UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Informatik

FB 4
Informatik

Networked RDF Graphs

Simon Schenk
Steffen Staab

Nr. 3/2007

Arbeitsberichte aus dem Fachbereich Informatik

Universität Koblenz-Landau • Campus Koblenz • Universitätsstraße 1 • 56070 Koblenz
Telefon +49 261 287-0 • http://www.uni-koblenz.de

Arbeitsberichte des Fachbereichs Informatik

I**SSN:** 1864-0346

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Paulus

Die Professoren des Fachbereichs:
Prof. Dr. Bátori, Jun.-Prof. Dr. Beckert, Prof. Dr. Burkhardt, Prof. Dr. Diller, Prof. Dr. Ebert, Prof. Dr. Furbach, Prof. Dr. Grimm, Prof. Dr. Hampe, Prof. Dr. Harbusch, Jun.-Prof. Dr. Hass, Prof. Dr. Krause, Prof. Dr. Lautenbach, Prof. Dr. Müller, Prof. Dr. Oppermann, Prof. Dr. Paulus, Prof. Dr. Priese, Prof. Dr. Rosentahl, Prof. Dr. Schubert, Prof. Dr. Staab, Prof. Dr. Steigner, Prof. Dr. Troitzsch, Priv.-Doz. Dr. von Kortzfleisch, Prof. Dr. Walsh, Prof. Dr. Wimmer, Prof. Dr. Zöbel

Kontaktdaten der Verfasser

Simon Schenk, Steffen Staab
Institut für Informatik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: sschenk@uni-koblenz.de; staab@uni-koblenz.de

Networked RDF Graphs

Simon Schenk, Steffen Staab
Institute for Computer Science
University of Koblenz
http://isweb.uni-koblenz.de
{sschenk, staab}@uni-koblenz.de

ABSTRACT

Networked graphs are defined in this paper as a small syntactic extension of named graphs in RDF. They allow for the definition of a graph by explicitly listing triples as well as by SPARQL queries on one or multiple other graphs. By this extension it becomes possible to define a graph including a view onto other graphs and to define the meaning of a set of graphs by the way they reference each other. The semantics of networked graphs is defined by their mapping into logic programs. The expressiveness and computational complexity of networked graphs, varying by the set of constraints imposed on the underlying SPARQL queries, is investigated. We demonstrate the capabilities of networked graphs by a simple use case.

1. INTRODUCTION

The Resource Description Framework (RDF) is a language for representing semantic information in the World Wide Web (cf. [18]). Being such a language it allows for referencing of the described resources leading to *networking of resource descriptions*.

RDF does not allow for referencing and reusing of RDF graphs. It is not possible to describe how data in one graph is re-used in another graph. The only connection from one graph A to another graph B is by using an URI in graph A that links to some URI the definition of which happens to be found in graph B[1]. Thereby, RDF does not allow for the circumscription of the interest that a publisher of graph A takes in the data found in graph B.

A mechanism that will allow such circumscribing within RDF graphs may provide many valuable possibilities, such as:

1. *reuse of RDF graphs* enabling the *dynamic* copying of contents from one graph to the other;
2. *viewing RDF graphs* in a way that is defined by another RDF graph;
3. *dynamic networking of RDF graphs.* RDF graphs will constitute dataspaces that do not only sit next to each other, but the meaning they describe will come from their dynamic networking.

The interested reader may note that analogous mechanisms of focusing into Web documents that are viewed by people have existed for long: they are the XPath descriptions, the stylesheet languages like XSLT, the notions of frames and pagelets. We think it is very natural and useful to transfer some of the concepts that are useful for data viewed and understood by humans to corresponding mechanisms for the data part of the Web that is supposed to be viewed and understood by computers.

This paper is about such reusing, viewing and dynamic networking of RDF graphs. For this purpose, we combine two recent advancements in the RDF domain, *i.e. named graphs* as described in [5] and the query language *SPARQL* [21]. We come up with a small, but powerful syntactic extension of RDF that we call *networked RDF graphs*. We investigate its semantics by a mapping to logic programs and show its benefits with a simple use case.

In the following, we start with a description of this use case. We introduce the foundations on which we build in Section 3. We define the abstract structure of networked RDF graphs in Section 4. In Section 5, we describe their semantics by a mapping into logic programs. Then, we explore the computational complexity of the model in section 6. We describe a first implementation and related work in sections 7 and 8.

2. USE CASE

Our running example in this paper is defined by a simple use case on maintaining information at an academic institution in a completely decentralized fashion and on publishing this data in the Semantic Web.[2]

Our university consists of departments (e.g. 'Department of computer science') that have different institutes (e.g. 'IFI – Institute for computer science') and each institute has one or several labs (e.g. 'ISWeb – Information Systems and Semantic Web', or 'AGAS – Active Vision Lab'). Once networked graphs are established, each lab will maintain and publish its membership site in an RDF graph using different technical platforms (and maybe even different ontologies). Every lab has researchers, e.g. Steffen and Simon are researchers at ISWeb and Richard is a researcher at ISWeb, but also at AGAS. Saqib is also a researcher at ISWeb, but he is an extern receiving his salary from another organization. Hence, Steffen and Richard are employees of IFI, but Saqib is not. The administrative staff who is not assigned to a particular lab, e.g. Ute, is still a member of the institute, IFI.

One objective of this use case is the maintenance of employment and research relationships in graphs specific to

[1]The semantics of `rdfs:seeAlso` is very weak.

[2]The reader may note that structurally similar use cases exist in E-learning (cf. [4]) or in semantic multimedia annotation.

each group (e.g. ISWebGraph for ISWeb). Another objective is to consolidate this information *dynamically* at other levels of granularity, e.g. at the level of IFI or at the department or university levels. Hence, the graph describing employments with the institute (IFIGraph) should not copy information, as this would be quickly outdated. It should rather refer to the appropriate statements provided by the labs in their corresponding graphs as well as add new relationships (e.g. about Ute).

3. FOUNDATIONS

In this section, we briefly describe some of the foundations based on which we define networked graphs.

3.1 RDF

RDF is a graph based knowledge representation language. The nodes in a graph are URIRefs, blank nodes or literals. Arcs between the nodes represent their relationships. The arcs are labeled with URIRefs, representing the property that holds between the two nodes. We simplify the RDF graph model (cf. [18]) here slightly in order to come up with a more concise formal characterization:

DEFINITION 1. *Let U be the set of URIRefs, L the set of RDF Literals and B the set of Blank Nodes as defined in [18]. U, L and B are pairwise disjoint. Let $R = U \cup L \cup B$. A statement is a triple in $R \times U \times R$. If $S = (s, p, o)$ is a statement, s is called the subject, p the predicate and o the object of S.*

DEFINITION 2. *A graph G is a set of statements. For every two graphs G_1 and G_2 the sets of blank nodes used in G_1 and in G_2 are disjoint.*

[18] defines a model theoretic semantics for RDF graphs based on inference rules which allow to derive new statements from a given RDF graph. They include, for example, transitivity of class membership. We refer the reader to [18] for a detailed description of RDF, which includes a definition of RDF and the RDF Schema language RDFS.

Parts of the ISWebGraph can be represented in RDF and noted in N3 as follows:

EXAMPLE 1.

```
{ u:Steffen u:worksAt u:ISWeb.
  u:Simon u:worksAt u:ISWeb.
  u:Saqib u:worksAt u:ISWeb.
  u:Saqib u:status u:externalResearcher.
  u:Richard u:worksAt u:ISWeb. }
```

3.2 Named Graphs

While the RDF recommendation does not allow refering to RDF graphs, named graphs introduced in [5] offer means to group a set of statements together into a graph and to refer to this graph using a URIRef. This way information about the graph can be expressed in RDF using its name as a subject or object:

DEFINITION 3. *A named graph is a pair (n, G) of a URIRef n and an RDF graph G.*

In our use case we refer to the graph of the institute of computer science by `u:IFIGraph`, to the graph of the administration by `u:IFIAdminGraph` and to the graph of the ISWeb lab by `u:ISWebGraph`. In N3, graphs are named by prepending them with a name, for example:

EXAMPLE 2.

```
u:IFIGraph { u:Steffen u:worksAt u:IFI.
  [...]
  u:Richard u:worksAt u:IFI.
  u:ISWeb u:workingGroupOf u:IFI. }

u:IFIAdminGraph { u:Ute u:worksAt u:IFI. [...]}

u:ISWebGraph { u:Steffen u:worksAt u:ISWeb.
  [...]
  u:Richard u:worksAt u:ISWeb. }
```

In addition, [5] contains a vocabulary to describe relationships between graphs like `subGraphOf`, but no machinery to detect or exploit such relationships — which is done in this paper.

