Computer Science Department
University of Crete

RUL: a Declarative Language for Updating RDF Data

Master's Thesis

Stavros Sahtouris
Thesis Supervisor: Vassilis Christophides, Assoc. Professor

April 2006
Heraklion, Greece

Ευχαριστίες

Η μεταπτυχιακή αυτή εργασία αποτελεί, κατά κάποιο τρόπο, το αποτέλεσμα μίας ολόκληρης πορείας που ξεκίνησε πολλά χρόνια πριν το μεταπτυχιακό πρόγραμμα αυτό καθ' εαυτό. Σ' αυτή την πορεία δεν ήμουν πάντοτε μόνος μου: υπήρξαν διάφοροι άνθρωποι που, με τον ένα ή τον άλλο τρόπο, με ώθησαν προς διάφορες κατευθύνσεις, με στήριξαν όταν χρειαζόμουν στηρίγματα, ήταν δίπλα μου όταν τους είχα ανάγκη, μου έδειξαν νέους τόπους, τοπία και οπτικές γωνίες ή, άλλες φορές, απλώς μου έκαναν παρέα.

Κατ' αρχήν, ευχαριστώ τον επόπτη μου Βασίλη Χριστοφίδη που με καθοδήγησε ουσιαστικά, με εμψύχωσε και ήταν πάντοτε εκεί όποτε τον χρειαζόμουν, καθ' όλη τη διάρκεια του μεταπτυχιακού. Δεν θα ήταν καθόλου υπερβολή αν έλεγα ότι αυτή η εργασία δεν θα είχε γίνει χωρίς τη βοήθειά του. Επίσης, ευχαριστώ τους γονείς μου που με στήριξαν σε όλες τις αποφάσεις μου, όχι μόνο οικονομικά, αλλά κυρίως με την αγάπη και το διαρκές ενδιαφέρον τους.

Ευχαριστώ τον Μανόλη Κουμπαράκη και τη Ματούλα Μαγιρίδου για τη συνεργασία τους κατά το σχεδιασμό της RUL και τους Δημήτρη Πλεξουσάκη και Γρηγόρη Αντωνίου για τις χρήσιμες παρατηρήσεις τους. Επίσης, ευχαριστώ τον Γρηγόρη Καρβουναράκη για τις ουσιαστικές και γρήγορες απαντήσεις του στις απορίες μου σχετικά με την RQL και τον καλογραμμένο του κώδικα, τη Σοφία Αλεξάκη που με μύησε σε μπόλικα μυστικά της RDF Suite, το Λευτέρη Σιδηρουργό για τις συζητήσεις μας και το Χάρη Γκίκα που μου υπέδειξε τα κλειδιά για αρκετές πόρτες γνώσης στην πληροφορική.

Ακόμα, θα ευχαριστήσω (με σειρά εμφάνισης) το Θανάση για τα μαγικά ταξίδια μας, το Χάρη για την καλόκαρδη υπομονή του, το Νίκο για τις υποδείξεις των επιπλέον διαστάσεων και τις βουτιές στη μεγάλη οθόνη, τη csdlista και τον Albert Hofmann, το Δημήτρη για τα τσιγάρα και τα ούζα που ήπιαμε μαζί, την αδερφή μου για πολλούς λόγους, τον Πάκο για τα νυχτερινά ντεριβέ. Τέλος, για λόγους που δεν έχουν γίνει ακόμα σαφείς θα ήθελα να ευχαριστήσω (αλφαβητικά) τους Αλέκο, Γιάννη, Δέσποινα, Ευθυμία, Μανόλη και Μίλτο και μερικούς ακόμα ανθρώπους που μάλλον δεν θα διαβάσουν ποτέ αυτό εδώ το ευχαριστήριο.

ΠΕΡΙΛΗΨΗ

Η διαχείριση αλλαγών σε περιγραφές πόρων που βασίζονται σε RDFS σχήματα έχει γίνει απαραίτητη στις σύγχρονες εφαρμογές του Σημασιολογικού Ιστού. Αποσκοπώντας στην ικανοποίηση αυτών των απαιτήσεων, προτείνεται μία δηλωτική γλώσσα διαχείρισης αλλαγών για γράφους RDF, η οποία βασίζεται στα παραδείγματα των γλωσσών επερωτήσεων και όψεων RQL και RVL. Η γλώσσα ονομάζεται RUL και σε αυτήν διασφαλίζεται ότι οι αλλαγές στους κόμβους και τις ακμές δεν παραβιάζει τη σημασιολογία του μοντέλου RDF ή των δεδομένων RDFS σχημάτων. Επιπλέον, η RUL υποστηρίζει καλά καθορισμένες αλλαγές στο επίπεδο των πόρων και των ιδιοτήτων τους καθώς και τη δυνατότητα πολλαπλών αλλαγών με ντετερμινιστική σημασιολογία. Επιπλέον, εκμεταλλεύεται πλήρως την εκφραστική δύναμη της RQL προκειμένου να καθορίσει τα όρια των μεταβλητών στους κόμβους και τις ακμές του RDF γράφου. Η γλώσσα υλοποιήθηκε στο πλαίσιο της RDF Suite ως επέκταση της RQL. Η υλοποίησή της βασίζεται σε μία γλώσσα αλλαγών σε βάσεις δεδομένων και παράγει SQL προτάσεις αλλαγών για τις αναπαραστάσεις που χρησιμοποιούνται στην RDF Suite.

ABSTRACT

Semantic Web applications are striving nowadays for managing changes of persistent resource descriptions created according to RDFS schemata. To cope with this demands, a declarative update language for RDF graphs is proposed, which is based on the paradigms of query and view languages RQL and RVL. This 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. The language has been implemented in the context of RDF Suite, as an extension of RQL. The implementation relies on a database update language and generates SQL update statements for the various database representations used in RDF Suite.

Contents

List of Figures

List of Tables

1

Introduction

Semantic Web applications are striving nowadays for managing changes of persistent resource descriptions created according to RDFS schemata [9, 28]. The majority of ontology-based authoring and annotation tools [2] 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 [3] have implemented suitable update APIs [7, 8, 24, 26]. 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 and sound primitives is a challenging issue.

The most interesting proposal so far is MEL that has been developed in the framework of QEL and it is based on Datalog [22]. 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 sub-graph centered approach but consistency of updates with respect to the employed RDFS schemata is not respected. Furthermore, no formal semantics or detailed behavior description have been given for MEL. The rdfDB Query Language [12] 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.

In this thesis, we propose a declarative update language for RDF graphs which is based on the paradigms of query and view languages RQL [14] and RVL [21]. Our language, called RUL ([19]), provides primitive and set-oriented updates. Update operations affect the class instances and/or property instances in a well defined way. RUL integrates smoothly with RQL and benefits from the typing data model and the powerful pattern matching the later provides. RUL comes with operation semantics defined in a declarative (chapter 3) as well as in a procedural (chapter 4) manner. It is a design choice of RUL to provide safe expressions and deterministic iteration semantics.

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 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 [4], or SPARQL [17]) offering less expressive pattern matching capabilities [13]. None of the RDF update languages proposed so far [12,22] supports the aforementioned functionality.

In chapter 2 we present the eight RUL operations and describe their syntax. We also describe informally their effects on the RDF graph. The RDF graph considered here consists of nodes, representing classes or class instances, and arcs representing properties, property instances or classification links between instances and classes/properties. The effects of RUL operations are described as sequences of insertions and deletions of nodes and arcs on this graph. The preconditions are described and the main effects of each operation are distinguished from the side effects. We explain the functionality of RUL operations with variables (set-oriented updates) as well as statements containing multiple operations. We also illustrate with examples the integration of RUL with RQL (or another RDF query language for that matter).

In chapter 3 we formally define the semantics of RUL operations and we focus

on the safe and deterministic set-oriented updates where we reason that the order of operations in a statement matters (statements with the same RUL operations in a different order have different semantics). Later, our update semantics are compared with the semantics of knowledge base updates, where it is proposed that RUL can be used as a low level update language for implementing a high level knowledge base update language. RUL is also compared with other RDFS update languages and proved to be more expressive. Last but not least, we present the world of database update languages, define the concept of expressive power and present how they are compared in the literature. We focus on two of them, namely on WL and SdetTL, as they are the most expressively powerful for the needs of RUL. We also explain the functionality of the provided database update operations as they are proposed in the literature and focus on the deterministic semantics of the two languages. We argue that WL is more suitable for implementing RUL, as its semantics easily capture the semantics of the RUL sub-operations (insertion and deletion of arcs and nodes on the RDF graph) as well as for performance reasons.

In chapter 4, the architecture of RUL implementation is explained. RUL has been developed as an extension of RQL implementation and follows most of its design principles, except that the returned result of a RUL statement is feedback to the user rather than the goal of the statement. RUL statements consist of an update operation part (the head) and a query part. We present the various database representations used in RDF Suite to store RDFS descriptions, and use WL programs to describe the implementation of each RUL operation according to each database representation. We also explain how we ensure the safe and deterministic semantics of the language in implementation. Finally, the translation to SQL is described and we present some optimization techniques used to improve the performance of the language.

An important design decision in the implementation level is the use of a temporary relation for storing the results of the evaluation of the query part of a RUL statement. We show how this principle is used to ensure safety and determinism. We also take benefit of it for optimizing the costly operations with schema variables.

1.1 Motivating example: a graphical RDF/S managment tool

In this chapter we consider a graphical user interface (GUI) for editing RDF/S description graphs (see figure 1.1).Like various RDF/S authoring tools, it can be used to navigate through an RDF/S schema graph using the mouse and select classes, properties, resources and property instances. The user can apply various update operations over the selected items by selecting them from a menu. Everyone using a personal computer is familiar with the semantics of these operations: a ”new” and a ”delete” for inserting and removing items from the graph, a ”copy and paste” operation for cloning items, a ”cut and paste” operation for moving items from one place to another and, finaly, a ”rename” operation for changing the URIs of various resources. The semantics of these operations as well as the restrictions to what the user can do over each kind of item are similar (and in some cases equivalent) to the semantics and preconditions of RUL update operations, so it is interesting to examine how these GUI operations over some specific items can be expressed with RUL expressions. The selection of one or more items from the graph in the GUI world is expressed with some query. In case of graphicaly represented RDF/S graphs, we are interested in the update operations applied over a graphical selection of items using RUL statements.

The ”new” GUI operation corresponds to the insertion of a new class or property instance in the RDF graph. This can be handled with an INSERT, whether

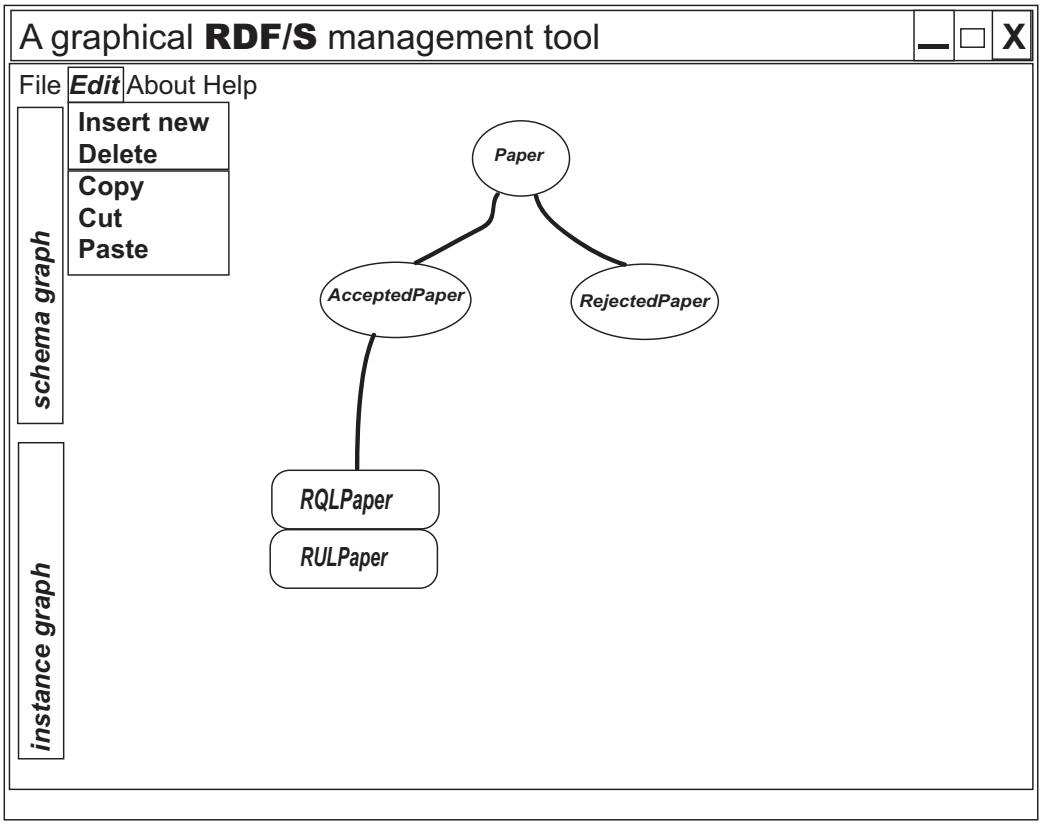

Figure 1.1: A fictional graphical user interface for managing RDF/s descriptions.

it is an insertion of a class or a property instance. The user selects the class or property he/she wants to be instantiated, and clicks on the ”new” selection from the menu. For example, the user selects $Paper$ and clicks on ”new” to insert a new $Paper$ resource. The corresponding RUL expression is the following:

```
INSERT Paper(&newPaperValue)
```

The side effects of the INSERT operation for this case do not cause any harm to the behavior of the GUI tool. If the $\&newPaperValue$ resource exists as an instance of a super-class of $Paper$, it is now also an instance of $Paper$.

The ’Wdelete’W GUI operation corresponds to the erasure of an instance or a classification link. We suppose that if a resource is an instance of a class (e.g. $AcceptedPaper$), it is also an instance of all super-classes of it (e.g. $Paper$ is a super-class of $AcceptedPaper$), although this information is often omitted in the graphical representation. For example, resource $\&RULPaper$ is also an instance of class $Paper$, although the link between them does not appear in figure 1.1. The semantics of the ”delete” GUI operation can described as the erasure of the resource and the instantiation links emanating from it or just the erasure of one instantiation link. In RUL we provide both functionalities. For example, the erasure of the instantiation link between a resource $\&r$ and a class C is captured by

```
DELETE C(&r)
```

while the instantiation link between a property P and a property instance between resource $\&source$ and resource $\&target$ is erased by

```
DELETE P(&subject, &object)
```

In RUL we also express more sophisticated erasures, e.g. the erasure of a set of instantiation links emanating from a specific resource.

The ”copy and paste” GUI operation is also handled with a RUL INSERT. If the user selects some resource $\&RULPaper$ that is an instance of the class $AcceptedPaper$ and pastes it to $RejectedPaper$, the following RUL expression captures the semantics of this operation:

```
INSERT RejectedPaper(&RULPaper)
```

If the user pastes the resource to a super-class of $AcceptedPaper$ (e.g. $Paper$), the expression is the same. RUL INSERT will not modify the description in that case, but this is exactly the behavior we want, because $\&RULPaper$ is already an instance of $Paper$.

If the ”copy and paste” GUI operation is applied over some property instance, the RUL INSERT for property instances captures again the semantics of the operation. It is possible, though, that the user might try to paste the copied instance to a property of which the domain and/or the range do not contain the source and/or the target of the property instance as instances, or they are of a different literal type. The desired behavior of the GUI tool would be to not allow the user to paste the property instance there. Because of the preconditions of RUL INSERT for property instances, RUL INSERT will return ”false” to the overlying GUI application so that it will be aware of the fact that this operation is not valid.

