How to Modify on the Semantic Web?

A Web Application Architecture for Algebraic Graph Transformations on RDF

Benjamin Braatz* and Christoph Brandt

SECAN-Lab, Université du Luxembourg
benjamin.braatz@uni.lu, christoph.brandt@uni.lu
http://wiki.uni.lu/secan-lab/

Abstract. In this paper, we show how formal methods of algebraic graph transformation can be made available in the technological environment of the Semantic Web. This new and promising approach allows to model and formulate transformation operations on a high level of abstraction. To demonstrate its feasibility, we develop a small book shop case study. We show how some of the necessary modification operations in this settings can be realised by the proposed SPARQL/Update language and by our rule-based graph transformation approach. In a comparison, we highlight the main differences between these two approaches. The most important benefit of our rule-based transformation approach is that it facilitates to move from application engineering to rule engineering and thereby support generic solutions, which lower the maintenance efforts by supporting the declarative specification of modification operations.

1 Introduction

The Semantic Web, based on the Resource Description Framework (RDF, as specified in [1] and related documents), is an emerging technology for managing and maintaining heterogeneous, distributed, structured data and metadata, which are stored as triples in RDF data stores or (from the more formal point of view) RDF graphs. While a lot of publications consider the deduction on RDF data by inference rules, the modification of RDF graphs has not been in the focus until recently. As an application scenario, we present a book shop example, show how the relevant data are represented in an RDF data store and motivate some modification operations, which are required in this context, in Sect. 2.

The SPARQL/Update language, which is proposed in [2,3] and currently developed, is one of the most promising approaches regarding modification operations on RDF. In Sect. 3, the operations from the application scenario are realised using SPARQL/Update. The development of SPARQL/Update is to a large extent practice-driven, i. e., the requirements of applications in the field of modifications motivate and determine the collection of language primitives.

* The first author is supported by the National Research Fund, Luxembourg, and cofunded under the Marie Curie Actions of the European Commission (FP7-COFUND).

F. Daniel and F.M. Facca (Eds.): ICWE 2010 Workshops, LNCS 6385, pp. 187–198, 2010.

In this paper, we propose the use of algebraic graph transformation (see [4] for a recent overview of the field) as an alternative technique to modify RDF graphs. The theoretical framework of algebraic graph transformation, which uses category theoretical methods to allow abstract formal reasoning about rule-based transformations, has been adapted to the needs of RDF in [5,6]. In Sect. 4, we show how the example operations can be implemented as algebraic graph transformation rules, while we discuss their realisation in a Semantic Web architecture in Sect. 5. This approach is theory-driven in the sense that the notion of transformation rule, which replaces the collection of language primitives found in SPARQL/Update, is motivated by a formal understanding of modifications on RDF data.

The two approaches to modifications on RDF, SPARQL/Update and algebraic graph transformation, are compared in Sect. 6, where we also shortly discuss other related work. Finally, in Sect. 7, we summarise the present work and give some possible lines of future work.

2 Application Scenario

Throughout the paper, we consider a book shop application based on Semantic Web technologies as an example scenario. The shop has two kinds of users: employees of the shop who can change the catalogue by adding and deleting books to and from it and customers who can buy these books by adding them to their shopping carts and later ordering the contents of the shopping cart.

For the representation of RDF data we use the Notation 3 (N3) syntax defined in [7]. Figure 1 shows an RDF representation of the book "The Lord of the Rings" by J. R. R. Tolkien on the one hand in N3 and on the other hand as a graphical visualisation with subjects and objects as vertices and triples as edges. Such an N3 representation might be returned when a client accesses the URI `http://shop.example.com/isbn/978-0-261-10238-5`. Representations of the same information in RDF/XML or HTML could, of course, also be provided by the application.

The representation of the shopping cart of a user is shown in Fig. 2, where we again give the N3 representation as well as a graphical visualisation. This representation (or an equivalent RDF/XML or HTML representation) will be returned when a client accesses the URI `http://shop.example.com/users/JohnDoe`. It should only be accessible if the corresponding user is authenticated with the application. The details of possible authentication schemes are outside the scope of this paper.

In the following, we will assume that the data store of our application is represented by a single RDF graph, which contains the graphs in Fig. 1 and 2 as parts. Furthermore, the exact selection of the whole RDF graph that is provided in response to a GET request has to be determined by some mechanism outside the scope of this paper.