3.3 SPARQL

SPARQL is a query language for RDF based on graph pattern matching, which is defined in [21]. In this paper we are only interested in SPARQL CONSTRUCT queries. A SPARQL CONSTRUCT query matches a graph pattern against one or more input graphs. The resulting variable bindings are embedded into a template description in order to generate new RDF data. In this paper we will use a normalised and restricted subset of SPARQL, which can express all SPARQL CONSTRUCT queries[3]. While [21] offers various ways to write complex graph patterns, we assume all complex patterns to be expressed as sequences of patterns of single statements.

DEFINITION 4. *A SPARQL CONSTRUCT query has the form*

```
CONSTRUCT <CONSTRUCT pattern>
[FROM NAMED <graph>]+
WHERE <WHERE pattern>
```

where `<CONSTRUCT pattern>` is a graph pattern and `<WHERE pattern>` is a filtered graph pattern as defined in definitions 18 and 17.

Definitions 17 and 18 can be found in appendix A.

A query is evaluated by matching the WHERE pattern against the graphs declared using FROM NAMED and computing bindings for the variables used in the WHERE pattern. The filters in the WHERE pattern are applied to every computed binding. If any filter returns *false* or *error*, the binding will not be used. A filter returns error, if it is invoked on parameters of the wrong type, or one of its parameters is *error*. In the following, two aspects of error handling in filters will be important: *! error* results in *error* and the BOUND filter is the only filter, which may derive a return value other than error from an unbound variable as a parameter.

The CONSTRUCT pattern may only use variables introduced by the WHERE pattern[4]. The result of a query is

[3]We do not allow to define the default graph using the "FROM" declaration. We will assume the default graph to be a Networked Graph, which can be referred to using its name and "FROM NAMED". Such a Networked Graph can be easily derived from a FROM NAMED declaration.

[4]In fact, arbitrary variables can be used, but only if all variables in a statement in the CONSTRUCT pattern have been bound in the WHERE pattern, this statement will be included in the result. Thus newly introduced variables in the CONSTRUCT pattern would not contribute to the result.

obtained by substituting all possible variable bindings for the variables in the CONSTRUCT pattern and building the union of these graphs.

4. NETWORKED GRAPHS

We now list requirements for an extension of the RDF toolset resulting from the use case. We define this extension, named Networked Graphs, and describe a language to define Networked Graphs.

4.1 Requirements

From the use case we derive the following requirements:

Nesting of named graphs. We need to import all statements from one graph to another one in order to allow for easy structuring of RDF data. In the use case all administrative personnel shall be described in one graph IFIAdminGraph, which is to be included in the institute's graph IFIGraph.

Views on RDF graphs. Views allow for reuse of parts of RDF data in different contexts. The IFIGraph shall only include those parts of the ISWebGraph, which do not describe external researchers.

In a semantic web setting, which can hardly be controlled, circular view definitions must be possible, as they can not be avoided.

Exchange of graph definitions. For the IFIGraph, only a description of how to derive its content from other graphs shall be stored, not its materialisation.

Compatibility with earlier RDF extensions. The machinery for easy formulation of RDF graphs shall be preserved and extended instead of being replaced.

Referencing of graphs. Referencing of graphs is a neccessary prerequisite for defining views and imports.

4.2 Abstract Syntax

The last requirement can already be fulfilled using named graphs, so we base the definition of Networked Graphs on named graphs and extend them by a view mechanism.

DEFINITION 5. *A Networked Graph is either*
a) a named graph $g = (n, G)$, or
b) a quadruple $g = (n, G, [g_1..g_n], v)$,
where n is a URIRef called the name of the Networked Graph, G is a graph, $[g_1, ..., g_n]$ is a list of Networked Graphs and v is a mapping from a list of Networked Graphs to an RDF graph called the view definition of g.

We define a function *deref* to access the contents of a Networked Graph given the Networked Graph definition itself or its name. *deref* carries out dereferencing of Networked Graphs as a logical operation, i.e. given a Networked Graph compute its content. This logical operation is opposed to dereferencing as an addressing operation, i.e. given a URIRef, download the corresponding file, because in contrast to named graphs, Networked Graphs are not neccessarily explicitly listed in a single file. In Section 5 we will describe, how *deref* can be computed for a set of Networked Graphs.

DEFINITION 6. *Let $g = (n, G, [g_1 \dots g_n], v)$ or $g = (n, G)$ be a Networked Graph.*
Then deref *is an overloaded function mapping from networked graphs and names of networked graphs into a set of statements, i.e. a graph:*

$$deref(g) = \begin{cases} G, \text{ if } g \text{ is a named graph} \\ G \cup v(g_1, ..., g_n) \text{ else.} \end{cases}$$

$$deref(n) = \begin{cases} G, \text{ if } g \text{ is a named graph} \\ G \cup v(g_1, ..., g_n) \text{ else.} \end{cases}$$

An example from our use case could be as follows:

EXAMPLE 3. *Let ISWebGraph and IFIAdminGraph be Networked Graphs. Then IFIGraph is a Networked Graph:*
IFIGraph = (`u:IFIGraph`,
{`u:ISWeb u:workingGroupOf u:IFI.`,
`u:IFI u:belongsTo u:CSDepartment.`},
[ISWebGraph, IFIAdminGraph], v), with
v(ISWebGraph, IFIAdminGraph) =
v_2(ISWebGraph)∪deref(IFIAdminGraph) and
v_2(ISWebGraph) = {`u:Steffen u:worksAt u:IFI.`
`[...] u:Richard u:worksAt u:IFI.` },

For reasons of simplicity, we will use the name of a Networked Graph interchangably with the Networked Graph itself. From the context it will be clear, whether the name or the Networked Graph is meant.

4.3 An RDF Language Extension for Networked Graphs

Apart from the name, we can write the complete definition of a Networked Graph in one SPARQL CONSTRUCT query, including all statements in G into the CONSTRUCT pattern, naming the list of Networked Graphs using FROM NAMED and encoding v in the WHERE and CONSTRUCT patterns. For example 3 we would have:

EXAMPLE 4.

```
CONSTRUCT {
  u:ISWeb u:workingGroupOf u:IFI.
  u:IFI u:belongsTo u:CSDepartment.
  ?s ?p ?o. ?person u:worksAt u:IFI. }
FROM NAMED u:ISWebGraph FROM NAMED u:IFIAdminGraph
WHERE { GRAPH u:IFIAdminGraph {?s ?p ?o.}.
  UNION GRAPH u:ISWebGraph {?person u:worksAt u:ISWeb}. }
```

Encoding RDF graphs in SPARQL construct queries, however, is not convenient. In order to be upwards compatible with named graphs, we use an encoding for Networked Graphs which is based on named graphs.

DEFINITION 7. *A Networked Graph c=(n,G,[c_1,...,c_n],v) is encoded in a named graph with name n. The contents of G are explicitly included in this named graph. The view definition is included in statements of the form:*
`n g:definedBy <query>`.[5]
where `<query>` is a literal containing a CONSTRUCT query. The datatype of `<query>` is `g:query`
We call such a statement a view definition statement.
We call the object literal of a view definition statement a subquery *of the Networked Graph definition. The view definition is the union of the subqueries.*

We use a special datatype for literals containing subqueries in order to recognise subqueries for the translation to logic programs described in chapter 5. Using a special datatype

[5]As namespace for the Networked Graph vocabulary we propose `http://isweb.uni-koblenz.de/ontologies/2006/11/ng#`

for queries is neccessary in order to discover query literals, even though they are not contained in a `definedBy` statement. Some Networked Graph could match such a literal in its view definition and compose a `definedBy` statement from it, which in turn must be included when computing the Networked Graph's content.

