RUL: A Declarative Update Language for RDF

M. Magiridou[1]*, S. Sahtouris[2], V. Christophides[2], and M. Koubarakis[1]*

[1] Dept. of Electronic and Computer Engineering, Technical University of Crete
GR73100 Chania, Greece
{magiridou,manolis}@intelligence.tuc.gr
[2] Institute of Computer Science FORTH, Vassilika Vouton P.O. 1385 Heraclion, Greece
{saxtouri,christop}@ics.forth.gr

Abstract. We propose a declarative update language for RDF graphs which is based on the paradigms of query and view languages RQL and RVL. Our language, called RUL, ensures that the execution of the update primitives on nodes and arcs neither violates the semantics of the RDF model nor the semantics of the given RDFS schema. In addition, RUL supports fine-grained updates at the class and property instance level, set-oriented updates with a deterministic semantics and takes benefit of the full expressive power of RQL for restricting the range of variables to nodes and arcs of RDF graphs.

1 Introduction

Semantic Web applications are striving nowdays for managing changes of persistent resource descriptions created according to RDFS schemata [1, 2]. The majority of ontology-based authoring and annotation tools [3] requires first to manually edit the resource descriptions and thereafter reloading them into an RDF Store from scratch. This approach offers rather limited functionality especially in the case of deletions and modifications. To overcome these limitations, some RDF Stores [4] have implemented suitable update APIs [5–8]. However, forcing developers to code in advance all possible updates of resource descriptions (using these APIs) is not a viable solution for dynamic Semantic Web applications employing non trivial RDFS schemata. In this context, designing a declarative update language offering complete (i.e., all valid RDF changes should be specifiable by one or by a sequence of update primitives from a minimal set) and sound (i.e., every primitive is guaranteed to maintain consistency of resource descriptions w.r.t. the employed RDFS schemata) primitives is a challenging issue.

In this paper, we propose a declarative update language for RDF graphs which is based on the paradigms of query and view languages RQL [9] and RVL [10]. Our language, called RUL, ensures that the execution of the update primitives on nodes and arcs neither violates the semantics of the RDF model (e.g., insert a property as an instance of a class) nor the semantics of a specific RDFS schema (e.g., modify the subject of a property with a resource not classified under its domain class). This main design choice has been made in order to take into account the fact that updates are fairly destructive operations and change

* Supported in part by European project OntoGrid - http://www.ontogrid.net.

the state of an RDF graph. Thus, type safety for updates is even more important than type safety for queries. The more errors we can catch at compile time the less costly runtime checks (and possibly expensive rollbacks) we need. The rest of RULs design choices concern (a) the *granularity* of the supported update primitives; (b) the *deterministic or not* behavior of the executed sequences of update statements; and (c) the *smooth integration* with an underlying RDF/S query language. To the best of our knowledge, RUL is the first declarative language supporting fine-grained updates at the class and property instance level, has a deterministic semantics for set-oriented updates and takes benefit of the full expressive power of RQL for restricting the range of variables to nodes and arcs of RDF data graphs. However, our design can be also immediately transferred to other RDF query languages (e.g., RDQL [11], or SPARQL [12]) offering less expressive pattern matching capabilities [13].

None of the RDF update languages proposed so far [14, 15] supports the aforementioned functionality. The most interesting proposal is MEL that has been developed in the framework of QEL and it is based on Datalog [14]. MEL primitive commands consist of a statement specification and an optional query constraint, declared as a QEL query. The granularity of the operations follows a subgraph-centered approach but consistency of updates w.r.t the employed RDFS schemata is not respected. Furthermore, no formal semantics or detailed behavior description have been given for MEL. The rdfDB Query Language [15] supports SQL-like updates (insert and delete) by following a statement-centered approach and does not integrate smoothly with the query language. In fact, the update operations can affect only specific statements without variables and thus their execution semantics is trivial.

From knowledge representation languages in the semantic data modelling tradition, Telos [16] is probably the closest to RDF. Telos has inspired the RDF data model behind the query language RQL and the hybrid framework of [17]. Work on update languages for Telos is reported in [16, 18, 19] However, the statements UNTELL and RETELL discussed in these papers concentrate on the temporal features of Telos and pay no attention to the many issues regarding update side-effects as discussed in this paper.

The rest of the paper is organized as follows. Section 2 introduces the syntax of RUL in an incremental, informal way by giving examples and intuitive explanations while Section 3 clarifies RULs formal semantics and in particular its deterministic behavior for set-oriented updates. Our conclusions as well as some challenges for future work are given in Section 4.

2 The Syntax of RUL

RUL can be used to express updates to RDF graphs i.e., insertions, deletions and modifications of nodes and arcs. We consider an RDF graph to follow the formal model for RDF proposed in [20]. The main constraints this model imposes to RDF Semantics [21] can be summarized as follows: *the domain and range of properties should always be defined, both of these declarations have to be unique* and *class and property definitions have to be complete.*