The question arises how the operations of employees and customers on this data store can be specified and implemented. As examples, we will consider the following operations:

```
@prefix book: <http://shop.example.com/isbn/>.
@prefix shop: <http://shop.example.com/scheme/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
@prefix curr: <http://bank.example.com/currencies/>.
book:978-0-261-10238-5 a shop:Book;
                       dc:title "The Lord of the Rings";
                       dc:creator [ a foaf:Person;
                                    foaf:name "J. R. R. Tolkien" ];
                       dc:publisher <http://www.harpercollins.com/>;
                       shop:price "17.95"^^curr:EUR.
```

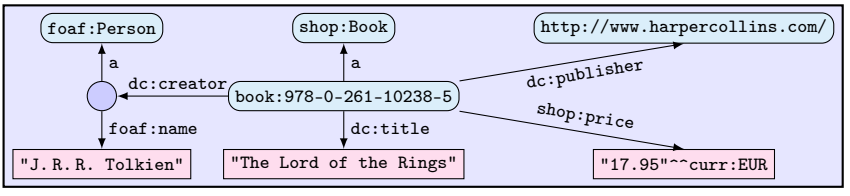

Fig. 1. Representation of a book in RDF

```
@prefix book: <http://shop.example.com/isbn/>.
@prefix shop: <http://shop.example.com/scheme/>.
@prefix user: <http://shop.example.com/users/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
user:JohnDoe shop:cart [ a shop:Cart;
                         shop:content book:978-0-261-10238-5,
                                      book:978-0-261-10221-7 ].
book:978-0-261-10238-5 dc:title "The Lord of the Rings".
book:978-0-261-10221-7 dc:title "The Hobbit".
```

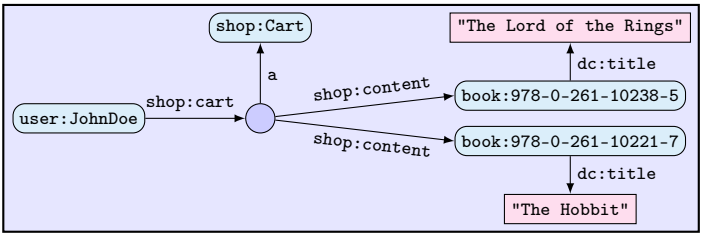

Fig. 2. Representation of a shopping cart in RDF

- The blank node representing the author in Fig. 1 is *replaced* by a URI for that author, such that other books of the author can also be linked to the same URI and a navigation to all books of an author becomes possible.
- A book of this author is *added* to the catalogue.
- A book is *added* to the shopping cart of a customer.

In the following sections, we will realise these operations first by the current version of the proposed SPARQL/Update language and then by the help of formal techniques from algebraic graph transformation.

3 Solution by SPARQL/Update

The SPARQL/Update language was proposed in [2] and is currently in the process of standardisation, where a working draft is available in [3]. It extends the RDF query language SPARQL (defined in [8]) by keywords for modifying the contents of an RDF triple store.

It can be used to realise the required operations on the data store as shown in Fig. 3 till 5. In Fig. 3, the blank node representing the author in Fig. 1 is replaced by the URI `author:Tolkien`. This is achieved by first binding the variable `?author` to this blank node in the `WHERE` clause, then removing the triples including the node as subject or object in the `DELETE` clause and, finally, inserting corresponding triples for `author:Tolkien` in the `INSERT` clause.

```
PREFIX book: <http://shop.example.com/isbn/>.
PREFIX author: <http://shop.example.com/authors/>.
PREFIX dc: <http://purl.org/dc/elements/1.1/>.
PREFIX foaf: <http://xmlns.com/foaf/0.1/>.
MODIFY
DELETE { book:978-0-261-10238-5 dc:creator ?author.
         ?author a foaf:Person;
                 foaf:name "J.R.R. Tolkien" }
INSERT { book:978-0-261-10238-5 dc:creator author:Tolkien.
         author:Tolkien a foaf:Person;
                        foaf:name "J.R.R. Tolkien" }
WHERE { book:978-0-261-10238-5 dc:creator ?author.
        ?author a foaf:Person;
                foaf:name "J.R.R. Tolkien" }
```

