SpeCS — An Efficient and Comprehensive SPARQL Query Containment Solver

Mirko Spasić and Milena Vujević Janičić
Department for Computer Science, Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
E-mails: mirko@matf.bg.ac.rs, milena@matf.bg.ac.rs

Abstract. The query containment problem is a fundamental computer science problem which was originally defined for relational queries. With the growing popularity of the SPARQL query language, it became relevant and important in this new context: reliable and efficient SPARQL query containment solvers may have various applications within static analysis of queries, especially in the area of query optimization. In this paper, we present a new approach for solving the query containment problem in SPARQL. It is based on reducing the query containment problem to the satisfiability problem in first order logic. It covers a wide range of SPARQL language constructs, including union of conjunctive queries, blank nodes, projections, subqueries, clauses from, filter, optional, graph, etc. It also covers containment under RDF SCHEMA entailment regime, and it can deal with the subsumption relation. We describe an implementation of the approach, an open source solver SpeCS and its thorough experimental evaluation on relevant benchmarks, Query Containment Benchmark and SQCFramework. The evaluation shows that SpeCS is highly efficient and that compared to state-of-the-art solvers, it gives more precise results, and in a shorter amount of time. In addition, SpeCS has the highest coverage of supported language constructs.

Keywords: SPARQL, query containment, FOL modeling, SMT solving, SpeCS solver

1. Introduction

Number of datasets in the form of Resource Description Framework (RDF) [1-3] is rapidly increasing, striving to accomplish the full power of Semantic Web [4-6]. For example, the Linked Open Data Cloud [7] registers more than 1,200 RDF data sources, from various domains, e.g. government, media, publications, geography, life sciences, etc. The number is even bigger if we add into consideration datasets that are not public. The constant growth of this number, and also growth of size of all the datasets, forces triplestores to be more efficient in order to supply users with requested data in a reasonable amount of time.

A huge number of benchmarks are designed in order to push the triplestores forward [8, 9], to generate the synthetic datasets suitable for benchmarking [10, 11], or to test the performance of Semantic Web tools and applications from various domains [12-14]. Also, from a database perspective, query optimizations within SPARQL (Simple Protocol and RDF Query Language), as a querying language and data access protocol [15, 16], are fundamental for achieving practical usability of these large amounts of data.

Within global query optimizations done by static analysis, the most important problem is query containment, a problem of deciding if each result of one query is also a result of another query (for any given dataset) [17, 18]. An example of two queries in query containment relation is given in Figure 1. Other important problems, like query equivalence and query satisfiability, can be reduced to the query containment problem. Namely, two queries are equivalent if their sets of answers over any dataset are equal. The problem of determining query equivalence can be reduced to the check if containment relation holds in both directions, i.e. the first query is contained within the second query and vice versa. Similarly, query containment solvers can be used for checking query satisfiability (a query is satisfiable if there exists at least one solution for it). This can be done by reducing the problem to the query containment (or equivalence) between the query in ques-
tion and a query for which it is known that there is no answer.

Determining relationship between two queries is highly beneficial. If two queries are equivalent, the query optimizer can evaluate one query instead of the other query, in cases when such choice is better from the efficiency perspective. Optimizers apply rewriting rules, changing the original graph patterns into new patterns. Such rewriting rules can be based on properties of particular SPARQL operators [19, 20], or can be seen as transformation rules for pattern trees, considered as query execution plans, reducing a query, when that is possible, into a normal form [21] which is PSPACE complete [19]. However, it is crucial to verify equivalence between the original query and the rewritten one, as unsound optimization rules may be undetected for a very long period of time and when finally activated, may cause significant problems [22, 23]. Solving query equivalence is important in other contexts as well, for example, as a support for query refactoring process, where the main goal is to improve the overall code quality, without affecting its original semantics [24]. Also, if a query is not satisfactory and it can be determined statically, without evaluating it, then this can save a significant amount of time, especially in a distributed environments (which is a dominant scenario nowadays) when federated queries are executed against multiple endpoints over a slow network [25].

There are additional applications of tools considering query containment in the context of relational databases, where query containment problem was originally defined [26]. These applications include information integration techniques [27, 28], information gathering [29, 30], integrity constraint checking and verification [31–33], and knowledge representation that is based on description logic or graph representation [34–37]. All these applications can be naturally transferred to the Semantic Web context.

Different SPARQL query containment solvers have been developed [21, 38–40], but there are still many open problems concerning language coverage, efficiency, reliability, reasoning under RDF SCHEMA entailment regime, etc. In this paper, we extend our previous work [41] (where we presented our initial ideas) and present a procedure for solving the containment problem in SPARQL by reducing it to the satisfiability problem in first order logic (FOL). We also present an open source implementation of this procedure, a tool named SPECS. We use efficient satisfiability modulo theories (SMT) solvers [42] as an underlying reasoning machinery. The proposed procedure covers the core fragment of SPARQL language. We present a thorough experimental evaluation of the proposed approach on two most important benchmarks. A comparison with the existing state-of-the-art SPARQL query containment solvers on these benchmarks shows that the proposed approach has both a better performance and a higher coverage of the supported language constructs.

We also detected, confirmed and fixed some important problems within these benchmarks. Benchmarks were used in evaluation, with their query pairs and specifications, and all solvers and their JAVA wrappers, together with the sheets containing results of the presented evaluation are available online [43].

**Overview of the paper.** Section 2 presents the related work. Section 3 defines SPARQL semantics and query containment problem. Section 4 gives the modeling of the core subset of SPARQL queries and modeling of the containment relation between conjunctive queries. Section 5 extends the proposed approach by adding support for graph patterns containing unions, negations, optional constructs, subqueries, graph constructs and larger subset of expressions and conditions. Section 6 gives an overview of the proposed approach in the context of subsumption relation, which is a weaker form of containment. Section 7 considers reasoning under RDF schema. Section 8 presents an implementation of the proposed approach and its evaluation, comparing it with available state-of-the-art solvers on the existing benchmarks. Section 9 gives final conclusions and presents possible directions for further work.
2. Related Work

Query containment problem was originally defined within relational databases and SQL language, already in 1977 [26]. Since then, many important theoretical results and research directions came from relational query containment. With increasing popularity of Semantic Web, investigating query containment within SPARQL has also become an important and active research topic.

Relational query containment. The query containment problem belongs to the class of undecidable problems [44], but over the last few decades, researchers have been actively working on identifying its decidable fragments [26, 45–47]. Query containment problem has been under consideration in different specific contexts and including support for different language constructs, like projection, join, positive selection and union operators [48] or expressions containing the difference operator [45]. Important contexts include the presence of inclusion dependencies [49] and special classes of queries, like terminal and positive conjunctive queries [50], as well as queries containing views [51, 52], grouping and aggregate functions [53]. Implementing a query containment checker that can be efficiently used in practice is very demanding [54]. Recent research includes solving containment problem using machine learning techniques [55] or satisfiability modulo theories [56]. New contexts of solving containment problem include checking equivalence of embedded SQL queries [57].

Graph query containment. In recent years, graph databases gained an increasing popularity and their query languages become an active research topic, especially problems related to the query containment. For example, containment, equivalence and satisfiability problems are considered for XPath queries, which are used for managing graph databases [58–64]. Research is done also on other, less popular, query languages [37, 65, 66]. However, most of research on graph query containment is done in the context of SPARQL.

Semantics of SPARQL. Reasoning about containment relation highly depends on the semantics that has been used. There are two different kinds of SPARQL semantics [15, 19] which are proved equivalent [67], that could be considered while checking relations between queries in question. The first one, primarily aimed at developers, has been defined in an operational manner [16]. Set based semantics [19] and bag based semantics [68] are examples of the second kind, and they are given as alternatives to the original operational semantics. However, they describe only a subset of the SPARQL language, i.e. these semantics include only simplified graph pattern constructs. Later, more complicated language features, e.g. subqueries [69, 70], minus operator and projections [68, 71], and negation [72] were added.

SPARQL query containment. The language expressiveness of SPARQL is the same as expressiveness of relational algebra [67]. Therefore, the containment problem in SPARQL is also undecidable, but, like in the relational case, for some classes of queries, it is decidable [73]. For example, a general complexity analysis of containment and equivalence for different fragments of SPARQL covering different language constructs, like..union, optional and projections shows that the problem is decidable and NP-complete for some fragments and it is undecidable for other fragments [17]. Related problems regarding equivalence of graph patterns are also considered [19, 20, 74]. These problems are approached by identifying normal forms of graph patterns and by examining transformation rules which keep equivalence. Procedures for deciding SPARQL query containment are applied within semantic query optimization in translation of a SPARQL query to an SQL query implemented in Ontop tool [75], and within OnGIS, a semantic query broker, in the context of geospatial data [76]. The containment problem can also be considered in a weaker form, i.e. subsumption, that represents a generalization of containment [17, 19, 21, 77].