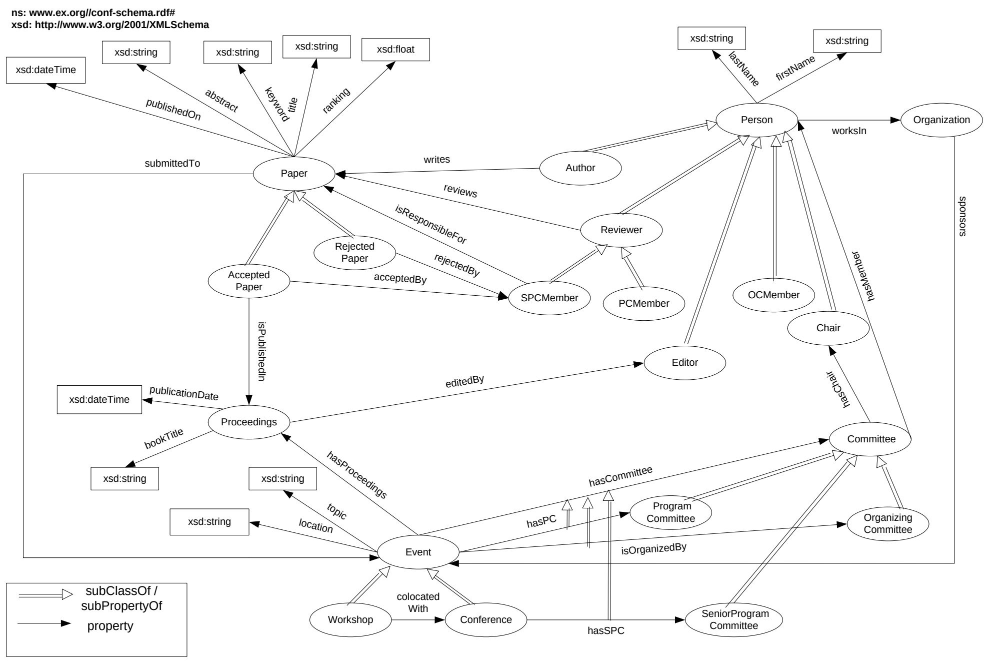

Fig. 1. RDF Schema in graphical form for the conference organization example

In this section, we present the syntax of RUL in an incremental, informal way by giving examples and intuitive explanations based on the RDF schema of Figure 1 dealing with the organization of scientific conferences, and Figure 2 where the effects and side-effects of each operation are analyzed in detail. Section 3 presents the formal semantics of RUL.

We assume that the vocabularies used in the RDF graphs have been defined using RDF Schema. RUL does not deal with schema updates. We also do not deal with blank nodes, containers, collections or reification in this paper.

2.1 Updating Class Instances

Instances of classes can be updated using the `INSERT, DELETE` or `MODIFY` statements. The syntax of the `INSERT` statement is as follows:

```
INSERT QualClassName(ResourceExp)
[FROM VariableBinding]  [WHERE Filtering] [USING NAMESPACE NamespaceDefs]
```

The `INSERT` operation introduces new nodes in an RDF graph and classifies them, or inserts new classification links for existing nodes. The expression `ResourceExp` denotes a node and can be a constant URI or a variable. In the former case, `ResourceExp` determines a unique graph node, while in the latter, the clause `FROM` determines the bindings of this variable (i.e., a set of nodes) as in RQL. The expression `QualClassName` denotes the class to which the new nodes will become instances or to which the new classification links from existing nodes will be created.

The clause `WHERE` gives as usual the filtering conditions for the variables bindings introduced in the clause `FROM`. The clause `USING NAMESPACE` gives a list of namespaces that disambiguate the use of names in the other clauses. The clauses `FROM, WHERE` and `USING NAMESPACE` are optional. In the rest of this paper, we show the `USING NAMESPACE` clause when we are presenting the syntax of RUL but avoid any namespace information in the examples for reasons of brevity (i.e., all the names employed in the examples are unique and they are defined in the schema namespace `ns` of Figure 1).

As in RQL, RUL distinguish between direct and indirect instances of a class `C` or property `P` (equivalently, between direct and indirect instantiation links). A resource node `r` is a *direct* instance of class `C` if it has been introduced in the graph by an appropriate update statement. A resource node `r` is an *indirect* instance of class `C` if r is a direct instance of a subclass of `C`. The definition is similar for properties. An RDF graph *has no redundancies* with respect to instantiation if there is no instance of a class or a property that is both a direct and an indirect instance. All the update operations defined below result in RDF graphs with no redundancies with respect to instantiation.

The *effects* and *side-effects* of an `INSERT` operation with the above syntax are presented graphically in Figure 2(I). If node `ResourceExp` exists in the graph and it is classified under a superclass of `QualClassName` (Case (I.1) in Figure 2), the effect of `INSERT` is that a new classification link is inserted between `ResourceExp` and `QualClassName`. In this case, the operation has the side-effect that the prior classification link is deleted (since it is implied by the new classification link). On the other hand, if `ResourceExp` exists in the graph and it is classified under a subclass of `QualClassName` (Case (I.3) where `D` is a subclass of `C`), the `INSERT` operation has no effects. Obviously, if the node exists as a direct instance of `QualClassName`, the operation has no effects too. Finally, if node `ResourceExp` exists in the graph and it is classified under a class which is not related through a subclass relation to `QualClassName` (Case (I.2)), the result is a multi-classified node (`&d1` is classified both under `B` and `D` classes) without any side-effects.

Example 1. Make the resource with URI `http://www.ex.org/paper1.pdf` an instance of the class `AcceptedPaper`:

```
INSERT AcceptedPaper(&http://www.ex.org/paper1.pdf)
```

As we explained above, this update operation will be effective only if the resource node `paper1.pdf` is not already an instance of class `AcceptedPaper` or one of its subclasses (if it had any). In other words, the execution of an `INSERT` operation leaves us with an RDF graph with no redundancies with respect to instantiation.

Example 2. Classify as reviewers all members of the OC of ISWC05:

```
INSERT Reviewer(X)
FROM {Y}isOrganizedBy.hasMember{X;OCMember}
WHERE Y = &http://www.iswc05.org
```

The above example demonstrates the use of variables in the `INSERT` clause and the use of RQL *path expressions* for navigating RDF graphs in the `FROM` clause. More precisely, variable `X` will be range restricted to instances of class `OCMember`