Fig. 3. SPARQL/Update query for replacing blank node by URI

In Fig. 4, the insertion of another book by the same author is realised by an `INSERT DATA` query, which does not use any variables, but just lists the triples that are to be added to the data store.

```
PREFIX book: <http://shop.example.com/isbn/>.
PREFIX shop: <http://shop.example.com/scheme/>.
PREFIX author: <http://shop.example.com/authors/>.
PREFIX dc: <http://purl.org/dc/elements/1.1/>.
PREFIX curr: <http://bank.example.com/currencies/>.
INSERT DATA { book:978-0-261-10273-6 a shop:Book;
                                     dc:title "The Silmarillion";
                                     dc:creator author:Tolkien;
                                     dc:publisher <http://www.harpercollins.com/>;
                                     shop:price "10.80"^^curr:EUR }
```

Fig. 4. SPARQL/Update query for adding a book to the catalogue

Finally, in Fig. 5, we are adding the book that was just added to the catalogue in the data store to the shopping cart of Fig. 2. Since the shopping cart itself is represented by a blank node, which is not adressable from outside the data

```
PREFIX book: <http://shop.example.com/isbn/>.
PREFIX shop: <http://shop.example.com/scheme/>.
PREFIX user: <http://shop.example.com/users/>.
INSERT { ?cart shop:content book:978-0-261-10273-6 }
WHERE { user:JohnDoe shop:cart ?cart }
```

Fig. 5. SPARQL/Update query for adding a book to a cart

store, we are using a `WHERE` clause again to bind the variable `?cart` to it and afterwards just insert the corresponding `shop:content` triple.

The given SPARQL/Update queries only realise concrete instances of the desired operations. They replace a particular blank node by a URI, add a particular book to the catalogue and put it in the shopping cart of a particular user. If the operations have to be provided for generic situations—replacing all blank nodes that represent the same author by a corresponding URI, adding an arbitrary book to the catalogue or putting an arbitrary book into the shopping cart of an arbitrary user—then we need some other technology in order to generate SPARQL/Update queries from given parameters.

Regarding a possible Web application architecture, such a generation of SPARQL/Update queries could be done either on the client side, using a public SPARQL/Update end point, or on the server side, leading to application-specific service interfaces.

The first alternative, providing a public SPARQL/Update end point, poses serious security issues since clients are able to request arbitrary modifications on the data store, where objectionable queries can only be distinguished from allowed queries by a deep analysis of their respective effect.

In both cases, client-side as well as server-side generation of queries, an assumed generation procedure will be embedded into specific application code that constructs the query code and also realises the user interface or a Web service. This complicates the handling of security issues, the analysis of the impact of queries on the data store and the maintenance of the operations.

4 Solution by Algebraic Graph Transformation

As an alternative to SPARQL/Update, we propose the use of algebraic graph transformation for modifying RDF data stores. A comprehensive treatment of the theory of algebraic graph transformation, more specifically of the "double-pushout" (DPO) approach to graph transformation, can be found in [4]. Since this approach is not applicable as-is to RDF graphs due to differences in the formalisation of graphs—the DPO approach uses graphs that allow multiple edges with identical labels between two nodes–, we have developed a variant of this approach in [5] and [6] and have also shown first theoretical results regarding the composition of RDF transformation rules.

In Fig. 6, an RDF transformation rule realising the replacement of blank node representations of authors by URIs is shown with its full formal structure and a

compact notation, which will be used in the following. A rule consists of a left-hand side L, an interface I and a right-hand side R. The interface is connected to left-hand and right-hand side by homomorphisms l and r, respectively, which are required to be injective by the theory and are, for practical purposes, always chosen to be inclusions. This also enables us to draw L, I and R into one diagram in the compact notation, where the nodes and triples that are only present in L are marked by {del} and those that are only present in R by {add}.