The ”cut and paste” GUI operation is more complicated. A ”cut and paste” when class instances have been selected can be viewed as an attempt to change the instantiation information of these instances. A resource is ”cut” means some instantiation links between the resource and the selected classes are erased. When the resource is ”pasted”, some other instantiation links are added between the resource and the selected classes. RUL REPLACE for classification of class instances can be used in that case. If the instance is multi-classified, a single RUL REPLACE is not enough to capture the semantics of such a ”cut and paste” oper-

ation. E.g., If the user wants to ”cut” the resource $\&RULPaper$ and paste it as an instance of $RejectedPaper$, the RUL expression is the following:

```
REPLACE $C1<-RejectedPaper(&RULPaper) Q($C1)
```

where Q is an RQL expression that returns all the classes that have $\&RULPaper$ as an instance. Similarly, a class variable can be used to denote that the resource is going to be ”pasted” under more than one classes.

In case of applying ”cut and paste” on property instances, the RUL REPLACE classification for property instances captures the semantics of the operation and provides the necessary preconditions when the operation should not be allowed. The affected property instance has to be a valid instance of the property under which is classified, otherwise the tool should not allow the operation. RUL REPLACE semantics is aware of this restriction.

Finally, a ”rename” GUI operation would be desired in some systems. The aim of this operation is to change the name of a URI or the value of a literal attribute. If the new name of a resource exists in the description base, the GUI tool should have to merge the equally named resources. This is captured by the semantics of RUL REPLACE for class instances.

If the user clicks on some literal value and desires to rename it, we indentify the value by refering to the property instance triplet it is part of. Then the user enters a new value, that replaces the old one. In RUL this is captured by the semantics of REPLACE for property instances.

```
REPLACE P(&someResource, "str1" <- "str2")
```

2

The syntax of RUL

RUL can be used to express updates to RDF graphs i.e., insertions, deletions and replacements of nodes and arcs.

An RDF graph contains various types of nodes and arcs. Classes are represented as nodes and properties as arcs between the class nodes. The class the node of which a property arc emanated from is ”the domain of the property” and the one that ends to is ”the range of the property”.

Classes and properties are related through IsA (subsumption) relations. These relations are represented by arcs. The class from the node of which an IsA arc emanates is a sub-class of the class to the node of which the IsA arc ends.

Property arcs are also connected with IsA arcs in the same way as class nodes. Of course, an arc connecting other arcs is not compatible with the semantics of a graph representation. In order to deal with this problem, we can view properties as triplets consisting of a domain arc, a property node and a range arc. The domain arc emanates from the domain class node and ends to the property node, while the range arc emanates from the property node and ends to the range class node. With that model in mind, we can connect property nodes with IsA arcs. We prefer to use a shortcut for that triplet, though, and represent a property as an arc. A property arc emanates always from exactly one node and ends to exactly another one. This node is either a class node or a node representing a class of literal values.

A class instance, sometime referred as ”a resource”, is also represented with a node. A resource is an instance of one or more classes. We say that a resource is a direct instance of the classes that do not have any sub-classes with this resource as an instance. The resource is an indirect instance of the classes that are super-classes of some classes with this resource as an instance.

If a resource is a direct instance of class, the resource node is connected to the class node through an arc called ”classification arc” or ”classification link”. A classification link emanated from a class instance node and ends to a class node. A class instance node is valid only if there is at least one classification arc emanating from it. If a resource is an indirect instance of a class, this relation is implied through the IsA arcs connecting the class with a sub-class for which the resource is a direct instance of.

Property instances are represented as arcs between class instance nodes, literal nodes, or both. A literal node is a node is not connected to any other node through classification links and represents a literal value. The class instance or literal value from the node of which a property instance arc emanates is called ”the source of the property instance” and the class instance or literal value to the node of which

a property instance ends is called ”the target of the property instance”.

Property instances are connected to the properties they are instances of, by classification links. Like in class instances, a property instance can be direct instance of some property and indirect instance of some other properties. Only the direct instantiation relation is represented by classification links. A classification link from property instances is an arc emanating from the property instance arc and ending to the property arc that this instance is a direct instance of. In order to be compatible with the semantics of graphs, we can view a property instance arc as a shortcut of the triplet ”source arc”-”property instance node”-”target arc”, where the source arc emanated from the source node of the property and ends to the property instance node and the target arc emanates from the property instance node and ends to the target node of the property instance. In that case, the classification link of property instance emanates from the property instance node and ends to the property node of which it is an instance of. As in the case of property arcs, we prefer to use a shortcut: the whole triplet is represented a property instance arc, and the classification links emanate from it and end to the property arc (which is also a shortcut).

In the figures of this chapter, a class node is drawn as a circle, while a resource node is a string starting with an ampersand (&). IsA arcs are solid arrows with a white head, while property arcs, as well as property instance arcs are solid arrows with a black head. The instantiation arcs are dashed arrows with white head. The property arcs and the property instance arcs are distinguished by the context: a property arc emanats and ends to class nodes, while a property instance arc emanats and ends to class instance nodes.

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 2.1 dealing with the organization of scientific conferences, and IMG_REF_HERE

where the effects and side-effects of each operation are analyzed in detail.

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.

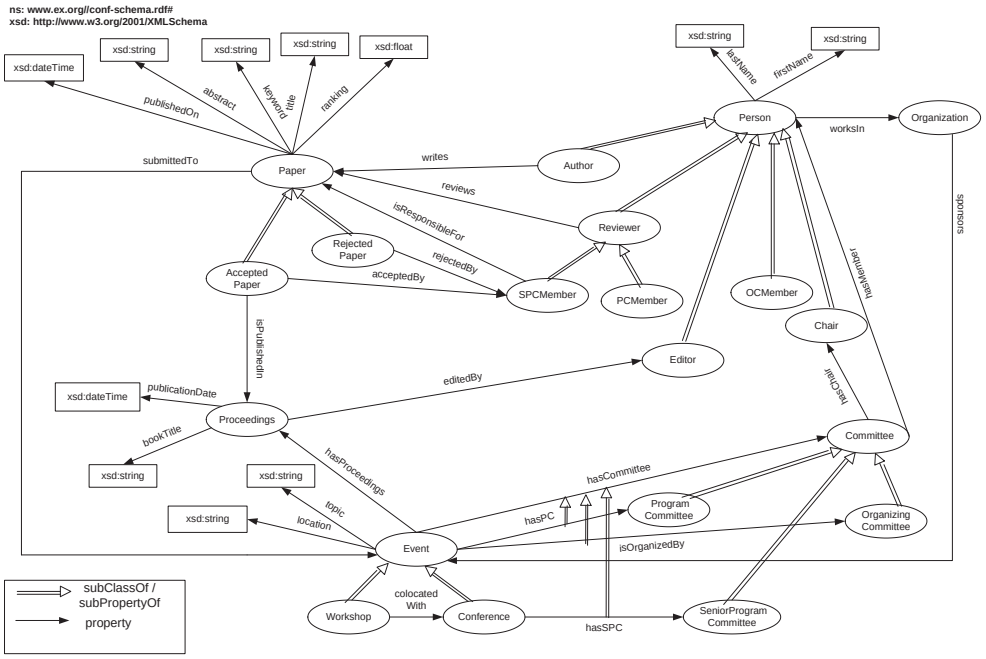

Figure 2.1: *The RDF schema of a sientific conference example will be used to illustrate and clarify the syntax of RUL*

The syntax of any RUL expression is as follows:

```
UPDATE SchemaStatement(ClassInstancesStatement)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
```

The update statement can be an INSERT, DELETE or REPLACE statement for class or property instances. The $SchemaStatement$ is a statement related

to schema variables or constants, while the $ClassInstancesStatement$ contains class instance variables or constants. These statements will later be examined in detail, and they are based on the statements described in [19]. More precisely, the INSERT and DELETE clauses described here are no different from the INSERT and DELETE statements in [19]. In this thesis we use the REPLACE clause instead of the MODIFY clause, but we also describe its behavior with more details, separating the case of modification to the resource or property instance from modification to the resource or property classification link.

For example, the first update statement we will examine is the INSERT statement for class instances, which is:

```
INSERT QualClassName(ResourceExp)
```

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. In short, an INSERT operation ensures that a resource is an instance of the specified class, as long as certain contraints are not violated.

As usual, the WHERE clause gives the filtering conditions for the variable bindings introduced in the FROM clause. 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 2.1).

As in the RDF Query Language (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 is an instance of C and it is not an instance of any subclass of C. 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.

It is a design choice of RUL to have a different 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.

2.1 Updating class instances

2.1.1 INSERT for class instances

The syntax of the INSERT statement for class instances 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 effects and side-effects of an INSERT operation with the above syntax are presented graphically in figure 2.2. A new node $ResourceExp$ can be created as a direct instance of $QualClassName$, as it is shown in figure 2.2, statement

(1). If node $ResourceExp$ exists in the graph and it is classified under a superclass of $QualClassName$ (fig. 2.2 statement (4)), 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 $QualClassName$ (fig. 2.2, statement (2), where C is a subclass of B), 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 the node $ResourceExp$ exists in the graph and it is classified under a class which is not related through a subclass relation to $QualClassName$ (fig. 2.2 statement (3)), the result is a multi-classified node ($\&r1$ is classified both under B and D classes) without any side-effect.

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
```

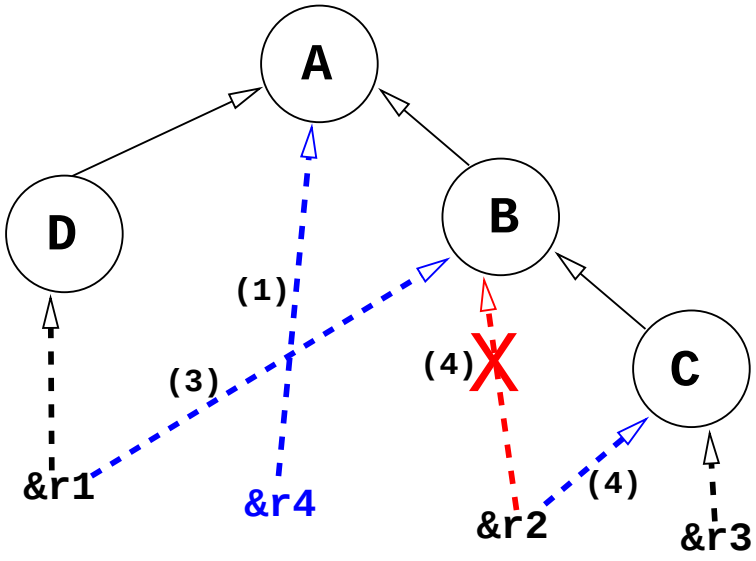

Figure 2.2: *Examples of some INSERT operations for class instances:*

(1) INSERT $A(\&r4)$

(2) INSERT $B(\&r3)$

(3) INSERT $B(\&r4)$

(4) INSERT $C(\&r2)$

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

2.1.2 DELETE for class instances

The syntax of the DELETE operation for class instances 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 (fig. 2.3). 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 fig. 2.3, statement (1), $\&r1$ is now classified under classes A and B). If $ResourceExp$ is multi-classified (e.g., $\&r4$ in 2.3.4), the classification links to classes not related to $QualClassName$ remain untouched (in fig. 2.3, statement (4), the classification link to A remains untouched). An interesting case of a deletion of a multi-classified resource is demonstrated in fig. 2.3, statement (5), where $\&r5$ is an instance of K through M. The classification link to M is removed, because M is a subclass of $QualClassName$ (in this case L), but the classification link to K is not removed as K is not related to L through subsumption. Finally,

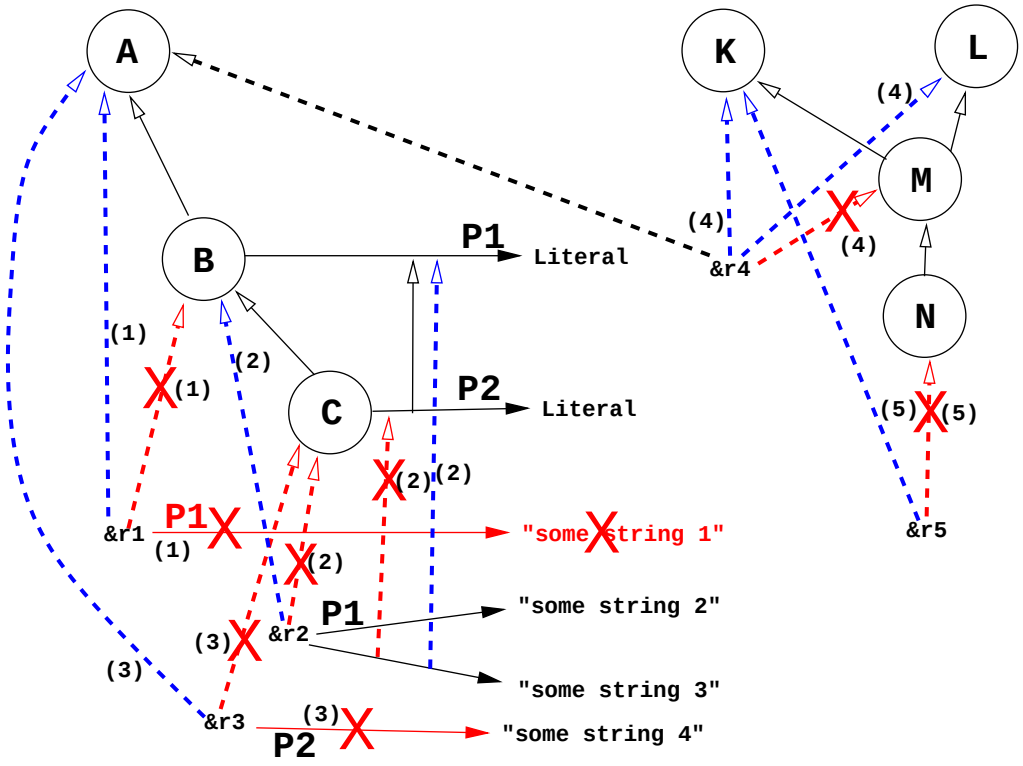

Figure 2.3: *Examples of some DELETE operations for class instances:*

(1) DELETE $B(\&r1)$

(2) DELETE $C(\&r2)$

(3) DELETE $B(\&r3)$

(4) DELETE $M(\&r4)$

(5) DELETE $L(\&r5)$

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 (resource removal).

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 (e.g. fig. 2.3 statement (1) and statement (2)), are also deleted by a DELETE property instance operation (which is described below in detail). 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 property instance $P1$ emanating from $\&r1$ in fig. 2.3, which is deleted (1) when the respective classification link is removed. The deletion of $\&r2$ in (2) has a more interesting side effect: the property instance $P2$ is generalized to an instance of $P1$ ($P1$ is a super-property of $P2$), while the property instance $P1$ remained untouched. In general, when a class instance is deleted, the property instance related to it, remain untouched if they are still valid ($P1$ in (2)). If this is not possible, they are generalized to their ancestor properties, if any ($P2$ in (2)), or completely removed ($P1$ in (1)). Finally, if a property instance cannot be generalized, despite the fact there is a super-property ($P2$ in (3) cannot be generalized because $r3$ is now an instance of A, therefore not in the domain of $P2$ or $P1$), the whole delete operation is aborted.

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).

2.1.3 REPLACE for class instances

The syntax of the REPLACE operation is:

```
REPLACE 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 effect of the REPLACE operation (fig. 2.4) is to completely remove the node(s) denoted by $OldResourceExp$ and then insert the node(s) denoted by $NewResourceExp$ as an instance of what $OldResourceExp$ used to be. What's more, the new node preserves all the property instances related to the old one. The insertion of $NewResourceExp$ has the same semantics as the INSERT operation presented earlier (see fig. 2.4 statement (2), where the inserted resource $\&r4$ is specialized to be instance of B).

