SPARQLing Constraints for RDF

Georg Lausen Michael Meier[*] Michael Schmidt[*]

Institut für Informatik, Universität Freiburg
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg i.Br., Germany

{lausen, meierm, mschmidt}@informatik.uni-freiburg.de

ABSTRACT

The goal of the Semantic Web is to support semantic interoperability between applications exchanging data on the web. The idea heavily relies on data being made available in machine readable format, using semantic markup languages. In this regard, the W3C has standardized RDF as the basic markup language for the Semantic Web. In contrast to relational databases, where data relationships are implicitly given by schema information as well as primary and foreign key constraints, relationships in semantic markup languages are made explicit. When mapping relational data into RDF, it is desirable to maintain the information implied by the origin constraints. As an improvement over existing approaches, our scheme allows for translating conventional databases into RDF without losing general constraints and vital key information. As much as in the relational model, those information are indispensable for data consistency and, as shown by example, can serve as a basis for semantic query optimization. We underline the practicability of our approach by showing that SPARQL, the most popular query language for RDF, can be used as a constraint language, akin to SQL in the relational context. As a theoretical contribution, we also discuss satisfiability for interesting classes of constraints and combinations thereof.

1. INTRODUCTION

The goal of the Semantic Web [3] is to support semantic interoperability between applications exchanging data on the web. In order to provide a standardized machine readable data format, different semantic markup languages have been proposed. Together with various OWL dialects [22], the World Wide Web Consortium has recommended the Resource Description Framework (RDF) [25] as a standard format for representing data in the Semantic Web.

RDF databases are collections of so-called *triples of knowledge* over certain resources, which are represented by URI references and literals. Each such triple (a, b, c) states a binary relationship, in which a appears in the role of the *subject*, b in the role of the *predicate* also called *property*, and c in the role of the *object*. An RDF database typically is represented by a graph. The edges of the graph are directed from the subject to the object, and labeled with the property. The W3C RDF recommendation [25] provides an abstract syntax for the knowledge triples and defines a small base vocabulary. RDF Schema (RDFS) [26] extends the RDF vocabulary to impose more structure on an RDF graph. Supplementary, a formal semantics for RDF and RDFS has been fixed in [27]. This semantics standardizes the meaning of the RDF and RDFS vocabularies and, in respect thereof, forms the basis for semantic interoperability.

The vision of the Semantic Web necessitates data being made available in a machine readable format. Today, large amounts of data reside in relational databases. Making this data accessible to Semantic Web applications thus requires a translation into semantic markup languages. Based on this requirement, up-to-date several approaches for mapping relational data to RDF have been proposed [2, 4, 6, 5].

Surprisingly, existing approaches to translating relational data into RDF mostly disregard relational constraints, in particular key and foreign key definitions. We argue that, in line with the prominent role of constraints in relational databases, constraints over RDF are important for multiple reasons. First, they can serve as a mechanism for restricting the number of valid states (RDF graphs) and, in respect thereof, are indispensable for both checking and maintaining data consistency. Second, constraints imply a semantics on the data. Queries typically exploit such constraints, e.g. build upon primary and foreign keys. Whenever constraints from the relational database are ignored, it might become hard to write appropriate queries. Finally, it is well-known that constraints in relational (and also deductive) databases offer valuable support for semantic query optimization techniques [15, 29, 10]. One might expect that similar optimizations are possible for queries over constrained RDF data. Exemplarily, we will show that SPARQL [30], a declarative RDF query language, which has been standardized by the W3C, is suitable for semantic optimization in a similar fashion as relational database languages are.

Based on the previous argumentation, which is further supported by an in-depth discussion of an example presented in [16], we suggest to extend RDF by constraints akin to relational databases. By the best of our knowledge, this is the first approach for fully integrating relational constraints into RDF. By maintaining origin key and foreign key constraints,

[*]The work of this author was funded by Deutsche Forschungsgesellschaft, grant GRK 806/2.

our scheme supports a mapping from relational databases to RDF without losing vital restrictions and key information.

On the one hand, our approach to constraints for RDF is strongly related to the way constraints are treated in SQL. We propose an explicit statement of key and foreign key constraints by extending the RDF vocabulary. Moreover, for the definition of general constraints we shall allow arbitrary query expressions in SPARQL akin to the `ASSERTION` clause in SQL. On the other hand, our constraints also include the constraints that can be expressed by the standard RDFS vocabulary, such as subclass and subproperty specifications. We will finally show that minor extensions to SPARQL are sufficient to make the language an excellent candidate for both expressing and checking constraints over RDF.

Contributions. We make the following contributions.

- We extend RDF by a set of constraints, which allow for mapping relational data to RDF without loss of vital restrictions and key constraints.
- We study the satisfiability for combinations of different types of constraints.
- We demonstrate that SPARQL is qualified for dealing with constraints. In particular, it can be used for extracting constraints from an RDF database and for checking an RDF database against sets of constraints.
- We exemplarily show that semantic optimization of SPARQL queries based on constraint-implied knowledge is a very promising optimization approach.

Structure. In Section 2 we introduce the basic formal framework for relational databases and RDF. Subsequently, we propose to extend the RDF vocabulary in Section 3, in order to be able to represent general keys, foreign keys, and other types of constraints in RDF. Section 4 addresses the satisfiability problem of constrained RDF vocabularies. To underline the practicability of our approach, we show in Section 5 that SPARQL can be used for both extracting constraints from RDF graphs and checking key and foreign key constraints. We also present SPARQL queries that check other natural constraints, such as min/max-cardinalities. Practical benefits of constraints for RDF are outlined in Section 6, where we show how SPARQL can be empowered by constraints in a similar way as SQL. In particular, we discuss a sample scenario, where semantic optimization of a SPARQL expression bases upon the knowledge of the respective constraints. Related work then is presented in Section 7, before we conclude with a short summary of our results and a discussion of future work in Section 8.

2. FRAMEWORK

We first shall introduce some basic terminology from relational databases, then, in larger detail, present a formal exposition of RDF, and finally present a simple, preliminary mapping from a relational database to an RDF graph.

A relational database schema $\mathcal{R}$ is descibed by a set of relational schemata identified by R, $\mathcal{R} = (R_1, ..., R_n)$. We use $\mathit{Att}(R_i)$ to denote the set of attributes of the relation symbol R_i. An instance I of $\mathcal{R}$ is a tuple $(I_1, ..., I_n)$, where for $i \in [n]$[1] I_i is a finite instance of R_i, i.e. a finite subset of the n-ary cartesian product over an underlying domain. An element $\mu \in I_i$ is called tuple. Let $a \in \mathit{Att}(R)$, we use $\mu.a$ to denote the value of the attribute a of the tuple μ.

A *key* over $\mathcal{R}$ is an expression of the form $R[a_1...a_k] \rightarrow R$, where $R \in \mathcal{R}$ and for $i \in [k]$ it holds that $a_i \in \mathit{Att}(R)$.[2] Let I be an instance of $\mathcal{R}$. I satisfies $R[a_1...a_k] \rightarrow R$ if and only if $\forall \mu_1, \mu_2 \in I\ (\bigwedge_{1 \leq i \leq k} (\mu_1.a_i = \mu_2.a_i) \rightarrow (\mu_1 = \mu_2))$.

A *foreign key* over $\mathcal{R}$ is an expression of the form $R[a_1...a_k] \subseteq R'[a'_1...a'_k]$, where $R, R' \in \mathcal{R}$, $\{a_1, ..., a_k\} \subseteq \mathit{Att}(R)$ and $\{a'_1, ..., a'_k\} \subseteq \mathit{Att}(R')$. R is called *child* and R' *parent* of the foreign key. Then I satisfies the foreign key $R[a_1...a_k] \subseteq R'[a'_1...a'_k]$ if and only if $\forall \mu_1 \in I\ \exists \mu_2 \in I'\ (\bigwedge_{1 \leq i \leq k} \mu_1.a_i = \mu_2.a'_i)$ and $I' \models R'[a'_1...a'_k] \rightarrow R'$.

Now we are going to introduce the basic RDF framework needed for this paper. The definitions are based on [21], however adapted for our purposes. Since we only discuss graphs obtained from translations of relational databases into RDF, all our graphs are free of blank nodes. Thus, in order to keep the framework simple, we do not consider blank nodes. Moreover, restricting ourselves to the database domain, we consider an RDF *vocabulary* $\mathcal{V} = (N_C, N_P)$, where N_C is a finite set of *classes* and N_P is a finite set of *properties*. Classes and properties are denoted by URI references. As we will see soon, relation schemata will be mapped to classes, while attributes will always be mapped to properties. Finally, we use literals in a less general way, because literals for us are simply attribute values taken from the tuples in the relations.

Given a vocabulary $\mathcal{V}$, an *interpretation* $\mathcal{I}$ of $\mathcal{V}$, $\mathcal{I} = (\Delta_\mathcal{I}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$ is given as

- $\Delta_\mathcal{I}$ is a possibly infinite, nonempty set, called *object domain*,
- Δ_D is a possibly infinite, nonempty set, called the *data domain*, which we assume to be disjoint from $\Delta_\mathcal{I}$, i.e. $\Delta_\mathcal{I} \cap \Delta_D = \emptyset$,
- $.^{\mathcal{I}_C}$ is the *class interpretation function* assigning to each class $A \in N_C$ a finite subset $A^{\mathcal{I}_C} \subseteq \Delta_\mathcal{I}$,
- $.^{\mathcal{I}_P}$ is the *property interpretation function* assigning to each property $Q \in N_P$ a finite subset $Q^{\mathcal{I}_P} \subseteq \Delta_\mathcal{I} \times (\Delta_\mathcal{I} \cup \Delta_D)$.

