83–92 (2003)
13. Calvanese, D., Giacomo, G.D., Lenzerini, M.: Conjunctive query containment and answering under description logic constraints. ACM Transactions on Computational Logic (TOCL) **9**(3), 22 (2008)
14. Calvanese, D., Ortiz, M., Simkus, M.: Containment of regular path queries under description logic constraints. In: IJCAI Proceedings-International Joint Conference on Artificial Intelligence, vol. 22, p. 805 (2011)
15. Calvanese, D., Rosati, R.: Answering Recursive Queries under Keys and Foreign Keys is Undecidable. In: Proc. 10th Int. Workshop on Knowledge Representation meets Databases (KRDB 2003), vol. 79, pp. 3–14 (2003)
16. Chandra, A.K., Merlin, P.M.: Optimal Implementation of Conjunctive Queries in Relational Data Bases. In: Proceedings of the ninth annual ACM symposium on Theory of computing, pp. 77–90. ACM (1977)
17. Chebotko, A., Lu, S., Jamil, H.M., Fotouhi, F.: Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. Tech. rep. (2006)
18. Chekol, M.W.: Static analysis of semantic web queries. Ph.D. thesis, Université de Grenoble (2012). Thesis
19. Chekol, M.W.: On the containment of SPARQL queries under entailment regimes. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA., pp. 936–942 (2016)
20. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: PSPARQL query containment. In: Proc. DBPL (2011)
21. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under RDFS entailment regime. In: Proc. IJCAR, pp. 134–148 (2012)
22. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: SPARQL query containment under SHI axioms. In: Proc. AAAI, pp. 10–16 (2012)
23. Chekol, M.W., Euzenat, J., Genevès, P., Layaïda, N.: Evaluating and benchmarking SPARQL query containment solvers. In: International Semantic Web Conference (2), pp. 408–423 (2013)

24. Cyganiak, R.: A relational algebra for SPARQL. Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170 p. 35 (2005)
25. Florescu, D., Levy, A., Suciu, D.: Query containment for conjunctive queries with regular expressions. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 139–148. ACM (1998)
26. Genevès, P., Layaïda, N.: A system for the static analysis of XPath. ACM Trans. Inf. Syst. **24**(4), 475–502 (2006)
27. Genevès, P., Layaïda, N., Schmitt, A.: Efficient static analysis of XML paths and types. In: Proc. PLDI, pp. 342–351. ACM (2007)
28. Glimm, B.: Using SPARQL with RDFS and OWL entailment. Reasoning Web. Semantic Technologies for the Web of Data pp. 137–201 (2011)
29. Groppe, J., Groppe, S., Kolbaum, J.: Optimization of SPARQL by using coreSPARQL. In: Proc. ICEIS, pp. 107–112 (2009)
30. Gutierrez, C., Hurtado, C., Mendelzon, A.O.: Foundations of semantic web databases. In: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 95–106. ACM (2004)
31. Hayes, P.: RDF semantics. W3C Recommendation (2004)
32. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: OWL 2 web ontology language primer. W3C Recommendation (2009)
33. Ioannidis, Y.E., Ramakrishnan, R.: Containment of conjunctive queries: Beyond relations as sets. ACM Transactions on Database Systems (TODS) **20**(3), 288–324 (1995)
34. Johnson, D.S., Klug, A.: Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. Journal of Computer and System Sciences **28**(1), 167–189 (1984)
35. Kostylev, E.V., Cuenca Grau, B.: On the semantics of SPARQL queries with optional matching under entailment regimes. In: The Semantic Web ISWC 2014, *Lecture Notes in Computer Science*, vol. 8797, pp. 374–389. Springer International Publishing (2014)
36. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: SPARQL with property paths. In: International Semantic Web Conference, pp. 3–18. Springer (2015)
37. Kozen, D.: Results on the propositional μ-calculus. Theoretical computer science **27**(3), 333–354 (1983)
38. Letelier, A., Pérez, J., Pichler, R., Skritek, S.: Static analysis and optimization of semantic web queries. In: Proc. PODS, pp. 89–100. ACM (2012)
39. Manola, F., Miller, E.: Resource description framework primer. Internet (2004)
40. Mateescu, R., Meriot, S., Rampacek, S.: Extending SPARQL with Temporal Logic (2009). URL `http://hal.archives-ouvertes.fr/inria-00404761/en/`. INRIA Research Report, RR-7056
41. Munoz, S., Pérez, J., Gutierrez, C.: Minimal deductive systems for RDF. In: The Semantic Web: Research and Applications, vol. 4519, pp. 53–67. Springer (2007)
42. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems (TODS) **34**(3), 16 (2009)
43. Pichler, R., Skritek, S.: Containment and equivalence of well-designed SPARQL pp. 39–50 (2014)
44. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the 16th international conference on World Wide Web, pp. 787–796. ACM (2007)
45. PrudHommeaux, E., Seaborne, A.: SPARQL query language for RDF (2008)
46. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: a SPARQL performance benchmark. In: Proc. ICDE, pp. 222–233 (2009)
47. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: Proc. ICDT, pp. 4–33. ACM (2010)
48. Serfiotis, G., Koffina, I., Christophides, V., Tannen, V.: Containment and minimization of RDF/S query patterns. In: The Semantic Web - ISWC 2005, *LNCS*, vol. 3729, pp. 607–623 (2005)
49. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: Proc. WWW Conference, pp. 595–604 (2008)
50. Tanabe, Y., Takahashi, K., Hagiya, M.: A decision procedure for alternation-free modal μ-calculi. pp. 341–362 (2008)

51. Tanabe, Y., Takahashi, K., Yamamoto, M., Tozawa, A., Hagiya, M.: A decision procedure for the alternation-free two-way modal μ-calculus. In: TABLEAUX, pp. 277–291 (2005)