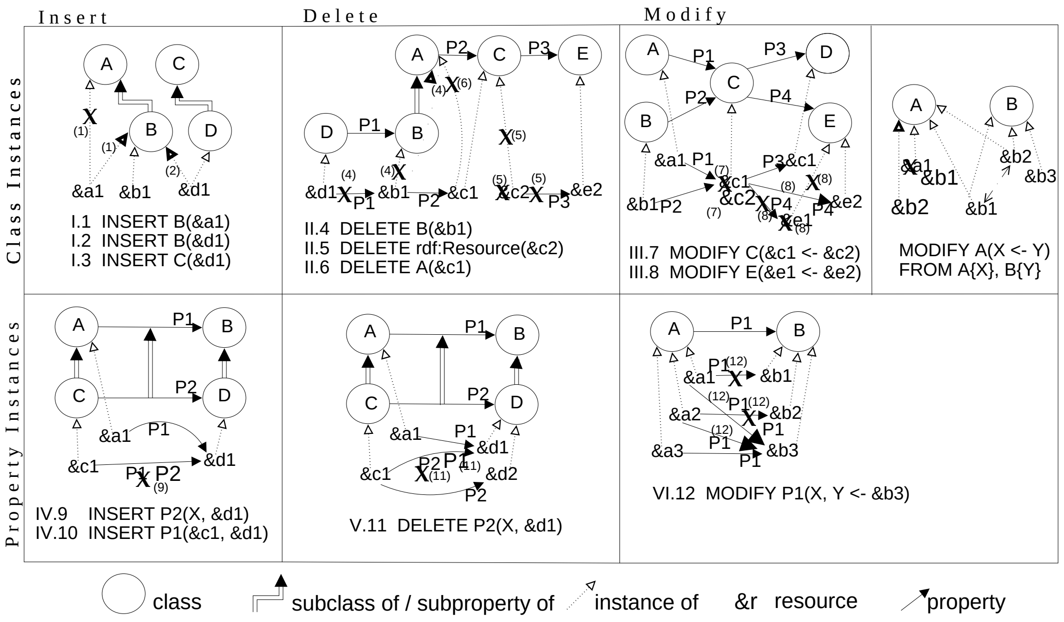

Fig. 2. RUL operations effects and side-effects

involved in the `OrganizingCommittee` of the ISWC05 `Event`. This update operation will multiply classify `OCMember` instances under the class `Reviewer`.

The syntax of the `DELETE` operation is as follows:

```
DELETE QualClassName(ResourceExp)
[FROM VariableBinding]  [WHERE Filtering]  [USING NAMESPACE NamespaceDefs]
```

The `DELETE` operation deletes classification links and possibly nodes from an RDF graph (Figure 2 (II)). The expression `ResourceExp`, which denotes the node from which the classification link to be deleted originates, can be a URI or a variable. The effect of the `DELETE` operation is to remove the direct or indirect classification link of `ResourceExp` to class `QualClassName` and replace it by the link of `ResourceExp` to all the immediate super-classes of `QualClassName` if any (e.g., in Figure 2(II.4), `&b1` is now classified under class `A`). If `ResourceExp` is multi-classified (e.g., `&c1` in Figure 2(II.6)), then the classification link of `ResourceExp` to `QualClassName` is deleted without being replaced by another. Finally, if `QualClassName` is the top of the class hierarchy `rdf:Resource`, the effect is the deletion of `ResourceExp` node along with all its classification links (Figure 2(II.5)).

It should be stressed that, all classification links that are added by a `DELETE` operation must take the semantics of `INSERT` into account, so that the resulting RDF graph remains without redundancies.

The side effects of `DELETE` in any of the above cases are caused by the changes in the classification of a node. To be more specific, all property arcs emanating from the note denoted by `ResourceExp` that have as domain (or range) a class, to which `ResourceExp` is no longer an instance, are also deleted. These side-effects are necessary to keep the graph consistent, since `ResourceExp` does no longer belong to the declared classification. To illustrate these, consider the properties `P1` and `P3` in Figure 2(II), which are deleted when the respective classification links are removed (statements (II.4) and (II.6)).

Example 3. Delete all papers submitted by the PC chair(s) of ISWC05:

```
DELETE Paper(X)
FROM {Y}writes{X}, {Z;Conference}hasPC.hasChair{Y}
WHERE Z=&http://www.iswc05.org
```

The above `DELETE` operation will be effective only if the node bindings of variable `X` are classified under the class `ns:Paper` or one of its subclasses (e.g., `AcceptedPaper`). It is worth noticing that these nodes will still be present in the output RDF graph of the previous update operation, but only as instances of the top class `rdf:Resource` (since `ns:Paper` has no other superclasses).

Finally, the syntax of the `MODIFY` operation is:

```
MODIFY QualClassName(OldResourceExp <- NewResourceExp)
[FROM VariableBinding]  [WHERE Filtering]  [USING NAMESPACE NamespaceDefs]
```

The expressions `OldResourceExp` and `NewResourceExp` can be constants or variables as in other statements. The arrow `<-` has the meaning of an *assignment* operation. The `MODIFY` operation is not a sequence of `DELETE` and `INSERT`. The effect of the `MODIFY` operation (Figure 2(III)) is to completely remove the node(s) denoted by `OldResourceExp` and then insert the node(s) denoted by `NewResourceExp` as an instance of `QualClassName`. The insertion of `NewResourceExp` has the same semantics as the `INSERT` operation presented earlier (see cases III.7 and III.8). The first side effect of `MODIFY` is that all properties emanating from (or ending at) the resource denoted by `OldResourceExp` are completely removed. The other side effect is that the previously removed properties became now properties emanating from (or ending at) the resource denoted by `NewResourceExp` (e.g., in Case III.8, property arc `P4` which ends at `&e1`, is removed, while another property arc `P4` which ends at `&e2`, is inserted).