SPARQL query containment solvers. There are several state-of-the-art query containment solvers:

- **SPARQL-Algebra (SA)** [21] is a solver for SPARQL query subsumption and equivalence. It is based on a tree-like form of SPARQL queries, called pattern tree, that can be considered as a query execution plan [78]. It is implemented on top of the non-optimized version of the ARQ SPARQL engine [79], by adding rules for graph transformations, support for canonical forms, and for testing query equivalence.
- **Alternation Free two-way \( \mu \)-calculus (AFMU)** [38] solves the query containment problem by reducing it to a satisfiability problem in a fragment of the \( \mu \)-calculus without alternation [80]. Different approaches for transforming queries into this fragment of \( \mu \)-calculus are proposed [18, 81]
and for each one a satisfiability tool [82] for μ-calculus is used.

- TreeSolver (TS) [39] implements a procedure for exploring containment, equivalence, overlapping and other relation types of NoSQL queries, such as XPath or Jaql [83, 84]. The solver can also be used to formulate and solve properties on any kind of tree-shaped structures, including SPARQL queries.

- Jena-Sparql-Api Graph-Isomorphism based query containment solver (JSAG) [40] is a bottom-up algebra-tree matching approach that is used for solving the problem of query containment by checking if there is a subgraph isomorphism between the normalized algebra expression tree of a sub-query and a subtree of a super-query.

3. SPARQL and Query Containment Problem

The SPARQL query grammar is precisely defined by W3C in the EBNF notation [15], while a simplified subset of this grammar is given in Figure 2 and discussed in this section. Data queried by SPARQL are stored in terms of internationalized resource identifiers (IRIS), which are generalisation of the uniform resource identifiers [85].

In the following text, we assume that V, I, B and L are given, mutually disjoint, countable sets: V is a set which contains variables, I is a set which contains IRIS, B is a set which contains blank nodes (denoting unidentified resources), and L is a set which contains literals. Elements of these sets correspond to RDF terms, i.e. var, iri, blankNode and rdfLiteral of the grammar given in Figure 2. Like in [38], we abbreviate any union of these sets as, for instance, IBL := I U B U L.

An RDF triple is an element of the set IB × I × IBL. A triple pattern is defined as a nonterminal TPattern by the grammar. Note that each triple pattern is an element of the set VIB × VI × VIBL, so each RDF triple is a triple pattern by its definition. Elements of a triple pattern are called subject, predicate and object, like the corresponding nonterminals. An RDF triple may be considered as a graph, where the subject and the object of the triple are nodes of the graph, while the edge is labeled by the predicate. An RDF graph is a graph constructed from a set of RDF triples sharing the same nodes. Graph patterns and query patterns are defined as nonterminals GPattern and QPattern by the grammar. Graph patterns may contain different operators, like ..filter and braces (other types of graph patterns are considered in Section 5). Different conditions (described by nonterminal Cond in the grammar) containing expressions (described by nonterminal Expr in the grammar) may be a part of a filter clause.

Definition 3.1 (RDF dataset) An RDF dataset D is a set

\{G_0, \{i_1, G_1\}, ..., \{i_n, G_n\}\}

where

- each G_k is an RDF graph, while sets of blank nodes between different graphs are disjoint, and
- each i_k is a different IRI.

A dataset must contain a graph G_d, which is called the default graph of D. It defines function df as

\[df(D) := G_d,\]

that maps a dataset D to its default graph. A dataset may contain zero or more pairs \(\{i_k, G_k\}\), which are called named graphs. They define function names as

\[names(D) := \{i_1, ..., i_n\},\]

that maps a dataset D to the set of IRIS of its named graphs. Named graphs also define function gr that has two parameters. First parameter is an RDF dataset D, which is, for readability reasons, written in subscript, that specifies the dataset containing a named graph. The second parameter is an IRI, and the function is defined as follows:

\[gr_D(i_k) := \begin{cases} G_k, & \text{if } (i_k, G_k) \in D \\ \emptyset, & \text{otherwise} \end{cases}\]

An RDF merge of a subset \(\{G_{k_1}, ..., G_{k_m}\}\) of RDF graphs\(^1\) from a dataset D is a graph defined as follows:

\[merge_D(G_{k_1}, ..., G_{k_m}) := \bigcup_{i=1}^{m} G_{k_i} \]

Similar to function gr, function merge has a parameter D, given in subscript, that specifies an RDF dataset containing RDF graphs G_{k_1}, ..., G_{k_m}.

\(^1\)Note that sets of blank nodes of different graphs G_k are disjoint.
A SPARQL query is executed against an RDF dataset denoted by $D$. A query may specify an additional dataset, called query dataset and denoted by $D$.

**Definition 3.2 (Query dataset $D$)** A query dataset $D$ which is specified by the from and from named clauses of a query executed against an RDF dataset $D$, is defined in the following way:

- If a query does not contain neither from nor from named clauses,
  
  $$D := D.$$  

- Otherwise, the dataset $D$ is defined by its default graph and its named graphs, where the default graph of $D$ is defined as follows:
  
  $$df(D) := \text{merge}(G_k,...,G_m),$$

  where IRIs $i_k,j \in \{1,...,m\}$ referred in the from clauses define graphs $G_k$ in RDF merge, i.e. it holds $G_k := gr_D(i_k), j \in \{1,...,m\}$.

- If a query contains from named clauses and no from clauses,
  
  $$df(D) := G_0,$$

  i.e. the default graph of $D$ is an empty graph.

If a query contains from clauses but does not contain from named clauses, the dataset $D$ consists of a default graph and zero named graphs. Otherwise, named graphs $\langle i_l,G_l \rangle$ of the dataset $D$ are defined by IRIs $i_l,j \in \{1,...,n\}$ referred in the from named clauses, i.e. each $G_l$ in the named graphs is defined as $G_l := gr_D(i_l), j \in \{1,...,n\}$.

Note that if some $i_k$ mentioned in the from clause does not belong to names($D$), by Definition 3.1 it holds $gr_D(i_k) = \emptyset$.

Each triple pattern of a query is matched against an RDF graph from the query dataset which is called an active graph. For the simplified grammar, given in Figure 2, the active graph is always equal to the default graph of the query dataset. However, when we extend the grammar with a graph clause (in Subsection 5.5), the active graph may become equal to the one of the named graphs as well.

Blank nodes in RDF graphs within a dataset and blank nodes in graph patterns have different semantics:

- A blank node within a dataset represents an anonymous resource: a resource for which an IRI or literal is not known, but by the blank node it is indicated that such resource exists.
- Blank nodes in graph patterns act as variables, not as references to specific blank nodes in the dataset.

In the following text, when blank nodes are treated in the same way as variables, it is explicitly stated.

Intuitively, a result of a query execution connects variables from the query pattern to values within the active graph. Additionally, for obtaining the result of a query, it is also necessary to connect blank nodes from the query pattern to the values within the active graph, but these values cannot be retrieved by a query. For the set of query variables (and blank nodes) there can be more than one set of values satisfying the condition given by the query. Therefore, a result of the query execution is a set of mappings from the set of variables $\mathcal{V}$ to the set $\mathcal{IBL}$. We follow the standard set semantics.
of a SPARQL query and of a graph pattern evaluation [19, 38, 67, 74].

In the following text, \( dom(m) \) denotes a domain of a mapping \( m \). The function \( \mu \) denotes a partial mapping from the set of variables and blank nodes \( \mathcal{VB} \) to the set \( \mathcal{I} \). The function \( \mu_{x \rightarrow c} \) denotes a mapping such that \( dom(\mu_{x \rightarrow c}) := \{ x \} \) and \( \mu_{x \rightarrow c}(x) := c \).

For each SPARQL construct \( c \), \( \var{c} \) denotes a set of variables (subset of \( \mathcal{V} \)) appearing in \( c \).

**Definition 3.3** (Compatible mappings) Mappings \( m_1 \) and \( m_2 \) are compatible if for each \( x \) such that \( x \in dom(m_1) \cap dom(m_2) \), it holds \( m_1(x) = m_2(x) \).

A set \( \mathcal{IBLe} \) is an extension of the set \( \mathcal{IBL} \) which contains an additional constant named \( err \), i.e.

\[
\mathcal{IBLe} := \mathcal{IBL} \cup \{ err \}.
\]

Intuitively, this constant is used to denote an error state which occurs when the evaluation of an expression is not possible in some specific context. Therefore, it is a value of an extension \([ [ ] ] \) of the mapping \( \mu \) on variables and blank nodes outside of \( dom(\mu) \), as defined in the following definition.