Based on a given interpretation we can introduce a corresponding RDF graph. Among the nodes of an RDF graph we distinguish between URIs and literals. Literals in our framework are the elements of the data domain, and URIs are used to identify the elements of the object domain. By requiring that the object domain and the data domain are disjoint, we assume that literal values cannot be identified by URIs [27]. Therefore, in an RDF graph resulting from our definitions, each occurrence of a literal will be represented by a unique node in the graph. This fact makes the following definitions a bit more complicated.

Let $\mathcal{I} = (\Delta_\mathcal{I}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$ be an interpretation. The *RDF graph* $G^\mathcal{I} = (N^\mathcal{I}, E^\mathcal{I})$ of $\mathcal{I}$ is a directed labeled graph, where

- for the set of nodes $N^\mathcal{I}$ we have $N^\mathcal{I} = N_C \cup \{a, b \mid (a, b) \in Q^{\mathcal{I}_P}, Q \in N_P, b \in \Delta_\mathcal{I}\} \cup \{a, b_{a,Q} \mid (a, b) \in Q^{\mathcal{I}_P}, Q \in N_P, b \in \Delta_D\}$, and

[1] By $[n]$, we denote the set $\{1, ..., n\}$ for any $n \in \mathbb{N} \backslash \{0\}$.

[2] We consider for each schema only one key, the primary key.

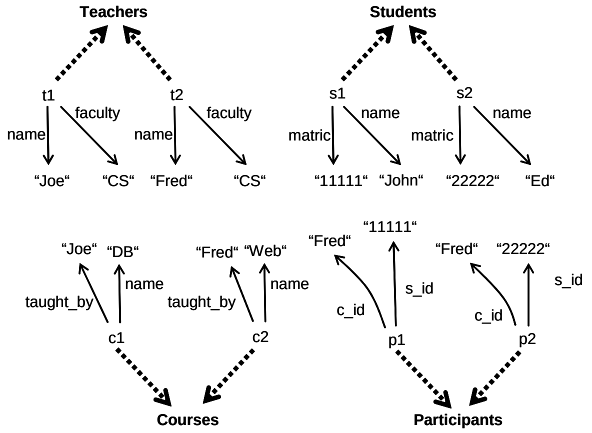

Figure 1: The result of mapping a relational database to an RDF graph by treating all tuples independently. Classes are indicated by bold font and typing by dotted edges.

- for the set of labeled arcs $E^{\mathcal{I}}$ we have $E^{\mathcal{I}} = \{(a, Q, b) \mid (a, b) \in Q^{\mathcal{I}_P}, b \in \Delta_{\mathcal{I}}\} \cup \{(a, Q, b_{a,Q}) \mid (a, b) \in Q^{\mathcal{I}_P}, b \in \Delta_D\} \cup \{(a, \mathtt{rdf:type}, C) \mid a \in C^{\mathcal{I}_C}, C \in N_C\}$.

Several approaches described in the literature elaborate on mappings of a relational database to an RDF graph (e.g. [2, 6, 5]). Our interest are generic mappings that can be applied automatically and make semantic properties of the relational database explicit in the resulting RDF graph. Powerful, explicitly defined mappings as described in [6, 5] are a different issue. It is not clear how both approaches can be combined.

We will now introduce a very naive mapping scheme, which maps all attribute values into the RDF data domain. As we will point out, this simple approach has severe deficiencies. In Section 3.1.1 we will improve on our naive mapping scheme, and finally present an approach that uses both the data and the object domain for mapping attribute values.

Let R be a relation schema, where $Att(R) = \{A_1, \ldots, A_k\}$. We introduce a class C_R and properties $Q_{R,A_1}, \ldots, Q_{R,A_k}$. Further let I be an instance of R. Then for every tuple $\mu = (a_1, \ldots, a_k)$ in I we set up a unique RDF node $n_\mu \in \Delta_I$ and a labeled edge $(n_\mu, \mathtt{rdf:type}, C_R)$. In addition, for every nonnull value $\mu.A$ of μ, $A \in Att(R)$, we introduce an edge $(n_\mu, Q_{R,A}, (\mu.A)_{n_\mu, Q_{R,A}})$, where, due to the value-based nature of the relational data model, $(\mu.A)_{n_\mu, Q_{R,A}}$ is a literal taken from the data domain. Let us denote an RDF graph constructed in the described way by $G^I_{Rel} = (N^I_{Rel}, E^I_{Rel})$.

We will demonstrate this approach by means of a small running example and then see immediately its deficiencies. We consider the well-known scenario of teachers, students, and courses. Starting from the scratch, we discuss the following relational sample instance.

Teachers

name	faculty
Joe	CS
Fred	CS

Students

matric	name
11111	John
22222	Ed

Courses

taught_by	name
Joe	DB
Fred	Web

Participants

c_id	s_id
Fred	11111
Fred	22222

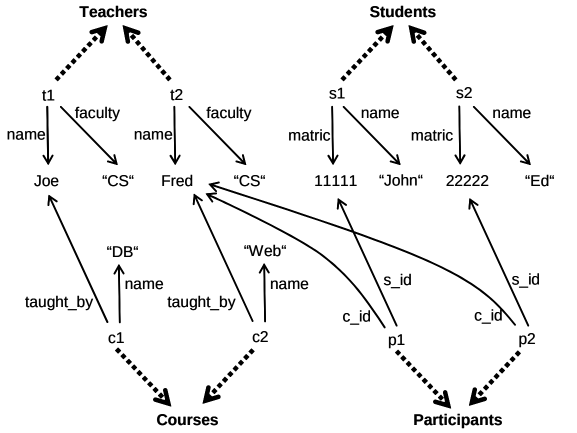

Figure 2: Key and foreign key values as URIs.

Key attributes are marked in bold font. Further, attribute `taught_by` is a foreign key w.r.t. `Teachers`, Attribute `c_id` is a foreign key w.r.t. `Courses`, and Attribute `s_id` is a foreign key w.r.t. `Student`.

Applying the mapping introduced so far results in the RDF graph depicted in Figure 1. The problem with the mapping is that information about key and foreign key constraints is completely lost. Key and foreign key constraints constitute valuable semantic information and we will subsequently show how such constraints can be stated in RDF.

3. CONSTRAINTS

3.1 Constraints Stated Inside RDF

3.1.1 Relational Constraints

We have argued that the mapping from a relational database to RDF (cf. Figure 1) as introduced so far is insufficient, as all key and foreign key information is completely lost. Moreover, as properties in RDF are multi-valued in general, we also lose the information that, according to our mapping, properties are resulting from attributes and therefore are functional (i.e. at most single-valued). In addition, whenever an attribute was defined `NOT NULL`, and consequently in the RDF graph the corresponding property must be defined for all objects of a certain class, then this information shall not be lost in the RDF representation. In particular, in Section 6 we will show that this kind of information is a valuable input for a semantic RDF query optimizer.

We start by showing how key and foreign keys can be represented in RDF. Obviously, to represent key and foreign key values in an RDF graph, we should use the object domain and not the data domain of RDF. Therefore, URIs are the appropriate means to represent key and foreign key values (cf. [6, 5]). For example, instead of introducing an edge from object s_2 to value 11111, we can lift the value 11111 of property `matric` to a URI and introduce an edge from $p1$ to that URI. This approach is demonstrated in Figure 2; note that URIs appear only once in the graph and consequently nodes previously representing foreign key and corresponding key values now have collapsed.

Applying this mapping to a given database instance, the resulting RDF graph is denoted by $G^I_{URI} = (N^I_{URI}, E^I_{URI})$. However, different to [6, 5], we argue that also key and foreign key constraints should be stated explicitly in the RDF graph, because otherwise it is difficult or even impossible to extract this important semantic information from a given RDF graph. To make this possible we need additional definitions and mechanisms, which we are going to present now.

Let $\mathcal{I} = (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$ be an interpretation. Let $C \in N_C$, $Q, Q_i \in N_P$, $1 \leq i \leq n$. First we define *functional* constraints. A functional constraint $Func(C, Q)$ defines a property Q to be functional over objects of class C. $\mathcal{I}$ satisfies $Func(C, Q)$, $\mathcal{I} \models Func(C, Q)$, if there holds:

$$\{x \mid \#\{y \mid (x, y) \in Q^{\mathcal{I}_P}\} \leq 1\} \supseteq C^{\mathcal{I}_C}.$$

A functional property Q is called *total* with respect to a class C, whenever on every object of class C property Q is defined. We write $Total(C, Q)$ and define $\mathcal{I}$ satisfies $Total(C, Q)$, $\mathcal{I} \models Total(C, Q)$, if there holds:

$$\{x \mid \#\{y \mid (x, y) \in Q^{\mathcal{I}_P}\} = 1\} \supseteq C^{\mathcal{I}_C}.$$

Next we formally define keys and foreign keys for RDF. We write $Key(C, [Q_1 \ldots Q_n])$ to state a key assigned to class C over properties $Q_1, \ldots, Q_n$ and we write $FK(C, [Q_1 \ldots Q_n], C', [Q'_1 \ldots Q'_n])$ to state a foreign key over the respective properties of child C and parent C'. Semantics of keys and foreign keys is as follows.