Example 4. The information that `paper1.pdf` is an accepted paper is incorrect. The correct information is that `paper101.pdf` has been accepted.

```
MODIFY AcceptedPaper(&http://www.ex.org/paper1.pdf <- &http://www.ex.org/paper101.pdf)
```

If `paper1.pdf` had title “The language SQL”, we could equivalently write:

```
MODIFY AcceptedPaper(X <- &http://www.ex.org/paper101.pdf)
FROM {X}title{Y}
WHERE Y="The language SQL"
```

2.2 Updating Property Instances

The `INSERT`, `DELETE` and `MODIFY` statements can also be used to update the properties of resources i.e., arcs in an RDF graph. The syntax of the `INSERT` statement in this case is as follows:

```
INSERT QualPropertyName(SubjectExp, ObjectExp)
[FROM VariableBinding] [WHERE Filtering] [USING NAMESPACE NamespaceDefs]
```

The above INSERT operation adds to resource node SubjectExp a new property arc that is an instance of property QualPropertyName and has value ObjectExp. SubjectExp and ObjectExp can be constants or variables with bindings determined in the FROM clause. In both cases RQL typing rules for triples must be respected: SubjectExp must evaluate to a URI, instance of the domain of property QualPropertyName, and ObjectExp must evaluate to a URI or literal value instance of the range of property QualPropertyName.

We now detail the semantics of this operation by referring to Figure 2(IV). The variable X, that has not been given a range, should be assumed to range over all nodes shown in the figure. As in the case of resources, if a property arc from SubjectExp to ObjectExp exists and it is an instance of a super-property of QualPropertyName (Figure 2(IV.9)), then the operation's effect is the deletion of the instantiation link of the arc and the introduction of a new link to QualPropertyName (e.g., the arc from &c1 to &d1 becomes an instance of property P2). However, when SubjectExp and ObjectExp are not instances of the domain and range of QualPropertyName this operation has no effect (e.g., P1 between &a1 and &d1 is not affected by the insertion IV.9). If the property arc exists as an instance of a subproperty of QualPropertyName (Figure 2 (IV.10)), then the operation has also no effect. It is obvious that there are no side-effects in this operation.

Example 5. Make "IR" a keyword of paper http://www.ex.org/paper1.pdf.

```
INSERT keyword(&http://www.ex.org/paper1.pdf, "IR")
```

Example 6. Make Oracle a sponsor of every database conference.

```
INSERT sponsors(&http://www.oracle.com, X)
FROM {X;Conference}topic{Y}
WHERE Y like "*database*"
```

Example 7. Make editors of the proceedings of ISWC05 the chair(s) of the PC and the chair(s) of the OC.

```
INSERT editedBy(X,Y)
FROM {Q}hasProceedings{X}, {Q}@P.hasChair{Y},
WHERE Q = &http://www.iswc05.org and (@P=isOrganizedBy or @P=hasPC)
```

This example demonstrates the use of *schema querying* in the FROM clause of RUL. Variables prefixed by @ are RQL *property variables* implicitly restricted to range over the set of all data properties.

The syntax of the DELETE operation is as follows:

```
DELETE QualPropertyName(SubjectExp, ObjectExp)
[FROM VariableBinding]  [WHERE Filtering]  [USING NAMESPACE NamespaceDefs]
```

As in the case of resources, the DELETE operation (Figure 2(V)) removes essentially the instantiation link between QualPropertyName and the property arc from SubjectExp to ObjectExp (e.g., the arc from &c1 to &d1 is not anymore an instance of `P2`) and inserts a link from the arc to the super-property of `QualPropertyName` (e.g., the arc from `&c1` to `&c2` becomes an instance of `P1`), as we discussed in the property `INSERT` operation. If the arc is not an instance of `QualPropertyName` (e.g. the arc from `&a1` to `&d1` not classified under `P2`), then the operation has no effect. This update operation has also no side-effects.

Example 8. Delete keyword "IR" from paper `http://www.ex.org/paper2.pdf`:

```
DELETE keyword(&http://www.ex.org/paper2.pdf, "IR")
```

Example 9. Remove assigned papers on web services from reviewer Smith:

```
DELETE reviews(&http://www.uni-ex.edu/~smith, X)
FROM {X}paperKeyword{Y}
WHERE Y like "*web services*"
```

Example 10. Delete all sponsors of ISWC05:

```
DELETE sponsors(X, &http://www.iswc05.org)
FROM Organization{X}
```

The syntax of the `MODIFY` operation is:

```
MODIFY QualPropertyName([OldSubjectExp <-] NewSubjectExp,
                        [OldObjectExp  <-] NewObjectExp)
[FROM VariableBinding] [WHERE Filtering] [USING NAMESPACE NamespaceDefs]
```

As we can see in Figure 2(VI), the effect of the operation is to delete the arc between the resources denoted by the `OldSubjectExp` and `OldObjectExp` and insert a new arc from `NewSubjectExp` to `NewObjectExp` (e.g., the arc between `&a1` and `&b1` is removed and a new arc between `&a1` and `&b3` is inserted). If `OldSubjectExp` (resp. `OldObjectExp`) or `NewSubjectExp` (resp. `NewObjectExp`) is not an instance of a class in the domain (resp. range) of `QualPropertyName`, the operation has no effect. If the arc from `NewSubjectExp` to `NewLObjectExp` already exists and it is an instance of `QualPropertyName`, it is not inserted (e.g., the arc between `&a2` and `&b3`), so that redundancies are avoided, as we discussed in the property `INSERT` operation. No other arcs are affected as a side-effect of the above operation.

Example 11. Change the keyword "IR" to "Information Retrieval" in the papers where this keyword appears:

```
MODIFY keyword(X, "IR" <- "Information Retrieval")
FROM Paper{X}
```

Example 12. Make the publication date of every accepted paper to be the same as the publication date of the proceedings where it is published:

```
MODIFY publishedOn(Y, Z <- X)
FROM  {Y;AcceptedPaper}isPublishedIn.publicationDate{X}, {Y}publishedOn{Z}
```

The above examples demonstrate the modification of a property's object. The following example illustrates a case where the subject of a property is updated.

Example 13. Pass all the reviews to be done by Prof. Smith to his Ph.D. student Jones:

```
MODIFY reviews(&http://www.ex.org/~smith <- &http://www.ex.org/~jones, Y)
FROM Paper{Y}
```

Example 14. The information "Oracle sponsors WWW 2005" in our graph is incorrect. The correct information is "Google sponsors ISWC 2005".

```
MODIFY sponsors(&http://www.oracle.com <- &http://www.google.com,
                &http://www.www05.org <- &http://www.iswc05.org)
```

This example demonstrates the change of both subject and object of a property.