**Definition 3.4** (Notation \([ [ ] ] \)) A value of an RDF term \( t \), an expression \( E \), and a triple pattern \( tp \), according to the mapping \( \mu \), in notation \([ [t] ]\)_\( \mu \), \([ [E] ]\)_\( \mu \), and \([ [tp] ]\)_\( \mu \), respectively, is a value from \( \mathcal{IBLe} \), defined in the following way:

\[
[t]_{\mu} := \begin{cases} 
    t, & t \in \mathcal{I}L \\
    \mu(t), & t \in \mathcal{VB} \text{ and } t \in dom(\mu) \\
    err, & t \in \mathcal{VB} \text{ and } t \notin dom(\mu) 
\end{cases}
\]

\[
[E]_{\mu} := \begin{cases} 
    [t]_{\mu}, & E = t, \ t \in \mathcal{VIL} \\
    [tp]_{\mu}, & tp \text{ is } \mathcal{S} \mathcal{P} \mathcal{O} \text{ and } s \in \mathcal{VIB}, \ p \in \mathcal{VI}, \ o \in \mathcal{VIBL} 
\end{cases}
\]

\[
[tp]_{\mu} := \begin{cases} 
    [s]_{\mu} | [p]_{\mu} | [o]_{\mu}, & tp \text{ is } \mathcal{S} \mathcal{P} \mathcal{O} \text{ and } s \in \mathcal{VIB}, \ p \in \mathcal{VI}, \ o \in \mathcal{VIBL} 
\end{cases}
\]

\[
\mu \text{ satisfies built-in condition } R, \ \text{denoted } m \models R, \ \text{if:}
\]

\[
\begin{align*}
[[E_1]]_\mu & \neq err, \\
[[E_2]]_\mu & \neq err \\
[[E_1]]_\mu & = [[E_2]]_\mu, \ \text{if not } \mu \models R_1, \ \text{R is } E_1 = E_2 \\
\mu & \models R_1 \text{ and } [tp]_\mu \models R, \ \text{R is } R_1 \land [tp]_\mu \\
\mu & \models R_1 \text{ or } [tp]_\mu \models R, \ \text{R is } R_1 \lor [tp]_\mu \\
\mu & \models R_1, \ \text{R is } (R_1)
\end{align*}
\]

The following definition specifies some operations over sets of mappings [19].

**Definition 3.6** (Operations over sets of mappings) Let \( \Omega_1 \) and \( \Omega_2 \) be sets of mappings. Operations union, join, difference and left outer-join are defined as follows:

\[
\begin{align*}
\Omega_1 \cup \Omega_2 := \{ m \mid m \in \Omega_1 \lor m \in \Omega_2 \} \\
\Omega_1 \bowtie \Omega_2 := \{ m_1 \cup m_2 \mid m_1 \in \Omega_1, m_2 \in \Omega_2, \ m_1 \text{ and } m_2 \text{ are compatible} \} \\
\Omega_1 \setminus \Omega_2 := \{ m_1 \mid m_1 \in \Omega_1 \text{ and for all } m_2 \in \Omega_2, \ m_1 \text{ and } m_2 \text{ are not compatible} \} \\
\Omega_1 \bowtie \Omega_2 := (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)
\end{align*}
\]

**Definition 3.7** (Evaluation of a graph pattern) Let \( \mathcal{D} \) be an RDF dataset, \( \mathcal{G} \) a graph within \( \mathcal{D} \), \( \mathcal{E} \) a triple pattern, \( \mathcal{GP}, \mathcal{GP}_1, \mathcal{GP}_2 \) graph patterns, and \( R \) a built-in condition. An evaluation of a graph pattern over the graph \( \mathcal{G} \) within the dataset \( \mathcal{D} \), denoted by \( \mathcal{D} \models [ \[ \mathcal{E} \] ] \), is defined recursively as follows:

\[
\begin{align*}
[\mathcal{E}]_\mu := \{ m \mid \text{dom}(\mu) = \text{var}(\mathcal{E}) \land [[\mathcal{E}]]_\mu \in \mathcal{G} \} \\
[\mathcal{GP}_1 . \mathcal{GP}_2]_\mu := [\mathcal{GP}_1]_\mu \bowtie [\mathcal{GP}_2]_\mu \\
\text{gp filter } R]_\mu := \{ m \in [\mathcal{GP}]_\mu \mid m \models R \} \\
\{ \text{gp} \}_\mu := [\text{gp}]_\mu
\end{align*}
\]

Note that a dataset \( \mathcal{D} \) that figures in \([ \[ \mathcal{E} \] ] \) is not relevant if we consider only graph patterns introduced so far. However, with support for additional graph patterns (Section 5.5), it becomes relevant.

**Definition 3.8** (Projection of a mapping) A projection of a mapping \( m \) to the set \( \mathcal{F} \), denoted by \( m|_\mathcal{F} \), is a mapping such that \( \text{dom}(m|_\mathcal{F}) = \text{dom}(m) \cap \mathcal{F} \) and for each \( x \in \text{dom}(m|_\mathcal{F}) \) it holds \( m|_\mathcal{F}(x) = m(x) \).

\(^2\) Different semantics are also possible, like bag semantics [19, 67, 68]. A comparison [19] of set and bag semantics presents the reasons why set semantics is of fundamental importance in the development and implementation of a query language.
Note that if domain of \( m \) and \( \tau \) are disjoint, \( m_\tau \) is an empty mapping, i.e. a mapping with an empty domain.

**Definition 3.9 (Projection operator over sets)** Let \( \Omega \) be a set of mappings, and \( \tau \) be a set of variables. Then, a projection operator over the sets \( \tau \) and \( \Omega \), denoted \( \Pi_{\tau}(\Omega) \), is defined as:

\[
\Pi_{\tau}(\Omega) := \{ m_\tau | m \in \Omega \}
\]

**Definition 3.10 (Evaluation of a query)** An evaluation of a query \( Q \) over a dataset \( D \) is denoted \( [Q]_D \) and is defined as

\[
[Q]_D := \Pi_{\var{Qpat}}([qpat]_{df(D)})
\]

where within the query \( Q \), \( qpat \) is a query pattern, \( \var{Qpat} \) is a set of distinguished variables, i.e. variables given by the select clause of the query \( Q \), and \( D \) is a query dataset.

Note that if select clause does not contain projections, i.e. query \( Q \) is a select + query, a set of its distinguished variables is equal to the set of all variables in its query pattern, i.e. \( \var{Qpat} \).

Each mapping that belongs to the set \( [Q]_D \) is called a solution mapping corresponding to a query \( Q \) over a dataset \( D \). Similarly, each mapping that belongs to the set \( [Q]_D \) is called a pattern instance mapping that corresponds to a graph pattern \( gp \) evaluated over the graph \( G \) within the dataset \( D \). Pattern instance mapping is a composition of an RDF instance mapping, which maps blank nodes to RDF terms from IBL, and a solution mapping, which maps query variables to RDF terms from IBL. In a case when a mapping \( \mu \) from \( [qpat]_{df(D)} \) has a disjoint domain with \( \var{Qpat} \), \([Q]_D \) contains the empty mapping, corresponding to the empty SPARQL solution. In a case when the set \( [Q]_D \) is an empty set, there is no result of a query evaluation.

**Definition 3.11 (Relevant variables \( \var{Qpat} \))** For an RDF dataset \( D \), a query \( Q \) and a mapping \( \mu \) such that \( \mu \in [Q]_D \), all variables from \( \text{dom}(\mu) \) are called relevant variables and are denoted by \( \var{Qpat} \).

Note that for grammar from Figure 2, the set equality \( \var{Qpat} = \var{Qpat} \cap \var{Qpat} \) holds.

**Definition 3.12 (Query satisfiability, unsatisfiability)** Query \( Q \) is satisfiable if there exist a dataset \( D \) and a mapping \( \mu \) such that \( \mu \in [Q]_D \). Otherwise, \( Q \) is unsatisfiable.

**Definition 3.13 (Query containment, super-query, sub-query, query equivalence)** Given two queries \( Q_1 \) and \( Q_2 \). \( Q_1 \) is contained in \( Q_2 \), denoted by \( Q_1 \subseteq Q_2 \), if for every RDF dataset \( D \), \([Q_1]_D \subseteq [Q_2]_D \) holds. \( Q_3 \) is called a super-query, while \( Q_1 \) is called a sub-query. \( Q_1 \) is equivalent to \( Q_2 \), if for every RDF dataset \( D \), it holds \([Q_1]_D = [Q_2]_D \).

Note that an unsatisfiable query is a sub-query of any other query.

**Definition 3.14 (Query containment problem)** For the given two queries \( Q_1 \) and \( Q_2 \), query containment problem is to determine whether \( Q_1 \) is contained in \( Q_2 \), i.e. whether \( Q_1 \subseteq Q_2 \) holds.

### 4. Modeling Conjunctive Queries in SPARQL

Grammar presented in Figure 2 contains the constructs that are most commonly used in SPARQL (conjunctive queries with filter clauses). In this section, we describe our approach for modeling SPARQL queries based on this grammar, i.e. a translation of queries to FOL formulas that are used for reasoning about query relations. Modeling queries containing subqueries, union and optional operators, and queries based on other extensions of this grammar is described in Section 5.

**Definition 4.1 (Theory signature \( \mathcal{L} \) corresponding to queries \( Q_1 \) and \( Q_2 \))** The signature \( \mathcal{L} \) of the FOL theory that is used for modeling a relationship between SPARQL queries \( Q_1 \) and \( Q_2 \) is given as a tuple

\[
(F_s, P_s, ar),
\]

where \( F_s \) is a set of function symbols, \( P_s \) is a set of predicate symbols, and \( ar \) is an arity function. More precisely:

- \( F_s := C \cup F \) is a set of function symbols, where:
  - \( C \) is a set of function symbols with arity zero, i.e. a set of constants corresponding to literals and IRIs that appear in the queries. It also contains a constant \( \text{err} \), corresponding to \( \text{err} \) in IBL.
  - \( F \) is an empty set. This set is a placeholder for a set of function symbols corresponding to built-in functions defined in SPARQL (some are introduced within Subsection 5.6, e.g. \text{datatype}).
– \( \mathcal{P}_s := \mathcal{P} \cup \{ \beta_d, \beta_n \} \) is a set of predicate symbols, where:

* \( \mathcal{P} \) is a set which contains the equality predicate symbol (\( = \)). This set can be extended to contain other predicate symbols corresponding to boolean functions defined in SPARQL (some are introduced within Subsection 5.6, e.g. isLiteral).