- $\mathcal{I}$ satisfies $Key(C, [Q_1, \ldots, Q_n])$,
$$\mathcal{I} \models Key(C, Q_1, \ldots, Q_n),$$
if, whenever $\exists o_1, o_2 \in C^{\mathcal{I}_C}$ such that $\exists v_i \in \Delta_{\mathcal{I}} \cup \Delta_D, 1 \leq i \leq n$, where $(o_1, v_i), (o_2, v_i) \in Q_i^{\mathcal{I}_P}$, then $o_1 = o_2$. Moreover, $\mathcal{I} \models Total(C, Q_i)$ must hold for all $i \in [n]$ as well.
- $\mathcal{I}$ satisfies $FK(C, [Q_1 \ldots Q_n], C', [Q'_1 \ldots Q'_n])$,
$$\mathcal{I} \models FK(C, [Q_1 \ldots Q_n], C', [Q'_1 \ldots Q'_n]),$$
if, whenever $o_1 \in C^{\mathcal{I}_C}$, then $\exists o_2 \in C'^{\mathcal{I}_C}$ such that $(o_1, v_i) \in Q_i^{\mathcal{I}_P}$ implies $(o_2, v_i) \in Q_i'^{\mathcal{I}_P}, 1 \leq i \leq n$. Moreover, it must hold $\mathcal{I} \models Key(C', [Q'_1, \ldots, Q'_n])$; thus the parent properties referred to must form a key.

To get a clean and general solution for expressing functional properties, keys, and foreign keys inside an RDF graph we now propose to make that information explicit. To this end, we extend the RDF vocabulary by a new namespace, which will be identified by prefix `rdfc`. This namespace extends the RDF vocabulary by two classes `rdfc:Key` and `rdfc:FKey`, allowing to type properties and to introduce new objects representing key and foreign key constraints.

We believe that explicitly stating constraints as part of an RDF graph will alleviate the construction of RDF processing technology. In Section 6 we will show how constraints can be used for optimizing RDF query expressions.

Let us first concentrate on keys and foreign keys. Keys and foreign keys may be built out of several components which are of type `rdf:Seq`. Using properties `rdfc:Key` and `rdfc:FKey` we are able to associate keys and foreign keys with the classes for which they apply and finally, by property `rdfc:ref` we link a foreign key to the key of the respective class. The construction we propose is akin to SQL and is demonstrated in Figure 3 with respect to the keys of classes `inProject` and `respFor` and the foreign key of class `respFor`.

Let $G^{\mathcal{I}}_{URI} = (N^{\mathcal{I}}_{URI}, E^{\mathcal{I}}_{URI})$ be an RDF graph under consideration. We add new nodes and arcs to the graph to make functional properties, keys, and foreign keys explicit. Starting from the graph depicted in Figure 2 we proceed as follows. As exemplified in Figure 3, for classes `Teachers`, `Students` and `Participants` we introduce objects `P_Key`, `T_Key` and `S_Key`, respectively. In addition we introduce objects `P_FKey1` and `P_FKey2` to represent the foreign keys of class `Participants`. We add corresponding typing edges from the newly introduced objects to class `rdfc:Key` and class `rdfc:FKey`, respectively. Next, we add the corresponding edges labeled with `rdfc:Key`, respectively `rdfc:FKey`, in order to relate a key and foreign key to its respective class; we add edges labeled with `rdfc:ref` to relate foreign keys to their parent keys. Finally, as keys and foreign keys are of type `rdf:Seq` as well, we add corresponding typing edges and also indicate the components of keys and foreign keys by the edges labeled with `rdf:_1` and `rdf:_2`.

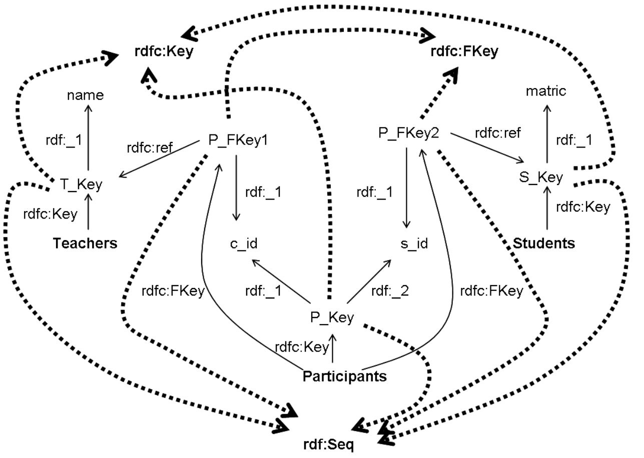

Figure 3: Keys and foreign keys of classes `Teachers`, `Students` and `Participants` are explicitly represented; class `Courses` is omitted.

To type properties, we introduce a new node for each property into the RDF graph. The property then is typed by class `rdf:Property`. To express functionality (respectively totality) of property Q over class C, we introduce an edge labeled `rdfc:funct` (respectively `rdfc:total`) from C to Q.

3.1.2 RDFS Constraints and More

Now let us first look at the kind of constraints that are part of RDFS [26], however used as axioms for inferencing in that context. We argue that inferencing is a separate issue; we will further comment on this difference in Section 7. Independently hereof, these constraints are interesting for checking an RDF graph as well. The following types of constraints can be used to express *subclass relationships, subproperty relationships* and *domain* and *range definitions* of properties. RDFS provides an appropriate vocabulary to express these constraints as part of an RDF graph.

Let be given a vocabulary $\mathcal{V} = (N_C, N_P)$ of RDF and a corresponding *interpretation* $\mathcal{I} = (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$. Let $C, D \in N_C$ and $R, S \in N_P$. Let ϕ be one of the constraints mentioned above. We define $\mathcal{I}$ satisfies ϕ, $\mathcal{I} \models \phi$, if depending on ϕ there holds:

$$\begin{array}{ll} SubC(C, D): & C^{\mathcal{I}_C} \subseteq D^{\mathcal{I}_C}, \\ SubP(R, S): & R^{\mathcal{I}_P} \subseteq S^{\mathcal{I}_P}, \\ PropD(R, C): & \{x \mid \exists y : (x, y) \in R^{\mathcal{I}_P}\} \subseteq C^{\mathcal{I}_C}, \\ PropR(R, C): & \{y \mid \exists x : (x, y) \in R^{\mathcal{I}_P}\} \subseteq C^{\mathcal{I}_C}. \end{array}$$

The following constraints are not part of RDFS, however are well-known from other representation languages. First we introduce *cardinality* constraints, which are well-known from Entity-Relationship modelling, respectively UML design of a relational database. Let $n \geq 0$ and $C \in N_C$, $R \in N_P$. We write $Min(C, n, R)$, respectively $Max(C, n, R)$, to specify that for any object $o \in C^{\mathcal{I}_C}$, on which a property R is defined, R associates o with at least, respectively at most, n other objects. Let ψ be either a *min-cardinality constraints* $Min(C, n, R)$ or a *max-cardinality constraints* $Max(C, n, R)$. We define $\mathcal{I}$ satisfies ψ, $\mathcal{I} \models \psi$, if there holds:

$$\begin{array}{ll} Min(C, n, R): & \{x \mid \#\{y \mid (x, y) \in R^{\mathcal{I}_P}\} \geq n\} \supseteq C^{\mathcal{I}_C} \\ Max(C, n, R): & \{x \mid \#\{y \mid (x, y) \in R^{\mathcal{I}_P}\} \leq n\} \supseteq C^{\mathcal{I}_C}. \end{array}$$

Next we introduce *subproperty-chain* constraints. This kind of constraints resembles path-constraints known from object-oriented databases, XML and also OWL 1.1 [23]. For a class $C \in N_C$ and properties $S, R_i \in N_{Po}$, $1 \leq n$, we introduce a constraint $SubPChain(C, R_1, \ldots, R_n, S)$, which allows to define a *subproperty-chain constraint.* Let $\circ$ denote the composition of binary relations. Let $\phi = SubPChain(C, R_1, \ldots, R_n, S)$. We define $\mathcal{I}$ satisfies ϕ, $\mathcal{I} \models \phi$, if there holds:

$$\{(x, y) \mid (x, y) \in R_1^{\mathcal{I}_P} \circ \ldots \circ R_n^{\mathcal{I}_P}, x \in C^{\mathcal{I}_C}\} \subseteq \\ \{(x, y) \mid (x, y) \in S^{\mathcal{I}_P}, x \in C^{\mathcal{I}_C}\}.$$

The final two kinds of constraints we shall discuss are mostly interesting for technical reasons (cf. Section 4). For a class C, the *singleton* constraint $Single(C)$ guarantees the existence of a single element. We define $\mathcal{I}$ satisfies $Single(C)$, $\mathcal{I} \models Single(C)$, if there holds $|C^{\mathcal{I}_C}| = 1$. The *anti-key* constraint $AntiKey(C, [Q_1 \ldots Q_n])$ states that properties $Q_1, \ldots, Q_n$ do not constitute a key for class C. We define $\mathcal{I}$ satisfies $AntiKey(C, [Q_1 \ldots Q_n])$, $\mathcal{I} \models AntiKey(C, [Q_1 \ldots Q_n])$, if $\exists o_1, o_2 \in C^{\mathcal{I}_C}$, $o_1 \neq o_2$, such that $\exists v_i \in \Delta_{\mathcal{I}} \cup \Delta_D, 1 \leq i \leq n$, where $(o_1, v_i), (o_2, v_i) \in Q_i^{\mathcal{I}_P}$.

3.2 Constraints Stated Outside RDF

To allow also application specific constraints, we propose to state them explicitly by RDF query language expressions. This approach is inspired by SQL, where general constraints can be defined in the CHECK-clause by means of appropriate SQL query expressions. We do not elaborate on this any further. However, in Section 5 we will provide corresponding SPARQL query expressions for all kinds of constraints discussed so far. While this is primarily done to demonstrate how constraint checking can be implemented, it also shows how constraints can be expressed in SPARQL in general.