We close this section by pointing out that it is a design choice of RUL to have one syntax for updates of instantiation links (unary predicates) and a *different syntax* for updates of property arcs (binary predicates) to remind the user of the *different semantics* of these operations (i.e., we do not believe that a uniform syntax based on triples and `rdf:type` would be appropriate for RUL).

2.3 More Expressive Updates

The syntax of RUL presented above allows us to express two kinds of updates: *primitive* ones where a node or arc of an RDF graph is inserted or deleted (with appropriate side-effects), and *set-oriented* ones where an atomic update of the same kind (e.g., an insertion) is performed repeatedly for all resource tuples calculated by evaluating the `FROM` and `WHERE` clauses of an `INSERT, DELETE` or `MODIFY` statement. Of course, by writing multiple RUL statements, we can also express *sequences* of such updates. In this section, we extend the above syntax to be able to express sequences of primitive updates inside a *single* RUL statement, and show with examples why such an extension is a useful feature of RUL. A discussion of the problems involved and how they can be addressed effectively is postponed until Section 3.3.

The first extension that we propose is to allow multiple atomic formulas, in an `INSERT, DELETE` or `MODIFY` clause. In this way, we can express sequences of primitive updates of the *same* kind.

Example 15. Make resource `&http://www.ex.org/paper3.pdf` authored by Smith an instance of class `Paper`.

```
INSERT Paper(&http://www.ex.org/paper3.pdf),
       writes(&http://www.uni-ex.edu/~smith, &http://www.ex.org/paper3.pdf)
```

Note that even in sequences of primitive insertions as in the above example, the order of execution of each individual update *does* matter (we cannot insert a property `writes` for resource `paper3.pdf` before we make it an instance of the range of `writes`). This is in direct contrast with updates in relational languages [22, 23] where order does not matter in sequences of updates of the same kind. Thus, the order of execution for update statements with multiple predicates is from left to right and the comma operator signifies sequence.

Example 16. Reject all papers with ranking less than 4, and add the SPC member responsible for the paper as the person who made the final recommendation.

```
INSERT RejectedPaper(X), rejectedBy(X,Y)
FROM {X;Paper}ranking{Z},
     {X}submittedTo.hasSPC.hasMember{Y;SPCMember}, {Y}isResponsibleFor{X}
WHERE Z < 4
```

This example shows clearly why the proposed enhancement of the RUL syntax is useful. In this case additions to the graph come "in pairs"; thus, the example is impossible to express without variables and sequencing.

Apart from sequences of updates of the same kind, RUL can also express sequences of updates of *different* kinds. This is done by allowing multiple `INSERT`, `DELETE` or `MODIFY` clauses before the `FROM` clause of an update statement.

Example 17. Form the Program Committee of ISWC06 by taking the set of all PC members of ISWC05 except those that reviewed less than 5 papers for ISWC05, and adding to this set the members of the OC of ISWC05.

```
INSERT hasPCMember(&http://www.iswc06.org#pc, X)
DELETE hasPCMember(&http://www.iswc06.org#pc, Y)
INSERT hasPCMember(&http://www.iswc06.org#pc, Z)
FROM   {W}hasPCMember{X}, {W}hasPCMember{Y},
       {W}hasOCMember{Z}
WHERE  W = &http://www.iswc05.org#pc
       and count(SELECT Q FROM {Y}reviews{Q}, {Q}submittedTo{W}) <5
```

This last extension to the syntax of RUL also allow us to express updates with effects that depend on the order of execution of the primitive updates captured by the clauses `INSERT`, `DELETE` or `MODIFY` (e.g., in Example 17, all the Program Committee members of ISWC05 have to be made Program Committee members for ISWC06 *before* those of them that reviewed less than 5 papers for ISWC05 are deleted). The order of execution for multiple update clauses in an RUL update statement is from top to bottom. Thus, update clauses with multiple predicates can be trivially translated into sequences of update statements with a single predicate. We will discuss these issues in detail in Section 3.3 where the semantics of set-oriented updates are discussed in every detail.

2.4 Safety

The presence of variables in RUL statements forces us to impose an easily verifiable syntactic notion of safety as in relational updates [22]. An RUL statement is *safe* if all the variables appearing in `INSERT`, `DELETE` or `MODIFY` clauses also appear in the `FROM` clause of the statement. Thus, if an RUL statement is safe, no new values can be inserted in the graph except the ones present in the update statement itself.

Example 18. Let us revisit Example 11. If the user writes the unsafe statement

```
MODIFY keyword(X, "IR" <- "Information Retrieval")
```

an RUL compiler can easily translate this into the safe statement of Example 11 since `domain(keyword)=Paper`. This is one of the benefits of adopting the RQL typing framework [9].

3 The Semantics of RUL

In this section we give a formal semantics to RUL. We start by defining the concepts of RDF that we need using the formal model introduced in [9]. The important contribution of [9] is the introduction of a rich type system for RDF and RDFS that has been proved valuable in the implementation of RQL. Because RUL updates are destructive operations that change the state of an RDF graph, type safety for RUL updates is even more important than type safety for RQL queries. The more errors we can catch at compile time, the less costly runtime checks (and possibly expensive rollbacks) we will need.

We start by defining the concepts of RDF graph and RDFS graph. We slightly modify the definitions of [9] to cover only the concepts of RDF used in this paper (we do not deal with blank nodes, containers, collections or reification).

Let LT be the set of XML Schema data types that can be used in RDF. Let T be the set of types in the RDF/S type system defined in [9]. Let $Values(T)$ be the set that includes all typed literals with types from T and all URIs.