* \( \beta_d \) and \( \beta_n \) are predicate symbols that model belonging of a triple to the default graph of an RDF dataset \( D \), or to a named graph specified by an IRI, respectively.

– The arity function \( ar \) is defined as follows:

\[
\text{ar}(\alpha) := \begin{cases} 
0, & \text{if } \alpha \in \mathcal{C}, \\
2, & \text{if } \alpha \in \mathcal{P}, \\
3, & \text{if } \alpha \in \mathcal{P}, \text{ and } \alpha \in \beta_d, \\
4, & \text{if } \alpha \in \mathcal{P}, \text{ and } \alpha \in \beta_n.
\end{cases}
\]

A set of variables, intuitively corresponding to variables and blank nodes from SPARQL queries \( Q_1 \) and \( Q_2 \), is denoted by \( V \).

### 4.1. Transforming Queries Into Formulas

A simple query, described by grammar given in Figure 2, consists of select, from, from named and where clauses. In this section we describe how to transform this kind of queries into a FOL formula \( \Phi \).

**Definition 4.2** (Function \( \sigma_t \)) A function \( \sigma_t \) maps terms from SPARQL queries into corresponding variables and constants from the described signature, i.e. the function \( \sigma_t \) is a bijective function that maps elements from \( \text{VIBLe} \) into \( V \cup \mathcal{C} \):

\[
\sigma_t(t) := \begin{cases} 
\epsilon(c \in \mathcal{C}), & \text{if } t = c \text{ and } c \in \text{IL}, \\
\nu(v \in V), & \text{if } t = v \text{ and } v \in \text{VB}, \\
\text{err}, & \text{if } t = \text{err}.
\end{cases}
\]

If a query is executed against an RDF dataset \( D \), note that, following the Definition 3.2, a default graph of a query dataset \( D \) can be:

– equal to the default graph of \( D \),
– an RDF merge of one or more graphs, or
– an empty graph \( G_D \).

**Definition 4.3** (Function \( cx \)) Assuming that a SPARQL query is executed against an RDF dataset \( D \), and that a query dataset is \( D \), a function \( cx \) maps an active graph \( G \) to the set of corresponding graph IRIs constants in the following way:

– if the active graph \( G \) is a default graph of the query dataset \( D \) (that is not equal to \( D \)), then\(^3\)

\[
cx(G) := \begin{cases} 
\sigma_t\{\{i_1, \ldots, i_k\}\}, & \text{if } G = \text{merge}_D(G_{i_1}, \ldots, G_{i_m}) \text{ and } \text{gr}_D(i_j) = G_{i_j}, j \in \{1, \ldots, m\}, \\
\emptyset, & \text{if } G = G_D.
\end{cases}
\]

– if the active graph \( G \) is a named graph of the dataset \( D \), then

\[
cx(G) := \sigma_t(\{i\}), \text{ where } \text{gr}_D(i) = G.
\]

Note that function \( cx \) is not defined on the default graph of the RDF dataset \( D \), as there is no IRI associated to the default graph of \( D \).

The following definition introduces a function \( \sigma \) for mapping SPARQL constructs into corresponding FOL formulas from the given SPARQL signature. The full notation of this function includes two parameters:

1) a term, an expression, a condition or a pattern, and
2) an RDF graph \( G \) used for matching the pattern within the query.

As mapping terms, expressions and conditions do not depend on the graph \( G \), to keep the formulas simple, we use \( \sigma \) with only one parameter (this is also the case when \( G \) is obvious from the context). When necessary, the second parameter is denoted in superscript, i.e. \( \sigma^G \).

**Definition 4.4** (Function \( \sigma \)) Function \( \sigma \) is defined recursively depending on its first argument, i.e., if the argument is:

– Term \( t \)

* if \( t \in \text{VIBLe} \), then:

\[
\sigma(t) := \sigma_t(t)
\]

– Expression \( E \)

\(^3\)Function \( \sigma_t \) is defined on \( \text{VIBLe} \), but in this definition, we abuse notation, and use it on a set of elements from its domain, with standard meaning: \( \sigma_t(A) = \{ \sigma_t(a) \mid a \in A \} \).
if $E$ is $e \in \text{VIL}_e$, then:

$$\sigma(E) := \sigma_1(e)$$

**Condition** $R$

if $R$ is an equality operator $= \text{comparing expressions } E_1 \text{ and } E_2$, the result is the corresponding equality predicate:

$$\sigma(E_1 = E_2) := \sigma(E_1) = \sigma(E_2)$$

if $R$ is a logical operator over conditions $R_1$ and $R_2$, the result is the corresponding logical connective of FOL:

$$\sigma(\neg R_1) := \neg \sigma(R_1)$$

$$\sigma(R_1 \land R_2) := \sigma(R_1) \land \sigma(R_2)$$

$$\sigma(R_1 \lor R_2) := \sigma(R_1) \lor \sigma(R_2)$$

if $R$ is a condition $R_1$ in parentheses, the result is the corresponding condition:

$$\sigma((R_1)) := \sigma(R_1)$$

**Pattern** $gp$, where an RDF graph $G$ is used for matching the pattern within the query

if $gp$ is a triple pattern $tp$ of the form $s \circ p \circ o$, where $s \in \text{VIB}_s, p \in \text{VIB}_p, o \in \text{VIB}_o$, the result is:

$$\sigma^G(tp) := \begin{cases} 
\beta_d(\sigma(s), \sigma(p), \sigma(o)), \\
\text{if } G \notin \text{dom}(cx) \\
\bigvee_{i,j \in \alpha(G)} \beta_1(\sigma(s), \sigma(p), \sigma(o), i), \\
\text{if } G \in \text{dom}(cx) \text{ and } \text{cx}(G) \neq \emptyset \\
\bot, \\
\text{if } G \in \text{dom}(cx) \text{ and } \text{cx}(G) = \emptyset 
\end{cases}$$

if $gp$ is a list of two patterns, $gp_1$ and $gp_2$, the result is the following conjunction:

$$\sigma^G(gp_1 \cdot gp_2) := \sigma^G(gp_1) \land \sigma^G(gp_2)$$

if $gp$ is a filter clause on a pattern $gp_1$ and a built-in condition $R$, the result is the following conjunction:

$$\sigma^G(gp_1 \text{ filter } R) := \sigma^G(gp_1) \land \sigma(R)$$

if $gp$ is a graph pattern $gp_1$ in braces, the result is the application of $\sigma$ to the pattern:

$$\sigma^G(gp_1) := \sigma^G(gp_1)$$

Note that the context cannot be changed in case of ., filter and braces clauses. An example of a formula corresponding to a query given in Figure 1 is given in Figure 3.

\[
\beta_d(x, a, \text{UndergradStudent}) \land \beta_d(x, \text{takesCourse}, y) \\
\land \beta_d(x, \text{name}, n_1) \land \beta_d(x, \text{phone}, t_0) \land n_2 = \text{Tim}.
\]

Fig. 3. Formula corresponding to the query given in Figure 1 (above). For readability, each $\text{IRI}$ literal and variable is denoted by a corresponding subscript $i$, $l$ and $v$, respectively.

We use $\exists a$ to abbreviate $\exists a_1 \ldots a_n$, and $\forall a$ to abbreviate $\forall a_1 \ldots a_n$, when $n$ is clear from the context.

**Definition 4.5** (Formula $\Phi(\alpha)$ corresponding to $Q$) Let $Q$ be a query with a query pattern $qpat$ specifying a query dataset $D$. Let $\alpha$ denote a set of some variables from $\mathcal{V}$, and let $\text{var}$ denote other free variables in the formula $\sigma(qpat)$, i.e. $\text{var} := \text{var}(\sigma(qpat)) \setminus \alpha$. Formula $\Phi(\alpha)$ corresponding to the query $Q$ is defined as:

$$\exists \text{var} \sigma^G(\alpha)(qpat).$$

If $\mathcal{V}$ denotes variables from $\mathcal{V}$ that correspond to the distinguished (or relevant) variables of the query $Q$, then, intuitively, variables $\text{var}$ within formula $\Phi(\mathcal{V})$ corresponding to the query $Q$ denote variables from $\mathcal{V}$ that correspond to variables from $\mathcal{V}$ that are used in the query but that are not selected.

### 4.2 Modeling Query Containment Problem

Modeling query containment problem includes defining formulas $\Theta$ and $\Psi$, and a relation $\sim$.

**Definition 4.6** (Formula $\Theta$) Let $Q_1$ be a query, let $\mathcal{PT}$ denote variables from $\mathcal{V}$ that correspond to the relevant variables of the query $Q_1$, and let $\Phi_1(\mathcal{PT})$ correspond to $Q_1$. Formula $\Theta$ is defined as:

$$\neg(\exists \mathcal{PT} \Phi_1(\mathcal{PT}))$$
Definition 4.7 (Relation \(\sim\)) Queries \(Q_1\) and \(Q_2\) are in relation \(\sim\), denoted by \(Q_1 \sim Q_2\), if relevant variables \(\forall \Phi_1(\overline{T})\) and \(\forall \Phi_2(\overline{T})\) of these queries are equal, i.e. if \(\forall \Phi_1 = \forall \Phi_2\) holds.