4. SATISFIABILITY

Relational constraints (Section 3.1.1), i.e. functionality of properties, keys, and foreign keys, as well as the constraints taken from RDFS (Section 3.1.2), i.e. subclass, subproperty, property range and domain restrictions, are stated as part of an RDF graph by means of a dedicated vocabulary. They are called *vocabulary* constraints in the sequel. All other kinds of constraints will be called *non-vocabulary* constraints.

Let $\mathcal{V}$ be a given RDF vocabulary and $\mathcal{C}$ a set of arbitrary constraints over $\mathcal{V}$. We call $\mathcal{V}_{\mathcal{C}} = (\mathcal{V}, \mathcal{C})$ a *constrained* RDF vocabulary. We will now discuss the satisfiability of a constrained RDF vocabulary $\mathcal{V}_{\mathcal{C}}$. We call $\mathcal{V}_{\mathcal{C}}$ *satisfiable* iff there exists an interpretation $\mathcal{I}$ of $\mathcal{V}$ which satisfies all the constraints in $\mathcal{C}$, where additionally, for some class C of $\mathcal{V}$ we have $C^{\mathcal{I}_C} \neq \emptyset$ and for some property P of $\mathcal{V}$ we have $P^{\mathcal{I}_P} \neq \emptyset$. Obviously, a constrained vocabulary whose set of constraints is not satisfiable should be revised, because certain classes and properties cannot be populated.

Assume first that $\mathcal{C}$ contains only vocabulary constraints. The question of satisfiability for this mixture of constraints turns out to be trivial, i.e. satisfiability is always guaranteed. To see this, let $o \in \Delta_I$ and define $C^{\mathcal{I}_C} = \{o\}$ and $R^{\mathcal{I}_P} = \{(o, o)\}$ for all classes and properties.

LEMMA 1. *Let $\mathcal{V}$ be a RDF vocabulary, $\mathcal{C}$ be a set of constraints over $\mathcal{V}$ containing arbitrary vocabulary constraints, max-cardinality, subproperty-chain and singleton constraints. There exists an interpretation $\mathcal{I}$ of $\mathcal{V}$, such that $\mathcal{I} \models \mathcal{C}$.*

When – in addition to the vocabulary constraints – we also allow arbitrary non-vocabulary constraints, satisfiability becomes undecidable in general. We are interested in the kind of mixtures of constraints that are responsible for undecidability, and those mixtures that still guarantee decidability when combined with the vocabulary constraints.

For undecidability, as known undecidable reference problem we refer to the implication problem for keys by keys and foreign keys in relational databases (Lemma 3.2 in [12]). Let $\mathcal{R}$ be a (relational) schema, Σ a set of keys and foreign keys over $\mathcal{R}$, and φ a key over $\mathcal{R}$. Does $\Sigma \models \varphi$, i.e. does any interpretation I which satisfies all the keys and foreign keys in Σ also satisfy the key φ?

From the undecidability of the implication problem $\Sigma \models \varphi$ we can immediately conclude the undecidability of the satisfiability problem with respect to $\Sigma \wedge \neg\varphi$. This means, whenever we allow a set of constraints being formed out of key, foreign key and anti-key constraints, we cannot algorithmically test for satisfiability. However, as anti-key constraints are not natural constraints from a practical point of view, other mixtures of constraints are still of interest. We can proof the following theorem.

THEOREM 1. *Let $\mathcal{V}$ be a RDF vocabulary, and let $\mathcal{C}$ be a set of constraints over $\mathcal{V}$ containing arbitrary vocabulary constraints, cardinality constraints and either subproperty-chain or singleton constraints. Then testing satisfiability of the constrained RDF vocabulary $\mathcal{V}_{\mathcal{C}}$ is undecidable.*

Appendix A provides the proof for singleton constraints. Note that we do not consider RDF blank nodes in this section. We claim that they do not play a role in our proofs. In particular, when considering decidability, a blank node states the existence of e.g. a resource, which could replace the blank node. In our undecidability proof, we reduce from a problem in relational databases, where we simply have plain values, which can be represented as resources in RDF. Consequently, blank nodes are immaterial for us here.

Next we investigate, whether restricting to only unary foreign keys makes the satisfiability problem easier. A foreign key $FK(C, [Q_1 \ldots Q_n], C', [Q'_1 \ldots Q'_n])$ is called unary if and only if $n = 1$. The following lemma shows that this is not the case. The proof of the lemma can be found in Appendix B.

LEMMA 2. *Let φ be a foreign key and $\mathcal{V}$ a RDF vocabulary. Then, there exists a vocabulary $\mathcal{V}'$ of RDF and a finite set Π of unary foreign key, key and subproperty-chain constraints such that φ is satisfiable with respect to $\mathcal{V}$ iff Π is satisfiable with respect to $\mathcal{V}'$.*

Let us now turn our focus towards decidable cases. The satisfiability of constrained RDF vocabularies, where subclass, subproperty, property domain and range, min-cardinality, max-cardinality, unary foreign key and unary key constraints are allowed, can be decided using a reduction to the description logic $\mathcal{ALCHIQ}$. For $\mathcal{ALCHIQ}$ it is known that satisfiability can be decided in exponential time [32].

Corollary 1. *Let $\mathcal{V}_\mathcal{C} = (\mathcal{V}, \mathcal{C})$ be a constrained RDF vocabulary, where $\mathcal{C}$ only contains subclass, subproperty, property domain and property range, min- and max-cardinality, as well as unary foreign key and unary key constraints. The satisfiability of $\mathcal{V}_\mathcal{C}$ can be decided in* ExpTime.

The translation to $\mathcal{ALCHIQ}$ is shown in Appendix C. An important property of description logics is that their expressions can be nested. Observe that in our translation to $\mathcal{ALCHIQ}$ neither arbitrary, nor arbitrarily nested expressions can occur. For this reason, we believe that even better complexity bounds can be shown, and we are planning to address this issue in future investigations.

As a final remark, we mention that there are also description logics that incorporate keys directly [11, 34, 18, 33, 17, 9]. However, these approaches typically do not provide the same set of constraints, e.g. number restrictions are missing.

5. SPARQLING CONSTRAINTS

In this section we show that the SPARQL query language is a good candidate for stating constraints and checking RDF graphs against constrained RDF vocabularies. For space limitations, we omit an introduction to SPARQL, but refer the interested reader to [30] for the language definition.

In Section 5.1 we show how SPARQL can be used to extract constraints that have been specified with the `rdfc` vocabulary. Then, in Section 5.2 we present queries for checking the validity of RDF graphs against different constraint types. We finally propose extensions to SPARQL, and discuss the complexity of constraint checking in Section 5.3.

5.1 Extracting Constraints

The following SPARQL query extracts all keys and foreign keys defined with the `rdfc` vocabulary on top of RDF.

```
SELECT ?keyname ?class ?keytype ?keyatt ?ref
WHERE {
  {
    ?class rdfc:Key ?keyname.
    ?keyname rdf:type ?keytype;
             ?seqrel ?keyatt.
    FILTER (?keytype=rdfc:Key &&
            ?seqrel!=rdf:type)
  } UNION {
    ?class rdfc:FKey ?keyname.
    ?keyname rdf:type ?keytype;
             ?seqrel ?keyatt;
             rdfc:ref ?ref.
    FILTER (?keytype=rdfc:FKey &&
            ?seqrel!=rdf:type  &&
            ?seqrel!=rdfc:ref)
  }
} ORDER BY DESC(?keytype) ?keyname
```

The first part of the Union extracts all keys. Here, variable *?keyname* binds to the different keys, *?class* binds to the class the current key is defined on, *?keytype* always binds to `rdfc:Key`, and *?keyatt* binds to all attribute URIs of the current key. As a consequence, for an n-ary key, the result contains n tuples, which only differ in their *?keyatt* value. Foreign key extraction in the second part of the Union is similar. There, variable *?keytype* binds to `rdfc:FKey`, and variable *?ref*, which was not used in the first part, binds to the name of the referenced key. Note that we sort the result in descending order by *?keytype* (i.e. list keys before foreign keys), and by *?keyname* in ascending order, to group keys by their name. Figure 4 shows the result of the extraction query executed on the graph in Figure 3.

?keyname	*?class*	*?keytype*	*?keyatt*	*?ref*
P_Key	Particip.	rdfc:Key	c_id	
P_Key	Particip.	rdfc:Key	s_id	
S_Key	Students	rdfc:Key	matric	
T_Key	Teachers	rdfc:Key	name	
P_FKey1	Particip.	rdfc:FKey	c_id	T_Key
P_FKey2	Particip.	rdfc:FKey	s_id	S_Key

Figure 4: Result of constraint extraction from the graph in Figure 3.

5.2 Checking Constraint Violation

We realize a check whether an RDF instance satisfies (a set of) constraints by querying constraint-violating situations in the respective RDF instance. Whenever no violation is detected, all constraints are satisfied. We use the SPARQL "Ask" query form to check violation of constraints. Ask queries are the Boolean counterparts of Select queries, i.e. they return *yes* if the specified query pattern contains one or more solution mappings, and *no* otherwise. We define constraint violation check queries as follows.

Definition 1
Let Q be a query and C be a constraint. We say that query Q *checks the violation of* C if, for each graph G, it returns *yes* if and only if G violates C. □

For simplicity, we study each type of constraint individually and discuss only queries that check violation of one constraint at a time. As it turns out, according to our definition all types of our constraints except anti-key constraints can be checked with SPARQL queries. We claim that, for the latter type, it is only possible to provide an Ask query that returns *no* (instead of *yes*) exactly if the constraint is violated. We will discuss implications, and finally propose according extensions to the SPARQL language in Section 5.3.