Definition 1. *An* RDFS graph *is a 6-tuple $S = (VS, ES, C, P, \prec, \Theta, \Lambda)$ where VS is a set of nodes, $ES \subseteq V \times V$ is a set of edges, C is a set of class names, P is a set of property names, $\prec$ is a partial order on $C \cup P$, $\Theta : VS \cup ES \to C \cup P$ is a function mapping nodes to classes and edges to properties, and $\Lambda : VS \cup ES \to T$ is a typing function that returns the type of each node or edge.*

Definition 2. *An* RDF graph *over the RDFS graph $(VS, ES, C, P, \prec, \Theta, \Lambda)$ is a quadruple $G = (V, E, \nu, \lambda)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, $\nu : V \to Values(T)$ is a value function that assigns a value from $Values(T)$ to each node in V and $\lambda : V \cup E \to 2^{C \cup P} \cup LT$ is a typing function which satisfies the following: (i) For each node a in V, λ returns a set of class or data type names $c \in C \cup LT$ such that $\nu(a)$ belongs to the interpretation of each c. (ii) For each edge $(a, b) \in E$, λ returns a property name $p \in P$ such that $(\nu(a), \nu(b))$ belongs to the interpretation of p.*

Note that λ contains all classes (resp. properties) that a node (resp. property arc) is an instance of directly or indirectly.

Let *Query* be the set of queries that can be expressed in RQL and $Tuple$ the set of tuples of arbitrary arity formed by elements of $Values(T)$. We assume that the function $\mathcal{E} : Query \times Graph \to Tuple$ gives the semantics of RQL query evaluation as defined in [9]. If q is an RQL query and G is an input RDF graph then the answer to query q is the set of tuples $\mathcal{E}(q, G)$.

Let *Graph* be the set of all possible RDF graphs and *Update* be the set of all possible updates that can be expressed in RUL. The semantics of RUL statements are captured by the semantic function $\mathcal{A} : Update \times Graph \to Graph$. When an update u is applied to a graph $G \in Graph$ and appropriate preconditions are satisfied, u affects a set of nodes and arcs of G and produces a new graph given by $\mathcal{A}(u, G)$.

An RUL update is called *primitive* if it is of the form `INSERT` $c(i)$, `DELETE` $c(i)$, `INSERT` $p(i, i)$, `DELETE` $p(i, j)$ where c is a class, p is a property and i, j are URIs. If τ and τ' are two updates then their *composition* is a *complex* update denoted by $\tau;\tau'$. The semantics of composition are given by the equation $\mathcal{A}(\tau;\tau',G) = \mathcal{A}(\tau',\mathcal{A}(\tau,G))$. Composition is an associative operation thus $\mathcal{A}(\tau_1;\cdots;\tau_n,G) = \mathcal{A}(\tau_n,\mathcal{A}(\ldots,\mathcal{A}(\tau_1,G)))$.

3.1 The Semantics of `INSERT`

Let $G = (V,E,\nu,\lambda)$ be an RDF graph over the RDFS graph $(VS,ES,C,P,\prec,\Theta,\Lambda)$. The effect of update `INSERT` $c(i)$ in G is captured by $\mathcal{A}(\texttt{INSERT}\ c(i),G) = (V',E,\nu',\lambda')$ where V', ν', λ' are defined as follows. If there is no node $a \in V$ with $\nu(a) = i$ then $V' = V \cup \{a_0\}$ where a_0 is a brand new node symbol. Additionally, ν' extends ν such that $\nu'(a_0) = i$ and λ' extends λ such that $\lambda'(a_0) = \{c\}$. On the other hand, if there is a node $a \in V$ with $\nu(a) = i$ then $V' = V$ and ν' is the same as ν. In this case, if $c \in \lambda(a)$ then $\lambda' = \lambda$. If $c \notin \lambda(a)$ but there exist classes $c_1,\ldots,c_k \in \lambda(a)$ such that $c \prec c_1,\ldots,c \prec c_k$ then λ' is the same as λ with the exception that $\lambda'(a) = (\lambda(a) \setminus \{c_1,\ldots,c_k\}) \cup \{c\}$. Otherwise, λ' is the same as λ with the exception that $\lambda'(a) = \lambda(a) \cup \{c\}$.

The preconditions for the execution of the primitive update `INSERT` $p(i_1,i_2)$ in G is that i_1 is a URI and instance of $domain(p)$, and i_2 is a URI or literal and instance of $range(p)$. The effect of this update is captured by $\mathcal{A}(\texttt{INSERT}\ p(i_1,i_2),G) = (V',E',\nu',\lambda')$ where V',E',ν' and λ' are defined as follows. If i_2 is a literal of type t and there is no $a \in V$ such that $\nu(a) = i_2$ then $V' = V \cup \{a_0\}$ where a_0 is a brand new node symbol such that $\nu'(a_0) = i_2$ and $\lambda'(a_0) = t$ (function ν' is identical to ν for all other values in its domain). Otherwise, $V' = V$ and $\nu' = \nu$. Now let $a_1,\ a_2 \in V'$ be nodes such that $\nu(a_1) = i_1$ and $\nu(a_2) = i_2$. If $p \in \lambda((a_1,a_2))$ then $E' = E$ and $\lambda' = \lambda$. If $p \notin \lambda((a_1,a_2))$ but there are properties $p_1,\ldots,p_k \in \lambda((a_1,a_2))$ such that $p \prec p_1,\ldots,p \prec p_k$ then $E' = E$ and λ' is the same as λ with the exception that $\lambda'((a_1,a_2)) = (\lambda((a_1,a_2))\setminus\{p_1,\ldots,p_k\})\cup\{p\}$. Otherwise, $E' = E \cup \{(a_1,a_2)\}$ and λ' is the same as λ with the exception that $\lambda'((a_1,a_2)) = \lambda((a_1,a_2)) \cup \{p\}$.