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

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

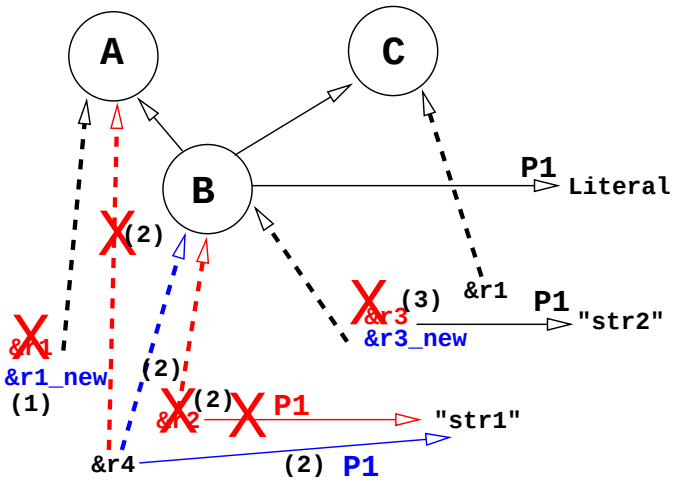

Figure 2.4: *Examples of some REPLACE operations for class instances:*

(1) REPLACE $A(\&r1$ <- $\&r1\_new)$

(2) REPLACE $B(\&r2$ <- $\&r4)$

(3) REPLACE $A(\&r3$ <- $\&r3\_new)$

If paper1.pdf had title "The language SQL", we could equivalently write:

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

It should be stressed that the REPLACE operation is not a sequence of DELETE and INSERT. The main difference between a REPLACE operation and a sequence of DELETE and INSERT operations is the different side effects.

The first side effect of REPLACE is that all properties emanating from (or ending at) the resource denoted by $OldResourceExp$ are completely erased. The other side effect is that the previously removed properties will become properties emanating from (or ending at) the resource denoted by $NewResourceExp$. In figure 2.4 statement (2), property arc $P1$ emanating from $\&r2$ and ending at literal value "str1", is removed, while another property arc $P1$ which ends at literal value "str1", is inserted, emanating from $\&r4$. In figure 2.4 statement (3), the property arc $P1$ is removed and then inserted with a new source instance.

In other words, REPLACE could be described as a resource erasure followed by a resource addition. The semantics of these operations is not the same as the semantics of the RUL INSERT and DELETE statements presented previously. More precisely, during the erasure, the resource is completely removed from the database, as long as it is originally an instance of QualClassName. During the addition operation, the new resource is inserted according to the corresponding RUL INSERT operation, with all the effects and side effects of an INSERT operation. Moreover, during the operation operation the property instances attached to the removed resource are modified as follows: If a property instance has the removed resource as source (or, similarly, target), the RUL REPLACE operation will cause the property to have the added resource as source (or target) instead. For example,

new values can be inserted with REPLACE, or existing ones can be specialized (instaciated under a sub-class of the class they where originaly instaciated).

For example, in figure 2.4, statement (1), the operation can be described as a removal of $\&r1$ and an insertion of a new resource $\&new\_r1$. Notice that $\&r1$ also an instance of C, but the REPLACE operation asks only the instance of A to be modified. Therefore, after the execution of the operation, $\&r1$ will still be an instance of C.

Another example is presented in figure 2.4, statement (2), where the resource $\&r2$ is removed and then the resource $\&r4$ is added instead. The property instance $P1$ is also removed but replaced with a new instance emanating from the inserted resource. The new resource is not new to the database. It is originally an instance of A, and after the operation it has been specialized to an instance of B (and an indirect instance of A).

In order to illustrate the difference of a REPLACE with a sequence of DELETE and INSERT, notice the following RUL statements:

(a) Replace the instance of B r2 with r4 ..... (b) Delete r2 from B and insert r4 to B

$$REPLACE\ B(\&r2 <- \&r4) \qquad DELETE\ B(\&r2)$$
$$INSERT\ B(\&r4)$$

After the execution of the sequence (b), $r2$ will be an instance of the super-classes of B, as this is the effect of DELETE, while in (a), r2 will be either completely removed or the classification link between r2 and (as well the super-classes of B) will be canceled. What's more, in (b) the property instance $P1(\&r2, "str1")$ would be removed, as the domain of $P1$ is B, while in (a) the property instance will be modified to $P1(\&r4, "str1")$.

2.1.4 REPLACE classification for class instances

REPLACE can also be used for modifying the classification of a class instance. In this case, the following syntax is used:

```
REPLACE OldQualClassName<-NewQualClassName(ResourceExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
```

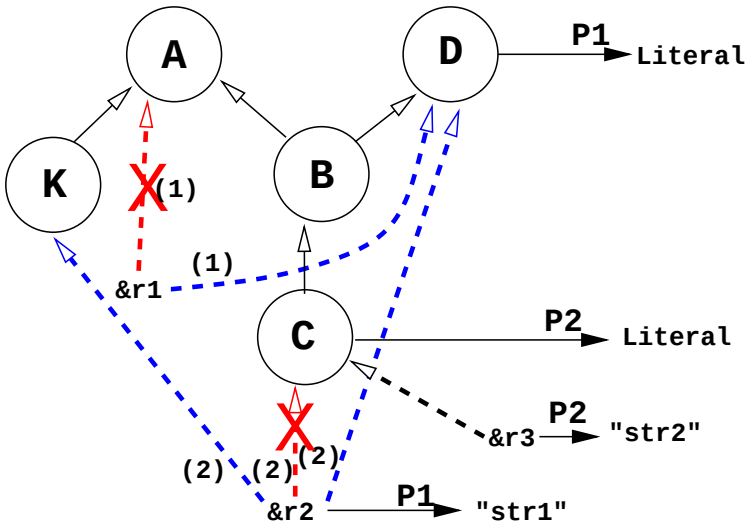

Figure 2.5: *Examples of some REPLACE-classification operations for class instances:*
(1) REPLACE A <- $D(\&r1)$
(2) REPLACE B <- $K(\&r2)$
(3) REPLACE B <- $K(\&r3)$
(4) REPLACE B <- $K(X)$ *WHERE* $B\{X\}$

This operation modifies a classification link that emanates from the class instance node of the class instance denoted by $ResourceExp$ and ends to the class node of the class denoted by $OldQualClassName$ or a node of a subclass of

it. The effect of the operation is to redirect the classification link so that it no longer ends to the node of class $OldQualClassName$, but it ends to the class node representing the class $NewQualClassName$.

In other words, the effect of this operation is that $ResourceExp$ is not anymore an instance of $OldQualClassName$, but an instance of $NewQualClassName$ (e.g. $\&r1$ is not anymore an instance of A, but an instance of D, in fig. 2.5 statement (1)). If there are property instances emanating from or ending at $ResourceExp$ because of their domain or range being $OldQualClassName$ or a subclass of it (e.g. the property instance $P1$ emanating from $\&r2$), then their domain or range should also be $NewQualClassName$ or a subclass of it (e.g. after the operation in fig. 2.5 statement (2), $\&r2$ is still an instance of D). Otherwise, the operation has no effect and it is aborted (e.g. the operation in 2.5.3 is aborted, because of the property instance $P2$).

In fig. 2.5, statement (4), the operation is aborted. As it will be analyzed later, this operation is equal to a sequence of the operations of statement (2) and (3). We have already seen that (3) is aborted, therefore (4) is aborted as well, for the same reason.

2.2 Updating property instances

2.2.1 INSERT for property instances

The INSERT, DELETE and REPLACE 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]
```

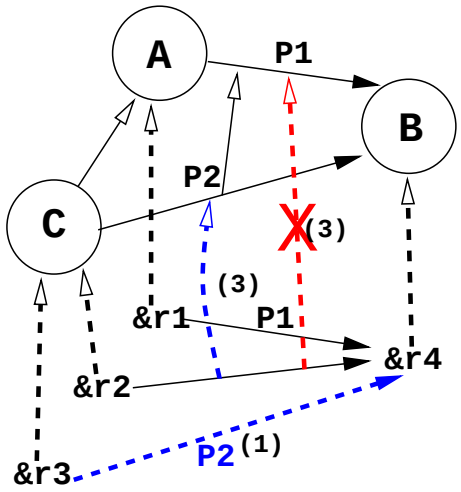

Figure 2.6: *Examples of some INSERT operations for property instances:*
(1) INSERT $P2(\&r3, \&r4)$
(2) INSERT $P2(\&r1, \&r4)$
(3) INSERT $P2(\&r2, \&r4)$
(4) INSERT $P1(\&r1, \&r4)$

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 2.6. 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 $QualPropertyName$ (fig. 2.6 statement (3)), then the operation's effect is the deletion of the instantiation link of the arc

and the introduction of a new link to $QualProperyName$ (e.g., the arc from $\&r2$ to $\&r4$ 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., the arc $P2$ between $\&r1$ and $\&r4$ is not inserted in fig 2.6 statement (2) and the operation has no effect). If the property arc exists as an instance of a sub-property of $QualPropertyName$, then the operation has also no effect (fig. 2.6 statement (4)). Last but not least, if there are not any instances of $QualPropetyName$ emanating from $SubjectExp$ and targeting to $ObjectExp$, a new property arc is inserted, provided that $SubjectExp$ and $ObjectExp$ are instances of the domain and range of the property (fig. 2.6 statement (1)). 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.

2.2.2 DELETE for property instances

The syntax of the DELETE operation is as follows:

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

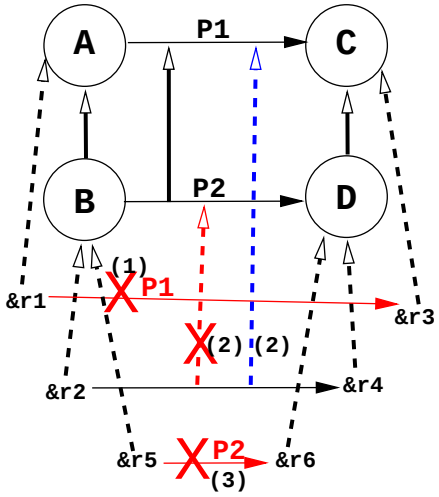

Figure 2.7: *Examples of some DELETE operations for property instances:*
(1) DELETE $P1(\&r1, \&r3)$
(2) DELETE $P2(\&r2, \&r4)$
(3) DELETE $P1(\&r5, \&r6)$

As in the case of resources, the DELETE operation (fig. 2.7) removes essentially the instantiation link between $QualPropertyName$ and the property arc from $SubjectExp$ to $ObjectExp$ (e.g., the arc from $\&r2$ to $\&r4$ in figure 2.7 statement (2) 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 $\&r2$ to $\&r4$ in

fig. 2.7 statement (2) becomes an instance of $P1$), as we discussed in the property INSERT operation. If the arc is not an instance of $QualProperty Name$ (or is not an existing arc), the operation has no effect. It is interesting to focus on the differences in the examples presented in fig. 2.7 statement (2) and statement (3). In both cases, the deleted property is an instance of $P2$. In the first case (fig. 2.7 statement (2)), the $QualProperty Name$ is $P2$, so the instance is deleted as an instance of $P2$ and therefore generalized to an instance of $P1$. In the second case (fig. 2.7 statement (3)), $QualProperty Name$ is $P1$, so the respecting instance is deleted as an instance of $P1$. The instance is deleted because there is no super-property of $P1$.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}
```

2.2.3 REPLACE for property instances

The syntax of the REPLACE operation is:

```
REPLACE QualPropertyName([OldSubjectExp <-] NewSubjectExp,
        [OldObjectExp <-] NewObjectExp)
[FROM VariableBinding]
```

```
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
```

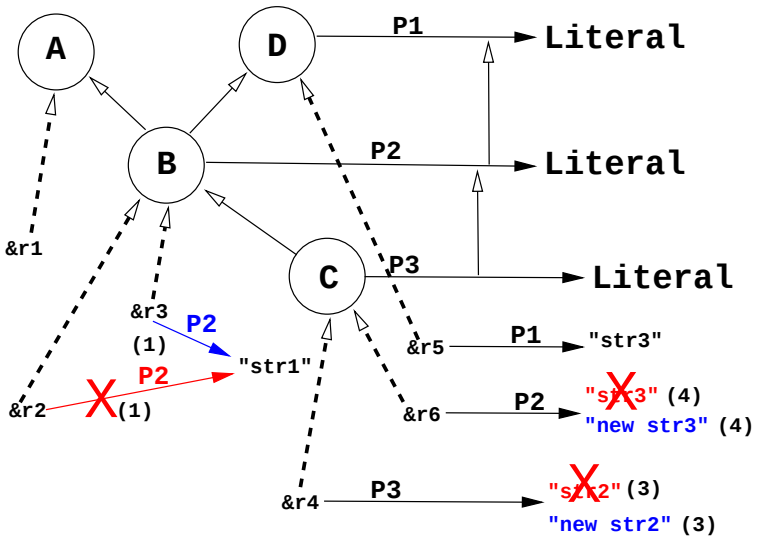

Figure 2.8: *Examples of some REPLACE operations for property instances:*
(1) REPLACE $P2(\&r2$ <- $\&r3, "str1")$
(2) REPLACE $P2(\&r2$ <- $\&r1, "str1")$
(3) REPLACE $P2(\&r4, "str2"$ <- $"new\ str2")$
(4) DELETE $P2(X, "str3"$ <- $"new\ str3")$ *FROM* DX

As we can see in figure 2.8, 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 $NewObjectExp$. The REPLACE statement can also be used to replace only the subject or the object of a property instance with a new one (e.g. in fig. 2.8 statement (1), the arc between $\&r2$ and "str1" is removed and a new arc between $\&r1$ and "str1" is inserted, so that the subject of this property is replaced). If $OldSubjectExp$ (resp. $OldObjectExp$) or $NewSubjectExp$ (resp. $NewObjectExp$) is not an instance of a class in the domain (resp. range) of $QualPropertyName$, the operation is aborted, as a precondition is violated (e.g., in fig. 2.8 statement (2), the operation has no effect as

$\&r1$ is not an instance of the domain of $P2$). If the arc from $NewSubjectExp$ to $NewLObjectExp$ already exists and it is a (direct or indirect) instance of $QualPropertyName$, it is not inserted, so that redundancies are avoided, as we discussed in the property INSERT operation. If there is an instance of a sub-property of $QualPropertyName$ (like $P3$ is a sub-property of $P2$ in figure 2.8 statement (3)), then the subject and/or subject of this instance is replaced by the new one, but the classification of the property does not change (e.g. the subject of $P3$ is now "new str3"). In general, the classification of a property instance affected by this operation should never change.

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

```
REPLACE 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:

```
REPLACE 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:

```
REPLACE 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".

```
REPLACE 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.

2.2.4 REPLACE for property instances classification

As in class instances, REPLACE can be used for modifying the classification of one or more property instances, e.g. to make an instance of a property become an instance of another property. In that case, the syntax of replace is as follows:

```
REPLACE OldQualPropertyName <-
        NewQualPropertyName (SubjectExp, ObjectExp)
[FROM VariableBinding]
[WHERE Filtering]
[USING NAMESPACE NamespaceDefs]
```

From the RDF graph point of view, the operation affects the classification links than emanated from the property instance arc representing the property instance $OldQualPropertyName(SubjectExp, ObjectExp)$, and ends to the property arc representiong the $OldQualPropertyNameproperty$. The effect of the operation is to redirect the classification link so it no longer ends to the arc of $OldQualPropertyName$ property, but instead it ends to the property arc representing $NewQualPropertyName$ property.