We now present check queries for the different types of constraints. We have verified the correctness of all queries with the ARQ SPARQL Processor for Jena [13].

5.2.1 Key Constraints

We start with the violation check for key constraints. According to Definition 1, we provide an `Ask` query that returns *yes* exactly if the key is violated. Given a key constraint *Key*(`C`,[`p1`,...,`pn`]), the idea is to check for two distinct objects that bind to the same nodes through `p1`, ..., `pn`.

Note that, by definition key constraints comprise totality constraints, more precisely one constraint *Total*(`C`,`pi`) for each property `pi`. We defer the discussion of totality constraints to Section 5.2.3, only showing how to encode the violation check for the uniqueness property. However, we emphasize that the totality checks can easily be integrated.

```
ASK {
  ?x rdf:type C.
  ?y rdf:type C.
  ?x p1 ?p1; [...]; pn ?pn.
  ?y p1 ?p1; [...]; pn ?pn.
  FILTER (?x!=?y)
}
```

The query introduces variables ?x and ?y, which are bound to objects of type C. Next, the same variables ?p1, ..., ?pn are used to bind these objects along properties p1, ..., pn. The Filter expression then asserts that variables ?x and ?y bind to different objects. Clearly, the body computes exactly all pairs of violating objects.

5.2.2 Foreign Key Constraints

Next, we encode the violation of foreign key constraints. A foreign key constraint FK(C,[p1,...,pn],D,[q1,...,qn]) specifies that the attributes [p1,...,pn] of objects of type C refer to an existing key [q1,...,qn] over class D. According to its semantics, a foreign keys is violated if (1) there is an object of class C that does not reference an object of class D through p1,...,pn, or if (2) the referenced properties do not constitute a key. The check for condition (2) has been discussed before, so we restrict ourselves to the check for condition (1).

```
ASK {
  ?x rdf:type C; p1 ?p1; [...]; pn ?pn.
  OPTIONAL {
    ?y rdf:type D; q1 ?p1; [...]; qn ?pn.
  } FILTER (!bound(?y))
}
```

As required, the query returns *yes* if there is an object $?x$ of type C for which no object $?y$ of type D exists s.t. $?x$ and $?y$ bind to the same nodes through properties p1, ..., pn, and q1, ..., qn, respectively. Only if, for an object of type C, no associated object of type D exists, variable $?y$ is unbound and the respective mapping will pass through the filter.

5.2.3 Cardinality, Functionality, and Totality

When checking for max-cardinality constraints of the form Max(C,n,p), we try to witness an object of type C with at least $n+1$ p-properties. To this end, we introduce $n+1$ "witness" variables ?p1, ..., ?pn+1 and try to bind them to distinct-valued p-labeled properties. The shortcut

$$\texttt{allDist([?p1,...,?pn])} \overset{def}{:=} \bigwedge_{1\leq i\leq n}(\bigwedge_{i<j\leq n}\texttt{?pi!=?pj})$$

enforces that variables ?p1, ..., ?pn are pairwise distinct. Violation of max-cardinality then is checked as follows.

```
ASK {
  ?x rdf:type C.
  ?x p ?p1; [...]; p ?pn+1.
  FILTER (allDist([?p1,...,?pn+1]))
}
```

To check min-cardinality constraints Min(C,n,p), we first compute all non-violating objects, i.e. those objects with at least n distinct p-property labels. In a second step, we "subtract" these non-violating objects from the set of all objects. Clearly, this strategy returns exactly the violating objects. We implement the approach as follows.

```
ASK {
  ?x rdf:type C.
  OPTIONAL {
    ?y rdf:type C.
    ?y p ?p1; [...]; p ?pn.
    FILTER (allDist(?p1,...,?pn) && ?x=?y)
  } FILTER (!bound(?y))
}
```

The part inside the OPTIONAL computes all constraint-satisfying objects of type C. This result is then negated under closed-world assumption, using a combination of operators OPTIONAL, FILTER, and (not) BOUND.

It is easy to see that both functionality and totality constraints are special cases of cardinality constraints. In particular, a functionality constraint Func(C,p) is equivalent to the cardinality constraint Max(C,1,p). A totality constraint $Total$(C,p) can be expressed by the conjunction of the cardinality constraints Max(C,1,p) and Min(C,1,p).

5.2.4 SubProperty-Chain Constraints

A subproperty-chain constraint $SubPChain$(C,p1,...,pn,q) enforces that, for each object o of type C, if there is a chain of properties p1,...,pn starting from o, then this chain always references a node that is also directly referenced via property q of o. Violation can be checked as follows.

```
ASK {
  ?x rdf:class C; p1 ?p1.
  ?p1 p2 ?p2.
  ?p2 p3 ?p3.
  [...]
  ?pn-1  pn ?pn.
  OPTIONAL { ?x q ?q. FILTER (?pn=?q) }
  FILTER (!bound(?q))
}
```

The outer statement binds variable $?x$ to RDF nodes of type C, and variables $?p1$ through $?pn$ along the property chain. Note that the outer part selects only those nodes for which such a property chain is defined, i.e. exactly those that may violate the constraint. For the latter, $?pn$ is bound to the node that, according to the constraint, must be determined by the q-property of $?x$. Then, inside the OPTIONAL clause, the fresh variable $?q$ is bound to the node referenced by property q if and only if this node is identical to the node ?pn has been bound to. The outer Filter expression finally asserts that exactly those result mappings, for which such a binding is not possible, appear in the result. Clearly, the result contains exactly the constraint-violating mappings.

5.2.5 Singleton Constraints

A singleton constraint $Single$(C) enforces that there is exactly one object of class C. In the following we assume that the RDF graph contains at least one triple. Otherwise, the constraint is trivially violated.[3] The singleton constraint vi-

[3] We claim that, in this case, a constraint violation check cannot be implemented.

olation check then is implemented as follows.

```
ASK {
  { ?x1 rdf:type C.
    ?x2 rdf:type C.
    FILTER (?x1!=?x2) }
  UNION
  { ?x1 ?y ?z.
    OPTIONAL { ?x2 rdf:type C. }
    FILTER (!bound(?x2))       }
}
```

Here, the first part of Union returns the non-empty result iff there is more than one object of type C, while the second returns the non-empty result iff there is no object of type C. Note that the second part fails on the empty graph.

5.2.6 Anti-key Constraints

An anti-keys of the form *AntiKey*(C,[p1,. . . ,pn]) states that properties p1, . . . , pn do *not* constitute a key for class C. While a "positive" key is satisfied if no two distinct objects with identical values for properties p1 through pn exist, an anti-key enforces the existence of at least one violating pair.

We claim that it is impossible to write a SPARQL query that returns *yes* exactly if the anti-key is violated. However, from the semantics of anti-keys it is clear that the violation check query of the related key constraint *Key*(C,[p1,. . . ,pn]) returns *yes* exactly if the anti-key holds, and *no* otherwise.

Although one could easily invert our definition of violation check queries for this special type of constraints, the combined check for violation of anti-key constraints and other constraint types within a single query, as required for checking full schema satisfaction, is impossible. Motivated by this observation, we next propose a straightforward language extension that empowers SPARQL to deal with this problem.

5.3 Wrapping Up

5.3.1 SPARQL and CHECK

It would be particularly nice to have a reserved construct for checking constraints, akin to the SQL Check clause. We propose to extend SPARQL by new clauses PosC ("Positive Constraint") for encoding "positive" constraints, and NegC ("Negative Constraint") for encoding "negative" constraints, such as anti-keys. The semantics of these clauses is defined as follows. PosC returns *yes* exactly if its body returns the empty result, while NegC returns *yes* exactly if its body returns the non-empty result, just like Ask.

We emphasize that, although anti-keys (the only negative constraints in this work) might rarely occur in practice, NegC is required for encoding other user-defined, negative constraints. In particular, the clause is essential to express that certain relationships between objects do *not* hold.

In addition to the clauses, we finally introduce a new query form called Check, which must consist of a (non-empty) list of PosC- and NegC-clauses. We also restrict PosC- and NegC-clauses to occur only inside lists of the new query form. The Check form then simply computes the logical conjunction of all clauses inside its list, i.e. returns *yes* if and only if all PosC and NegC clauses do so.

The user then might encode positive constraints in a PosC clause, i.e. in form of an expression that returns the empty result exactly if the constraint holds. Negative constraints are represented by NegC clauses containing expressions that return the non-empty result exactly if the constraint holds. Clearly, by merging all PosC and NegC constraint clauses into a single Check query, we obtain a query that returns *yes* iff all specified constraints are satisfied. Using this novel construct, we can easily specify a single SPARQL query that checks whether a constrained schema (which may consist of both positive and negative constraints) is satisfied.

5.3.2 Complexity of Constraint Checking

We have shown that SPARQL queries can implement constraint checks. Hence, the SPARQL semantics implies an upper bound for the complexity of constraint checking. It is known that SPARQL is PSpace-complete [24]. However, it is reasonable to assume that the number of constraints is considerably smaller than the size of the data, and that the size of constraints is limited. The complexity of checking a fixed number of constraints then is given by the complexity of evaluating fixed-size queries. This complexity, also called data-complexity, is in LogSpace for SPARQL [24].

LogSpace is known to be contained in PTime, thus we conclude that evaluation of fixed-size SPARQL queries can be realized efficiently. Note that LogSpace is exactly the data complexity of First-Order Logic, which is strongly related to Relational Algebra and SQL.