Definition 4.8 (Formula \(\Psi\)) Let \(Q_1\) and \(Q_2\) be queries, let \(\forall \Phi_1\) denote variables from \(\mathcal{V}\) that correspond to the relevant variables of the query \(Q_1\), and let \(\forall \Phi_2(\overline{T})\) and \(\Phi_2(\overline{T})\) correspond to queries \(Q_1\) and \(Q_2\) respectively. Formula \(\Psi\) is defined as:

\[
\forall \overline{T} (\Phi_1(\overline{T}) \Rightarrow \Phi_2(\overline{T}))
\]

Query containment problem is known to be undecidable if query \(Q_2\) contains projections [17]. Therefore, we assume that \(Q_2\) is a select * query, or all variables in its query pattern appear in the select clause. We reduce the query containment problem of such queries to the following three checks:

1. a validity check of a formula \(\Theta\)
2. a check if \(Q_1 \sim Q_2\) holds
3. a validity check of a formula \(\Psi\)

More precisely, we conclude that \(Q_1 \subseteq Q_2\) holds if any of the following two conditions is satisfied:

- Check (1) is satisfied. Intuitively, this condition corresponds to the unsatisfiability of \(Q_1\), i.e. by Definition 3.12, for any dataset \(D\), \([Q_1]^D = \emptyset\). Therefore, \([Q_1]^D \subseteq [Q_2]^D\) holds for any query \(Q_2\).
- Both checks (2) and (3) are satisfied. Intuitively, these conditions check if each mapping from \([Q_1]^D\) belongs to \([Q_2]^D\). Check (2) validates a domain of such mappings, i.e. relevant variables, while check (3) validates the rest.

The validity check of formulas \(\Theta\) and \(\Psi\) can be performed by an SMT solver (by checking the unsatisfiability of a negation of the corresponding formula).

The proposed approach is both sound and complete, as stated by the following theorem. The proof of this theorem requires introducing a number of additional definitions and non trivial lemmas and is out of scope of this paper.

Theorem 4.1 (Soundness & Completeness) Query \(Q_1\) is a sub-query of query \(Q_2\) (\(Q_1 \subseteq Q_2\)) if and only if

1. \(\Theta\) is valid, or
2. \(Q_1 \sim Q_2\) holds and \(\Psi\) is valid.

5. Extensions

In this section, we support modeling of SPARQL queries containing some additional (more complex and advanced) constructs, like union, diff, optional and graph operators, subqueries and built-in functions within new types of expressions and conditions.

5.1. Modeling Queries With union Operator

One important SPARQL operator is union operator, which extends the grammar from Figure 2, as shown in Figure 4. It provides a possibility to combine different graph patterns, where one of two alternative graph patterns can match, i.e. the semantics of union operator is specified by the following definition that extends Definition 3.7.

![Fig. 4. Adding union operator into the grammar from Fig. 2](image)

Definition 3.7(u) (Evaluation of a graph pattern) ... An evaluation of a graph pattern over the graph \(G\) within the dataset \(D\), denoted by \([\cdot]^D\), is defined recursively as follows:

\[
\ldots := \ldots
\]

\[
[gp_1 \text{ union } gp_2]^D := [gp_1]^D \cup [gp_2]^D
\]

Any graph pattern \(gp\) containing union operator, can be reduced to the equivalent pattern of the form:

\[
gp^1 \text{ union } \ldots \text{ union } gp^n,
\]

called a simple normal form of a graph pattern, where all \(gp^i(1 \leq i \leq n)\) are union-free graph patterns [74]. Therefore, we perform such transformation on query patterns of \(Q_1\) and \(Q_2\), for which the containment relation is considered. Then, union operator can appear only outside the scope of other operators.

Suppose that \(Q_1\) and \(Q_2\) are of the following form

\[
Q_1 = \text{select } \overline{dv} \text{ where } \{ qpat_1 \},
\]

\[
Q_2 = \text{select } \star \text{ where } \{ qpat_2 \},
\]
where query patterns $qpat_1$ and $qpat_2$ are in simple normal form consisting of graph patterns $gp_1^1 (1 \leq i \leq m)$ and $gp_2^1 (1 \leq j \leq n)$ respectively. Instead of containment between queries $Q_1$ and $Q_2$, we consider queries $Q_1^1 (1 \leq i \leq m)$ and $Q_2^2 (1 \leq j \leq n)$, defined as:

$$Q_1^1 = \text{select } \exists v \text{ where } (gp_1^1),$$

$$Q_2^2 = \text{select } * \text{ where } (gp_2^2),$$

where $from$ and $from$ named clauses of queries $Q_1$ and $Q_2$ are propagated to the queries $Q_1^1$ and $Q_2^2$, respectively. Following the semantics of the operator $\text{union}$, we reduce the query containment problem of queries $Q_1$ and $Q_2$ to the checks if for each $i (1 \leq i \leq m)$ there exists $j (1 \leq j \leq n)$ such that

$$Q_1^1 \subseteq Q_2^2.$$

5.2. Operator $diff$

There are different types of a negation in SPARQL, including the operator $\text{minus}$ (which is a part of the SPARQL standard) and the operator $diff$ (which is not part of the SPARQL standard yet, but has very small and subtle differences to $\text{minus}$ [71]). Here, we consider the operator $diff$, in order to reduce the operator $\text{optional}$ in Subsection 5.3 to more simple operators, i.e. $\text{union}$, and $diff$.

The syntax of $diff$ operator is given in Figure 5, while its semantics is given by Definition 3.7(d).

![Fig. 5. Adding GPattern operator into the grammar from Fig. 2](image)

**Definition 3.7(d)** (Evaluation of a graph pattern) ... An evaluation of a graph pattern over the graph $G$ within the dataset $D$, denoted by $[ \cdot ]^D_G$, is defined recursively as follows:

$$\ldots := \ldots$$

$$[gp_1 \text{ diff } gp_2]^D_G := [gp_1]^D_G \setminus [gp_2]^D_G$$

Transforming of a pattern with $diff$ operator into a corresponding FOL formula is given by extending the Definition 4.4.

**Definition 4.4(d)** (Function $\sigma$) Function $\sigma$ is defined recursively depending on its first argument, i.e., if the argument is:

- $\ldots$

- Pattern:

  * $\ldots$

    - a $diff$ clause, the result is the following conjunction:

      $$\sigma(gp_1 \text{ diff } gp_2) :=$$

      $$\sigma(gp_1) \land (\forall \pi \neg \sigma(gp_2)),$$

      where $\pi$ denotes all variables from $V$, that appear in $\sigma(gp_2)$, but not in $\sigma(gp_1)$.

5.3. Modeling Queries With optional Operator

All graph patterns explained so far demand to be matched completely. However, it is very useful to have possibility to add an information to the solution where the information is available, but do not reject the solution because some part of the query pattern does not match. Optional matching provides this facility. The SPARQL query grammar is extended as presented in Figure 6. The semantics of optional operator is given in Definition 3.7(o) that extends Definition 3.7.

![Fig. 6. Adding optional operator into the grammar from Fig. 2](image)

**Definition 3.7(o)** (Evaluation of a graph pattern) ... An evaluation of a graph pattern over the graph $G$ within the dataset $D$, denoted by $[ \cdot ]^D_G$, is defined recursively as follows:

$$\ldots := \ldots$$

$$[gp_1 \text{ optional } gp_2]^D_G := [gp_1]^D_G \triangleright [gp_2]^D_G$$

A pattern $pat$ is well-designed [17, 19, 74] if for all its subpatterns $gp$ such that

$$gp = gp_1 \text{ optional } gp_2,$$
it holds that all variables appearing in \( gp_2 \), but not in \( gp_1 \), cannot appear in \( \text{pat} \) anywhere else except in \( gp_2 \). Concerning both theoretical aspects and practical efficiency of query evaluation, it is important that queries contain only well-designed graph patterns [17, 19]. Therefore, we also consider such queries. Also, for the operator \( \text{diff} \), we assume similar features. Note that graph patterns being introduced by reducing well-designed optional patterns and connected by the operator \( \text{diff} \) will satisfy this condition.

According to the given semantics, operator \( \text{optional} \) can be reduced to a union of a conjunction and a negation. Therefore, instead of considering queries that contain graph patterns connected by the operator \( \text{optional} \), i.e. \( gp_1 \) \( \text{optional} \) \( gp_2 \) we syntactically replace them with a union of their conjunction (operator \( . \)) and negation (operator \( \text{diff} \)):

\[
\text{gp}_1 \cdot \text{gp}_2 \ \text{union} \ \text{gp}_1 \ \text{diff} \ \text{gp}_2
\]

As union, conjunction and negation are already considered, the operator \( \text{optional} \) is fully covered.

### 5.4. Subqueries as Graph Patterns

Subqueries allow embedding queries within other queries in order to facilitate a composition of new queries and a reuse of existing queries. Subqueries are simpler than queries because they cannot contain \( \text{from} \) and \( \text{from named} \) clauses, and they are used in a place of a graph pattern [69]. To support subqueries, we extend the SPARQL query grammar in the way presented in Figure 7. The semantics is given the Definition 3.7(s), according to literature [69, 70].