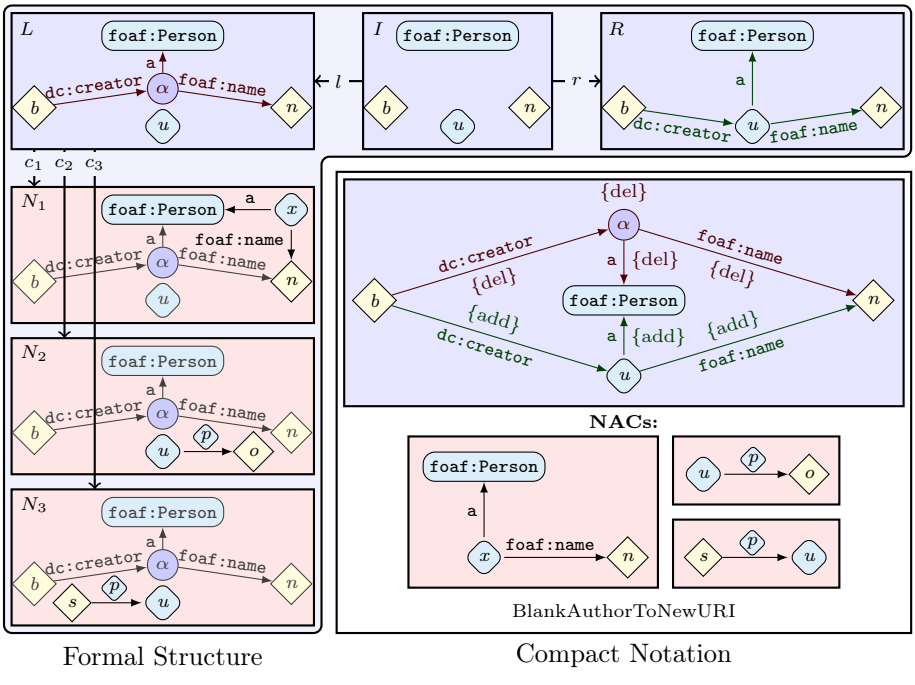

Fig. 6. RDF transformation rule for replacing blank node by new URI

The patterns used in these rules do not only contain URIs, literals and blank nodes, but also two kinds of variables. The general variables, depicted by sharp diamonds, can be assigned to URIs, literals and blank nodes, while the URI variables, depicted by rounded diamonds, can only be assigned to URIs. The variable sets of L, I and R are required to be bijectively mapped by l and r, which intuitively means that they are identical for the whole rule.

The additional blank nodes and triples in the left-hand side L of an RDF transformation rule are the ones that will be deleted when the rule is applied, while the additional blank nodes and triples in the right-hand side R are created. For example, the rule in Fig. 6 deletes a blank node α representing an author with name n of a book b with the corresponding triples and creates equivalent triples for a URI u representing that author. Observe that only blank nodes can be deleted and created, while URIs and literals are assumed to be globally and

eternally given. This is reflected in the requirement that variables, which can be assigned to URIs and literals, have to be the same in L, I and R which means that they are neither deleted nor created.

In addition to this, a transformation rule contains a set of negative application conditions (NACs), which describe situations in which the rule cannot be applied. These are formalised as extensions of the left-hand side by the structures that are forbidden, where the irrelevant parts of L are omitted in the compact notation. For instance, the NAC N_1 in Fig. 6 forbids that the rule introduces a new URI for an author if there is already another URI (represented by the URI variable x in the NAC) with the same name. The NACs N_2 and N_3 require that the URI that is used for the author (represented by the URI variable u in the rule) is not already used in any other triple as subject or object.

Figure 7 shows an application of this transformation rule, where the predicates in the lower row are abbreviated and the dots symbolise that the graph might be much larger than the shown part. The left-hand side L of the RDF transformation rule is found in the host graph G by a match homomorphism m, which also assigns the variables. The effect of the rule is now described in terms of constructions from category theory (see, e. g., [9] for an introduction). The deletion is obtained by a so-called minimal pushout complement (MPOC) D, which intuitively just deletes all additional blank nodes and triples of the left-hand side; this is only possible if the deleted blank nodes are not part of other triples in G (*dangling condition*) and deleted blank nodes are not identified to other blank nodes or variables (*identification condition*). The insertion is then obtained by a pushout (PO), i. e., a disjoint union of D and R over the interface I as common subpart.

Recall that the NAC N_1 forbids the application of the rule BlankAuthorToNewURI in Fig. 6 if there is already another URI for that author. Therefore, we need a second rule BlankAuthorToExURI, shown in Fig. 8, to allow the replacement of a blank node author representation by an already existing URI.