6. EXPLOITING CONSTRAINTS

We now turn towards a discussion of the practical benefits of constrained RDF vocabularies. In particular, we highlight different properties of constraints that might be exploited for semantic query optimization (SQO). Although SPARQL is the language of our choice, similar optimization approaches may be equally useful for other query languages over RDF. While an exhaustive discussion of SQO for SPARQL over constrained RDF instances is beyond the scope of this paper, our intention here is to demonstrate that constraints over RDF, just like constraints in relational databases, indeed offer various resources for semantic query optimization.

We consider the scenario of teachers, students, and courses that has been introduced in Section 2. Recall that the following RDF constraints were defined (cf. Figure 3).

Keys

Class	Attributes
Teachers	[name]
Courses	[taught_by]
Students	[matric]
Particip.	[c_id,s_id]

Foreign Keys

Class	Attributes	Referenced Key
Courses	[taught_by]	Teachers
Particip.	[c_id]	Courses
Particip.	[s_id]	Students

In addition, let us assume that the name property of class Students is (exactly) single-valued, i.e. *Total*(Student,name) holds. As argued in Section 3, this constraint might result from a NOT NULL constraint for the name attribute in the relational database. Since both keys and foreign keys in the relational database are restricted to be NOT NULL, we also have *Total*(C,p) for each key and foreign key property p over its class C.[4] Recalling that NULL values in the relational database are ignored in the RDF graph, we also know that the remaining properties q have either cardinality one or zero; for those, we derive constraints of the form *Func*(C,q).

We now discuss optimization approaches for the following SPARQL query, assuming that the constraints above hold.

```
SELECT ?teachername ?coursename ?studentname
WHERE {
  ?course rdf:type Courses;
          taught_by ?teachername;
          name ?coursename.
  ?participant rdf:type Participants;
               c_id ?teachername;
               s_id ?studentmatric.
  ?teacher rdf:type Teachers;
           name ?teachername.
  OPTIONAL {
    ?student rdf:type Students;
             matric ?studentmatric;
             name ?studentname.
  }
}
```

The SPARQL operator "." is the equivalent of the relational Join operator, while operator Optional by idea is very similar to the relational Left Outer Join [24]. More precisely, the Optional operator joins its inner expression with the outer one, thereby retaining outer result mappings for which no join partner exists. Note that ";" is not an operator but only a syntactical convention, i.e. each block denotes a sequence of triple patterns with the same subject, connected through operator ".". Hence, the blocks in the query only function as logical arrangements of patterns.

Stage 1: Operator replacement. We now show that the Optional operator can be replaced by operator ".", thus basically we replace the more complex Left Outer Join by a standard Join.[5] As in relational databases, this clears the way for other optimizations, such as join reordering.

Outside the Optional clause, three (logical) blocks introduce variables *?course*, *?participant*, and *?teacher*. These blocks are interconnected through variable *?teachername*. Variable *?studentmatric* then connects the blocks outside with the block inside the Optional. From a relational perspective of view, we compute the left outer join between the outer and inner part on variable *?studentmatric*.

Basing on this observation, it is easy to see that operator Optional could be replaced by "." if, for *each* binding of variable *?studentmatric* in the outer part, there is join partner in the inner part, i.e. inside the Optional clause all variables can always be bound. Note that variable *?studentmatric* outside is bound to the value of the foreign key property s_id on top of Participants, which is also the value of the key property matric of a Students object. It follows that variable *?student* is always bound to the Students object identified by *?studentmatric*. From the constraint *Total*(Students,name), we conclude that also *?studentname* will inevitably be bound. Hence, the Optional clause is redundant, because a join is always possible. Consequently, the following query is equivalent under the given constraints.

```
SELECT ?teachername ?coursename ?studentname
WHERE {
  ?course rdf:type Courses;
          taught_by ?teachername;
          name ?coursename.
  ?participant rdf:type Participants;
               c_id ?teachername;
               s_id ?studentmatric.
  ?teacher rdf:type Teachers;
           name ?teachername.
  ?student rdf:type Students;
           matric ?studentmatric;
           name ?studentname.
}
```

Stage 2: Removing redundant joins. Another optimization approach arises from the fact that foreign key constraints, which link together objects of different types, may imply **rdf:type** relations on variables in the query. While in relational databases, type information are valuable for query optimization, in the RDF model these information always constitute an additional join, thus it is desirable to remove redundant triple patterns that enforce such relationships.

We observe that the second block of the query binds variable *?participant* to an object of type Participants. Now let us consider the foreign key property s_id in the second block. This property enforces that variable *?studentmatric* is always bound to a key value of a Students object. It is clear that, according to this key restriction for variable *?studentmatric*, variable *?student* in the fourth block will automatically be bound to the Students object identified by its designated key *?studentmatric*. Hence, the key and foreign key constraints enforce that the node, to which variable *?student* is bound, must be of type Students. Consequently, we can remove ?student rdf:type Students from the query. Arguing similarly, we also drop the type restrictions for *?teacher* and *?course*. We get the following.

```
SELECT ?teachername ?coursename ?studentname
WHERE {
  ?course taught_by ?teachername;
          name ?coursename.
  ?participant rdf:type Participants;
               c_id ?teachername;
               s_id ?studentmatric.
  ?teacher name ?teachername.
  ?student matric ?studentmatric;
           name ?studentname.
}
```

In principle, the elimination of redundant triple patterns comes along with a simplification of the query. In case that there are indices defined on property rdf:type of objects, it might not always be an advantage to eliminate triples that specify rdf:type relationships, as they might provide fast access paths to the triples. However, given that such indices are available, one could instead try to exploit the constraints to derive implied rdf:type relations on variables, in order to create efficient access paths. Therefore, in both situations constraints can be very useful for query optimization.

Next, we observe that, according to the second block in the query, variable *?teachername* is bound to the key of a Teachers object. Hence, the variable identifies exactly one Teachers object, which will be bound to variable *?teacher*.

[4] Note that this totality restriction is also captured by the semantics of key constraints.

[5] The SPARQL semantics defines operator Optional as a combination of Join and Set Minus, while operator And is put down to a Join [24].

But *?teacher* does not occur in the result, and consequently, we can simply remove `?teacher name ?teachername` from the query without affecting the output. This finally results in the following simplified query expression.

```
SELECT ?teachername ?coursename ?studentname
WHERE {
  ?course taught_by ?teachername;
          name ?coursename.
  ?participant rdf:type Participants;
               c_id ?teachername;
               s_id ?studentmatric.
  ?student matric ?studentmatric;
           name ?studentname.
}
```

Stage 3: Join Reordering. Triple patterns that are connected through operator "`.`" can be evaluated in arbitrary order. Changing the evaluation order of joins might, just like in relational databases, minimize the size of intermediate results and significantly speed up query evaluation [1]. Also in this regard, constraints offer valuable optimization resources. In particular, we can exploit totality, functionality, and general cardinality constraints.

While an exhaustive discussion of triple pattern reordering for the example scenario lies beyond the scope of this paper, we exemplarily discuss the execution order of the first two triple patterns `?course taught_by ?teachername` and `?course name ?coursename`. First recall that variable *?course* is always bound to `Courses` objects, and that we have two constraints *Total*(`Course`,`taught_by`), as well as *Func*(`Course`,`name`) defined on top of the properties mentioned in the triple patterns. According to these constraints, the result size of the first pattern is given by the number of `Courses` objects, while for the second pattern, the result size is smaller or equal to the number of `Courses` objects. In order to minimize the size of intermediate results, it is preferable to evaluate the patterns in reversed order.

To summarize, we have shown by example that constraint knowledge can significantly contribute to different semantic query optimization approaches for SPARQL.

7. RELATED WORK

Relational Databases and RDF. Relational databases are typically used in the context of RDF as a back end to store the RDF-statements, e.g. in Jena [14] and Sesame [28]. The underlying, very basic idea common to these approaches is to save all RDF triples in a relational table with schema (`subject`,`predicate`,`object`), on top of which SQL can be used to access (parts of) RDF triple patterns efficiently.

It has repeatedly been observed that mappings from the relational to the RDF data model are important to bring forward the vision of the Semantic Web [2, 35, 4, 31]. For instance, in [2] Tim Berners-Lee emphasizes the role of such mappings and discusses relations between both data models. With the same focus, the W3C has proposed universal approaches to define n-ary relations on the Semantic Web [35]. Bizer [4] presents a language for the specification of mapping rules. These ideas are implemented in the D2RQ system [5]. Rather than materializing translated RDF graphs, user-defined mapping rules imply a virtual RDF graph over the relational database. The system finally offers support for SPARQL queries on top of the virtual graph. A similar approach, called RDF views, is implemented in the Virtuoso system [6]. However, all these approaches mostly disregard the issue of mapping relational key and foreign key constraints. Moreover, the focus of these systems is on complex, user-defined mappings, while we are interested in an automated, generic translation of relational databases.

Constraints. Constraints for the Semantic Web have been studied before in different contexts of description logics (cf. [9, 18, 36]), respectively OWL-DL [22]. However, having a different focus, virtually all existing approaches model constraints as axioms. The integration of these axioms into inference systems then forms the basis for deriving constraint-satisfying models, given that such models exist. Thus, unlike constraints in the relational context, axioms do not immediately restrict the state space of the database. In particular, with this approach it is not explicitly possible to check whether a model satisfies a set of constraints. This fundamental difference has recently been clarified in [20], and [19, 8] present possible ways of integration. However, all these approaches differ from our approach, as they emphasize the description logic point of view. In contrast, we focus on model checking, and consequently constraints in form of axioms are not suited for our purpose.