In other words, this operation is used to change the classification of the instances ($SubjectExp$, $ObjectExp$) of $OldQualPropertyName$ so that they become instances of $NewQualPropertyName$, as presented in figure 2.9. This

operation has no effect if some preconditions are not satisfied. One precondition is that the domain and the range of $OldQualPropertyName$ must be of the same type as the domain and range, respectively, of $NewQualPropertyName$. For example, if the range of the first is string and the other is integer, then the operation has no effect. Another example is presented in figure 2.9 statement (4), where the first property has a literal range, while the second has a class. Another precondition is that if the domain/range is a class the subject and object of the respecting property instances must be class instances of the domain/range of $NewQualPropertyName$ (e.g. in fig. 2.9 statement (1), $\&r2$ is not an instance of the range of $P2$, so the operation is aborted).

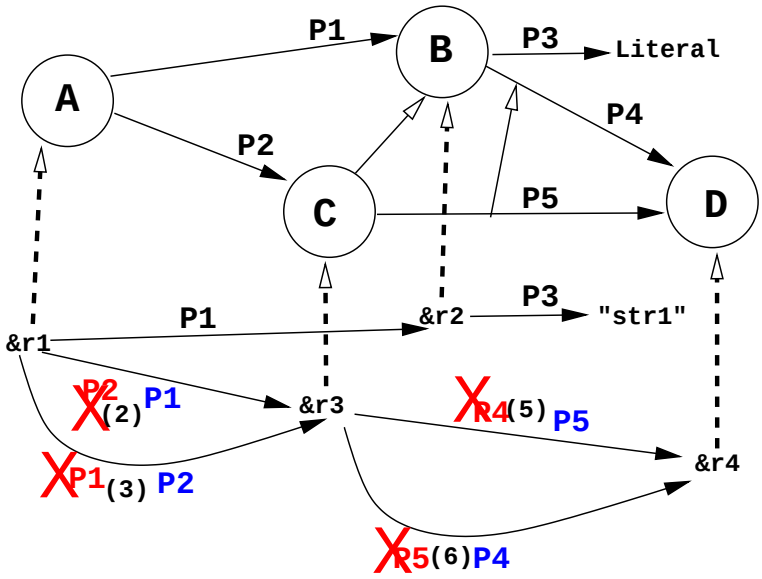

Figure 2.9: *Examples of some REPLACE-classification operations for property instances:*

(1) REPLACE $P1$ <- $P2(\&r1, \&r2)$

(2) REPLACE $P2$ <- $P1(\&r1, \&r3)$

(3) REPLACE $P1$ <- $P2(\&r1, \&r3)$

(4) REPLACE $P3$ <- $P1(\&r2, "str1")$

A REPLACE operation for property classifications can have the effect of an INSERT or a DELETE operation if $OldQualProperyName$ and $NewQualProperyName$ are related through subsumption. For example, in figure 2.9 statement (5), $P5$ is a subclass of $P4$, so the operation has the same effect as a DELETE operation. In figure 2.9 statement (6), $P4$ is a super-class of $P5$, so the operation has the same effect as an INSERT operation. This observation does not hold in case of REPLACE for class instance classification, because a modification of a class instance might affect the property instances attached on it, while the opposite is not true.

In figures 2.9 statement (2) and (3) we present some examples of updates that cannot be made using an INSERT or a DELETE operation.

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 REPLACE 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.

The first extension that we propose is to allow multiple atomic formulas, in an INSERT, DELETE or REPLACE 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 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 comes ”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 REPLACE clauses before the FROM clause of an update statement. In this case, the order of execution is from top to bottom.

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
```

Sequences of update operations of the same kind, seperated by a commna operator, can be placed in the same statement with other operations of the same or different kind. In this case, the order of execution is still from top to bottom and from left to right. The RUL statement of example 18.a is not equivalent to the one of example 17, because the order of INSERT and DELETE operations has changed. Example 18.a is equivalent to example 18.b, though.

Example 18.a: This statement is not equivalent to example 17

```
INSERT hasPCmember(&http://www.iswc06.org#pc, x),
       hasPCmember(&http://www.iswc06.org#pc, z)
DELETE hasPCmember(&http://www.iswc06.org#pc, y)
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
```

Example 18.b: Statements in 18.a and 18.b are equivalent

```
INSERT hasPCmember(&http://www.iswc06.org#pc, X),
       hasPCmember(&http://www.iswc06.org#pc, Z)
```

```
DELETE hasPCmember(&http://www.iswc06.org#pc, Y)
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 sequence of the INSERT, DELETE or REPLACE clauses (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 statements is from top to bottom. Thus, update clauses with multiple operations can be trivially translated into sequences of update statements with a single operation.

3

The semantics of RUL

The purpose of RUL is to provide update functionality on RDF/s description graphs commiting to a number of RDF/s schemata. In this section we explore the world of update languages, stressing out the features we are interested in, so that the design choices of RUL can be justified. More precisely, we focus on two families of update languages and present their features. We, then, select the semantics that is more suitable to RUL from the aspect of expressive power and ensure that the semantics of RUL is deterministic. The formal semantics of the language, based on the semantics of RQL, is presented afterwards, with some illustrative examples.

3.1 Formal 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 [14]. The important contribution of [14], when compared with other formal models of RDF e.g., the RDF semantics by Hayes [23], is the introduction of a rich type system for RDF and RDFS that has been proved valuable in the specification 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. The slight differences of [14, 15] from the RDF semantics in [23]) do not affect the issues covered in this work.

We start by defining the modeling constructs of an RDF resource description and schema graph. We slightly modify the definitions of [14] to cover only the concepts of RDF used in this thesis (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 [14]. 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 $C \cup P$, $\Theta : VS \cup ES \rightarrow C \cup P$ is a function mapping nodes to classes and edges to properties, and $\Lambda : VS \cup ES \rightarrow T$ 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, $\nu : V \rightarrow Values(T)$ is a value function that assigns a value from $Values(T)$ to each node in V and $\lambda : V \cup E \rightarrow 2^{C \cup P} \cup LT$ 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.

- 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.
- 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.

Thus, in a logical sense an *RDF graph* as defined above corresponds to the completion of the corresponding logical theory.

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 $\mathcal{E} : Query \times Graph \rightarrow Tuple$ gives the semantics of RQL query evaluation as defined in [14]. 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 is captured by the semantic function $\mathcal{A} : Update \times Graph \rightarrow Graph$. 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 is given by the equation $\mathcal{A}(\tau;\tau',G) = \mathcal{A}(\tau',\mathcal{A}(\tau,G))$. Composition is an associative operation thus $\mathcal{A}(\tau_1;\cdots;\tau_n,G) = \mathcal{A}(\tau_n,\mathcal{A}(\ldots,\mathcal{A}(\tau_1,G)))$.

The following notation is used repeatedly in the rest of this sections, which formalize the semantics of the various RUL operations:

- $S = (VS, ES, C, P, \prec, \Theta, \Lambda)$ is an RDFS schema graph.
- $G = (V, E, \nu, \lambda)$ be an RDF graph over the schema graph S.
- c is a class, i, i_1, i_2 are URI references and p is a property.
- x is a variable, b is a variable binding expression and f is a filtering condition.

3.1.1 The semantics of `INSERT`

Let $G = (V, E, \nu, \lambda)$ be an RDF graph over the RDFS graph $(VS, ES, C, P, \prec, \Theta, \Lambda)$.

Definition 3: The effect of update `INSERT` $c(i)$ in G is captured by $\mathcal{A}(\texttt{INSERT}\ c(i), G) = (V', E, \nu', \lambda')$ where $V',\ \nu',\ \lambda'$ 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 $\lambda'(a_0) = \{c\}$.
- 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 or literal and instance of $domain(p)$, and i_2 is a URI or literal and instance of $range(p)$.

Definition 4: 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:

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

3.1.2 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.

Definition 5: The effect of this update is captured by $\mathcal{A}(\texttt{DELETE}\ c(i), G) = (V', E', \nu, \lambda')$ where $V',\ E', \lambda'$ 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 $\lambda'(a) = \lambda(a) \setminus C_1$.
- 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, $E' = E \setminus (\{(a, b) :\ \lambda((a, b)) = p \wedge (\exists c_1 \in C_1) domain(p) = c_1\} \cup \{(b, a) :\ \lambda((b, a)) = p \wedge (\exists c_1 \in C_1) range(p) = c_1\})$. □

The preconditions for the execution of the primitive update DELETE $p(i_1, i_2)$ in G is that i_1 is a URI reference and instance of $domain(p)$, and i_2 is a URI reference or literal and instance of $range(p)$.

Definition 6: The effect of this update is the generalization of properties $\mathcal{A}(\texttt{DELETE}\ p(i_1, i_2), G) = (V, E', \nu, \lambda)$ where E' is defined as follows. Let $a_1,\ a_2 \in V$ be nodes such that $\nu(a_1) = i_1$ and $\nu(a_2) = i_2$. Then $E' = E \setminus \{(a_1, a_2)\}$. □

The semantics of DELETE statements with multiple predicates can then be easily defined as in the case of INSERT using composition.

3.1.3 The semantics of REPLACE

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 REPLACE $c(i, j)$ in G is that i is an instance of class c.

Definition 7: The effect of this update operation dealing with class instantiation is captured by $\mathcal{A}$(REPLACE $c(i, j), G) = (V, E', \nu, \lambda')$ where E', λ' are defined as follows.

Let $a \in V$ be the node with $\nu(a) = i$ and b the node with $\nu(b) = j$.

- If $c \in \lambda(a)$ and C_1 is the set $\{c_1 : (c_1 \preceq c \vee c_1 \succ c) \wedge c_1 \in \lambda(a)\}$, C_2 is the set $\{c_2 : c_2 \notin C_1 \wedge c_2 \in \lambda(a)\}$ and C_c the set $\{c_c : c_c \in C_1 \wedge c_c \not\preceq c_2 \wedge c_2 \in C_2\}$, let $C_n c$ be the set $\{c_n c : c_n c \in C_c \wedge c_n c \notin \lambda(b)\}$. Then $\lambda'(a) = \lambda(a) \setminus C_c$ and $\lambda'(b) = \lambda(b) \cup C_n c$.

In addition $E' = E \cup (\{(b, r) : \lambda(a, r) = p \wedge (\exists c_n \in C_n) domain(p) = c_n\} \cup \{(d, b) : \lambda(d, a) = p \wedge (\exists c_n \in C_n) range(p) = c_n\} \setminus \{(a, r) : \lambda(a, r) = p \wedge (\exists c_n \in C_n) domain(p) = c_n \setminus \{(d, a) : \lambda(d, a) = p \wedge (\exists c_n \in C_n) range(p) = c_n\})$. □

In order to understand the meaning of the above formal descriptions, we can see $REPLACE$ as a two step operation. The first step is the removal of i from the set of nodes that are instances of any ancestor or descedant of c.The second step is an addition operation that can be described as an $INSERT\ c(j)$ operation followed by a sequence of $INSERT\ p(k, l)$ operations for such p, k, l that p is each property with instances adjusted to the node i, and either k is i or l is i. The formal semantics of $INSERTp(k, l)$ are given later in this chapter. Note that the operation of the first step is not a $DELETE$ operation and, therefore, the $REPLACE$ operation is not a sequence of $DELETE\ c(i);\ INSERT\ c(j)$. The difference

between the first step of the $REPLACE\ c(i,j)$ and the $DELETE\ c(i,j)$ operation is that the described values are completely removed from all the nodes having a sumsumption relationship with c, even if $c \neq rdf : Resource$.

The $REPLACE\ p(i,i',j,j')$ operation, dealing with property replacements, can also be described as a two-step operation in the same fashion.

Definition 8: The effect of the operation is captured by $\mathcal{A}(\mathsf{REPLACE}\ p(i,i',j,j'),G) = (V',E',\nu,\lambda')$ where $V',\ E',\lambda'$ are defined as follows. Let $a,a',b,b' \in V$ be the nodes with $\nu(a)=i,\nu(a')=i',\nu(b)=j,\nu(b')=j'$.

- If $p \in \lambda((a,b))$ then let P_1 be the set $\{p_1 : (p_1 \preceq p \vee p_1 \succ p) \wedge p_1 \in \lambda((a,b))\}$, P_2 be the set $\{p_2 : p_2 \notin P_1 \wedge p_2 \in \lambda((a,b))\}$ and P_p the set $\{p_p : p_p \in P_1 \wedge p_p \not\preceq p_2 \wedge p_2 \in P_2\}$. Now, let P_np be the set $\{p_np : p_np \in P_p \wedge p_np \notin \lambda((a',b'))\}$. Then, $\lambda'((a,b)) = \lambda((a,b)) \setminus P_p$ and $\lambda'((a',b')) = \lambda((a',b')) \cup P_np$.

□

Definition 9: The $REPLACE\ c,c'(i)$ operation, named "replace classification for class instances", is captured by $\mathcal{A}(\mathsf{REPLACE}\ c,c'(i),G) = (V',E',\nu,\lambda')$ where $V',\ E',\lambda'$ are defined as follows.

- If $c' \preceq c$, the semantics is exactly equal to $INSERT\ c'(i)$.
- If $c' \succ c$, C_{mindle} is the set $\{c_m : c_m \preceq c' \wedge c_m \preceq c\}$ and $c_up : c_m \preceq c_up \wedge c_m \in C_{mindle} \wedge c_m \in C_{mindle}$, then the operation is exactly equal to $DELETE\ c_up(i)$.
- Otherwise, let $a \in V$ be the node with $\nu(a) = i$. If $c \in \lambda(a)$ then let C_1 be the set $\{c_1 : (c_1 \preceq c \vee c_1 \succ c) \wedge c_1 \in \lambda(a)\}$, C_2 be the set $\{c_2 : c_2 \notin C_1 \wedge c_2 \in \lambda(a)\}$ and C_c the set $\{c_c : c_c \in C_1 \wedge c_c \not\preceq c_2 \wedge c_2 \in C_2\}$. Then λ' is the same as λ with the exception that $\lambda'(a) = \lambda(a) \setminus C_c$. The rest of the effects are captured by the formal semantics of $INSERT\ c'(j)$

□

Definition 10: In the case of REPLACE $p, p'(i,j)$, namely the "replace classification for property instances", the semantics is captured by $\mathcal{A}(\texttt{REPLACE}\, p, p'(i,j), G) = (V', E', \nu, \lambda')$ where $V',\ E',\ \nu', \lambda'$ are defined as follows. Let $a, b \in V$ be the nodes with $\nu(a) = i, \nu(b) = j$.

- If $p \in \lambda((a,b))$ then let P_1 be the set $\{p_1 : (p_1 \preceq p \vee p_1 \succ p) \wedge p_1 \in \lambda((a,b))\}$, P_2 be the set $\{P_2 : P_2 \notin P_1 \wedge p_2 \in \lambda((a,b))\}$ and P_p the set $\{p_p : p_p \in P_1 \wedge p_p \not\preceq p_2 \wedge p_2 \in P_2\}$. Then λ' is the same as λ with the exception that $\lambda'((a,b)) = \lambda((a,b)) \setminus P_p$. The rest of the effects are captured by the formal semantics of $INSERTp'(i,j)$

□

3.1.4 Set-Oriented Updates

The syntax of RUL allows us to express set-oriented updates using variables in the INSERT, DELETE or REPLACE clause.

The semantics of update statements with a single INSERT, DELETE or REPLACE 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 REPLACE is slightly more involved, as it can be considered a two-step operation. In the case of $REPLACE\ c(x,y)$ with variables, the two steps are splited. The first step, that is the erasure of the instation link, is evaluated for all values of x. The second step is an $INSERT\ c(y)$ operation for every values binded to y, independently of

the evaluation of x. This is nessecary in order to ensure that the semantics is deterministic, as it was the case with WL_{SPJ}.