The semantics of `INSERT` statements with multiple predicates in the `INSERT` clause can now be defined using composition as follows:

$$\mathcal{A}(\texttt{INSERT}\ c_1(i_1),\ldots,c_n(i_n),p_1(j_1,{j_1}'),\ldots,p_m(j_m,{j_m}'),\ D) =$$
$$\mathcal{A}(\texttt{INSERT}\ c_1(i_1);\cdots;\texttt{INSERT}\ c_1(i_k);\texttt{INSERT}p_1(j_1,{j_1}');\cdots;\texttt{INSERT}p_m(j_m,{j_m}'),\ D).$$

3.2 The semantics of `DELETE`

Let $G = (V,E,\nu,\lambda)$ be an RDF graph over the RDFS graph $(VS,ES,C,P,\prec,\Theta,\Lambda)$. The precondition for the execution of the primitive update `DELETE` $c(i)$ in G is that i is an instance of class c. The effect of this update is captured by $\mathcal{A}(\texttt{DELETE}\ c(i),G) = (V',E',\nu,\lambda')$ where V', E',λ' are defined as follows. Let $a \in V$ be the node with $\nu(a) = i$. If $c =$ `rdf:Resource` then $V' = V \setminus \{a\}$ otherwise $V' = V$.

If $c \in \lambda(a)$ then let C_1 be the set $\{c_1 : c_1 \preceq c \wedge c_1 \in \lambda(a)\}$. Then λ' is the same as λ with the exception that $\lambda'(a) = \lambda(a) \setminus C_1$. In addition, $E' = E \setminus (\{(a,b) : \lambda((a,b)) = p \wedge (\exists c_1 \in C_1)domain(p) = c_1\} \cup \{(b,a) :\ \lambda((b,a)) = p \wedge (\exists c_1 \in C_1)range(p) = c_1\})$.

If $c \notin \lambda(a)$ but there is a class c' such that $c' \prec c$ and $c' \in \lambda(a)$ then λ' is the same as λ with the exception that $\lambda'(a) = (\lambda(a) \setminus C_1) \cup C_2$ where $C_1 = \{c_1 \in \lambda(a) : \ c' \preceq c_1 \preceq c\}$ and $C_2 = \{c_2 \in \lambda(a) : \ c \prec c_2 \wedge \neg(\exists c_3)(c \prec c_3 \prec c_2)\}$. In addition, $E' = E \setminus (\{(a,b) : \ \lambda((a,b)) = p \wedge (\exists c_1 \in C_1) domain(p) = c_1\} \cup \{(b,a) : \lambda((b,a)) = p \wedge (\exists c_1 \in C_1) range(p) = c_1\})$.

In a similar way, one can define the semantics of `DELETE` for the case of properties. The semantics of `DELETE` statements with multiple predicates can then be easily defined as in the case of `INSERT` using composition. Finally, the semantics of `MODIFY` can also be defined similarly and are omitted.

3.3 Set-Oriented Updates

The syntax of RUL allows us to express set-oriented updates using variables in the `INSERT, DELETE` or `MODIFY` clause as we showed with examples in Section 2.

The semantics of update statements with a single `INSERT, DELETE` or `MODIFY` clause with variables can easily be defined using the operation of composition and function $\mathcal{E}$ that formalizes the evaluation of RQL queries. For example,

$$\mathcal{A}(\texttt{INSERT}\ c(x)\ \texttt{FROM}\ b(x)\ \texttt{WHERE}\ f(x), D) = \mathcal{A}(\texttt{INSERT}\ c(i_1); \cdots ; \texttt{INSERT}\ c(i_k), D)$$

where $i_1, \ldots, i_k$ are URIs such that $\mathcal{E}(\texttt{SELECT}\ x\ \texttt{FROM}\ b(x)\ \texttt{WHERE}\ f(x), D) = \{(i_1), \ldots, (i_k)\}$. The semantics can be given similarly if we have a predicate $p(x, y)$ in the `INSERT` clause. The same holds for statements with a single `DELETE` clause with variables. The case of `MODIFY` is slightly more involved: the order of execution of the actions involved is as it was explained in Section 2.

The situation becomes more complex when we consider multiple predicates in an `INSERT`, `DELETE` or `MODIFY` clause, or multiple `INSERT`, `DELETE` or `MODIFY` clauses in a single update statement. Obviously, clause order matters in this case as we have already demonstrated in Example 15 where we consider multiple updates of the same kind without variables. The following examples illustrate the issues involved when multiple updates of different kinds are allowed.

Example 19. Let us assume an RDFS schema with two classes `A` and `B` and an RDF graph with a single node with URI `i1` that is an instance of class `A` (so class `B` has no instances). Let us now consider the following statements:

```
(1)   DELETE B(X) INSERT B(X)                    (2)   INSERT B(X) DELETE B(X)
      FROM A{X}                                        FROM A{X}
```

The effect of Statement (2) is to leave class `B` in the same state (i.e., with no instances) while Statement (1) forces `i1` to become an instance of `B` as well.

There is also a deeper issue regarding the order of execution for the different tuples of values of the variables that satisfy the `FROM` and `WHERE` clauses.

Example 20. Let us revisit the above example and introduce a new class `C` in the schema, and a second resource node with URI `i2` that is an instance of class `C`. Let us now consider the following statement:

```
DELETE B(X) INSERT B(Y)
FROM A{X}, C{Y}
WHERE X != Y
```

The set of tuples satisfying the `FROM` and `WHERE` clause are `(i1,i2),(i2,i1)`. The following orders of execution are possible for the `DELETE-INSERT` block:

```
DELETE B(i1); DELETE B(i2); INSERT B(i2); INSERT B(i1)
DELETE B(i1); INSERT B(i2); DELETE B(i2); INSERT B(i1)
DELETE B(i2); INSERT B(i1); DELETE B(i1); INSERT B(i2)
```

These different orders result in different states of the graph. In the first case class `B` ends up with instances `i1, i2`, in the second case it has instance `i1`, and in the third case it has instance `i2`.