**Definition 3.7(s)** (Evaluation of a graph pattern) ... Let \( Q_{sq} \) be a subquery. An evaluation of a graph pattern over the graph \( G \) within the dataset \( D \), denoted by \( [[Q_{sq}]]_D^G \), is defined recursively as follows:

\[
[[Q_{sq}]]_D^G := [[Q_{sq}]]_D^D
\]

SPARQL query evaluation is bottom-up, meaning that the inner most subquery is evaluated first. Then, the results of a subquery are projected up to the outer query, and only distinguished variables are visible (in scope) to the outer query.

According to the semantics of a subquery, any query \( Q \) containing a subquery \( Q_{sq} \) (whose query pattern and distinguished variables are denoted by \( qpat \) and \( \text{dv} \), respectively) can be reduced to an equivalent query \( Q' \) that does not contain this subquery. The construction of such query \( Q' \) follows: instead of graph pattern \( \{ Q_{sq} \} \) within query \( Q \), graph pattern \( qpat' \) can be used, where \( qpat' \) is obtained from \( qpat \), by renaming all the variables from \( \text{var}(qpat) \setminus \text{dv} \), by introducing new fresh variables.

Note that query containment problem is undecidable if query \( Q_2 \) contains projections. Therefore, to keep decidability, it is necessary to consider only cases where subqueries are not present in the query \( Q_2 \), or for subqueries that are present in the query \( Q_2 \) all of its variables are projected up to \( Q_2 \).

### 5.5. Accessing Graph Names and Restricting by Graph IRI

By a SPARQL query, it is possible to match the graph patterns against each of the named graphs in the dataset and form solutions which have the corresponding variable bound to IRI of the graph being matched. This is done by \text{graph construct}. Its second feature restricts the matching to be applied against a specific named graph by supplying the graph IRI. This sets the active graph to the graph named by the IRI. The SPARQL query grammar is extended in the way presented in Figure 8.

**Fig. 8.** Adding graph operator into the grammar from Fig. 2

**Definitions 3.7 and 4.4** are extended as follows.

**Definition 3.7(g)** (Evaluation of a graph pattern) ... Let \( i \) be an IRI, and \( x \) a variable. An evaluation of a
graph pattern over the graph \( G \) within the dataset \( D \), denoted by \([\cdot]_D^G\), is defined recursively as follows [67]:

\[
\text{... ::= ...}
\]

\[
[\text{graph} \times \{\text{gp}\}]_D^G := \bigcup_{i \in \text{names}(i)} \{[\text{gp}]_{i \to D}^G \bowtie \{\mu_{k \to 1}\}\}
\]

\[
[\text{graph} \ i \ \{\text{gp}\}]_D^G := \{[\text{gp}]_{i \to D}^G\}
\]

**Definition 4.4(g) (Function \( \sigma \))** Function \( \sigma \) is defined recursively depending on its first argument, i.e., if the argument is:

- ... Pattern:

Let \( I_n \) be a set containing all graph IRI constants corresponding to the named graphs of the dataset \( D \) specified by the query, i.e., \( I_n := \{\iota_1, \ldots, \iota_k\} \), where \( \iota_j := \sigma(i_j) \), and IRIs \( \iota_j \) are given in the from named clauses of the query, or, in case when there is neither from named nor from clause, the set \( I_n \) is equal to \( \sigma_i(\text{names}(D)) \). If there is no from named clause, but there is some from clause, this set is empty, i.e., \( I_n := \emptyset \).

* ... 

* a graph pattern \( \text{gp}_1 \) that should be matched against each of the named graphs, the result is the following formula:

\[
\sigma(\text{graph} \times \{\text{gp}\}) := \bigvee_{i \in \iota_k} (\sigma_{i \to D}^G(\{\iota\})^{\bigcup (i)}(\text{gp}_1) \land \sigma(x) = i),
\]

\[
\land, \quad \sigma(x) \neq i \land \mu_{k \to 1}
\]

* a graph pattern \( \text{gp}_1 \) that should be matched against the specific named graphs, the result is the following formula:

\[
\sigma(\text{graph} \ i \ \{\text{gp}\}) := \begin{cases} 
\sigma_{i \to D}^G(\{\iota\})^{\bigcup (i)}(\text{gp}_1), & \mu_{k \to 1} \in I_n \\
\land, & \sigma(x) \notin I_n
\end{cases}
\]

### 5.6. Extending Expressions and Conditions

SPARQL language supports a wide range of built-in functions that are used within expressions and conditions. Two of them are presented in this subsection, while the others can be treated in a similar manner. The

SPARQL query grammar is extended in the way presented in Figure 9.

Definitions 3.4, 3.5, 4.1 and 4.4 are extended to include these new types of conditions and expressions. According to the SPARQL semantics [15], the following function is needed in order to extend Definition 3.4.

**Definition 5.1 (Semantics of function \( \text{dt} \))** Let \( l \) be a literal, function \( \text{dt} : L \to I \) is defined as follows:

\[
\text{dt}(l) := \begin{cases} 
i, & l \text{ is a typed literal, with } "i \text{ in suffix, } i \in I, \\
\text{xsd:string}, & l \text{ is untyped literal.}
\end{cases}
\]

**Definition 3.4(d) (Notation \([[:]]]) \) ...** A value of an expression \( E \), according to the mapping \( \mu \), in notation \([[:]][\mu] \), is a value from \( \text{IBLe}, \) defined in the following way:

\[
[[\text{datatype}(E_1)]][\mu] := \begin{cases} 
\text{dt}([[E_1]][\mu]), & [[E_1]][\mu] \neq \text{err} \\
\text{err}, & \text{otherwise}
\end{cases}
\]

**Definition 3.5(l) (Relation \( \vdash \)) ...** A mapping \( \mu \) satisfies built-in condition \( R \), denoted \( \mu \vdash R \), if:

\[
\mu \vdash R := \{\cdots, \text{...}, \cdots \}
\]

**Definition 4.1(d) (Theory signature \( L \)) ...**

- \( \mathcal{F} := \mathcal{C} \cup \mathcal{F} \) is a set of function symbols, where:

  * ... 

  * \( \mathcal{F} := \{\text{datatype}\} \).

- \( \mathcal{P} := \mathcal{P} \cup \{\beta_{n}, \beta_{d}\} \) is a set of predicate symbols, where:

  * \( \mathcal{P} := \{\cdots, \text{isliteral}\} \).
from J (Query subsumption) Definition 3.13(subsumption) The subsumption relation is defined as follows.

Given two queries \( Q_1 \) and \( Q_2 \), \( Q_1 \) is subsumed by \( Q_2 \), denoted by \( Q_1 \sqsubseteq Q_2 \), if for every RDF dataset \( D \), for each mapping \( \mu \) from \([Q_1]^{D}\) there exists an extension \( \mu' \), such that \( \mu' \) belongs to \([Q_2]^{D}\).

6. Subsumption Relation

Instead of containment, many authors deal with subsumption relation [17, 19, 21, 77], which can be considered as a weaker form of containment, practically applicable in the case of incomplete SPARQL query results. Instead of requirement that every mapping \( \mu \) from \([Q_1]^{D}\) belongs to \([Q_2]^{D}\) in Definition 3.13, subsumption relation is defined as follows.

Definition 3.13(subsumption) (Query subsumption) Two queries \( Q_1 \) and \( Q_2 \) are in relation \( \sqsubseteq \), denoted by \( Q_1 \sqsubseteq Q_2 \), if for every RDF dataset \( D \), for each mapping \( \mu \) from \([Q_1]^{D}\) there exists an extension \( \mu' \), such that \( \mu' \) belongs to \([Q_2]^{D}\).

An example of query pair satisfying subsumption relation is given in Figure 10.

Unlike the containment relation, if two queries satisfy the subsumption relation in both directions, they do not have to be equivalent (if either of them contains union or projection operator) [17]. While considering subsumption, query \( Q_2 \) can contain projections, as the subsumption problem is not undecidable like containment in this particular case [17]. Also, queries can contain all the constructs described so far.

We reduce the subsumption relation check to the same checks as used for the containment problem, but with the relation \( \sim \) (Definition 4.7) redefined in a weaker form (\( \subseteq \) instead of \( = \)), i.e., into a relation \( \sim \).

Definition 4.7(subsumption) (Relation \( \sim \)) Queries \( Q_1 \) and \( Q_2 \) are in relation \( \sim \), denoted by \( Q_1 \sim Q_2 \), if a set of relevant variables \( \{ \forall \upsilon \} \) of \( Q_1 \) is a subset of a set of relevant variables \( \{ \forall \upsilon \} \) of \( Q_2 \), i.e. if \( \{ \forall \upsilon \} \subseteq \{ \forall \upsilon \} \) holds.

Soundness and completeness (dual to Theorem 4.1) can be proved for subsumption as well.