The situation becomes more complex when we consider multiple predicates in an `INSERT`, `DELETE` or `REPLACE` clause, or multiple `INSERT`, `DELETE` or `REPLACE` clauses in a single update statement. Obviously, clause order matters in this case as we have already demonstrated, e.g. when 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.

Let us assume an RDFS schema with three 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.

Let us revisit the above example and introduce a new class `C` and a second graph node with URI `i2` that is an instance of class `B`. Let us now consider the following statement:

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

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

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

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

These different orders result in *different states* of the graph. In the first case class `C` ends up with no instances, in the second case it has instance `i2`, and in the third case it has instance `i1`.

Similar issues arise with `REPLACE` even in the presence of a *single* `REPLACE` clause with variables. Let us revisit the previous Example and consider the following statement:

```
REPLACE B(X <- Y)
FROM A{X}, C{Y}
WHERE X != Y
```

We have already stated that the REPLACE statement is not equivalent to a sequence of a $DELETE$ and an $INSERT$, but it can be viewed as a two-step operation consisting of an erasure and an addition procedure. It is easy to see that, although it is an erasure instead of a $DELETE$, the problem of the danger for non-determinism remains.

The solution is to split each `REPLACE` statement to an erasure operation followed by an addition operation and execute all removals corresponding to the variable bindings first, followed by the corresponding insertions. The side-effects of primitive `REPLACE` statements as defined in section 2 are also taken into account. The removal as well as the addition operation differ in the case of $REPLACE$ for instances and the case of $REPLACE$ for instance classification, but as far as it concerns derminism, the problems that have to be solved are the same. A detailed explanation on how the removal and the addition procedure is implemented in each of these cases of $REPLACE$ can be found in section 4. The core idea is that the implementation of $REPLACE$ as an erasure and an addition can be handled in the same way as a RUL statement with a $DELETE$ and an $INSERT$.

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 $Update \times Graph \times Graph$. Non-deterministic update languages have been considered in the past

for other data models e.g., by Abiteboul and Vianu for the relational model [5, 6]. It is a design choice of RUL to avoid non-determinism.

We solve the dilemma of examples such as the above by adopting a semantics similar to the one proposed in [20] 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.

3.2 The semantics of knowledge base updates

The update operations for knowledge bases have different semantics whenever the world described by the base is static or dynamic. A static world does not change and the update operations are used when we are obtaining new information about it or lose confidence in some beliefs. A dynamic world can evolve and the update operations consist of bringing the knowledge base up to date whenever a change occurs.

The fundamental update operations in static world are called ”revision” and ”contraction”, while in a dynamic world they are called ”update” and ”erasure” ([16]). ”Revision”

and ”update” are operations that modify the knowledge base by adding a sentence, while ”contraction” and ”erasure” are used to remove a sentence. When using the operations that deal with a static world, the world itself does not change, but our perception of the world does. Thus, ”revision” and ”contraction” are used when some new information about the real world has been disclosed, forcing us to change our conceptualization of the world in order to represent it in a more accurate manner. But this is not the only change possible, because the real world might change as well. In this case, the knowledge base should be adapted to the new reality. The semantics of this kind of change is quite different, and are captured by ”update” and ”erasure”. ”Update” is similar to ”revision” (it refers to addition of information) while erasure is similar to contraction (it refers to removal of information). However, they both apply when the world dynamically changes, which makes them substantially different from their static counterparts.

We notice that there is no exact mapping between the above update operations and the RUL operations we propose ([10]). The reason for this lack of mapping between these two sets of operations lies on a fundamental difference underlying their definition: the two approaches reflect a different viewpoint on how a change should be interpreted and handled, which renders them incomparable.

Knowledge base update operations are fact-centered (as opposed to modification-centered): a new fact represents a certain need for the evolution of ontology. The ontology engineer (or some automatic sensor or similar device) should identify the type of the new fact, i.e., whether it changed the real world or not and whether it added knowledge or added uncertainty by casting doubt on some existing knowledge (removal of knowledge). These two facts constitute the change. This change is then fed into the system which should identify the actual modifications to perform upon the ontology to address the new fact and perform these modifications automatically. In RUL we are not interested in the fact itself that initiated the change. Rather, we are interested on the actual modifications that should be physically performed upon the ontology in response to this new fact. A belief change system would identify the new fact and decide on the modifications that should be performed upon the ontology, but the modification itself would be performed

by a low-level tool like RUL.

This analysis shows that the two approaches are not directly comparable, as they are based on a different paradigm. As a result, the comparison of the results of RUL (modification-centered approach) with the results of a tool based on some belief change technique (fact-centered approach) would not make much sense. Instead, it is interesting to explore the usefulness of RUL in the design of a belief change management system.

We will use the world of figure 3.1 as an example. In this world, $John$ is an adult and has a child, $Marry$, who is happy. If we add the sentence "$Marry\ is\ unhappy$", then the sentence "$Marry\ is\ Happy$" has to be reconsidered.

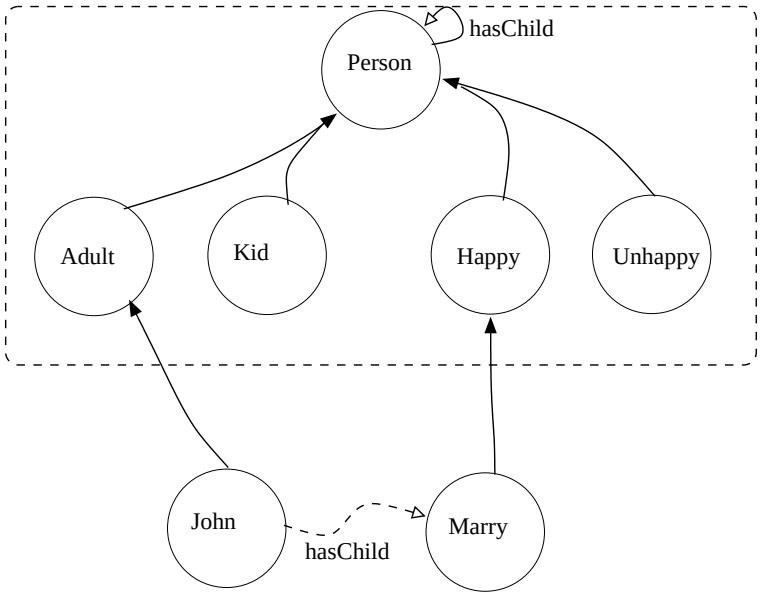

Figure 3.1: An example of a knowledge base description represented as graph.

An "update" or "revision" operation that adds the sentence "$Marry\ is\ unhappy$" would probably remove the sentence "$Marry\ is\ Happy$". This effect is captured by the semantics of RUL REPLACE classification:

```
REPLACE Happy <- Unhappy (&Marry)
```

Now, let us use the operation for adding the sentence "$Marry\ is\ a\ kid$". The addition of that sentence might not affect the other sentences of the model, because the classes

$Happy$ and Kid are not disjoint. Therefore, the semantics of this operation are captured by RUL INSERT:

```
INSERT Kid(&Marry)
```

If the case of adding the sentence "$Marry\ is\ Unhappy$", it is possible that the property instance $hasChild$ is removed. This effect is captured by a RUL DELETE for property instances.

As another example, a "contraction" operation for the sentence "$Marry\ is\ Happy$" could be captured by the semantics of REPLACE classification:

```
REPLACE Happy <- Unhappy (&Marry)
```

whilefor "$Johnisaperson$" by the semantics of a DELETE:

```
DELETE Person(&John)
```

In the later case, the property $hasChild$ will be also removed, as a side effect of the RUL operation, which could probably be consistent with the semantics of the knowledge base update operation.

In general, the semantics of knowledge base updates cannot be described with sequences of RUL operations, but a high level knowledge base update language can rely on the low level update operations provided by RUL, in the same sense as RUL operations rely on database update operations.

The description of multiple knowledge base update operations, e.g. operations for sets of sentences ([11]) with RUL, is a challenging issue. Knowledge base update operations do not directly correspond to RUL ones. The designer of the knowledge base update language should be able to group couples of update operations and sentences by the sequence of RUL statement they are implementing with (e.g. group together the sentences of an "erasure" that can be described with a RUL DELETE). Then, the high level knowledge update language can take advantage of the set-oriented semantics of RUL. The details of such an approach are out of the scope of this thesis, and can be considered future work.

3.3 The semantics of other RDFs update languages

The update languages proposed so far are MEL ([22]), rdfDB query language ([12]) and, of course, RUL ([19]).

The most interesting proposal is MEL that has been developed in the framework of QEL and it is based on Datalog. 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 sub-graph centered approach but consistency of updates with respect to the employed RDFS schemata is not respected. Furthermore, no formal semantics or detailed behavior description have been given for MEL. More precisely, MEL supports three update operations, namely insert, delete and update, which modify RDF triplets of the form ”subject-property-object”.

One difference between MEL and RUL is that in our approach the class instances can be handled independently to the property instances, while in MEL an update statement must be specified as a triplet update. For example, if a resource $\&RULpaper$ must be inserted as an instance of the class $Paper$, in MEL this could be achieved by inserting the triplet ”$Paper$: $\&RulPaper$ - P - O”, where P and O are variables denoting properties and the resources this properties end to, respectively, but there must be some query constraints for variables P and O, so that the resource $\&RULPaper$ is inserted as a subject of some property instances. According to the language description, the resource $\&RULPaper$ cannot be inserted without being related with some property instance, which functionality is supported in RUL.

Because of the ability of RUL to handle resources independently, the semantics of the MEL insert, delete and update operations is different to the semantics of RUL INSERT, DELETE and REPLACE operations respectively. We can compare the semantics of MEL with the semantics of RUL update operations for property instances.

The MEL insert and RUL INSERT-for-property-instances operations share the same semantics only if the subject and the object of the inserted property instance exist in the description base. In RUL INSERT operation, the insertion of a property instance is not

allowed in that case, while in MEL this is a way to insert new class instances.

The MEL delete and RUL DELETE-for-property-instances operations differ because of the RUL DELETE side effects. More precisely, the deleted instance in MEL is erased so that it is not an instance of the specified property or any ancestor of it. We have seen that in RUL DELETE we usually erase only the classification link that ends to the property and we insert a new classification link from the instance to the closest ancestor of the property.

The MEL update and RUL REPLACE-for-property-instances operations differ in the same way that MEL insert and RUL INSERT-for-property instances differ. It is possible to insert new resources in the description base by using the MEL update operation, while in RUL REPLACE this is prohibited. We don'Wt know if the side effects of RUL REPLACE operation are also side effects of MEL update, as the exact semantics of the MEL operations are not described.

In general, RUL is expressively more powerful than MEL. Apart from the differences and limitation described above, MEL does not support something similar to the RUL REPLACE classification operation. MEL and RUL share the same notion of safety in set-oriented update statements, but we do not know if MEL semantics is deterministic, as this issue has not been studied. Therefore, we cannot compare the set-oriented semantics of the languages.

The rdfDB Query Language 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.

3.4 Semantics of database update languages

Update languages on structured data are presented in this section. The expressive power and determinism are the features of update languages we are interested in. An update language provides update operations so that an update operation over a database instance will result to a modified database instance. Intuitively, an update language can be modeled as a

mapping from a database instance to another. More formally, given an input schema R and an output schema S, an update language is a subset of $instanceOf(R)XinstanceOf(S)$. Note that an update language that modifies only the data of a description base (like RUL) can be a subset of $instanceOf(R)XinstanceOf(R)$.

Abitebul and Vianu ([6]) give a formal definition of the update operation, with respect to the deterministic features of it. They state that a non-deterministic update from R to S is a subset of $instanceOf(R)XinstanceOf(S)$ which is recursively enumerable, and C-generic for some finite C. A finitely non-deterministic update from R to S is a non-deterministic update r such that for each instance I over R, the set $(J|(I,J) \in t)$ is finite. A deterministic update (from R to S) is a mapping from $instanceOf(R)$ to $instanceOf(S)$ which is partially recursive, and C-generic for some finite C. Our definition is a simplified explanation of this formal one.

Let R and S be database schemas, and let C be a finite set of constants.

Definition 3: ([25]) A mapping q from $inst(R)$ to $inst(S)$ is C-generic if and only if for each database instance I over R and each permutation ρ of the set of constants that is the identity on C, $\rho((q)I)) = q(\rho(I))$. When C is empty, we simply say that the query is generic. □

Genericity states that the query is insenitive to renaming of the constants in the database (using the permutation ρ). It uses only the relationships among constants provided by the database and is independent of any other information about the constants. The set C specifies the exceptional constants named explicitely in the query. These cannot be renamed without changing the effect of the query.

The core characteristic of an update language is its expressive power. The concept of expressive power has been defined and analyzed in the literature ([25], [6], [20], [18]) and depends on the functionality of the update language as well as on if the language is deterministic.

In the following we deal with database update languages. A database update language provides modifications on the data of a database with a specific schema. We do not deal with languages that modify the schema or perform updates independent to the database

schema.

3.4.1 The family of database update languages

An update operation $op_R(t1, t2, ...)$ on a relation R, modifies the relation R according to the values stored in the tuples $t1, t2, ...$ A primitive update operation is an operation where the tuples $t1, t2, ...$ are constant values. The tuples $t1, t2$, etc. are of type R.

A very primitive update language is LST ([18]) supporting the following syntax:

$stmt := stmt; stmt$
$| \ insert_R(t)$
$| \ delete_R(t)$

where $insert_R(t)$ means "insert the tuple t in relation R" and $delete_R(t)$ stands for "remove any tuple t from the relation R". The absence of an iteration construct is the distinguishing feature of this language.

A language with an iteration construct is SdetTL ([6], [18]). It is obvious that iteration means support for non-primitive updates.

$stmt := stmt; stmt$
$| \ insert_R(t)$
$| \ delete_R(t)$
$| \ erase_R$
$| \ while \ x : Q(x) \ do \ stmt$

The difference between $delete$ and $erase$ is that the former removes the tuple t from the relation R, while the later erases the whole relation R. The $erase$ functionality in SdetTL is nessecary because it cannot be expressed otherwise, as we explain in the next paragraphs.

The semantics of the while construct is not trivial. Here $Q(x)$ is a query in some query language and x a variable binding (or a set of variable bindings). For every x satisfying

Q, the statement $stmt$ is executed as a primitive operation. When the $stmt$ statement has been executed for all values bound to x, the resulting database state is the union of the effect of each atomic $stmt$ execution. The procedure is repeated again on the new database state, until there are no values of x satisfying Q.

In detail, let $t1, t2, ...$ be the result of the query Q. Let $stmt$ be a sequence of primitive update statements so that $stmt_R(t1)$ results to a database state $R1$, $stmt_R(t2)$ to a database state $R2$, etc. Note that in this context, each $stmt$ is executed over the initial database state R, and not over any intermediate states. The result of a while construct is the parallel execution of the following statements: $stmt_R(t1), stmt_R(t2), ...$. The initial database is now modified to a new database state R' given by the union of each separate state:

$R' \leftarrow R1 \cup R2 \cup ...$

Q is evaluated again, over R'. If the result of the execution of Q is not an empty set, the procedure is repeated, resulting to a new database state R'', etc.

Another interesting update language is WL ([20]), with the following syntax:

$stmt := stmt; stmt$

$| \; insert_R(a)$

$| \; delete_R(a)$

$| \; replace_R(a, c)$

$| \; if \; Q \; then \; stmt$

$| \; foreach \; x : Q(x) \; do \; stmt$

Again, Q is a query in some query language and x a set of variable bindings, but the semantics of $foreach$ is different from the one of the $while$ construct in SdetTL. If $stmt$ is an atomic operation, then for each value bound to x, the $stmt$ is executed. Each $stmt$ execution affects the database state modified by the previous one.