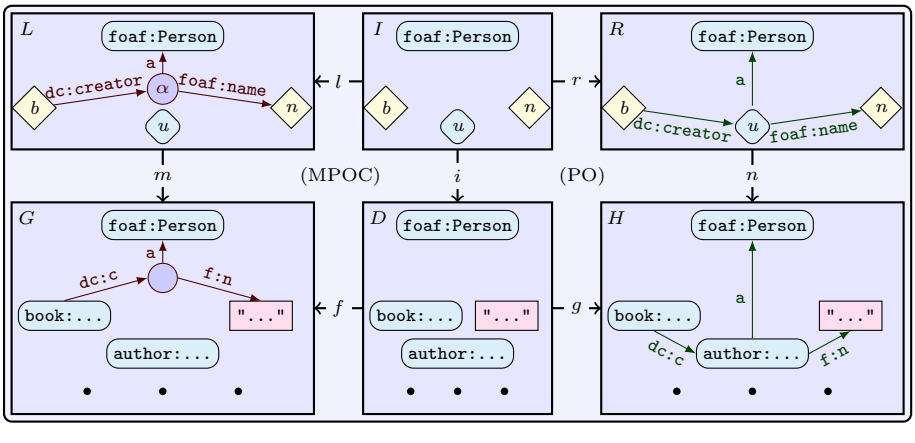

Fig. 7. Application of RDF transformation rule

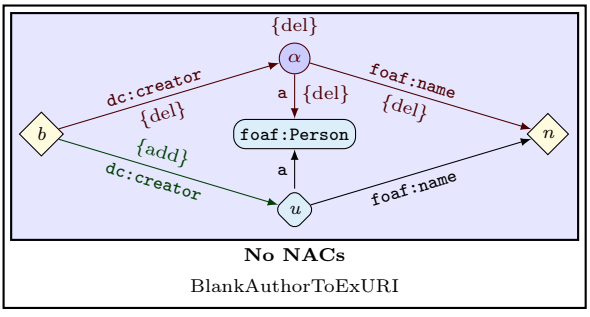

Fig. 8. RDF transformation rule for replacing blank node by existing URI

This rule uses a URI—represented by the URI variable u—that is already of type `foaf:Person` and connected to a `foaf:name` in the match of the left-hand side.

For the second example operation, adding a book to the catalogue, we specify several transformation rules in Fig. 9. The first one, AddBook, is used to insert the minimal required information, a title and the price, into the RDF data store. The NACs ensure that the URI for the new book is not already in use. The rules AddAuthorToBook and AddPublisherToBook can then be used to add authors and a publisher as optional information.

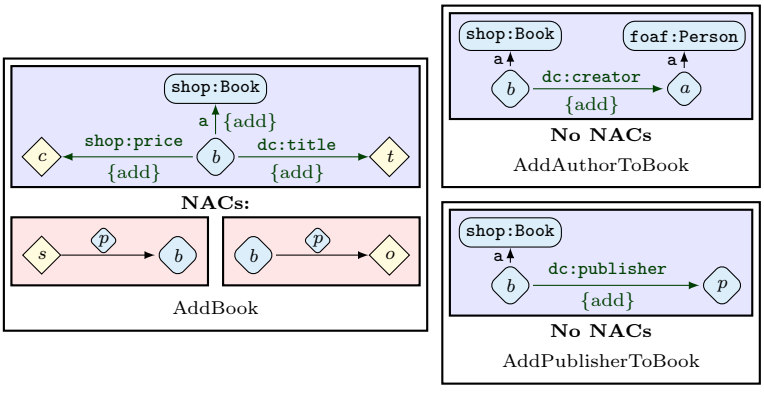

Fig. 9. RDF transformation rules for adding a book to the catalogue

Finally, the rather simple rule AddBookToCart adds a book to the shopping cart of a user (identified by the match of the URI variable u).

In this section, we have seen how graph transformation rules are formally structured and applied to RDF graphs as formal models of triple stores. Moreover, we have modelled the example operations from the application scenario

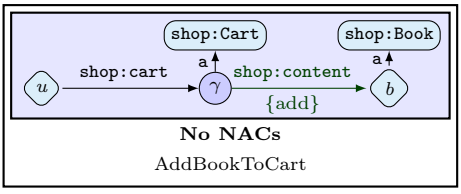

Fig. 10. RDF transformation rule for adding a book to a cart

by transformation rules. In the next section, we will sketch an architecture for providing and executing graph transformation rules using standard (Semantic) Web techniques.