In the past, semantic query optimization techniques on top of constraints has been studied extensively for relational databases, and also for deductive databases like Datalog (e.g. [15, 29, 10]). Resulting approaches to constraint-based optimization have found broad acceptance and, nowadays, various query engines exploit such techniques. However, the optimization of SPARQL under constraints has not been proposed before, last but not least because an approach for expressing constraints in RDF was missing up-to-date.

The theoretical results in Section 4 are related to work on the incorporation of key constraints into description logics [9, 18, 36]. Functional dependencies and unary foreign key constraints for description logics have already been studied in [36] (unary foreign key constraints are called "inverse features" there). Our undecidability result is of different nature as we do not use union of concepts. The undecidability proof in [9], on the other hand, uses negation and union of concepts. We claim that these features can not be expressed in our framework.

8. CONCLUSION AND FUTURE WORK

We have proposed to extend RDF by typical constraints from relational databases, such as primary and foreign key restrictions. Constrained RDF offers support for the translation of relational databases into RDF without losing vital information encoded in general restrictions and key constraints. The benefit of our novel approach is twofold. First, in maintaining original constraints from the relational source database we preserve important information that are indispensable for asserting data consistency. Second, as shown by example, constraints offer valuable resources for semantic query optimization. In this respect, constraints may improve the performance of query engines on top of RDF data.

As a theoretical contribution, we have analyzed the satisfiability of constrained RDF vocabularies over different classes of constraints. Although, for combinations of arbitrary types of our constraints the satisfiability problem is undecidable, checking whether an RDF graph satisfies a set of constraints can be realized in LogSpace. We also have discussed relationships to the description logic $\mathcal{ALCHIQ}$, to indicate a decidable class of constraints for our framework.

In order to underline the practicability of our approach, we have demonstrated that the SPARQL query language can deal with constraints in different ways. On the one hand, SPARQL can easily be used to extract constraints defined on top an RDF schema, since our approach naturally embeds constraints into the RDF data definition using a fixed RDF vocabulary. On the other hand, SPARQL can be used for checking RDF graphs against all types of constraints.

Our novel ideas open up a broad area of interesting research topics. First, the study of other types of constraints for RDF might be worth being considered, e.g. constraints in the context of XML data. Second, efficient algorithms for checking instances against constraints might be of practical interest. Finally, an algorithmic approach to semantic query optimization under constrained RDF should be developed, in order to speed up query engines in the Semantic Web.

9. REFERENCES

[1] C. K. Abraham Bernstein and M. Stocker. OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation. Technical Report, University of Zurich, 2007.

[2] T. Berners-Lee. Relational Databases on the Semantic Web. http://www.w3.org/DesignIssues/RDB-RDF.html.

[3] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. Scientific American, May 2001.

[4] C. Bizer. D2R MAP – A Database to RDF Mapping Language. In *WWW (Posters)*, 2003.

[5] C. Bizer, R. Cyganiak, J. Garbers, and O. Maresch. D2RQ: Treating Non-RDF Relational Databases as Virtual RDF Graphs. User Manual and Language Specification.

[6] C. Blakeley. Mapping Relational Data to RDF with Virtuoso's RDF Views, 2007. OpenLink Software.

[7] A. Borgida and G. E. Weddell. Adding Uniqueness Constraints to Description Logics (Preliminary Report). In *Deductive and Object-Oriented Databases*, pages 85–102, 1997.

[8] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Can OWL model football leagues? In *OWLED-07*, 2007.

[9] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Identification Constraints and Functional Dependencies in Description Logics. In *IJCAI*, pages 155–160, 2001.

[10] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based Approach to Semantic Query Optimization. *ACM Transaction on Database Systems*, 15(2):162–207, 1990.

[11] D. Dinh-Khac. Two Types of Key Constraints in Description Logics with Concrete Domains. Master thesis, TU Dresden, 2006.

[12] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *ACM*, 49(3):368–406, 2002.

[13] ARQ SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/.

[14] Jena2 Database Interface – Release Notes. http://jena.sourceforge.net/DB/index.html.

[15] J. J. King. QUIST: A System for Semantic Query Optimization in Relational Databases. *Distributed systems, Vol. II*, pages 287–294, 1986.

[16] G. Lausen. Relational Databases in RDF. In *Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases.*, 2007. To appear.

[17] C. Lutz, C. Areces, I. Horrocks, and U. Sattler. Keys, Nominals, and Concrete Domains. In *IJCAI*, pages 349–354, 2003.

[18] C. Lutz and M. Milicic. Description Logics with Concrete Domains and Functional Dependencies. In *ECAI*, pages 378–382, 2004.

[19] B. Motik, I. Horrocks, and U. Sattler. Adding Integrity Constraints to OWL. In *OWLED-07*, 2007.

[20] B. Motik, I. Horrocks, and U. Sattler. Bridging the Gap Between OWL and Relational Databases. In *WWW*, pages 807–816, 2007.

[21] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal Deductive Systems for RDF. *The Semantic Web: Research and Applications*, pages 53–67, 2007.

[22] OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref/.

[23] OWL 1.1 Web Ontology Language. http://www.webont.org/owl/1.1/.

[24] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPAQRL. In *CoRR Technical Report cs.DB/0605124*, 2006.

[25] Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/. W3C Recommendation, February 10, 2004.

[26] RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/. W3C Recommendation, February 10, 2004.

[27] RDF Semantics. http://www.w3.org/TR/rdf-mt/. W3C Recommendation, February 10, 2004.

[28] openRDF.org – home of Sesame. http://www.openrdf.org/documentation.jsp.

[29] S. T. Shenoy and Z. M. Ozsoyoglu. A System for Semantic Query Optimization. In *SIGMOD*, pages 181–195, 1987.

[30] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/. W3C Proposed Recommendation, November 12, 2007.

[31] N. Stojanovic, L. Stojanovic, and R. Volz. A reverse engineering approach for migrating data-intensive web sites to the Semantic Web. In *IFIP 17th World Computer Congress*, pages 141–154, 2002.

[32] S. Tobies. Complexity Results and Practical Algorithms for Logics in Knowledge Representation. PhD thesis, RWTH Aachen, Germany, 2001.

[33] D. Toman and G. E. Weddell. On Path-functional Dependencies as First-class Citizens in Description Logics. In *Description Logics*, 2005.

[34] D. Toman and G. E. Weddell. On the Interaction between Inverse Features and Path-functional Dependencies in Description Logics. In *IJCAI*, pages 603–608, 2005.

[35] Defining n-ary Relations on the Semantic Web, 2006. http://www.w3.org/TR/swbp-n-aryRelations/.

[36] G. E. Weddell. A Theory of Functional Dependencies for Object-Oriented Data Models. In *DOOD*, pages 165–184, 1989.

APPENDIX

A. PROOF OF THEOREM 1

Due to Lemma 3.2 in [12] the implication problem for keys by keys and foreign keys is undecidable, thus it suffices to show that its complement can be reduced to our problem here. Therefore, take an arbitrary relational schema $\mathcal{R} = (R_1, ..., R_n)$ and Σ a set of keys and foreign keys over $\mathcal{R}$ and $\varphi = R[X] \rightarrow R$ a key, where $X = A_1...A_k$ and $\{B_1, ..., B_l\} = \mathit{Att}(R) \backslash \{A_1, ..., A_k\}$. Note that $R = R_{i_0}$ for some $i_0 \in [n]$. We encode $\mathcal{R}$ into a constrained RDF vocabulary $\mathcal{V}_{\mathcal{R}} = ((N_C, N_P), \mathcal{C})$ such that

$$\text{there exists } I \models \mathcal{R} \text{ satisfying } \Sigma \wedge \neg\varphi \Leftrightarrow \mathcal{V}_{\mathcal{R}} \text{ is satisfiable.}$$

This gives us the desired reduction from the implication problem for keys by keys and foreign keys to our satisfiability problem.

$\mathcal{V}_{\mathcal{R}}$ has the following components.

$$\begin{aligned}
N_C &= \{r, R', \mathit{single}, R_1, ..., R_n\} \\
N_P &= \textstyle\bigcup_{i=1}^{n} \mathit{Att}(R_i) \,\dot\cup\, \{m, m'\} \\
\mathcal{C} &= \{\mathit{PropR}(m, R'), \mathit{SubC}(R', R), \mathit{PropR}(m', r)\} \\
&\cup \{\ \mathit{Key}(C, [B]) \mid C[B] \rightarrow C \in \Sigma\ \} \\
&\cup \{\ \mathit{FK}(C, [B], D, [B']) \mid C[B] \subseteq D[B'] \in \Sigma\ \} \\
&\cup \{\mathit{Key}(R', [B_1, ..., B_l]), \mathit{FK}(R', [X], \mathit{single}, [X])\} \\
&\cup \{\mathit{Single}(\mathit{single}), \mathit{Min}(r, 2, m)\} \\
&\cup \{\ \mathit{Max}(\mathit{single}, 1, R), \mathit{Min}(\mathit{single}, 1, R) \mid R \in X\ \} \\
&\cup \{\ \mathit{Min}(C, 1, R) \mid C \in \mathcal{R}, R \in \mathit{Att}(C)\ \} \\
&\cup \{\ \mathit{Min}(C, 1, m') \mid C \in N_C\ \} \\
&\cup \{\ \mathit{Max}(C, 1, R) \mid C \in \mathcal{R}, R \in \mathit{Att}(C)\ \}
\end{aligned}$$

Assume that $I \models \bigwedge \Sigma \wedge \neg\varphi$. We incrementally construct a model $\mathcal{I}$ of $\mathcal{V}_{\mathcal{R}}$. Choose $i \in [n]$ and $\mu \in I_i$. Add μ to $R_i^{\mathcal{I}_C}$ and $(\mu, \mu(A))$ to $A^{\mathcal{I}_P}$ for all $A \in \mathit{Att}(R_i)$. Furthermore, we add the tuple (μ, spy) to $m^{\mathcal{I}_P}$, in order to later ensure the violation of φ.