If $stmt$ is a sequence of atomic updates $stmt1; stmt2; ...$, then for each value bound to x, $stmt1$ is executed, affecting the state of the database. After $stmt1$ has been executed for all values assigned to x, $stmt2$ is executed over the modified database for the same

values. Then $stmt3$ follows, and so on. Under this light, the if construct is just a special case of $foreach$ ([18]).

3.4.2 Comparison of the semantics of the iteration constructs

Thee two iteration constructs presented previously have different semantics. Iteration constructs are important, because they can extent an update language to take benefit of the expressiveness of a query language. Therefore, an update language relies on the iteration constructs in order to provide non-primitive, set-oriented updates.

A more formal and descriptive definition for the update operations: Let I_S be an instance of the database schema S, R a relation in that schema and $t1, t2, ...$ some valid tuples of R. An update operation $op(I_S, R, t1, t2, ...)$ is a subset of $\{I_S\} \times instances(S)$, where $instances(S)$ is the set of database instances over S.

To begin with, in the $while$ construct, the query Q is the condition of the iteration, so it might be evaluated more than one times (one per iteration step). This construct can be viewed as a two-level iteration: an iteration based on the query condition and an iteration over the values satisfying the query. In the first-level iteration, each time, the query is evaluated over the current database state. In the $foreach$ construct, the query is evaluated only once and the iteration occurs only on the retrieved values. The meaning of the foreach construct is that the query is used as a filter for the values of x, rather than a condition that must hold. What's more, the query is evaluated only over the initial, input database state, rather than the intermediate, modified states produced by the atomic updates.

A general example might help to illustrate the above:
(1) $while\ x : Q(x)\ stmt_R$
(2) $foreach\ x : Q(x)\ stmt_R$

Let I_0 be the initial database state, $t_{I_0}1, t_{I_0}2, ...t_{I_0}n$ the result of the evaluation of Q over I_0, and R a relation described in the database schema.

(1) In the $while$ case, the resulting database instance is the following:

$I_1 = stmt(I_0, R, t_{I_0}1) \cup stmt(I_0, R, t_{I_0}2) \cup ...$ The next step is to evaluate Q over

I', resulting the following set of tuples: $t_{I_1}1, t_{I_1}2, ...$ The new database state:

$I_2 = stmt(I_1, R, tI_1 1) \cup stmt(I_1, R, t_{I_1}2) \cup ...$

The procedure is repeated until there is a database state I_m for which Q returns an empty set.

(2) In the $foreach$ case, the resulting database instance is the following:

$I_1 = stmt(\ stmt(\ stmt(I_0, R, t_{I_0}1), R, t_{I_0}2)...), R, t_{I_0}n)$

The meaning of the above formula is that the $stmt$ statement for $t_{I_0}2$ operates over the database instance produced as a result of the $stmt$ statement for $t_{I_0}1$.

In the example above it is clear that, in the $while$ case, each atomic statement produced by the iteration is executed only on the initial database state, while in the $foreach$ case, each atomic update operates over the result of the previous one. If the statement in the body of the iteration expression is not a single primitive operation, but a sequence of primitives and/or non-primitive ones operations, then the order of that sequence does not matter in case of the $while$ construct, but it is meaningful in the case of $foreach$.

For example:

(1) $while\ x : Q(x)\ stmt1_R; stmt2_R$

(2) $foreach\ x : Q(x)\ stmt1_R; stmt2_R$

$stmt1$ and $stmt2$ affect the same relation R (although this is not important in this context). Let's take a snapshot from the execution of the above iterated statements, while they modify the database state I using the tuple $t1$ as input:

(1) $I' = stmt1(I, R, t1) \cup stmt2(I, R, t1)$

(2) $I' = stmt2(stmt1(I, R, t1), R, t1)$

Somewhere in the process, $stmt1$ modifies R based on $t1$ (e.g. deletes $t1$ from R) and $stmt2$ operates on R based on $t1$ as well. In (1), $stmt1$ and $stmt2$ are executed in parallel and over the initial state of R. In (2), $stmt1$ changes R so that $stmt2$ operates on a modified R.

Now lets consider the following iteration expressions, where the order of $stmt1$ and

$stmt2$ is reversed:
(3) $while\ x : Q(x)\ stmt2_R; stmt1_R$
(4) $foreach\ x : Q(x)\ stmt2_R; stmt1_R$

The expression (3) is equivalent to (1), while the expression (4) is not equivalent to (2), as the order of execution does matter in case of $foreach$. To illustrate this, let's take the same snapshot from the execution of (3) and (4):
(3) $I' = stmt1(I, R, t1) \cup stmt2(I, R, t1)$
(4) $I' = stmt1(stmt2(I, R, t1), R, t1)$

In (2), $stmt1$ operates over I, while $stmt2$ operates over the result of $stmt1$. In (4), $stmt2$ operates over I and $stmt1$ on the result of $stmt2$.

3.4.3 Expressive power

Definition of expressive power for update languages ([18]): A database update language L1 is more expressive than a database update language L2 if L1 can express a superset of the mappings expressible in L2. More formally, let S be a database schema, $instances(S)$ be the set of database instances over S and u_{L1}, $u_{L2} \in instances(S) \times instances(S)$ be the sets of all mappings expressible in database udpate langues L1 and L2 respectively, then L1 is more expressive that L2 if and only if $u_{L2} \subset u_{L1}$. We say that L1 is as expressive as L2 if $u_{L2} \subseteq u_{L1}$ and $u_{L2} \subseteq u_{L1}$.

There are cases of languages that cannot be compared in terms of expressive power. More formaly, if there is a subset $u'_{L1} \subseteq u_{L1}$ and a subset $u'_{L2} \subseteq u_{L1}$ so that $u'_{L1} \nsubseteq u_{L2}$ and $u'_{L2} \nsubseteq u_{L1}$, then the update languages L1 and L2 are expressivily incomparable to each other, which means that each language can express a set of mappings that other cannot.

We have already seen than in order to provide non-primitive updates, an update language relies on a query one. The selection of the query language can affect the expressive power of the update language. More precisely, the expressive power of an update lan-

guage depends on two factors:
- the semantics of the supported update operations
- the power of the underlying query language.

An update language can be notified as U_Q, where U is a set of update semantics and Q a query language. For example, WL_{SPJ} is WL with a Select-Project-Join conjuctive query language. The various classes of update languages of that form and the expressive relation between them is illustrated in figure (3.2). The expressive power of an update language, e.g. WL, may change according to the underlying querying language, for example, WL based of $Fixpoint$ (WL_{FO}) is more powerful that WL based on conjuctive SPJ (WL_{SPJ}).

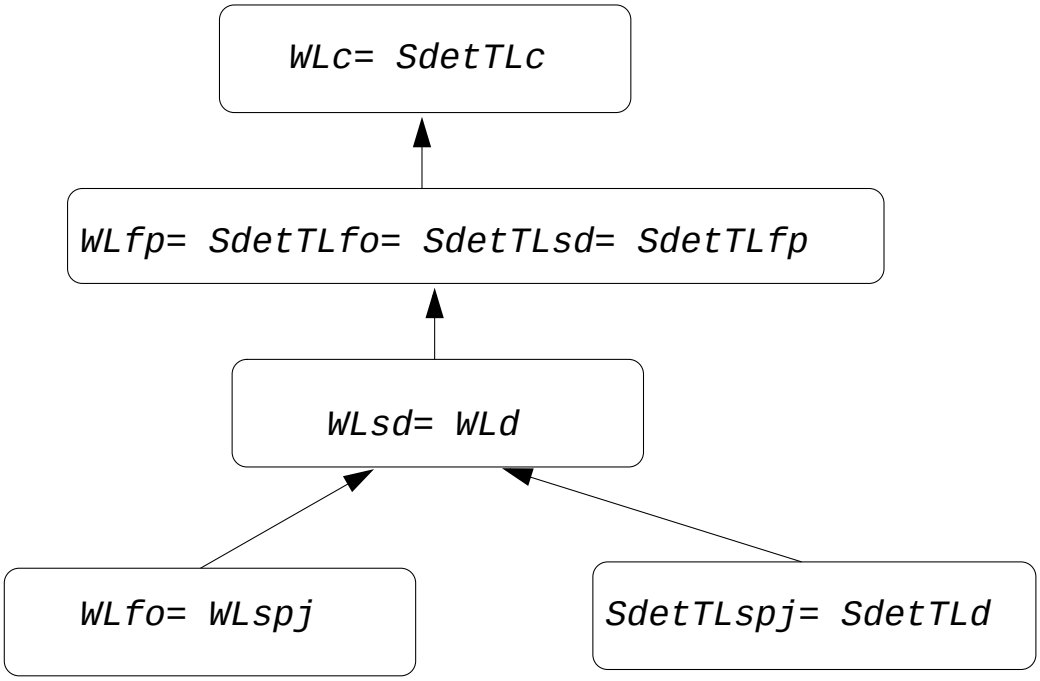

Figure 3.2: Classification of database update languages [18]

RUL is based on RQL, which is an SPJ query language with transitive closures on subsumption relationships on classes an properties. We are interested in WL_{SPJ} and $SdetTL_{SPJ}$, because these are the more expressive families of update languages that use

an SPJ underlying query language, as shown in fingure (3.2). Unfortunately, WL_{SPJ} and $SdetTL_{SPJ}$ are incomparable in terms of their expressive power. RUL is based on WL_{SPJ}, because its semantics is more suitable as it is analysed later.

In particular, both languages support a primitive $insert$ and $delete$ operation, but the semantics of these operations differ, if put in an iteration construct. An iterated $insert$ operation in $SdetTL$ is the union of the atomic $insert$s and this is equivalent to an iterated $insert$ operation in WL. For example, a sequence of $insert_R(t1), insert_R(t2)$ will result to a modified relation R that will contain both $t1$ and $t2$ tuples. This result will be the same in WL and $SdetTL$ languages.

The different behavior is exhibited in the case of the delete operation. In specific, the iterated delete operation in $SdetTL$ is the union of the effect of each atomic delete. Under this light, a set of delete operations over the same relation will result to an output database instance that is the same as the initial one, therefore the operation will have no effect ([18]). More precisely, if the effect of an atomic delete operation $delete_R(t1)$ is the removal of a tuple $t1$ from R and the effect of $delete_R(t2)$ is the removal of a tuple $t2$ from R, the effect of the operation is the union of the results: $R \leftarrow \{R - \{t1\}\} \cup \{R - \{t2\}\}$, according to the semantics of $while$. The union of an output instance of relation of R where a tuple $t1$ has been removed, and another output instance or relation R where another tuple $t2$ has been removed is the initial instance of R where no tuples have been removed. Therefore R remains unchanged.

On the contrary, WL can be used to describe a destructive iterated delete operation. A WL $delete_R(t1)$ will remove tuple $t1$ from R and the $delete_R(t2)$ operation following, will also remove $t2$ from the modified relation. At the end of the iteration, all deleted tuples will be missing and the database instance will have been modified: $R \leftarrow \{R - \{\{t1\} \cup \{t2\}\}$. In general, an iterated $insert$ operation is equivalent in both languages, while the iterated $delete$ operation is meaningful only in WL. $SdetTL$ introduces the $erase$ construct to deal with this problem. The $erase$ construct is used to empty a relation. An $erase$ followed by an $insert$ can be used to simulate an iterated delete operation equivalent to the one that is expressible in WL. In every step of the iter-

ation, the temporary output relation is the result of an $erase$ that empties the relation and the insertion of some tuple that should not be removed. Therefore,

WL: $foreach\ x : Q(x)\ delete_R(x)$

is equivalent to

SdetTL: $erase_R\ while\ x : Q'(Q(x))\ insert_R(x)$

where $Q'(x) = R \wedge \neg Q(x)$: a query that returns the tuples of R which do not satisfy Q.

Therefore, SdetTL does not lack the desired feature of an iterated delete operation, as long as Q' can be expressed in the underlying query language for every Q, which is not the case with SPJ. In general, $SdetTL$ and WL are comparable only if the underlying language supports Q', in which case $SdetTL$ is more powerful that WL ([18]). It is a fact that we cannot express Q' in SPJ, and $SdetTL_{SPJ}$ cannot provide an iterated $delete$ construct for removing specific tuples from a relation.

RUL is based on WL because the removal of values is a desired effect. More precisely, the ability to remove tuples from certain relations in the database, according to the values retrieved by a query, is needed to implement the effect of RUL DELETE and REPLACE operations, as well as the side effects of the RUL INSERT operation. A detailed explanation of how the iterated database delete operation is used to implement RUL operations is given in chapter 4 The main advantage of $SdetTL$, through, is that its semantics is always deterministic. We will deal with the non-deterministic aspects of WL expressions later and we will present the deterministic semantics of WL, as it has been studied in the literature ([20], [18]).

3.4.4 Determinism

An update language is deterministic if it supports only deterministic update operations. An update operation $op(I_S, R, t1, t2, ...) \subset \{I_S\} \times instances(S)$ is deterministic if for each initial database state I_S over each database schema S, there is exactly one resulting database instance I'_S so that $op(I_S, R, t1, t2, ...) \subseteq \{I_S\} \times \{I'_S\}$.

It is trivial that primitive database update operations are always deterministic, as they deal with the addition or removal of a single tuple in a single relation. The $SdetTL$

$erase$ operation is, also, obviously deterministic, because the result of an erase operation is always the same (an empty relation). Therefore, only the iteration constructs might entail the danger of non-determinism.

It has been shown that the iteration construct of $SdetTL$ is always deterministic ([6]). In specific, $while$ produces a set of two-level iteration steps, as described in the previous section. At the first level the query is deterministically evaluated over the initial database. At the second level, each inner operation is executed over the initial database, producing a temporary database state. After all steps are completed for one query evaluation, the new overall database state is the union of the separate states produced by each two-level operation. The process is repeated with another query evaluation over the new database state, until the query/condition is not satisfied.

The result of each second-level iteration is the union of the result of the produced atomic operations over the initial relations. The result of this union is always the same, regardless the order of execution of the produced intermediate operations. As for the first-level iteration, it can be viewed as a state transition. Each transition is deterministically depended on the previous one, as long as the underlying query language is also deterministic. The union operation after the second-level iteration is the key feature that ensured the determinism of the state transitions.

We need to show how WL could also be implemented with deterministic semantics. The core idea is to define properly the semantics of the $foreach$ construct. We have seen that in WL, each produced update operation is affected by the result of the operation executed before. Because of this characteristic, the order of execution of the statements is important, if the semantics of WL must be deterministic.

For example the following statements will have a different effect if executed over the same initial database state I:

Database state $I = R\{0, 1, 2, 3\}$
A database with one relation R that contains four tuples.

Query $Q : ans(x) \leftarrow R(x) \cap (x > 2)$
A query that returns the values of R that are greater than 2

The result of the query is $Q(x) = \{\{3\}\}$

The two statements:
(1) $foreach\ x : Q(x)\ insert_R(x); delete_R(x)$
(2) $foreach\ x : Q(x)\ delete_R(x); insert_R(x)$

Statement (1) produces the following update operations:
$insert_R(\{3\}); delete_R(\{3\});$

so that at the end of the execution of (1), the database state will be
$I' = R\{0, 1, 2\}$

Statement (2) produces the following update operations:
$delete_R(\{3\}); insert_R(\{3\});$

so that after the execution of (2) the database state is I" = I, because the deleted tuple $\{3\}$ is inserted afterwards.

A $foreach$ produces a sequence of atomic statements, one for each value set retrieved by the query. Although a deterministic query language always returns the same result set for the same query over the same database state, the order of the results in the set can vary. In other words, the same query Q might returns always the same set of results each time it is evaluated over the same database instance, but the order of the results in the set may change from time to time. If this order is used to produce a sequence of update statements, then determinism is in lost.