Resulting from this syntax, existing machinery for working with named graphs can be used for serialisation and exchange of Networked Graphs. Additionally, a non Networked Graph aware repository can still interpret the explicit statements in a Networked Graph, providing upwards compatibility.

We define a shorthand notation for a common subquery: The predicate `g:contains` is used as a shorthand for defining complete inclusion of all statements from the object graph of the statement into the subject graph. For example `u:IFIGraph g:contains u:IFIAdminGraph.` is a shorthand notation for `u:IFIGraph g:definedBy "CONSTRUCT {?s ?p ?o.} FROM NAMED u:IFIAdminGraph WHERE GRAPH u:IFIAdminGraph {?s ?p ?o.}".`

The following Networked Graph collects information required by the institute in our use case into the IFIGraph, refined from example 4 to importing only non-external researchers:

Example 5.

```
u:IFIGraph {
  u:ISWeb u:workingGroupOf u:IFI.
  [...]
  u:IFIGraph g:contains u:IFIAdminGraph.
  u:IFIGraph g:definedBy
    "CONSTRUCT { ?person u:worksAt u:IFI. }
     FROM NAMED u:ISWebGraph
     WHERE {
       GRAPH u:ISWebGraph {?person u:worksAt u:ISWeb.}.
       OPTIONAL {
          GRAPH u:ISWebGraph {?person u:status ?x.}.
          FILTER (?x = u:externalResearcher). } .
       FILTER (NOT BOUND(?x))."}. }
```

`definedBy` statements are intended to carry two meanings. First, there is the extensional meaning that the `definedBy` statements exists. Second, there is its intended intensional meaning, i.e. the evaluation of the view it defines. The second meaning will only be incurred if the graph in which a `definedBy` statement occurs (intensionally or extensionally) is identical with the subject of this statement.

To have a syntax exclusively based on RDF we could use an RDF vocabulary for expressing queries like the one proposed in [12], but that is out of scope of this paper.

5. SEMANTICS

Basically, the Networked Graph semantics is an extension of the SPARQL semantics to include nested and recursive queries. A Networked Graph, which contains only one statement describing a non-recursive query, maps directly to a SPARQL query. We define the semantics of Networked Graphs through a mapping of a set of SPARQL queries to a normal logic program. This also provides a logic translation of SPARQL. We apply the well founded semantics by Gelder et al. [26] to this program in order to deal with non-monotony in circular Networked Graph definitions.

5.1 Well Founded Semantics

The well founded semantics assigns a unique model to every normal logic program P using a three valued logic. A well founded model assigns to every atom a truth value of *true*, *false* or *unknown*. Atoms with an unknown truth values are those which do neither hold nor can be easily assumed not to hold. The model assigned to P under the well founded semantics is the minimal model of all such three valued models that fulfil P (cf. [26] for a detailed introduction of the well founded semantics).

As a result, we still have a unique model for programs, where two valued semantics would run into alternating fixed points or multiple models. This property will be very helpful for circular Networked Graph definitions.

Other possibilities to assign a semantics to Networked Graphs like stable model or first order semantics exist. The latter, however, will not coincide nicely with negation in SPARQL in the case of recursive Networked Graph definitions and the former does not determine a single, unique model without further constraints.

5.2 Filtering Statements with Negation

SPARQL focuses on filtering statements based on the availability of other statements. The following example query searches for all researchers of ISWeb who also happen to be external Researchers:

Example 6.

```
CONSTRUCT {?p u:worksAt u:IFI.} FROM NAMED u:ISWebGraph
WHERE {
  GRAPH u:ISWebGraph {?p u:worksAt u:ISWeb.}.
  GRAPH u:ISWebGraph {?q u:status u:externalResearcher.}.
  FILTER (?p = ?q). }.
```

The negation of such a kind of filter expression as used in example 6 cannot be used to select one statement based on the absence of another statement. Therefore, selecting all researchers of ISWeb who are not external researchers needs a rather involved check whether some optional variable remains unbound as shown in the following example.

Example 7.

```
CONSTRUCT {?p u:worksAt u:IFI.} FROM NAMED u:ISWebGraph
WHERE {
  GRAPH u:ISWebGraph {?p u:worksAt u:ISWeb.}.
  OPTIONAL { GRAPH u:ISWebGraph {?p u:status ?s.}.
             FILTER (?s = u:externalResearcher). }.
  FILTER (! BOUND(?s)). }
```

This consideration can be summarized as follows.

Definition 8. *A SPARQL query uses* BOUND negation, *if the following pattern occurs in the WHERE pattern:*

```
<patternx> OPTIONAL { <pattern> }.
<patterny> FILTER (! BOUND(?t)). <patternz>
```

where in angle brackets we have filtered graph patterns and `?t` is a variable introduced by `<pattern>`. `<patternx>`, `<patterny>` and `<patternz>` are parts of the query, which surround the BOUND negation.

Based on this definition we conclude:

Proposition 1. *Using BOUND negation in a filter is the only way to select statements in SPARQL based on the absence of other statements.* □

PROOF: Negation in SPARQL can not be expressed in graph patterns. However, in the presence of OPTIONAL graph patterns, some variables may be left unbound. Hence we can model negation by failure to search for non-existent statements: We need to formulate a filtered graph pattern matching the statements, which shall not exist, and then apply a FILTER to one or more of the variables introduced by this pattern. This filter needs to check whether the variable is unbound. This construct models negation by failure in SPARQL.

Apart from the bound filter, all filters will return an error, when invoked on an unbound variable. As *! error = error*, we can never obtain a *true* filter result from such a filter expression. Hence it will either trivially fail, or, if it is part of a disjunction, it will be irrelevant for the result. The truth value of *error || A* can only be *error* or true, only depending on the value of A. ■

Note that also in the presence of UNION some variables from one of the subpatterns of the UNION expression may be left unbound. In contrast to OPTIONAL patterns, this does not mean we failed to bind these variables, but that the query evaluation bound the variables in the other branch of the UNION.

5.3 Mapping to Logic Program

For the following translation from SPARQL to a logic program we assume the RDF/S extensions of all graphs to be materialised. We do not cover the RDF/S semantics here. It can be included by combining our work with the entailment rules in [18]. We briefly describe in appendix C how to modify our translation in definition 14 to also capture RDF/S semantics. Please note that when including RDF/S semantics, the results regarding computational complexity need to be revised to account for the complexity of RDFS inferencing.

We start to define the translation of SPARQL queries with a definition of a translation for filter expressions. We will use the following normal form for filter expressions:

DEFINITION 9. *A filter expression is in conjunctive normal form, if it is a conjunction of disjunctions of filter literals as defined in definition 17.*

EXAMPLE 8. *Let* $F_i(i \in \{1..n\}), G_j(j \in \{1..m\})$ *be filter literals. Then the following filter expression is in conjunctive normal form:*
$FILTER((F_1||...||F_n)\&\&(G_1||...||G_m))$

Analogously to boolean formulas every filter expression can be translated into conjunctive normal form.

LEMMA 1. *Every filter expression can be translated into an equivalent conjunctive normal form.* □

Normalization can be achieved comparable to normalization of boolean formulas by exploiting double negation, distribution and DeMorgan's laws. A proof that these laws hold can be found in appendix B.

Now we describe how to translate SPARQL queries into logic programs. First we define some auxiliary functions and data structures.

DEFINITION 10.