5 Web Application Architecture for Algebraic Graph Transformation

Transformation rules can themselves be represented in RDF. Figure 11 shows an RDF representation of the rule BlankAuthorToExURI from Fig. 8. The rule is identified by the URI `http://shop.example.com/rules/BlankAuthorToExURI` and its constituents are connected to it by the `gt:preserve`, `gt:delete` and `gt:create` predicates, where triples are reified using the `gt:subject`, `gt:pred` and `gt:object` predicates. Variables and blank nodes in the rule are represented by blank nodes with corresponding types `gt:UVar`, `gt:Var` and `gt:Blank`, where variables also have a `gt:name`, which will be used to identify them in requests.

```
@prefix rule: <http://shop.example.com/rules/>.
@prefix gt: <http://rdfgratra.example.org/>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
rule:BlankAuthorToExURI a gt:Rule;
                        gt:preserve [ a gt:Triple;
                                      gt:subject _:1; gt:pred a; gt:object foaf:Person ],
                                    [ a gt:Triple;
                                      gt:subject _:1; gt:pred foaf:name; gt:object _:2 ];
                        gt:delete [ a gt:Triple;
                                    gt:subject _:3; gt:pred dc:creator; gt:object _:4 ],
                                  [ a gt:Triple;
                                    gt:subject _:4; gt:pred a; gt:object foaf:Person ],
                                  [ a gt:Triple;
                                    gt:subject _:4; gt:pred foaf:name; gt:object _:2 ],
                                  _:4;
                        gt:add [ a gt:Triple;
                                 gt:subject _:3; gt:pred dc:creator; gt:object _:1 ].
_:1 a gt:UVar; gt:name "author".
_:2 a gt:Var; gt:name "name".
_:3 a gt:Var; gt:name "book".
_:4 a gt:Blank.
```

Fig. 11. RDF representation of transformation rule BlankAuthorToExURI

Such a representation of the rule shall be the response to a GET request on its URI. Its execution is—in accordance with the principles of RESTful Web applications in the sense of [10]—triggered by a POST request. A (partial) match is given by using the `gt:name` attributes as the names of POST variables. For a given partial match, there are essentially three possibilities:

- The match is total or there is only one possibility to complete it to an applicable total match. In this case, the rule is executed and results in a response with the image of the right-hand side as content.
- The match is already not allowed (even if it still might be partial) due to a NAC, the dangling or the identification condition. This results in a description of the violated condition in the response.
- There are multiple possible completions of the given partial match. The response contains the possible completions of the match (if they are already restricted enough to make this feasible). It would also be reasonable to offer a possibility to execute a rule on all possible extensions of a given partial match—especially if they only differ in the matches of blank nodes, which cannot be identified from the outside.

The storage of the applicable transformation rules on the server allows for an advanced security architecture. It can be decided per transformation rule which users are allowed to execute it—akin to stored procedures in relational databases. If the users are also adminstrated in the RDF data store then this is easily achieved by a `gt:allowed` predicate and corresponding triples relating the rules and the users or user groups.

A more intricate problem arises in the case of the rule AddBookToCart in Fig. 10. All customers should be able to execute this rule, but only on their own cart. This can be solved by introducing a `gt:bind` predicate for variables, which is a replacement for `gt:name` and has the effect that the match for a variable is not chosen by the user but by the environment—in this case binding u to the URI of the user who is currently logged in.

In the following section, we will compare the solution by SPARQL/Update from Sect. 3 and the proposed solution by the help of formal techniques using algebraic graph transformation, given in Sect. 4 and the current section.

6 Comparison and Related Work

The differences between the SPARQL/Update approach and our algebraic graph transformation approach for modifying RDF graphs are summarised in Table 1. SPARQL/Update queries are supposed to be generated by applications in an ad hoc manner, while graph transformation rules are pre-defined and applied by providing a match of the left-hand side. The language primitives of SPARQL/Update are derived from the needs arising in practice, while the concepts of graph transformation are derived from a category theoretical model of transformation. All in all, this leads to SPARQL/Update being a language that is supposed to be used in application engineering, while algebraic graph transformation enables

Table 1. Comparison of solutions

SPARQL/Update	Algebraic Graph Transformation
Ad hoc generation of queries for *specific* modifications	Application of *generic*, *pre-defined* rules via matches
Practice-driven: language primitives derived from practical needs	*Theory-driven:* concepts derived from categorial model of transformations
Language used in *application engineering*	Formalism enabling *rule engineering*

rule engineering, which can be done independently of the application context on a higher level of abstraction.