Theorem 4.1(subsumption) (Soundness & Completeness) Query \( Q_1 \) is subsumed by query \( Q_2 \) (\( Q_1 \sqsubseteq Q_2 \)) if and only if

1. \( \Theta \) is valid, or
2. \( Q_1 \sim Q_2 \) holds and \( \Psi \) is valid.
7. General Purpose Axioms and Modeling RDF

SCHEMA

The definition of an RDF dataset does not restrict the relationship between the named graphs and the default graph. If a query does not specify an RDF query dataset D, by from and from named clauses, the query processor interprets the query in terms of the default graph, while its content is also determined by the query processor. A common implementation of a query processor is to include all the triples from named graphs in the default graph as well. A general purpose axiom Default graph (given in Figure 11) is added to model this relationship: if a triple tp is present in a named graph G, it is also present in the default graph of dataset D.

RDF SCHEMA constrains the interpretation of graphs and is naturally modelled by axioms. Therefore, if the query containment should be analyzed with respect to an RDF SCHEMA, for each RDF SCHEMA entry a corresponding axiom, presented in Figure 11 is added. The schema can be in the following forms:

- SubC rdfs:subClassOf C
  Meaning: Class SubC is a subclass of class C, i.e. each instance of class SubC is an instance of class C. We model it with Axioms Subclass.

- SubP rdfs:subPropertyOf P
  Meaning: Property SubP is a subproperty of property P, i.e. each instance with property SubP which is equal to some value, has property P with the same value. We model it with Axioms Subproperty.

- P rdfs:domain C
  Meaning: Domain of property P is a set of instances of class C, i.e. if an instance has a property P, it has to be an instance of class C. We model it with Axioms Domain.

- P rdfs:range C
  Meaning: Range of property P is a set of instances of class C, i.e. if an instance has a property P, its value has to be an instance of class C. We model it with Axioms Range.

Note that a, C, SubC, P, and SubP are IRIS, i.e. a, C, SubC, P, SubP ∈ I. Therefore σ is well defined, i.e. σ(a), σ(C), σ(SubC), σ(P) and σ(SubP) appearing in our formulas are the corresponding constants from C. Also note that the proposed axioms naturally follow the corresponding SPARQL semantics.

8. Implementation and Experimental Evaluation

The proposed approach is implemented within our solver SPECS that is publicly available and open source [43]. In this section, we present implementation details and a thorough evaluation of SPECS that contains experimental results and a comparison with related state-of-the-art tools.

8.1. Implementation Details

SPECS is implemented in C++ (about 2,500 lines of code). It uses tools Lex and Bison for lexical and syntax analysis. It supports queries described by the simplified grammar given in Figure 2 and its extensions given in Section 5. Several other syntax constructs that are semantically equivalent to the specified ones are also supported, like grouping triple patterns sharing the same subject by using predicate-object lists, or triple patterns sharing both, the same subject and the same object by using object lists. The solver is modular and support for additional modeling can be added easily. It
generates the query-based containment conditions. If an RDF SCHEMA is specified, it generates axioms corresponding to the concrete rules.

All formulas are generated in the SMT-lib format [86]. Therefore, different solvers/provers can be used. In the experimental evaluation, we used both the SMT solver Z3 [87] (version 4.5.0) and the FOL prover Vampire [88] (version 4.2.2). The solver Z3 showed better performance (it was approximately 50% faster\(^5\)). Therefore, we present only the solving times corresponding to the usage of the solver Z3.

The profiling of SPECS with Z3 solver on benchmarks used in evaluation shows that 3.83% of the overall time is spent on query parsing, 4.03% on formula construction, while the most demanding part is solving the formula, i.e. approximately 92.14%.

8.2. Benchmarks Used in Evaluation

To the best of our knowledge, there are two different SPARQL query containment benchmarks, Query Containment Benchmark [38] and SQCFramework [89, 90]. We use both benchmarks in our evaluation.

Query Containment Benchmark (QC Bench) [38] contains a fixed number of query containment problems with synthetic queries handcrafted by its designers. Some of problems are labeled as positive, i.e. containment relation is satisfied between two queries, while the others are labeled as negative. The benchmark contains three test suites with increasing difficulty, each covering an increasing number of language features:

- Conjunctive Queries with No Projection (CQNoProj): This suite contains 20 different test cases (nop1-nop20), 9 positive and 11 negative, where query patterns are made of basic graph patterns, and queries are conjunctive without projections.
- Union of Conjunctive Queries with Projection (UCQProj): This suite contains 28 test cases (p1-p28), 13 positive and 15 negative, where queries can have union pattern and projections in select clauses.
- Union of Conjunctive Queries under RDFS reasoning (UCQrdfs): This suite is similar to the previous one regarding SPARQL constructs, but it deals with RDFS reasoning, i.e. the containment is considered with respect to a schema. There are 28 different tests (rdfs1-rdfs28), 12 positive and 16 negative.

The benchmark also contains a single test case with cyclic queries, but benchmark designers have not included it in any of QC Bench test suites. Three out of four solvers described in Section 2 and used in our experimental evaluation, AFMU, TS and JSAG, do not support cyclic queries, while SA supports cyclic queries [25], but solving this query is considerably slower than solving other queries. Concerning SPECS, it successfully solves it, and the solving time is similar to the solving times of other tests.

We noticed two problems in the benchmark related to the following test cases:

p24: In this test case, the query containment between queries Q20b and Q20a (Figure 12) is considered. Based on the benchmark specification, Q20b should be contained in Q20a. However, as the solution mapping of Q20b corresponding to the second operand of union contains more variables in its domain than the corresponding solution mapping of Q20a, the containment relation does not hold. Regarding other solvers, AFMU and TS are not able to provide the results within the specified timeout (20000ms), SA does not support projections and therefore it cannot solve the test case p24, while JSAG is aligned with SPECS and classifies it as a negative test case.

rdfs21: According to the benchmark specification, the query containment between queries Q41e and Q41a under RDF SCHEMA C3 (Figure 13) should be present. This is not valid, because the prefix clause of Q41e is different compared to the one in Q41a, SA and JSAG do not support RDF reasoning (and therefore do not try to solve this test case), while AFMU and TS ignore prefix clauses and classify this particular test case according to the wrong specification.

Therefore, we have changed the expected answers for tests p24 and rdfs21. Also, we have added two additional positive test cases, named p24a and rdfs21a, based on p24 and rdfs21, but with projection ?x in the select clauses and the same prefixes, respectively. These changes fix the aforementioned problems.

\(^5\)This should not be considered as the comparison of solvers Z3 and Vampire.
**SQCFrameWork** is a framework that generates customized SPARQL query containment benchmarks [89, 90] based on real SPARQL query logs. The framework is able to produce benchmarks of different sizes and according to the user-defined criteria which SPARQL construct should be considered⁶, for example, `union`, `optional`, `graph`, subqueries, etc. Also, in this framework, a user can specify the structural features of the SPARQL queries in terms of number of triple patterns, number of projections and join variables, join vertex degree, etc.

The authors also provide pregenerated benchmarks. These benchmarks are split into two groups based on query logs used for their generation:

- DBpedia and
- Semantic Web Dog Food (SWDF).

Each group has five test suites:

- KMeans++,
- DBSCAN+KMeans++,
- FEASIBLE.

---

### SPARQL according to the user-defined criteria which is able to produce benchmarks of different sizes and

Each group has five test suites: queries used for their generation:

- DBpedia and
- Semantic Web Dog Food (SWDF).

Each group has five test suites:

- KMeans++,
- DBSCAN+KMeans++,
- FEASIBLE.

---

⁶A deep analysis of query logs from DBPedia SPARQL endpoint shows that non-conjunctive queries containing `union` and/or `optional` operators are widely used [91], which justifies this feature of the framework.

---

**Fig. 12. Test case p24 from QC Bench**

```sparql
prefix : <http://example.org/>
select * where {
  ?x a :Student .
  [ ?x :name ?y ] union
  [ ?x :ssn ?ssn ] union
  [ ?x :sex ?sex ] union
  [ ?x :memberOf ?dept ] union
  [ ?x :emailAddress ?email ] union
  [ ?x :age ?age ]
  ?x :takesCourse ?course .
}
```

---

**Fig. 13. Test case rdfs21 from QC Bench**

```sparql
prefix : <http://example.org/>
select * where {
  ?x a :Student .
  [ ?x :name ?y ] union
  [ ?x :nickName ?z ] union
  [ ?x :telephone ?tel ] union
  [ ?x :memberOf ?dept ] union
  [ ?x :emailAddress ?email ] union
  [ ?x :age ?age ]
  ?x :takesCourse ?course .
}
```

---

- FEASIBLE-Exemplars and
- Random selection,

which are obtained by different clustering algorithms. For each test suite they generated benchmarks of sizes 15, 25, 50, 75, 100, and 125 (for SWDF), and benchmarks of sizes 2, 4, 6, 9, 12, 15 (for DBpedia), where the specified sizes correspond to the number of super-queries, while the total number of tests is significantly larger. In total, the whole corpus contains 78,359 test cases (2,792 from DBPedia, and 75,567 from SWDF).

The specification states that all test cases are positive (the query containment relation holds). The whole corpus in the form of RDF data is available online³.

In one third of the query pairs, this benchmark considers the containment relation in a subsumption form. Also, although definitions from Section 3 assume that projections variables from related queries should be named equally, it is not the case in this benchmark (an example is presented in Figure 14). As most of the query pairs (93.63%) are built with differently named projection variables, we enhanced our solver to deal with such queries. From a practical point of view, renaming of projection variables is highly justified.