(1) Let c be a statement pattern
(2) Let A, B, P be filtered graph patterns.
(3) Let C be a graph pattern.
(4) Let X, Y be filter expressions.
(5) Let x, y, z be resources.
(6) Let u, v, w be variables of a logic program.
(7) Let g and r be names of Networked Graphs[6]*.*
(8) Let ext *be a mapping from a Networked Graph to its graph component, G.*
(9) Let s, p, o be mappings from a statement to its subject, predicate and object.
(10) Let varsInt *be a mapping from a filtered graph pattern to the set of variables* introduced by *this pattern.*
(11) Let resources *be a mapping from a SPARQL expression to the set of resources used in this expression.*
(12) Let literals *be a mapping from a SPARQL expression to the set of literals used in this expression.*
(13) Let variables *be a mapping from a SPARQL expression to the set of variables* used in *this expression.*
(14) Let filters *be a mapping from a SPARQL expression to the set of filter operators used in this expression.*

EXAMPLE 9. *Let g be* IFIGraph *and A the following part of the construct pattern in example 5:*

```
GRAPH u:ISWebGraph {?person u:status ?x.}.
FILTER (?x = u:externalResearcher).
```

Then
ext(*g*) = {

```
u:ISWeb u:workingGroupOf u:IFI.
[...]
u:IFIGraph g:contains u:IFIAdminGraph.
u:IFIGraph g:definedBy "CONSTRUCT [...]" },
```

but `u:Steffen u:worksAt u:ISWeb` ∉ ext(*g*).
varsInt(*A*) = {`?x`}, *opposed to*
variables(*A*) = {`?person, ?x`}.
resources(*A*) = {`u:IFIGraph, g:contains, u:IFIAdminGraph., g:definedBy`}.
filters(*A*) = {" = "}.

We will use a mapping function *m* which is a mapping from SPARQL expressions to parts of a logic program. We will successively extend *m* until we can translate all SPARQL CONSTRUCT queries. We start with the translation of filter expressions:

DEFINITION 11. *A filter expression X in conjunctive normal form is translated into a part of a logic program as follows:*

1. *Define a set of constants* Const *and a bijection res* : *resources*(*X*) ↔ *Const.*
2. *Define a set of variables* Var *and a bijection var: variables*(*X*) ↔ *Var.*
3. *Define a set of predicates* F *and a bijection f* : *filters*(*X*) ↔ *F. Map the BOUND filter operator to the predicate* bound.
4. *Let* $m = res \cup var \cup f$.

[6] *r* is the name of the graph resulting from a query, which is usually unknown. When the query is used in a `definedBy` statement, the name of the graph is known upfront.

5. *Translate filter definitions* $op(a_1, ..., a_i)$ *to* $m(op)(m(a_1), ..., m(a_o))$.

6. *Translate filter definitions* $!(op(a_1, ..., a_i))$ *to* $\neg m(op)(m(a_1), ..., m(a_o))$.

7. *Translate the locial connectives* $X||Y$ *to* $(m(X) \vee m(Y))$ *and* $X\&\&Y$ *to* $(m(X) \wedge m(Y))$.

EXAMPLE 10. *The filter expressions in example 7 translate to* $eq(s, externalResearcher)$ *and* $\neg bound(s)$*, assuming* w *is given as follows:*
$w(\text{"="}) = eq$.
$w(\text{"BOUND"}) = bound$.
$w(\text{"u:externalResearcher"}) = externalResearcher$.
$w(\text{"?s"}) = s$.

During this translation we have simplified the filter expression such that the third possible return value *error* is no longer used. This is possible, as filter expressions which evaluate to *error* are treated as if they evaluate to $false$ in SPARQL:

LEMMA 2. *Filter expressions in conjunctive normal form are evaluated equally in SPARQL and in first order logic after translation according to definition 11.* □

A proof can be derived easily from the truth tables for logical connectives given in [21], see appendix B.

THEOREM 1. *The translation of filter expressions is sound and complete.* □

PROOF: Soundness follows directly from proposition 2 and the fact that in two valued logic the mapping in definition 11 is purely syntactical. To prove completeness, consider that we can translate every filter expression, because definition 11 provides a mapping for every constructor from definition 17, and proposition 1 holds. ■

The filter translation is used in translating SPARQL construct queries.

DEFINITION 12. *An arbitrary SPARQL CONSTRUCT query* Q *can be translated to a first order predicate logic program* P *as follows:*

1. *Let* m' *be defined like* m *in definition 11, except that* Var*,* $Const$ *and* F *are derived from* Q *and the statements of all graphs in the dataset of* Q.

2. *Let* m *be given by the union of* m' *and the mapping rules described in table 1.*

3. *Translate all filter expressions of* Q *into conjunctive normal form.*

4. *For every blank node creation in the CONSTRUCT pattern, introduce a new skolem function* f_i *and add to* m *the tuple* $(bc, f_i(u, ..., w))$ *where bc is the SPARQL expression introducing the new blank node and* $u, .., w$ *are the variables in the translation of the CONSTRUCT pattern.*

5. *Compute* $P = m(Q)$.

6. *Apply Lloyd-Topor transformation*[7] *to* P.

Table 1: Mapping from SPARQL to a logic program

SPARQL Syntax	FOL Syntax
m(CONSTRUCT {C} WHERE P)	$\big[\forall c \in C$: "$t(m(r), m(s(c)),$ $m(p(c)), m(o(c)) \leftarrow m(P)$" $\big[\forall v \in variables(c)$: ", $bound(m(v))$"$\big]\big]$ "$bound(x) \leftarrow \neg isNull(x)$ " "$isNull(null) \leftarrow$" "$t(u, u, m(g : uses), v)$ $\leftarrow t(u, u, m(g : uses), w),$ $t(w, w, m(g : uses), v)$"
m(FROM NAMED g)	$\big[\forall c \in ext(g)$: "$t(m(g), m(s(c)),$ $m(p(c)), m(o(c))) \leftarrow''\big]$ "$t(m(r), m(r), m(g : uses), m(g))$ $\leftarrow$"
m({P})	"$(m(P))$"
m(GRAPH g {x y z.}.)	"$t(m(g), m(x), m(y), m(z))$"
m(A. B.)	"$((m(A)), (m(B))$"
m(A UNION B)	"$(((m(A))$ " $\big[\forall v \in varsInt(B) \backslash$ $varsInt(A)$: ", $isNull(m(v))$"$\big]$) " $\vee ((m(B))$" $\big[\forall v \in varsInt(A) \backslash$ $varsInt(B)$: ", $isNull(m(v))$"$\big]$))
m(A OPTIONAL B)	"$(m(A), (m(B) \vee$ $\big[\forall v \in varsInt(B)$: "$isNull(m(v)),$"$\big]$) if A uses BOUND negation "$\neg m(B))$", else *"true)"*
m(FILTER X)	see definition 17

Expressions are evaluated as follows:

1. Evaluate the mapping m for all constants and (quantified) variable bindings into strings.
2. Evaluate [...]-expressions to concatenate the evaluations it contains into strings.
3. Concatenate all resulting strings.

Now we extend the translation to sets of Networked Graphs which use each other in their view definitions.

DEFINITION 13. *We say a Networked Graph* r *uses another Networked Graph* g*, written* uses(r,g)*, if*
a) g *occurs in a FROM NAMED declaration in some query defining* r*, or*
b) some Networked Graph used by r *uses* g.
An interdependence set V *is a set of Networked Graphs where* V *contains all Networked Graphs in the transitive closure of the* uses *relation for every Networked Graph in* V.

A Networked Graph in an interdependence set can only be dereferenced at the same time as, or after all Networked Graphs it depends on are dereferenced. Hence, we translate all queries used in all Networked Graphs in an interdepen-

[7] cf. [14] Or the more efficient version as presented in [9]

dence set into a single logic program for evaluation. Please note that a named graph can be dereferenced immediately.