It is possible to give *non-deterministic* semantics to RUL that allow all of the above executions. In this case $\mathcal{A}$ must be allowed to be a *relation* i.e., a subset of $\mathit{Update} \times \mathit{Graph} \times \mathit{Graph}$. Non-deterministic update languages have been considered in the past for other data models e.g., by Abiteboul and Vianu for the relational model [22, 23]. For practical reasons we have chosen to avoid non-determinism in RUL.

We solve the dilemma of examples such as the above by adopting a semantics similar to the one proposed in [24] where a procedural language with a `for each` iterator for deductive database updates is proposed. Let $U_1, \ldots, U_n$ be `INSERT` or `DELETE`. The semantics of updates with multiple `INSERT` or `DELETE` clauses with variables is captured by the following:

$$\mathcal{A}(U_1\ c_1(x_1) \cdots U_n\ c_n(x_n)\ \texttt{FROM}\ b(x_1, \ldots, x_n)\ \texttt{WHERE}\ f(x_1, \ldots, x_n), D) =$$
$$\mathcal{A}(U_1\ c_1(i_1^1); \cdots ; U_1\ c_1(i_1^k); \cdots ; U_n\ c_n(i_n^1); \cdots ; U_n\ c_n(i_n^k), D)$$

where $i_1^1, \ldots, i_n^1, \ldots, i_1^k, \ldots, i_n^k$ are URIs such that

$$\mathcal{E}(\texttt{SELECT}\ x_1, \ldots, x_n\ \texttt{FROM}\ b(x_1, \ldots, x_n)\ \texttt{WHERE}\ f(x_1, \ldots, x_n), D) =$$
$$\{(i_1^1, \ldots, i_n^1), \ldots, (i_1^k, \ldots, i_n^k)\}.$$

In other words, the `FROM` and `WHERE` clauses are evaluated first to compute a set of valid bindings. Then, each one of the `INSERT` or `DELETE` statements is executed in turn for *all* elements of the set of bindings. The semantics can be given similarly if multiple class or property predicates are allowed in the `INSERT` or `DELETE` clauses. Since update clauses with multiple predicates are trivially translated into sequences of update statements with a single predicate then our semantics cover this case as well.

4 Conclusions

We have presented an expressive declarative language for updating RDF graphs while ensuring that insertion/deletion/modification of nodes and arcs violates the semantics neither of the RDF model nor of the specific RDFS schema. More precisely, we have carefully designed the effects and side-effects of each RUL operation to always result in a consistent state of the updated graph. There is an ongoing implementation of RUL using the existing ICS-FORTH RQL code base. In future work, we plan to precisely characterize the expressive power of the language we have developed (in the spirit of [22, 23]) and consider schema updates and schema evolution. Our work on RUL should be considered as a first necessary step towards this direction.

References

1. Das, A., Wu, W., McGuinness, D.: Industrial Strength Ontology Management. In: The Emerging Semantic Web. (IOS Press)
2. May, W., J. Alferes, F.B.: Towards Generic Query, Update, and Event Languages for the Semantic Web. In: Proc. 2nd PPSWR. (2004)
3. Perez, A.G.: A Survey on Ontology Tools. Deliverable 1.3 IST Project OntoWeb (2002)
4. Magkanaraki, A., Karvounarakis, G., Christophides, V., Plexousakis, D., Anh, T.: Ontology Storage and Querying. ICS-FORTH Technical Report No 308 (2002)
5. Seaborne, A.: An RDF NetAPI. In: Proc. 1st ISWC. (2002) 399–403
6. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Proc. 1st ISWC. (2002)
7. Oberle, D., Volz, R., Motik, B., Staab, S.: KAON Server Prototype. Deliverable 6, IST Project WonderWeb (2002)
8. Sarkar, S., Ellis, H.: Five Update Operations for RDF. Rensselaer at Hartford Technical Report, RH-DOES-TR 03-04 (2003)
9. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: A declarative query language for RDF. In: Proc. 11th WWW. (2002)
10. Magkanaraki, A., Tannen, V., Christophides, V., Plexousakis, D.: Viewing the Semantic Web Through RVL Lenses. In: Proc. 2nd ISWC. (2003)
11. Seaborn, A.: RDQL - A Query Language for RDF. (http://www.w3.org/Submission/RDQL)
12. Clark, K.: SPARQL Protocol for RDF. http://monkeyfist.com/kendall/sparql-protocol/ (2004)
13. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A Comparison of RDF Query Languages. In: Proc. 3rd ISWC. (2004) 502–517
14. Nejdl, W., Siberski, W., Simon, B., Tane, J.: Towards a Modification Exchange Language for Distributed RDF Repositories. In: Proc. 1st ISWC. (2002) 236–249
15. Guha, R.V.: Rdfdb ql. (http://www.guha.com/rdfdb/query.html)
16. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing Knowledge about Information Systems. ACM Transactions on Information Systems **8** (1990) 325–362
17. Nejdl, W., Dhraief, H., Wolpers, M.: O-Telos-RDF: a Resource Description Format with Enhanced Meta-Modeling Functionalities Based on O-telos. (In: Workshop on Knowledge Markup and Semantic Annotation at the 1st K-CAP)
18. Koubarakis, M., Mylopoulos, J., Stanley, M., Jarke, M.: Telos: Features and Formalization. Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto (1989)
19. Plexousakis, D.: Semantical and Ontological Considerations in Telos: a Language for Knowledge Representation. Computational Intelligence **9** (1993) 41–72
20. Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In: Proc. 2nd SemWeb. (2001)
21. Hayes, P.: RDF Semantics. http://www.w3.org/TR/rdf-mt/ (2004)
22. Abiteboul, S., Vianu, V.: A Transcation Language Complete for Database Update and Specification. In: Proc. 6th PODS. (1987) 260–268
23. Abiteboul, S., Vianu, V.: Procedural and Declarative Database Update Languages. In: Proc. 7th PODS. (1988) 240–250
24. Wallace, M.: Compiling Integrity Checking Into Update Procedures. In: Proc. 12th IJCAI. (1991) 903–908