Moreover, the formal apparatus of graph transformation allows to reason about the effects of transformations ex ante, define languages of allowed graph structures by grammars, compose transformation rules to model complex operations and analyse dependencies in transformation sequences.

In [11], the *Delta* ontology for differences between RDF graphs is developed. This approach leads to a structure that is very similar to our notion of RDF transformation rules. A preserved graph (part) is connected to a deleted part by a `delta:deletion` predicate and to an inserted part by a `delta:insertion` predicate, where the possibility to nest graphs inside graphs in N3 is used. While our motivation is to provide a generic transformation facility, Delta puts its focus on documenting and possibly reapplying changes between RDF graphs. Up to now, a formal semantics for Delta differences is not formulated. Furthermore, they are not intended to be used in multiple situations, which is why an equivalent to the application of a rule *via a match* is missing.

In both cases, SPARQL/Update as well as Delta, it might be worthwhile to consider algebraic graph transformations as a formal foundation for the concrete languages.

In [12], the *Griwes* framework for knowledge representation languages is proposed. It aims at providing a common formal model for formalisms such as RDF or Conceptual Graphs. Griwes provides a notion of mappings and proofs between graphs, which corresponds to the morphisms in our approach, but it does not have a notion of transformation rule or modification operation. While it would be possible to formulate a similar transformation concept on the level of Griwes, we prefer to work directly on the RDF graphs to reduce the formal overhead.

7 Conclusion and Future Work

In this paper, we have proposed algebraic graph transformation as a means to modify on RDF data stores. This leads to a shift from modifications that are developed as part of the Web application engineering to modifications that are specified in a dedicated rule engineering process and, thus, facilitate among other things the reuse and maintenance of transformation rules.

In future work, the relations between transformations and inferences should be examined. More specifically, situations in which inferred triples are deleted shall be analysed, since this leads to the counter-intuitive result that the deleted triple will be inferred again at the next possibility and is, thus, not effectively deleted. Similar situations also arise in SPARQL/Update.

On the theoretical side, the rich theory that is available in the field of algebraic transformation can be used. However, as already shown in [5,6], this requires some adaptations.

References

1. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax (W3C Recommendation). In: World Wide Web Consortium, W3C (February 2004), http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/
2. Seaborne, A., et al.: SPARQL Update: A language for updating RDF graphs (W3C Member Submission). In: World Wide Web Consortium, W3C (July 2008), http://www.w3.org/Submissions/SPARQL-Update/
3. Schenk, S., Gearon, P.: SPARQL 1.1 Update (W3C Working Draft). In: World Wide Web Consortium, W3C (October 2009), http://www.w3.org/TR/2009/WD-sparql11-update-20091022/
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006), doi:10.1007/3-540-31188-2
5. Braatz, B., Brandt, C.: Graph transformations for the Resource Description Framework. In: Ermel, C., Heckel, R., de Lara, J. (eds.) Proc. GT-VMT 2008. Electronic Communications of the EASST, vol. 10 (2008), http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/158
6. Braatz, B.: Formal Modelling and Application of Graph Transformations in the Resource Description Framework. Dissertation, Technische Universität Berlin (2009)
7. Berners-Lee, T.: Notation 3: A readable language for data on the Web. World Wide Web Consortium, W3C (October 2007), http://www.w3.org/DesignIssues/Notation3
8. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (W3C Recommendation). In: World Wide Web Consortium, W3C (January 2008), http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
9. Adámek, J., Herrlich, H., Strecker, G.E.: Abstract and Concrete Categories: The Joy of Cats. Dover, New York (2009), http://katmat.math.unibremen.de/acc/
10. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Dissertation, University of California, Irvine (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
11. Berners-Lee, T., Connolly, D.: Delta: an ontology for the distribution of differences between RDF graphs. In: World Wide Web Consortium, W3C (August 2009), http://www.w3.org/DesignIssues/Diff
12. Baget, J.F., et al.: Griwes: Generic model and preliminary specifications for a graph-based knowledge representation toolkit. In: Eklund, P., Haemmerlé, O. (eds.) ICCS 2008. LNCS (LNAI), vol. 5113, pp. 297–310. Springer, Heidelberg (2008), doi:10.1007/978-3-540-70596-3_21