There are two possible semantics for the foreach construct that contains more than one inner statements, like the one following:

$foreach\ x : Q(x)\ stmt1; stmt2; stmt3$

Option 1: all the inner statements are executed in order for each value retrieved by Q. According to the first option we execute $stmt1$, $stmt2$ and $stmt3$ (in that order) for the first value retrieved by Q, then repeat for the next value, etc.

Option 2: each inner statement is executed for all values of Q before any execution of the statement following. According to this option we execute $stmt1$ for all values of Q, then $stmt2$ for the same values, and finally $stmt3$.

If the retrieved results of Q are $\{x1, x2, x3\}$, then the following are the sequences of update operations produced in each case:

Option 1:
$stmt1(x1); stmt2(x1); stmt3(x1);$
$stmt1(x2); stmt2(x2); stmt3(x2);$
$stmt1(x3); stmt2(x3); stmt3(x3);$

Option 2:
$stmt1(x1); stmt1(x2); stmt1(x3);$
$stmt2(x1); stmt2(x2); stmt2(x3);$
$stmt3(x1); stmt3(x2); stmt3(x3);$

We will show that the semantics described in option 2 is deterministic, while the semantics in option 1 is not.

First, let's prove that the semantics described in option 2 is deterministic:

It is enough to show that if a statement $stmt$ is deterministic, then a sequence of statements $stmt(x1); stmt(x2); ...$ is equivalent to any reordering of this sequence. The $stmt$ statement can either be an $insert$, a $delete$ or a $foreach$.

If it is an $insert$, then it is trivial that any order of the same $insert$ operations will have the same effect (values $x1$, $x2$, etc. will be inserted). The same holds for any ordering

of the same sequence of $delete$ operations (values $x1$, $x2$, etc. will be erased).

We need to show that the order of a sequence of $foreach$ statements is also deterministic, when these statements are produced from another $foreach$ statement. This is the case of nested $foreach$. It has been shown, though, that any nested foreach statement can be flatten ([20]) by pushing the query of each nested foreach statement up to the query of the first level statement:

$foreach\ x : Q1(x)\ do$

$stmt1(x);\ foreach\ y : Q2(x, y)\ do\ stmt2(y)$

$\rightarrow$

$foreach\ x, y : Q1(x) \cap Q2(x, y)\ do\ stmt1(x); stmt2(y)$

Therefore, the semantics of option 2 is deterministic, because each foreach statement produces sequences of statements of the same type (namely $insert$ or $delete$) grouped together.

In order to prove that option 1 semantics is not deterministic ([20]), we can use an example, as the following:

Database state $I : R1\{1, 2, 3\}$, $R2\{2, 3, 4\}$

The Query $Q : ans(x, y) \leftarrow R1(x) \cap R2(y)$

This is the foreach statement:

$foreach\ x, y : Q(x, y)\ do\ insert_{R1}(x); delete_{R1}(y)$

Case 1: Q returns the results in that order: $\{(2, 2), (3, 3), (2, 3), (3, 2)\}$ producing the causing the following operation sequence:

$insert_{R1}(2); delete_{R1}(2)$ state of R1: $R1\{1, 3\}$

$insert_{R1}(3); delete_{R1}(3)$ state of R1: $R1\{(1\}$)

$insert_{R1}(2); delete_{R1}(3)$ state of R1: $R1\{(1, 2\})$

$insert_{R1}(3); delete_{R1}(2)$ state of R1: $R1\{(1, 3\})$

resulting this database state I': $R1\{1, 3\}, R2\{2, 3, 4\}$.

Case 2: Q returns the results in that order: $\{(2,2),(3,3),(3,2),(2,3)\}$

$insert_{R1}(2); delete_{R1}(2)$ state of R1: $R1\{(1,3\})$

$insert_{R1}(3); delete_{R1}(3)$ state of R1: $R1\{(1\})$

$insert_{R1}(3); delete_{R1}(2)$ state of R1: $R1\{(1,3\})$

$insert_{R1}(2); delete_{R1}(3)$ state of R1: $R1\{(1,2\})$

resulting this database state I': $R1\{1,2\}, R2\{2,3,4\}$.

In these two cases, the resulting database is different. It is not necessary to continue with examples presenting cases of non-determinism, but it is interesting that in this example there are even more possible resulting database states, for different orders of the query result set. More cases of non determinism have been investigated in the literature ([6], [20], [18]).

WL with deterministic semantics in the $foreach$ construct is possible, if we chose option 2. The semantics of $insert$ and $delete$ is obviously deterministic. WL in its original form contains a $replace_R(x,y)$ construct, which can be also viewed as a complex operation consisting of a $delete$ and an $insert$:

$replace_R(x,y) := delete_R(x); insert_R(y)$

In case of a $foreach$ containing a replace construct, we have to deal with replace as if it actually was a separate $delete$ followed by a separate $insert$ statement, otherwise the $replace$ statement won't be deterministic. A $replace$ operation may either be deterministic or primitive, but not both.

The later observation is important for specifying the exact semantics of the $replace$ statement. In fact, an iterated $replace$ is translated as an iterated $insert$ followed by an $iterated$ delete. For example, if a $foreach$ statement $replace$s the values $(1,2,3)$ of a relation with the values $(2,3,4)$ respectively, then the order of execution is the following: values 1, 2 and 3 are removed from the relation and then values 2, 3 and 4 are inserted.

Compared to the previous declarative update languages for relational databases, RUL has been designed for updating RDF/s description. For example, the semantics of WL_{SPJ}

is not sufficient to describe a language that affects RDF data, because it lacks the ability to directly deal with concepts derived from the RDF/S model, like IsA relationships of class/property inheritance. In section 4, we will use WL_{SPJ} to describe the implementation of RUL over various database representations, where we deal with similar problems while implementing the deterministic semantics of RUL. As there are many analogies in the semantics of RUL and the semantics of the previously presented database update languages, we will chose a deterministic update language to implement RUL over various database representations of RDF/S descriptions. As we will see, the deterministic semantics of RUL rely on the deterministic semantics of the chosen database update language, but there are also some issues concerning determinism that are not directly related to the later semantics.

3.4.5 Selecting a database update language

RUL is implemented over a database udpate language. We have already seen that the desired feature of determinism is supported in both WL_{SPJ} and $SdetTL_{SPJ}$. The operations of RUL can be implemented with any of the above database update languages, as they both provide enough epxressive power and they are both deterministic.

We prefer to implement RUL with WL_{SPJ} for performance reasons. More precisely, the iteration operation of $SdetTL$ requires multiple evaluations of the same query over different states of the underlying database instance, while in WL the query is evaluated only once. What's more, according to the semantics of WL, the update operations inside a foreach clause directly affect the database, while in $SdetTL$ the effects are computed and stored in a temporary place until the iteration is completed. After the completion of the iteration in $SdetTL$, the temporarily stored effects have to be merged and applied to the database instance. The performance disadvantage of WL is that the results that are retrieved by the query have to be stored in a temporary place. This is necessary in order to achieve the deterministic semantics of WL. Compared to $SdetTL$ this is not a disadvantage, though, as in the later there are also some information that have to be temporarily stored. The size of the temporarily stored information in $SdetTL$ depends on

the size of the retrieved query results, but in SdetTL this information has to be stored as many times as the the evaluations of the query.

Another reason for chosing WL is that its iteration semantics are similar to the set-oriented semantics of RUL. In RUL, the RQL query is evaluated. Then each RUL operation is applied over the retrieved results. This is exactly what happens with the WL udpate operations that are nested inside a $foreach$ clause. In chapter 4, we will see how this similarity will prove handy in implementing the set-oriented semantics of RUL.

4

The implementation of RUL

RUL has been implemented as part of the RDF suite ([1]). RUL implementation follows the paradigm or the RQL implementation and the design decisions taken are, as much as possible, compatible with the design principles of RQL. Therefore, the architecture of RUL, presented in figure 4.1, is very similar to the one of RQL, as we show in the later. An RUL interpreter translates the queries into SQL statements, which are then executed. The parts of a RUL statement that can be expressed with an RQL query, are actually translated and executed by the RQL interpreter. RDF schema and data in RDF suite are stored in the underlying DBMS. At the moment there are three alternative database representations ([29]) that are all supported by the RUL interpreter. The RUL to SQL translation is affected by the selected database representation.

4.1 RUL vs RQL implementation

RUL is implemented as an extension of RQL. The key components of the later are a syntax parser, a graph constructor, an RDF/s validation and, finally, an SQL statement generator module. RUL extents its of these components to support the RUL functionality.

In RUL design, the update and the query parts of a RUL statement can be identified and separated, as it has been presented in chapter 2. The INSERT, DELETE and REPLACE parts are the heads of a RUL statement and they are the only reserved words that do not appear in RQL. The rest of a RUL statement, namely the FROM, the WHERE and the NAMESPACE clauses are identical to the ones appearing in RQL. It was trivial to modify the RQL parser to verify the syntax of RUL statements and produce a syntactical tree.

RQL then produces a graph based on the syntax tree, by finding the relations of the various parts of the input statement, that are represented as tree nodes and connect them by adding extra arcs where there are relations we want to represent. As far as it concerns RUL, the graph constructor module has been extended to manage the INSERT, DELETE and REPLACE statements, and let RQL deal with the constants and variables present in an RUL clause, as if they where part of a SELECT clause. More precisely, the INSERT, DELETE or UPDATE clause of the statement is represented by a graph node, under which the constants and variables related to it are hanged. In RUL, we are interested in the identification of these variables in the rest of the statement and also to check that each variable that appears in the head of an RUL statement, also appears in the FROM clause. These functionalities are acquired by reusing the corresponding functionalities already implemented in the RQL graph constructor.

In the example illustrated in figure 4.2,

```
DELETE Paper(X)
FROM {Y}writes{X}, {Conference}hasPC.hasChair{Y}
```

the head of the query is $DELETE\ Paper(X)$, where Paper is a constant denoting an RDF class and X a resource variable. The constructed graph relates the variable X

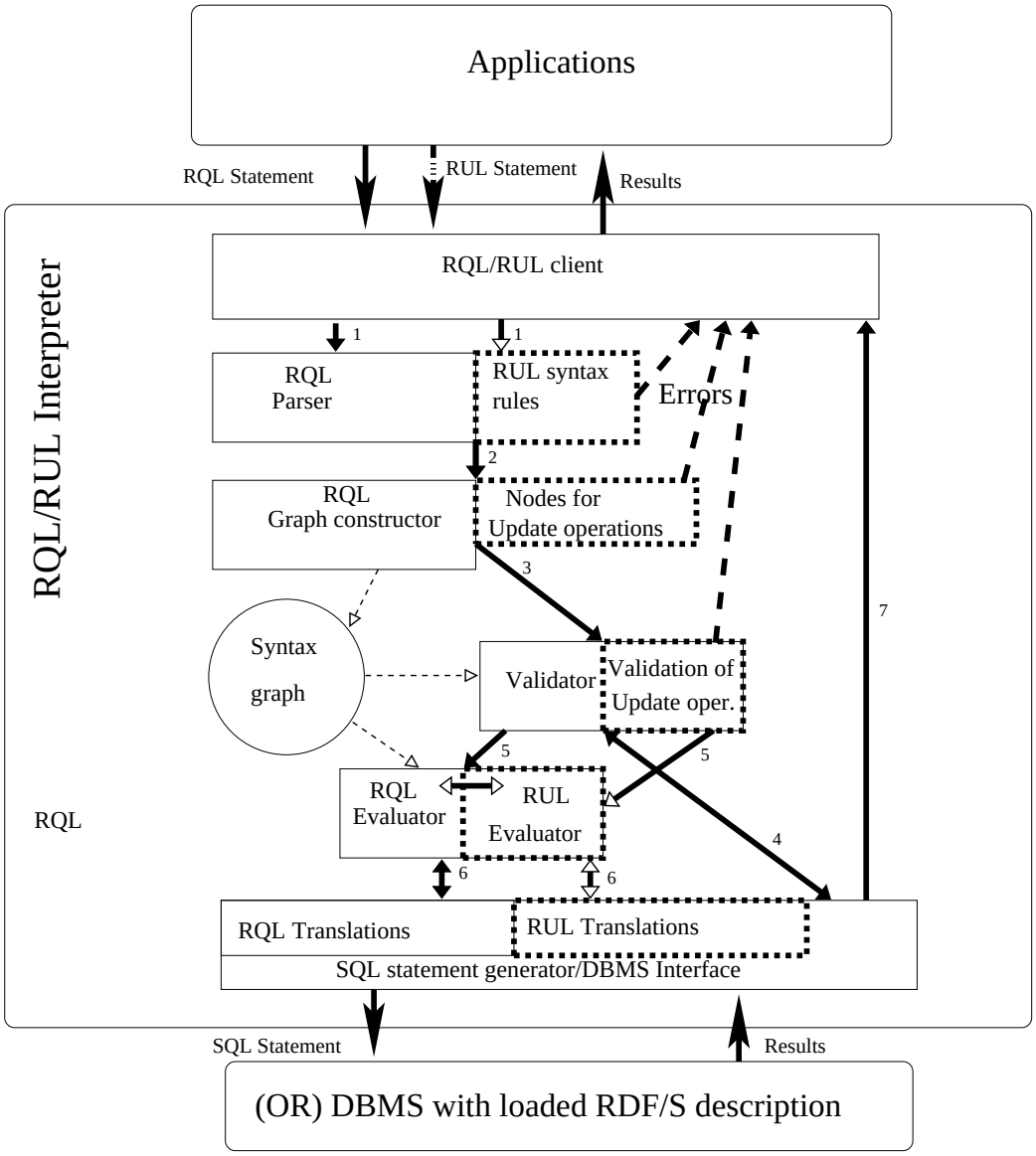

Figure 4.1: *The RUL statements are sent to the client. Parser and graph constructor modules of RQL are extented to handle the RUL syntax. They parse it and construct a syntax graph, that contains nodes for udpate operations. The RQL validator module is also extended to validate the RUL parts of the statement. The validation is performed against the underlying database. The RQL parts of the RUL statement are evaluated first by the RQL evaluator (against the database). The update operations are, then, translated into SQL statements and sent to the database as well. The result is sent to the RQL/RUL client and returned to the user application in an RDF/XML form.*

of the head with the X appearing in the $FROM$ clause. RQL graph constructor module is more sophisticated than that, but the rest of the details of the RQL graph constructor module do not affect RUL implementation and they are ommited because the separation of the update and querying part of RUL statements allows the querying to be handled by the existing RQL implementation.

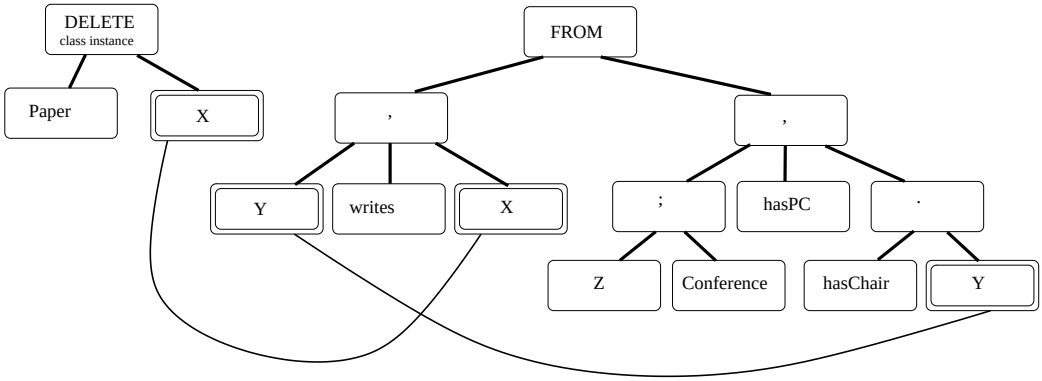

Figure 4.2: *The syntax graph constructed by RQL/RUL graph constructor for the statement of the example. Some arcs connect the various apearences of the same variable in the statement.*

The next step of the interpretation of the RUL statement is the validation of the components of the constructed graph. Here, each constant or variable hanging under the update node is checked against the database description, by performing SQL queries and checking the results. Recall that each constant or variable appearing in a RUL statement head must be of a class, property, resource or literal type. In the example presented above, the $DELETE - class - instance$ statement must be followed by a class name or variable and a class instance name or variable. For example, during the validation process, the database is asked if there exists a "$Paper$" class.