DEFINITION 14. *An interdependece set V is translated to a corresponding logic program P as follows:*

Let m be defined as in definition 12, except that $Const$, Var and F are derived from all Networked Graphs and all queries in view definition statements in V. For every literal q of type `g:query` *in $Q = \bigcup_{\forall g \in V} literals(g)$, translate the query contained in q into a logic program P' as defined in definition 12. In every rule $t(g,s,p,o) \leftarrow T$ in P', add to T an atom $t(g,g,w(definedBy),w(q))$.*[8] *If o has the datatype* `g:query`*, add o to Q.*[9] *Add P' to P.*

Example 5 will result in the following program (before application of Lloyd-Topor transformation), assuming a literal mapping of resource and variable names:

EXAMPLE 11.
t(IFIGraph, ISWeb, workingGroupOf, IFI) ←
...
t(g, s, p, o) ← t(IFIAdminGraph, s, p, o),
t(g, g, definedBy, "CONSTRUCT ...")
t(g, person, worksAt, IFI)
← t(ISWebGraph, person, worksAt, ISWeb),
((t(ISWebGraph, person, status, s),
eq(s, externalResearcher)) ∨ isNull(s)),
¬ bound(s),
t(g, g, definedBy, "CONSTRUCT ...")
bound(x) ← ¬ isNull(x)
isNull(null) ←
t(u, u, uses, v) ← t(u, u, uses, w), t(w, w, uses, u)
t(IFIGraph, IFIGraph, uses, ISWebGraph)←
t(IFIGraph, IFIGraph, uses, IFIAdminGraph)←

The set of true $t(g,s,p,o)$ atoms determines the true statements `s p o.` of the Networked Graph with the name `g`. Having determined the true statements we need to translate them back to RDF:

DEFINITION 15. *Let m^{-1} be the inverse of the mapping $res \cup var$, P a program resulting from the translation of an interdependece set V and* `g` *the name of a Networked Graph in V.*

deref(g) is the set of statements `m⁻¹(s) m⁻¹(s) m⁻¹(s).` *obtained from the bindings computed for the goal $\leftarrow t(m(g),s,p,o)$ from program P.*

5.4 Properties of the Semantic Model of Networked Graphs

Now we discuss possible limitations of the expressive power of view definitions and interesting properties of the resulting fragments of Networked Graphs. We will also show, why the well founded semantics is useful for Networked Graphs.

5.4.1 Alternating Fix Points

We have introduced the well founded semantics very briefly. To understand why we have chosen the well founded semantics, please consider the circular view definition below:

A person is considered a man if he is not a woman and a woman if she is not a man. In two valued logic `ex:g1` could alternate between {`ex:joe a ex:person.`} and {`ex:joe a ex:person. ex:joe a ex:man.`}. The resulting logic program has two stable models. The well founded semantics solves this problem by assigning an unknown truth value to the statement {`ex:joe a ex:man.`}.

EXAMPLE 12.

```
ex:g1 {
  ex:joe a ex:person.
  ex:g1 g:definedBy
    "CONSTRUCT {?p a ex:man.} FROM NAMED ex:g2
      WHERE { GRAPH g2 {?p a ex:person.
        OPTIONAL {?p2 a ex:woman. FILTER (?p2 = ?p).}.
      FILTER (! BOUND(?p2)).}." . } }

ex:g2 {
  ex:g2 g:definedBy
    "CONSTRUCT {?p a ex:woman. ?p a ex:person}
      FROM NAMED ex:g1
      WHERE { GRAPH g1 {?p a ex:person.
        OPTIONAL {?p2 a ex:man. FILTER (?p2 = ?p).}.
      FILTER (! BOUND(?p2)).}." . } }
```

5.4.2 Value Creation

Recursive SPARQL views with blank node creation in the CONSTRUCT pattern can lead to infinite models as shown below for a very simple view resulting in an infinite set of statements in graph `ex:g3` . To avoid this, we forbid the creation of blank nodes in CONSTRUCT patterns of view definitions. Please notice that the use of existing blank nodes, bound in the WHERE pattern, is safe.

```
ex:g3 {
  ex:joe a ex:person.
  ex:g3 g:definedBy
    "CONSTRUCT {?p ex:parent [a ex:person].
                ?p a ex:person.} FROM NAMED ex:g3
      WHERE { GRAPH ex:g3 {?p a ex:person.}.}" . }
```

We argue that in many cases, including the use case presented here, this restriction does not limit the usability of Networked Graphs. This limitation corresponds to similar restrictions in the relational calculus and in DL-safe rules [17].

In the remainder of this paper we only consider Networked Graphs without value creation. When considering value creation the complexity results need to be adjusted to datalog with function symbols.

5.4.3 Monotonic Networked Graphs

Using the result from proposition 1, we can introduce a limitation to the view definition language which results in Monotonic Networked Graphs, by forbidding BOUND negation.

The reader may notice, that we do not completely forbid the use of negation, as it is still possible in other filter expressions. The mapping of the monotonic subset of Networked Graphs to logic programs results in stratified programs. We will consider the consequences in section 6.

DEFINITION 16. *A view definition is called* monotonic *if it does not use BOUND negation.*

LEMMA 3. *Let P be a normal program resulting from an interdependence set including only monotonic view definitions. Then P is stratified.* □

[8] This makes sure, that a query is only used for computing a named context g, if g contains the query in its view definition.

[9] Here the creation of `g:definedBy` statements in CONSTRUCT patterns is handled.

PROOF: Stratification of P is only hindered by the negative dependence of t on itself resulting from the translation rule for OPTIONAL patterns. The mapping of OPTIONAL patterns, for example $m(B) \vee (\neg m(B) \wedge isNull(m(v)))$, must include $\neg m(B)$ to make sure that v is only bound to *null*, if B does not match the dataset. This is important to correctly handle BOUND negation.[10] If no BOUND negation is used, $\neg m(B)$ is not produced. Then P can be stratified. ■

5.4.4 Non-Recursive Networked Graphs

Another possible limitation is to forbid (direct or indirect) recursion in view definitions. Then the program can be made non-recursive. A non-recursive program only consists of predicates which occur only in the heads of rules in their own stratum [7].

LEMMA 4. *Let P be a normal program resulting from an interdependence set including only non-recursive view definitions. Then P can be rewritten to be non-recursive.* □

PROOF: Translate P into a program P' by 1) introducing a new 3-ary predicate t_n for every networked graph n such that $t_n(s,p,o)$ iff $t(n,s,p,o)$ in P and 2) substituting every $t(n,s,p,o)$ in P by $t_n(s,p,o)$. Filter predicates are non-recursive. Now we can assign a level mapping to the t_i, starting bottom-up from named graphs and following the *uses* relations backwards, such that the index in the head of a rule is always higher than in the body. Hence, P' is non-recursive. ■

As non recursive view definitions correspond to sets of SPARQL queries, which do not influence each other, this result means we can express SPARQL without value creation in Datalog without function symbols.

6. COMPLEXITY

We now investigate the data complexity of Networked Graph evaluation. The data complexity is defined as the complexity of computing a model for a variable extensional knowledge base given a fixed query. Data complexity is of particular interest, as usually the query is short compared to the knowledge base and translations of Networked Graph definitions are fixed, because all query literals are known in advance.