Choose $\mu_1, \mu_2 \in I_{i_0}$ such that μ_1 and μ_2 witness the violation of φ, thus they agree on their values of $A_1, ..., A_k$ but disagree on some value of some other attribute of R. Add $(\mathit{spy}, \mu_1), (\mathit{spy}, \mu_2)$ to $m^{\mathcal{I}_P}$, spy to $r^{\mathcal{I}_C}$ and μ_1, μ_2 to $R'^{\mathcal{I}_C}$.

Finally, for any object o created so far, we also add (o, spy) to $m'^{\mathcal{I}_P}$. The resulting interpretation is a model of $\mathcal{V}_{\mathcal{R}}$.

Conversely, assume that $\mathcal{I} \models \mathcal{V}_{\mathcal{R}}$. We incrementally construct a model I of $\Sigma \wedge \neg\varphi$. Choose some $B \in N_C$ such that $B^{\mathcal{I}_C} \neq \emptyset$ and $x \in B^{\mathcal{I}_C}$. Then, there exists $\mathit{spy} \in r^{\mathcal{I}_C}$ such that $(x, \mathit{spy}) \in m'^{\mathcal{I}_P}$. Furthermore, there exist $\mu_1, \mu_2 \in \Delta_{\mathcal{I}}$ such that $\mu_1 \neq \mu_2$, $(\mathit{spy}, \mu_1), (\mathit{spy}, \mu_2) \in m^{\mathcal{I}_P}$ and $\mu_1, \mu_2 \in R'^{\mathcal{I}_C}$. This shows that the following construction is not trivial.

For any $i \in [n]$ and $\mu \in R_i^{\mathcal{I}_C}$ do the following construction. For all $A \in \mathit{Att}(R_i)$ there exists $b_A \in \Delta_{\mathcal{I}}$ such that $(\mu, b_A) \in A^{\mathcal{I}_P}$. Add $\{\ \mu[A \mapsto b_A] \mid A \in \mathit{Att}(R_i)\ \}$ to I_i.

We show that $I := (I_1, ..., I_n) \models \Sigma \wedge \neg\varphi$. $I \models \Sigma$ is satisfied due the definitions of keys and foreign keys in our RDF framework. It remains to show that φ is violated. The objects μ_1, μ_2 are suitable candidates for this. As they are members of the class R' and $\mathit{Key}(R', [B_1, ..., B_l]) \in \mathcal{C}$ they disagree on at least one value among $\{B_1, ..., B_l\}$. We need to see why they agree on the values of the attributes from X. Choose $A \in X$ and a, $b \in \Delta_{\mathcal{I}}$ such that $(\mu_1, a), (\mu_2, b) \in A^{\mathcal{I}_P}$. This is possible because of the min-cardinality constraint $\mathit{Min}(R, 1, A)$. Choose $s \in \mathit{single}^{\mathcal{I}_C}$ and observe that $|\mathit{single}^{\mathcal{I}_C}| = 1$. Due to $\mathit{FK}(R', [X], \mathit{single}, [X])$, a and b are referenced by s via the property A. From $\mathit{Max}(\mathit{single}, 1, A)$ it follows that $a = b$. Thus, φ is violated, which concludes the proof. □

B. PROOF OF LEMMA 2

For simplicity of notation, we will identify the functions $.^{\mathcal{I}_C}$ and $.^{\mathcal{I}_P}$ in an interpretation with their graph.

Let $\varphi = \mathit{FK}(C, [R_1...R_n], D, [R'_1...R'_n])$ a foreign key[6] and $\mathcal{V}' := (N_C, N_P \,\dot\cup\, \{R_\varphi\})$. Π is given by

$$\begin{aligned}
&\{\mathit{FK}(C, [R_\varphi], D, [R_\varphi]), \mathit{Key}(D, [R'_1, ..., R'_n])\} \\
&\quad \cup \{\ \mathit{SubPChain}(C, R_\varphi, R_i, R_i) \mid i \in [n]\ \} \\
&\quad \cup \{\ \mathit{SubPChain}(D, R_\varphi, R_i, R'_i) \mid i \in [n]\ \}.
\end{aligned}$$

Let $\mathcal{I} = (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$ be an interpretation such that $\mathcal{I} \models \varphi$. For any $x \in C^{\mathcal{I}_C}$ choose one arbitrary but unique, $y_x \in D^{\mathcal{I}_C}$ such that $\forall i(\forall o_i \in \Delta_{\mathcal{I}} : (x, o_i) \in R_i^{\mathcal{I}_P}$, then $(y_x, o_i) \in R_i'^{\mathcal{I}_P})$, i.e. y_x serves as a witness for x that the foreign key is satisfied. Define $\mathcal{J}_{\mathcal{I}} := (\Delta_{\mathcal{I}} \,\dot\cup\, O, \Delta_D, .^{\mathcal{I}_C}, .^{\mathit{Prop}})$, where $O := \{\ o_x \mid x \in C^{\mathcal{I}_C}\ \}$ and

$$\begin{aligned}
.^{\mathit{Prop}} := .^{\mathcal{I}_P} &\cup \{\ (R_\varphi, x, o_x) \mid x \in C^{\mathcal{I}_C}\ \} \\
&\cup \{\ (R_\varphi, y_x, o_x) \mid x \in C^{\mathcal{I}_C}\ \} \\
&\cup \{\ (R_i, o_x, a) \mid i \in [n], x \in C^{\mathcal{I}_C}, (x, a) \in R_i^{\mathcal{I}_P}\ \}.
\end{aligned}$$

It is standard to verify that $\mathcal{J}_{\mathcal{I}} \models \Pi$.

Let $\mathcal{I} = (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}_C}, .^{\mathcal{I}_P})$ be an interpretation such that $\mathcal{I} \models \Pi$. $\mathcal{J}_{\mathcal{I}}$ is defined as $\mathcal{J}_{\mathcal{I}} := (\Delta_{\mathcal{I}}, \Delta_D, .^{\mathcal{I}_{Cl}}, .^{\mathcal{I}_{Prop}})$, where

$$\begin{aligned}
.^{\mathcal{I}_{Cl}} &:= \{\ (C, A) \in .^{\mathcal{I}_C} \mid C \in N_C, A \in \Delta_{\mathcal{I}}\ \}, \\
.^{\mathcal{I}_{Prop}} &:= \{\ (R, A, B) \in .^{\mathcal{I}_P} \mid R \in N_P, A, B \in \Delta_{\mathcal{I}}\ \}.
\end{aligned}$$

We show that $\mathcal{J}_{\mathcal{I}} \models \varphi$. Let $o \in C^{\mathcal{I}_C}$ and $o_1, ..., o_n \in \Delta_{\mathcal{I}}$ such that $\forall i \in [n] : (o, o_i) \in R_i^{\mathcal{I}_P}$. We need to show that $\exists o' \in D^{\mathcal{I}_C}$ such that $\forall i \in [n] : (o', o_i) \in R_i'^{\mathcal{I}_P}$. We know that $\mathcal{I} \models \mathit{SubPChain}(C, R_\varphi, R_i, R_i)$ and that the property R_i is total for all elements in class C for all $i \in [n]$. Due to $\mathcal{I} \models \mathit{FK}(C, [R_\varphi], D, [R_\varphi])$, R_φ is total for all elements in class C, too. So, $\exists o_{group} \in \Delta_{\mathcal{I}}$ such that $\forall i \in [n] : (o_{group}, o_i) \in R_i^{\mathcal{I}_P}$. Again, due to $\mathcal{I} \models \mathit{FK}(C, [R_\varphi], D, [R_\varphi])$ there exists some $o' \in D^{\mathcal{I}_C}$ such that $(o', o_{group}) \in R_\varphi'^{\mathcal{I}_P}$. As we know that $\mathcal{I} \models \mathit{SubPChain}(D, R_\varphi, R_i, R'_i)$ for $i \in [n]$ we have that $(o', o_i) \in R_i'^{\mathcal{I}_P}$. Thus, $\mathcal{J}_{\mathcal{I}} \models \varphi$, which concludes the proof. □

C. REDUCTION TO ALCHIQ

Our framework	$\mathcal{ALCHIQ}$ construct
$\mathit{SubC}(C, D)$	$C \sqsubseteq D$
$\mathit{SubP}(R, S)$	$R \sqsubseteq S$
$\mathit{PropD}(R, C)$	$\exists R.\top \sqsubseteq C$
$\mathit{PropR}(R, C)$	$\exists R^-.\top \sqsubseteq C$
$\mathit{Min}(C, n, R)$	$C \sqsubseteq \geq nR$
$\mathit{Max}(C, n, R)$	$C \sqsubseteq \leq nR$
$\mathit{FK}(C, [R], D, [S])$	$\exists R^-.C \sqsubseteq \exists S^-.D$
$\mathit{Key}(C, [R])$	$\exists R^-.C \sqsubseteq \leq 1R^-.C$

The translation of $\mathit{Key}(C, [R])$ was inspired by [7].

[6] Without loss of generality we can assume that $n \geq 2$, otherwise we are done.