Evaluation of SpeCS confirms that 78.92% of the test cases (61,840 test cases) are positive test cases. However, for the remaining 21.08% (16,519 test cases,

³https://github.com/dice-group/sqcframework/blob/master/SQCFrameWork-benchmarks.7z
all of them from the SWDF test suites), SpeCS finds that the query containment relation is not satisfied.\(^8\) We analyzed these test cases and classified reasons for violating query containment relations into the following four groups:

1. In the expression of filter clause, a variable ?num (instead of ?s) is used, while it has not been assigned/used within a triple pattern (120 test cases), causing a syntax error in the query.

2. Super-queries have a specific value within limit clause, although it is not present, or it has a greater value in the sub-query (2,699 test cases). Therefore, in such situations, the containment relation is also violated, as some super-query’s solutions can be excluded from the result set, while being present in the result set of sub-query.

3. Set of graphs specified with from clauses in the sub-query is not subset of graphs from super-query (178 test cases). In some of them an error in prefix clause caused the differences in from clauses in the queries or totally different graphs have been queried, while in the rest super-query has a single from clause, while sub-query does not contain any from clause and it queries the default graph.

4. A sub-query is not contained in the specified super-query (13,522 test cases). One such example, that demonstrates a problem that occurs in many test cases, is presented in Figure 15. Here, sub-query uses a graph pattern within optional clause, while the same pattern is a mandatory within the super-query. Another example, that is also present in many test cases, includes using a different IRI instead of predicate rdf:type.

For the first group of queries, the syntax errors are fixed by changing ?num with ?s. Regarding the second group, although SpeCS can notice this kind of situations, for evaluation purposes we decided to align SpeCS with other solvers and in such situations we ignore the limit clauses. The rest of query pairs that do not satisfy the containment relation (13,700 test cases), we labeled as negative test cases, i.e, we changed the benchmark specification. The other solvers used in evaluation do not support the majority of these negative cases, while SA does not support any (as these queries contain projection variables). AFMU and TS support only 52 particular tests from group (3), but they classify these tests as positive test cases. JSAG supports only 57 tests. It supports 25 tests from group (3) and, like AFMU and TS also classifies these tests as positive test cases. JSAG supports 32 tests from group (4), and here it correctly confirms that the containment relation is not satisfied. AFMU, TS and JSAG provide incorrect results on supported test cases from group (3) because they ignore the specified graphs within from clauses of queries.

We have also run SpeCS with all these test cases but in different direction, in order to explore how many super-queries are contained in a specified sub-query. There are 4,773 of such query pairs, i.e. 6.09%.

\(^8\)The framework authors are notified and aware of these problems.
8.3. Comparison with Relevant State-of-the-art Solvers

As described in Section 2, there are several state-of-the-art solvers for checking query containment problem. Comparison of all relevant features supported by the solvers is summarized in Table 1, while a detailed information regarding experimental evaluation is presented in Table 2. All of these solvers are used within experimental evaluation.

SA [21] solver covers a well-designed sub-class of SPARQL graph patterns, which contains, compared to our approach, smaller set of language constructs (see Table 1 for details). It does not analyze containment with respect to a RDF SCHEMA supported by SPECS. SA strictly follows the definition of subsumption relation and renaming of variables are not allowed. This work is primarily focused on theoretical complexity examination of subsumption and equivalence relation, while the soundness and completeness of the approach are not considered.

AFMU [38] solver reduces the containment problem to a satisfiability problem, like it is done in our approach. However, it uses $\mu$-calculus, whose solvers are slower than SMT solvers. The set of covered language construct within AFMU is also smaller compared to ours (see Table 1 for details). AFMU does not support subsumption relation and renaming of projection variables, but containment reasoning regarding schema is available. Soundness and completeness of this approach are proved.

TS [39] approach is similar to the approach within AFMU solver, as it uses $\mu$-calculus, but there are some differences in the query modeling. Its disadvantages compared to our approach are the same as in the previous case.

JSAG [40] solver does not support neither some constructs available within SPECS, nor containment under schema, but subsumption and renaming of projection variables are supported. Unlike the other approaches limited to the checking of the containment relation between two queries, JSAG is capable to enumerate all containment mappings among a set of queries. Soundness and completeness of the approach are not proved.

8.4. Experimental Setup

All the experiments were run on the machine with Intel® Core™ i7-4790 CPU @ 3.60GHz processor (8 cores) and 16GB RAM memory, running Ubuntu 18.04.3 LTS operating system.

All experiments are completely reproducible. Tools SA, AFMU, TS, software suitable for running query containment tests and QC Bench are downloaded from the benchmark site [92]. To evaluate JSAG (which is available at Github [93]) and SPECS [43], we have implemented additional JAVA wrappers. A possibility to execute other benchmarks generated by SQCFramework is also added. All of these improvements are publicly available together with detailed results (including execution logs and measured times) of the performed evaluation [43].

8.5. Evaluation Results on Query Containment Benchmark

The upper part of Table 2 contains the summary of the evaluation based on QC Benchmark. SA solver is the fastest one in the CQNoProj test suite, but it has one test incorrectly solved, and it does not support additional SPARQL constructs present in the other test suites. SPECS is the only solver that solves correctly all the test cases (without time limits and incorrect results). Also, it is much faster than AFMU, TS and JSAG, i.e. by a factor that takes values between 2.13 and 41.64.

8.6. Evaluation Results on SQCFramework

The lower part of Table 2 contains the summary of the evaluation based on SQCFramework generated
Table 2

<table>
<thead>
<tr>
<th>Test suites</th>
<th>QC Bench</th>
<th>SQCFramework</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQNoProj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#TO/#Unsupp/#Incorr</td>
<td>4/0/1</td>
<td>0/0/0</td>
</tr>
<tr>
<td>#Correct</td>
<td>16 19</td>
<td>0 20 20 20</td>
</tr>
<tr>
<td>AVG in ms</td>
<td>114.88</td>
<td>5.12 338.68 36.95 12.88</td>
</tr>
<tr>
<td>UCQProj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#TO/#Unsupp/#Incorr</td>
<td>11/0/0</td>
<td>0/0/0 3</td>
</tr>
<tr>
<td>#Correct</td>
<td>18 0 27</td>
<td>26 29 29</td>
</tr>
<tr>
<td>AVG in ms</td>
<td>63.15</td>
<td>- 883.87 45.22 21.23</td>
</tr>
<tr>
<td>UCQrdfs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#TO/#Unsupp/#Incorr</td>
<td>0/0/2</td>
<td>11/0/0 29 3 0 0</td>
</tr>
<tr>
<td>#Correct</td>
<td>27</td>
<td>0 0 20 29 29</td>
</tr>
<tr>
<td>AVG in ms</td>
<td>139.08</td>
<td>- 153.14 19.59</td>
</tr>
</tbody>
</table>

For a comparison purpose, we use the largest test suite from each of 10 groups presented in Subsection 8.2, containing together 19,592 query pairs. Solver SA does not support any of these tests, while solvers AFMU, TS and JSAG do not support the majority of them. The total percentages of successfully solved test cases per solver (within the specified time, i.e. 20000ms), in descending order, are:

<table>
<thead>
<tr>
<th>Solver</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECS</td>
<td>100.00%</td>
</tr>
<tr>
<td>JSAG</td>
<td>12.09%</td>
</tr>
<tr>
<td>TS</td>
<td>5.97%</td>
</tr>
<tr>
<td>AFMU</td>
<td>5.50%</td>
</tr>
<tr>
<td>SA</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Apart from being able to correctly solve all the test cases (one order of magnitude more than others), SPECS is also the fastest solver in all the test suites (by
a factor between 1.14 and 4.29). Note that an important goal of the SPECS implementation was to support all the language constructs present in this benchmark, as this benchmark reflects real SPARQL usage scenarios. However, although on this benchmark SPECS achieves 100% coverage, as SPECS does not cover the whole SPARQL language, it is possible to develop some new test cases that SPECS does not support.

9. Conclusions and Further Work

In this paper we presented a new approach for efficient automated reasoning about query containment problem in SPARQL. The approach reduces the query containment problem to the satisfiability problem in FOL and therefore enables the usage of state of the art SMT solvers. It covers a wide range of language constructs, e.g. conjunctive queries, filter, union, optional, graph clauses, blank nodes, projections, subqueries, built-in functions, etc. The containment problem was considered in both standard and subsumption form. Reasoning under RDF SCHEMA entailment regime is also presented.

We implemented the presented approach as an open source tool SPECS, and illustrated its performance on two different corpora: three test suites of Query Containment Benchmark and ten test suites generated by SQCFramework. The presented evaluation shows that compared to all other available state-of-the-art solvers, SPECS has both a better efficiency and a broader scope of covered language constructs. We identified some problems in the existing benchmarks, and successfully solved them. Within implementation, we also supported renaming of projection variables, as such feature is important and useful in practice.

There are several possible directions for further work: to extend the language coverage while keeping the soundness and completeness of the approach where possible, reasoning about containment considering a wider set of axioms (e.g. SHI axioms), applying the similar approach for different graph query language (e.g. XPath, GQL). We will also consider the possibility of applying the presented approach in the context of code refactoring.

Acknowledgments

This work was partially supported by the Serbian Ministry of Science grant 174021 and by COST action CA15123.

References