6.1 General Networked Graphs

First, we discuss how much the translation to a logic program adds to the overall complexity:

LEMMA 5. *The length of the logic program resulting from the translation in definitions 12 and 14 is in*

$O(|G| + \sum |C||2^l||W|)$, where $|G|$ is the number of statements in the graphs declared using FROM NAMED and for every query in the interdependence set's Networked Graph definitions, $|C|$ is the number of statement patterns in the CONSTRUCT pattern, l is the maximum level of nestings of OPTIONAL and UNION patterns, and $|W|$ is the length of the WHERE pattern. □

PROOF: The mapping of FROM NAMED adds exactly one fact for every statement in the graph declared using FROM NAMED. Every query is translated into a single rule of a logic program. Hence we have a number of clauses corresponding to the number of queries used in the view definitions. Each of these rules is then transformed into a set of normal clauses using Lloyd Topor transformation. First, for every atom in the head of a clause, an separate clause is generated. This results in $|C|$ clauses. Then for every logical OR in the body of a clause, Lloyd Topor transformation results in two new clauses. As logical ORs are only introduced by mappings of UNION and OPTIONAL patterns, this step generates at most $|2^l|$ clauses. Qualified graph patterns and filter expressions map to a number of literals of the logic program wich is equal to the number of qualified statement patterns in the qualified graph pattern or the length of the filter expression respectively. Unbound OPTIONAL patterns are replaced by a list of *isNull* atoms for the unbound variables, if any. This list has at most three times the length the OPTIONAL graph pattern. Thus the overall length of the resulting clauses is bound by $3|W|$. ■

With some simple optimisation, however, we can do better:

LEMMA 6. *The translation of interdependence sets to logic programs can be optimised to have complexity in $O(|G| + |C||W|)$.* □

PROOF: For every two rules resulting from Lloyd-Topor transformation of a UNION or OPTIONAL pattern we can remove one rule:

A rule binding a variable to *null*, which is used in the CONSTRUCT statement pattern trivially fails, as it contains the subgoals *bound(x)* and *isNull(x)* for some variable x.

Else, if a BOUND filter is applied to a variable introduced in a nested OPTIONAL of UNION pattern, we can add an atom *bound(x)* for every optional variable x to the body of the rule binding the optional variables. Now one of the two rules under consideration is trivially false, as it contains *isNull(x)* and *bound(x)* or *bound(x)* and $\neg$*bound(x)*.

Else, we only need to consider the rule which does not bind the optional variables. The alternative rule is irrelevant, because

1. we can never derive a return value of "true" from a filter on an unbound variable or
2. if no filter is applied to an optional variable, the OPTIONAL pattern can not not influence the result.

■

As we consider the data complexity here, we can conclude that the translation of non-query statements contributes linearly in the size of the input graphs.

The data complexity of computing the well founded semantics of a given function free normal logic program is known to be polynomial. Whether computing the well founded semantics in linear time is possible is still open. However, algorithms are known, which are $O(|P||A|)$ where $|P|$ is the the overall length of the program P, i.e. sum of

[10] Without $\neg m(B)$, v could be bound to *null*, even though binding the variables in $m(B)$ is possible. Hence, we would generate an additional binding. If no BOUND negation is used this does not matter, because the rules binding variables to null are not used to infer true statements, as all rules contain a subgoal *bound(v)* for every variable v used in the head of the rule.

the number of literals of all program clauses and $|A|$ is the number of atoms used in P (cf. [1], [7]).

THEOREM 2. *The data complexity of Networked Graph computation without value creation in the general case is quadratic in the size of the Networked Graphs used in the interdependence set.* □

PROOF: This theorem directly follows from [1], the fact that $|A| \leq |P|$ and lemma 6. ■

6.2 Monotonic Networked Graphs

Similarly for monotonic Networked Graphs we can conclude:

THEOREM 3. *The data complexity of Monotonic Networked Graph computation without value creation is linear in the size of the Networked Graphs used in the interdependence set.* □

PROOF: The data complexity of computing the minimal model for a stratified normal program with recursion is linear in the length of the program [1], resulting from the linear complexity of Horn programs [11]. Lemma 3 shows that the program resulting from the translation of monotonic networked graphs is stratified and, as above, the translation contributes linearly to the data complexity. ■

6.3 Non-Recursive Networked Graphs

Non-recursive Networked Graphs directly correspond to SPARQL queries.

THEOREM 4. *The data complexity of Monotonic Networked Graph computation without value creation is LOGSPACE in the size of the Networked Graphs used in the interdependence set.* □

PROOF: This theorem directly follows from lemma 4 and LOGSPACE complexity for non-recursive datalog with negation (cf. [7]). ■

In this case we also have LOGSPACE complete combined complexity (cf. [7]) for non-recursive datalog with negation. This result corresponds to results by Perez et al. [19].

7. IMPLEMENTATION

We have implemented a SAIL supporting general Networked Graphs without value creation for the Sesame 2.0 RDF repository[11]. A SAIL is a Storage And Inference Layer, a stackable module in Sesame for either storing RDF, for example in a relational database, or doing inferencing. In the latter case, infered statements can be added to an underlying SAIL for materialised storage or be infered on demand. The Networked GraphSAIL reuses the Sesame API to a large extent and only exploits the XSB Prolog engine for evaluating the generated normal programs. We use Sesame Contexts (corresponding to named graphs) of an underlying SAIL for storing statements. View definitions are parsed into an internal query model using the Sesame SPARQL parser. Then the rules of the logic program are generated, while the facts (the t predicate and filter predicates) are evaluated as Java Messages, i.e. on the Java side using the usual Sesame methods.

[11]http://isweb.uni-koblenz.de/Research/NetworkedGraphs.

Using this architecture, we do not need to translate large amounts of facts, and we can use Sesame mechanisms for introducing new filters or adding different kinds of inferencing, for example RDFS.

For the future, we also plan to distribute computations of Networked Graphs to different SPARQL endpoints, allowing for a true networking of RDF graphs.

8. RELATED WORK

Semantics of RDF and SPARQL. de Bruijn et al. provide a mapping of RDF and OWL to first order predicate logic and prove this mapping equal to normative RDF [8]. Based on this mapping they can formulate goals corresponding to simple SPARQL graph pattern matching. Translation of filter expressions and of optional patterns is not discussed.

Perez et al. provide a formal semantics of a core fragment SPARQL [19]. Similar to our mapping, they map blank nodes to variables and ignore the RDF/S vocabulary. Their semantics does not include queries to multiple graphs, a crucial fragment for our purposes. In future work it will be interesting to investigate the relation between their semantics ond our logic mapping.

Rules. Networked Graphs allow to use SPARQL to formulate rules. The Semantic Web Rule Language SWRL [13] combines OWL with Horn rules. The resulting combined language is very expressive and allows to formulate undecidable problems. Therefore a subset, DL-safe rules (c.f. [16]) has been proposed. SWRL allows to infer new statements from a given graph, but not to have a different view on this graph. Both of this is possible with Networked Graphs.

Polleres and Schindlauer propose to map SPARQL queries to Datalog. Their approach most probably will be very similar to our mapping, but does not provide SPARQL based views. At the time of writing of this paper only a poster abstract is available [20].

Grouping RDF Statements. RDF reification [18] can be used to make statements about RDF statements. A resource is created which has three properties refering to the subject, predicate and object of the reified statement. This resource can then be subject or object of statements. RDF reification has been criticized much because of its weak semantics. Particularly we can not infer that the reified statement exists. Additionally, reification leads to very large graphs: Every statement requires additional three statements to reify it and at least one statement to describe it.

Various approaches have been proposed to automatically decompose RDF graphs into small meaningfull parts, for example RDF molecules [10], minimum self contained graphs [25] and concise bounded descriptions [23]. However, these approaches do not claim to select pieses of an RDF graph which are meaningful for an application, for example in our use case all information about a single researcher. While automatically computing a sensible subgraph is impossible in the general case, at the time of creation of an RDF graph it is usually very easy for a user or a tool to describe a set of statements as belonging together.

Named graphs were mainly developed for trust ensurance and signing of RDF graphs. For these applications, one must only assure that a graph contains exactly the signed

information. In this paper we show how Networked Graph descriptions can be stored in named graphs, allowing for reuse of the infrastructure for named graphs. Using named graphs, enforcing for example graph inclusion is not possible: u:IFIGraph cannot automatically include all statements of u:IFIAdminGraph. We extend the static named graphs with a more dynamic view mechanism, but can still harness the existing work and infrastructure for named graphs.

Context. Stoermer et al. use named graphs plus semantic extensions of RDF to implement a system using modal logic for statements in RDF graphs [24]. The RDF semantics is extended such that statements are only true in the context of a graph called a context. Between these graphs compatibility relations can be defined and infered. They are then used for reasoning. For example if context A extends context B we can query context A instead of B to get a more complete answer. However, contexts do not allow for the reuse of RDF data, for example by importing statements. They focus on describing relationships among graphs and using them for reasoning.

Sintek and Decker present TRIPLE [22]. Triple allows to formulate models[12] containing rules describing the semantics of some knowledge representation language and knowledge bases. Models can be parametrised with other models, such that for example RDFS semantics can be applied to some knowledge base by parametrising the rdf model with the knowledge base. This mechanism also allows to apply arbitrary first-order rules. Sintek and Decker express part of the RDF/S semantics in TRIPLE, while Networked Graphs are built upon named graphs and the upcoming standard SPARQL and thus are upward compatible to the existing semantic web infrastructure.

C-OWL [2] is an extension of the web ontology language OWL with local contexts. These contexts are connected with bridge rules allowing for translation from one context to another. However, C-OWL aims at translations between local contexts, not at reuse of RDF data. Thus C-OWL can not be applied to the use case introduced here.

Chein et al. describe nesting of conceptual graphs in [6]. They provide an extension of conceptual graphs to express complex knowledge, where statements may be the object of other statements. Semantic graphs can be mapped to first order logic. Nested graphs can not be used for composing more complex graphs. The knowledge represented in a nested graph is pertinent to this graph and not available in the surrounding graph. The objective of this extension is to express quoting of the nested statements.

Views. Volz et al. [27] as well as Magkanaraki et al. [15] propose views for RDF. They can be used to define "virtual" classes, properties and instances based on graph patterns. They employ the RDFS semantics to ensure that neccessary class relations are also included in the view. The purpose is to provide a view to a graph using a different ontology rather than defining graph contents based on other graphs. These classical views are not oriented towards the networked structure of the semantic web. Reuse and exchange of views across the borders of single RDF repositories is not easily possible.

[12]The term model is misleading here, because it should not to be understood in the sense of a logical model. A more appropriate term would be context.

9. CONCLUSION

We have introduced Networked Graphs as a means for describing RDF graphs that are partially derived from other graphs using SPARQL queries. Thus, we have fulfilled the requirements derived from our running use case — which is just one of the many cases that will benefit from dynamic networking between RDF graphs.

As a new paradigm Networked Graphs raise new issues as regards the details as well as the broad vision of dynamically networking graphs.

With regard to the details, the precise semantics of SPARQL constitutes a moving target and we will have to investigate the relation between our semantics for SPARQL and proposals like [19] more closely. Also, there will be further issues about reasoning on relationships among Networked Graphs based on Networked Graph definitions when SPARQL queries are defined in RDF itself.

With regard to the broad vision, Networked Graphs provide a new paradigm for information integration. In this paper, we have defined a Networked Graph by combining a closed world semantics with a kind of Global-as-view approach for integrating information from other graphs. On the other hand there can also be scenarios, where each researcher defines her affiliation and the graph about lab membership is updated according to her view definition of the IFIGraph in a kind of Local-as-view approach. We conjecture that such an approach may require new machinery, including parameters for graph names in queries as well as trust relationships for updates on the view definition of the IFIGraph.

Acknowledgements. We would like to thank Carsten Saathoff and Thomas Franz for inspiring discussions about views for RDF and Lynda Hardman and Raphael Troncy of CWI for their feedback on an early version of this paper.

This research was supported by the European Commission under contract FP6-027026, Knowledge Space of semantic inference for automatic annotation and retrieval of multimedia content - K-Space. The expressed content is the view of the authors but not necessarily the view of the K-Space project.

10. REFERENCES

[1] K. A. Berman, J. S. Schlipf, and J. V. Franco. Computing the Well-founded Semantics Faster. In *Proceedings of the 3rd International Workshop on Logic Programming and Non-monotonic Reasoning*, pages 113–126. MIT Press, 1995.

[2] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. Contextualizing Ontologies. *Journal of Web Semantics*, 1(4):325–343, 2004.

[3] R. J. Brachman and H. J. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufman, 2004.

[4] J. Brase and W. Nejdl. Annotation for an open learning repository for computer science. In *Annotation for the Semantic Web*, pages 212–227. IOS Press, 2003.

[5] J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 613–622, New York, NY, USA, 2005. ACM Press.

[6] M. Chein, M. L. Mugnier, and G. Simonet. Nested Graphs: A Graph Based Knowledge Representation Model with FOL Semantics. In *Principles of Knowledge Representation and Reasoning*, pages 524–535. Morgan Kaufman, 1998.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing. Surveys.*, 33(3):374–425, 2001.

[8] J. de Bruijn, E. Franconi, and S. Tessaris. Logical reconstruction of normative RDF. In *OWL: Experiences and Directions Workshop*, Galway, Ireland, November 2005. CEUR Workshop Proceedings. http://www.debruijn.net/publications/owl-05.pdf [2006-10-20].

[9] S. Decker. *Semantic Web Methods for Knowledge Management*. Phd thesis, University of Karlsruhe, Februar 2002.

[10] L. Ding, T. Finin, Y. Peng, P. P. da Silva, and D. L. McGuinness. Tracking RDF Graph Provenance using RDF Molecules. Technical report, UMBC, April 2005. http://ebiquity.umbc.edu/paper/html/id/240/.

[11] W. Dowling and J. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[12] S. Egorov. RDF Graph Patterns and Templates. http://vocab.org/riro/gpt.html [2006-10-20], 2006.

[13] I. Horrocks et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. http://www.w3.org/Submission/SWRL/ [2006-10-20], 2004.

[14] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[15] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the semantic web through RVL lenses. *Journal of Web Semantics*, 1(4):359–375, 2004.

[16] B. Motik, I. Horrocks, R. Rosati, and U. Sattler. Can OWL and Logic Programming Live Together Happily Ever After? In *5th International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 501–514. Springer, 2006.

[17] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with rules. *J. Web Sem.*, 3(1):41–60, 2005.

[18] Patrick Hayes (ed.). Rdf semantics. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/ [2006-10-20].

[19] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *5th International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2006.

[20] A. Polleres and R. Schindlauer. SPAR2QL: From SPARQL to rules. In *International Semantic Web Conference (Posters Track)*, 2006.

[21] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ [2006-10-20], 2006.

[22] M. Sintek and S. Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2002.

[23] P. Stickler. CBD - Concise Bounded Description. http://swdev.nokia.com/uriqa/CBD.html [2006-10-20], 2005.

[24] H. Stoermer, I. Palmisano, D. Redavid, L. Iannone, P. Bouquet, and G. Semeraro. RDF and Contexts: Use of SPARQL and Named Graphs to Achieve Contextualization. In *Proccedings of the 2006 Jena User Conference*, 2006.

[25] G. Tummarello, C. Morbidoni, P. Puliti, and F. Piazza. Signing individual fragments of an RDF graph. In *WWW (Special interest tracks and posters)*, pages 1020–1021. ACM, 2005.

[26] A. van Gelder, K. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[27] R. Volz, D. Oberle, and R. Studer. Implementing Views for Light-Weight Web Ontologies. In *7th International Database Engineering and Applications Symposium*, pages 160–169. IEEE Computer Society, 2003.

APPENDIX

A. SPARQL DEFINITION

In the following the fragment of SPARQL used in this paper is defined. We start with the definition of filter expressions.

DEFINITION 17. *A* filter expression *consists of the keyword "FILTER" followed by a filter definition.*

Let V be a set of variables. Let $x, y \in R \cup V$ and F, G be filter definitions.

A filter definition is inductively defined as follows:

(1) $(x = y)$, $(x! = y)$, $(x < y)$, $(x <= y)$, $(x > y)$, $(x >= y)$ are filter definitions, called filter atoms.

(2) $func(x, ..., y)$ is a filter definition, if $func$ identifies a filter function as defined in [21, chapter 11]. $func(x, ..., y)$ is a filter atom.

(3) $(!F)$, $(F\&\&G)$, $(F||G)$ are filter definitions.

$=, != , ..., >=, !, \&\&, ||, func$ are called filter operators.

Let H be a filter atom. Then H and $!H$ are filter literals.

We will also write filter operators in prenex form, for example *!= (x,y)* instead of *x != y*.

EXAMPLE 13.

```
FILTER ((?x = "abc") &&
        (!(BOUND(?b))) &&
        http://example.org/somefilter(?x, ?y))
```

is a filter expression. `(?x = "abc")` *applies rule (1),* `http://example.org/somefilter(?x, ?y)` *applies rule (2) and the whole filter expression is formed by applying rule to form a filter definition(3) and adding the word* `FILTER`.

Filter expressions are used in filtered SPARQL patterns to filter bindings resulting from matching a graph pattern:

DEFINITION 18. *Let V be a set of variables.*
(1) A statement pattern *is a triple $(R \cup V) \times (U \cup V) \times (R \cup V)$.*
(2) A graph pattern *is a sequence of statement patterns.*
(3) A qualified statement pattern *is the keyword GRAPH followed by a URIref or a variable denoting a graph and a statement pattern in curly brackets.*
(4) A qualified graph pattern *is a sequence of qualified statement patterns.*
(5) A filtered graph pattern *is either*
a) a sequence containing one or more qualified graph patterns zero or more filter expressions and zero or more filtered graph patterns or
b) an OPTIONAL graph pattern or
c) a UNION graph pattern.
Filtered graph patterns are grouped in curly brackets.
(6) An OPTIONAL graph pattern *is two filtered graph patterns connected by the keyword OPTIONAL.*
(7) A UNION graph pattern *is two filtered graph patterns connected by the key word UNION.*
(8) All of the above are SPARQL expressions.

EXAMPLE 14.

```
{GRAPH u:IFIGraph {u:IFIGraph a ng:Graph.}
 GRAPH u:IFIGraph {u:IFIGraph g:contains ?x.}
 FILTER (?x = u:IFIAdminGraph).
} UNION {
 GRAPH u:ISWebGraph {u:Steffen u:worksAt u:ISWeb.}
}
```

is a UNION graph pattern consisting of a filtered graph pattern and a qualified statement pattern. It is also a SPARQL expression and a filtered graph pattern.
`u:IFIGraph g:contains ?x.`
is a statement pattern.

In this paper we use the following vocabulary:

DEFINITION 19. *Let x, y be filtered graph patterns. We say x is a subpattern of y, if x occurs in y.*

EXAMPLE 15.

```
GRAPH u:ISWebGraph {u:Steffen u:worksAt u:ISWeb.}
```

is a subpattern of the UNION graph pattern in example 14.

DEFINITION 20. *We say a filtered graph pattern x introduces a variable v, if v does not occur in qualified statement patterns outside x and v is not introduced by some subpattern of x.*

EXAMPLE 16.
In example 14, the following SPARQL expression introduces the variable `?x`*:*

```
{GRAPH u:IFIGraph {u:IFIGraph a ng:Graph.}
 GRAPH u:IFIGraph {u:IFIGraph g:contains ?x.}
 FILTER (?x = u:IFIAdminGraph).}
```

To evaluate a query, the WHERE pattern is matched against the graphs declared using FROM NAMED, computing all possible variable bindings. A qualified statement pattern succeeds, producing bindings for variable components of the pattern, if the graph named in the pattern contains a statement matching the subject, predicate and object of the pattern. We say a filtered graph pattern succeeds, if it produces bindings for all variables it introduces. An OPTIONAL graph pattern succeeds even if no suitable bindings can be computed for its right subpattern. In this case the variables introduced by the right subpattern are left unbound and only the bindings of the left subpattern are computed. A UNION pattern succeeds, if one of its subpatterns succeeds. The UNION pattern uses the bindings of the succeeding subpattern. If some filter expression instantiated with the bindings computed within the corresponding filtered graph pattern returns false or an error, the corresponding binding fails. A filter expression returns an error, if it is invoked with parameters of a wrong type or with a parameter returning an error. The only filter accepting unbound variables as parameters is the BOUND filter, which returns true, if a variable is bound and false else. Please refer to [21] regarding the behaviour of logical connectives in the presence of errors.

B. NORMALISATION OF FILTER EXPRESSIONS

The proof of lemma 1 is analogous to the proof that arbitrary boolean formulas can be translated into disjunctive normal form (cf. [3])[13].

LEMMA 1. *Every filter expression can be translated into conjunctive normal form.* □

PROOF: We show that the double negative law, the distributive law and DeMorgan's law hold in the presence of the *error* return value for filters. Evaluation of logical connectives in SPARQL filters is defined in [21].
Double Negative Law
As we can see below, the double negative law holds.

A	$\neg\neg A$
t	t
f	f
e	e

Distributive Law
As we can see from the following truth tables, the distributive law holds. Only extensions to two valued logics are shown and wlg. we leave out symmetric cases for B and C.

[13]The translation described in [3] results in an equal formula, which can be exponentially longer than the input formula. Please note that there exist other translations, only preserving satisfiability instead of equality, which are of linear complexity and would suffice here. However, the neccessary basics are the same as proved here.

A	B	C	$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C)$	$(A \wedge B) \vee (A \wedge C)$
t	f	e	t	t	e	e
t	t	e	t	t	t	t
f	t	e	e	e	f	f
f	f	e	f	f	f	f
e	t	e	e	e	e	e
e	f	e	e	e	e	e

DeMorgan's Law

As we can see from the following truth tables, DeMorgan's law holds. Again, only extensions to two valued logics are shown and wlg. we leave out symmetric cases.

A	B	$\neg(A \vee B)$	$\neg A \wedge \neg B$	$\neg(A \wedge B)$	$\neg A \vee \neg B$
t	e	f	f	e	e
f	e	e	e	t	t
e	e	e	e	e	e

■

Evaluation of normalised filter expressions

Now we prove lemma 2.

LEMMA 2. *Filter expressions in conjunctive normal form are evaluated equally in SPARQL and in first order logic after translation according to definition 11.* □

Without limiting generality we consider a filter expression $F = (A \vee B) \wedge C$ only consisting of one conjunction and one disjunction. Obviously it is in disjunctive normal form. We can construct any filter expression in concunctive normal form from expressions of this structure by replacing A or B with arbitrary disjunctions and C with another formula of this structure. Only extensions to two valued logics are shown and symmetries for A and B are left out.

A	B	C	F in SPARQL	F in FOL
t	f	e	e	f
t	t	e	e	f
f	t	e	e	f
f	f	e	f	f
e	t	e	e	f
e	f	e	e	f
e	e	e	e	f
e	t	t	t	t
e	f	t	e	f
e	e	t	e	f
e	t	f	f	f
e	f	f	f	f
e	e	f	f	f

As *error* is treated like *false* in the end result of a filter evaluation, proposition 2 holds. ■

C. INCLUDING RDF/S SEMANTICS

RDF/S semantics can be included to the logic program P as follows:

- Add a rule $t(g, g, rdf : type, ng : graph) \leftarrow t(g, s, p, o)$ to P
- For every axiomatic RDF and RDFS triple `s p o.` as defined in [18] add a rule $t(g, s, p, o) \leftarrow t(g, g, rdf : type, ng : graph)$ to P (As there are infinitely many `rdf:_ n`, `n`$\in \mathbb{N}$, we only add rules for those axiomatic statements containing some `rdf:_ n` to P, which are used in some Networked Graph in the dependence set. Hence, formally we do not cover the complete RDFS semantics. The result, however, is the same.)
- When mapping literals to constants, add facts *isLiteral(x)*, *isWellTypedLiteral(x)*, *isPlainLiteral(x)* with the obvious meaning where appropriate.
- Add a translation of the production rules given in chapter 7 of [18] to the program, extended to (graph, subject, predicate, object) quadruples.

When RDF/S semantics is included we need to reconsider the complexity of the resulting logic program.

Bisher erschienen

Arbeitsberichte des Fachbereichs Informatik
(http://www.uni-koblenz.de/fb4/publikationen/arbeitsberichte)

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J.Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe“
(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop "Reengineering Prozesse" – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005