Because of the RQL architecture, it was easy to extent this module to support RUL statements validation. As a matter of fact, all validation queries used in RUL where already implemented for the needs of RQL, so it was enough to call the corresponding high-level validation methods when needed. For example, RUL is aware that "Paper" is a class name, therefore it calls the method of RQL validator that checks if this name is

stored as a class name.

The next step, the translation to SQL, is the most interesting. The variables appearing in the $FROM$ clause are evaluated against the database. RUL is implemented in the same fashion as the $SELECT-FROM-WHERE$ queries of RQL are. One reason for this design decision is the obvious similarity of the $INSERT$, $DELETE$ or $REPLACE$ and the $SELECT$ clause. Both clauses appear in the head of the statement, and, mostly, each variable appearing in any of these clauses must also appear in the $FROM$ clause, according to the semantics of both languages.

The other reason for this similarity is the way RQL performs the $SELECT-FROM-WHERE$ statements. Each variable appearing in the $SELECT$ clause is recursively evaluated and stored in a temporary database relation. This seems to be a slow-down factor for RQL, but there are good reasons for this engineering choice. First of all, RQL supports nested queries, so the storage of an evaluated query in a temporary relation is a good solution that reduces implementation complexity. What's more, storing the results in intermediate relations gives the capability of joins and other operations between the results of various (nested) queries. Another reason for this choice is that RQL queries containing scehma and data retrieval cannot be executed "on the fly", so that multiple SQL queries have to be executed against the database for a single variable. In that case, the temporary relation is used as a place to collect the results of its query, instead of keeping them in the main memory.

Apart from the advantages in the implementation of RQL, RUL also stored the result of a query statement in a temporary relation. This is due to the fact that the deterministic semantics of RUL require the query to be executed only once and only over the initial database state, which means that the query results should not be affected by the updates in process. As we will further detail later in this chapter, this design choice has been proven to be crucial for implementing the deterministic semantics of the language.

The evaluation module responsible for the evaluation of the variables appearing in the $FROM$ clause by taking into account all the filtering conditions appearing in the $WHERE$ clause. This evaluation is performed by the existing RQL code. For each

variable appearing in the head of an update clause, the values retrieved by the evaluation are stored in a temporary relation (according to a specific database schema). Then, the execution of the update operations takes place. For each update clause, the respecting code for an update statement is executed for the values of the temporary relation.

We could use an example to illustrate this.

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

The variable evaluation is presented in table 4.1.

Table 4.1: Variables X, Y and Z are evaluated, producing the following results:

X	Y	Z
$\&p1$	$\&a1$	$\&http://www.iswc05.org$
$\&p1$	$\&a2$	$\&http://www.iswc05.org$
$\&p2$	$\&a1$	$\&http://www.iswc05.org$
$\&p2$	$\&a3$	$\&http://www.iswc05.org$
$\&p3$	$\&a4$	$\&http://www.iswc05.org$

The corresponding temporary relation for DELETE can be found in table 4.2

Table 4.2: Temporary relation for DELETE

$operation-id$	$class-name$	$class-instance$
1	$Paper$	$\&p1$
1	$Paper$	$\&p2$
1	$Paper$	$\&p3$

and for REPLACE, in table 4.3

The resulting SQL queries that perform the operations depend on the database representation used to store RDFS graphs, but for the shake of the example we can suppose

Table 4.3: Temporary relation for REPLACE

$operation - id$	$class - name$	$class - instance - 1$	$class - instance - 2$
2	$Paper$	$\&p1$	$\&someAuthor$
2	$Paper$	$\&p2$	$\&someAuthor$
2	$Paper$	$\&p3$	$\&someAuthor$
2	$Paper$	$\&p4$	$\&someAuthor$

that the instances of each class are stored in a relation named tc<Class-Name>, as shown in table 4.4 and that the delete operation has to delete the instances from there.

Table 4.4: A possible class instance DB relation for class Paper

URI
$\&p1$
$\&p2$
$\&p3$
$\&p4$
$\&p5$

The SQL query that performs the operation might look like this:

```
DELETE FROM tcPaper
WHERE tcPaper.URI = tempDELETE.Class-Instance
```

In fact, all operations are stored in one relation, with the columns presented in table 4.5

Table 4.5: The temporary relation tempUpdate

$operationid$	$id1$	$id2$	$resource1a$	$resource2a$	$resource1b$	$resource2b$

Each of these columns is used to match the needs of each update operation, and most operation make use of only a few of these columns. The first column, $operationid$, is used to separate each update operation from each other. In the above example, $DELETE$ was referred with operation id 1, and $REPLACE$ with 2. If there are more than one update statements of the same kind (e.g. two $DELETE$s), they are assigned a different operation id. For example, the following statement

```
DELETE Paper(&p1), Paper(X) INSERT Paper(X)
FROM Paper{X}
```

can be viewed as three update operations:

```
DELETE Paper(&p1)

DELETE Paper(X) FROM Paper{X}

INSERT Paper(X) FROM Paper{X}
```

and its one is assigned a different operation id. This means that if two update operations share the same variable, the values retrieved for this variable are stored twice in the temporary relations used by RUL.

The other columns of the update relation have a slightly different meaning according to the kind of operation. $INSERT$ class instance operation uses $id1$ to store the class name and $resource1a$ to store the corresponding class instance. $INSERT$ property instance operation uses $id1$ to store the property name, $resource1a$ for the source of the property instance, and $resource2a$ for the target.

Once the variables get evaluated and stored in the temporary relation, the last step of the evaluation module is the creation of the SQL statements that implement the update. This is the most important part of RUL implementation and will be detailed in the sequel. In general, the update statements benefit from the existence of the temporary relation by bulk updating the corresponding relation of the underlying database representations. After the variable evaluation, the produced SQL statements are depended only on two factors: (a) the kind of the RUL update statement and (b) the RDF/s database representation used.

In the actual RUL implementation, as well as in RQL, it is common to store the result of intermediate schema traversal queries in temporary relations. It is very likely that during the execution of the query part of a complex RUL operation, an intermediate relation for storing schema queries might has been created by RQL, so it is reused for storing the extra ancestors.

The result of an RUL statement is a Boolean. If the operation was executed successfully and the preconditions described in chapter 2 hold, the result is "$true$", otherwise

it is "$false$". The result is returned in and RDF/XML form, like the result of an RQL statement. The difference between the result of a RUL statement and an RQL statement is that in RUL the result is just feedback to the user. In RQL it is the purpose of the language to answer the query, while the purpose of a RUL statement is to modify to the database according to the used request. What's more, an RQL/RUL statement is always executed in a transaction, which is handled in a different way in RUL and RQL. More precisely, in RQL the transaction is always aborted after the execution of the statement is completer and the results have been returned to the used. In RUL the transaction is aborted only if at least one of the update operations has returned false. The abortion of the transaction means that all the operations are also aborted and no effects or side effects have affected the database. If all the update operations return true, the transaction is commited and the database is modified.

4.2 The database representations of RDF/s

RDF schema and data in RDF Suite are stored in a (Object) Relational DBMS. The database representation for RDF/s affects the performance of both querying and updating process. It has been stated that the final SQL statement produced by the interpreter is depended on (a) the kind of update operation and (b) the underlying database representation. Three representations are used in RDF Suite ([29]). The first is called **schema-specific** representation, the second is named **schema-specific no-IsA** and the last is the **hybrid** representation.

4.2.1 Representation of the RDF schema

A part of the database representation is dedicated to store and preserve the schema information. In RUL we focus on the IsA relations between classes and between properties, as well as the domain/range types for the property members. Figure 4.3 presents the relations of the representations that RUL is aware of.

The "subclass" and "subproperty" relations are used to store the classes and the properties, respectively, as well as the IsA relationships between them. For each class or prop-

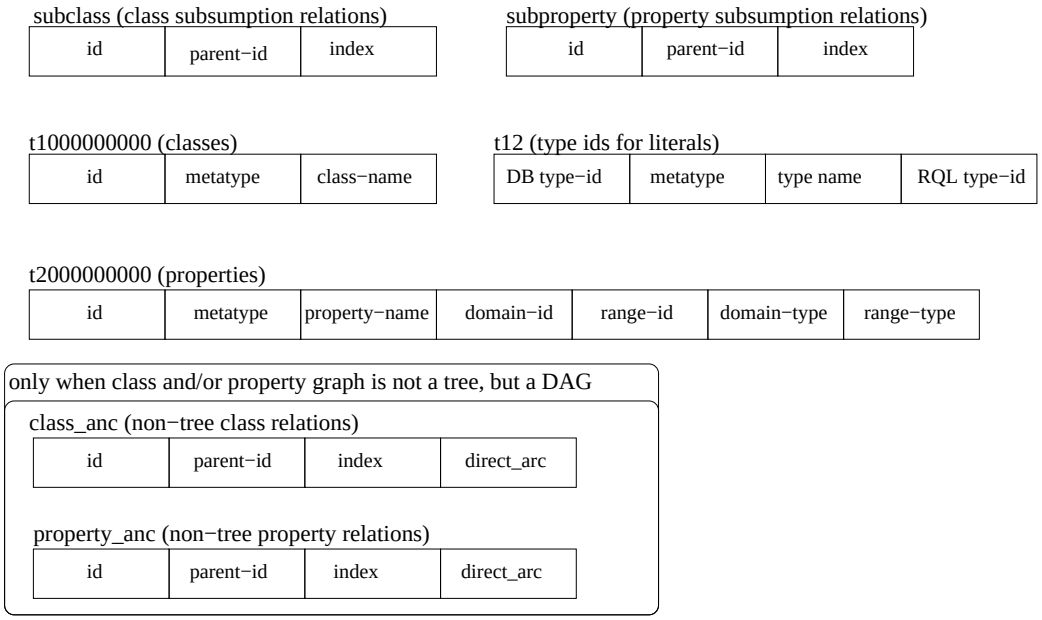

Figure 4.3: *The subsumption relations between class and properties are stored in subclass and subproperty relations respectively. The class and property ids and names are stored in t1000000000 and t2000000000 relations respectively. The relation t12 is used for storing the various type ids used by RSSDB and RQL to represent literal types. The class_anc and property_anc relations are used only if the class and/or property graph is not a tree, but a dag, and they are similar to subclass and subproperty relations, respectively.*

erty, there is a two-integer label. This label is used to describe the graph of the classes and properties ([27]). The first integer, stored in column ”id”, is a unique id produced by post-ordering the class graph. The second number, called ”index” is the smaller id of the descendants of the class, or equal to the id of the class if it is a leaf. There two numberings, one for classes and one for properties. The ”parent-id” field contains the id of the parent class.

The relations t1000000000 and t2000000000 are used to store the details of the classes and properties respectively. The class relation consists of a ”id” column, a ”metatype” column and the name of the class. The t2000000000 relation contains four more fields, two for the domain and, symmetrically, two for the range of the property, namely the ”domain-type”, ”domain-id”, ”range-type” and ”range-class”. The ”domain-type” (respectively ”range-type”) field is used to specify if the domain (range) of the property is a class or a literal object. If it is a class, then the ”domain-id” contains the id of the class that is the domain (similarly range) of the property, otherwise it is the literal type (e.g. integer, character string, floating point number, date) of the property.

We have already described how the ”id” and ”index” fields comprise a unique label for each class or property. This label is also used to describe the subsumption relations between the various classes and properties, in the case of a tree-structured hierarchy. If the class/property hierarchy is a Directed Acyclic Graph, the label describes only a cover tree if the graph, which is the initial graph without some selected edges (**??**). The edges removed are described in a separate relation, named ”class_anc” for classes and ”property_anc” for properties. The ”id” and ”index” fields of these relations are the id and index of a class that is a descendant of another class through a non-tree edge. The ”parent id” is the id of this non-tree ancestor. The last field is true if the subsumption relation between the class with id and the class with parent id is direct or false if it is implied by some other non-tree edge.

All three representations used in RDF Suite describe the RDF schema in the same way. The relations presented here are only a part of the database scheme actually used, but they are enough for implementing RUL, as they efficiently describe the class and

property IsA relations and contain all the information needed to check the constraints of any update operation.

4.2.2 Schema specific representation

In the schema specific representation, there is a separate relation for storing the instances of each class or property. Each of these relations contains one column, if it is used to store class instances, and two columns (source and target) if it is for property instances. For example (fig. 4.4), the instances of class $AcceptedPaper$ are stored in a different relation than the ones of $Rejectedpaper$. The instances of the class $Paper$ are stored in another distinct relation.

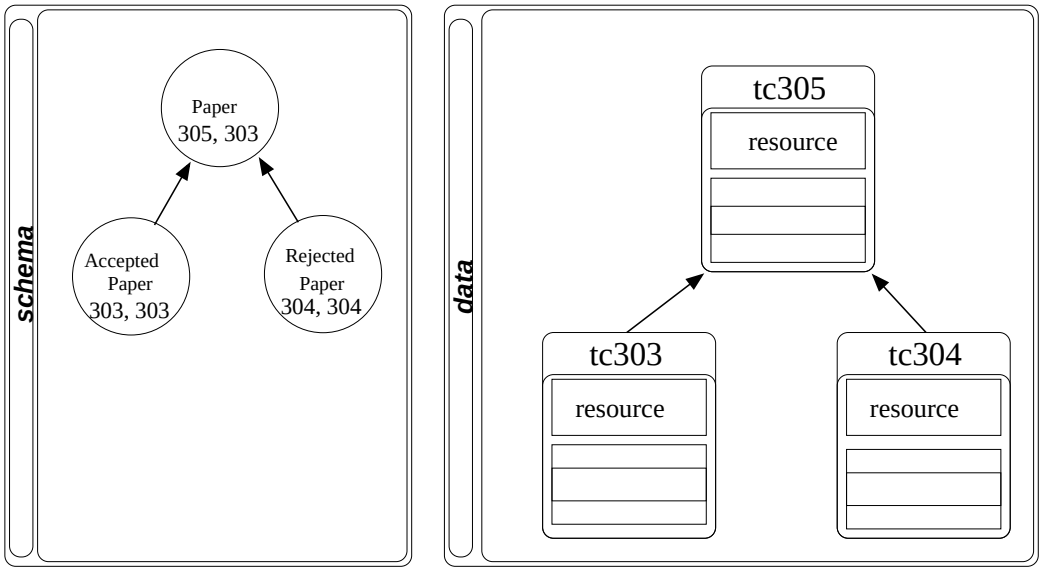

Figure 4.4: The class instances of AcceptedPaper are stored in tc304, of Rejected-Paper in tc303 and of Paper in tc305. The relations are connected with inheritance links, so that the tuples in tc303 or tc304 are also tuples of tc305. The couple of numbers under the name of each class is the label of the class, namely the id and the index.

The instance relations of classes or properties related through subsumption are also related using the inheritance feature between relations supported by any ORDBMS. In

our example, the class ”Paper” is a super-class of both ”Accepted Paper” and ”Rejected Paper”, therefore the relations for the two sub-classes inherit the instance relations of the ”Paper” class. If a class instance is physically added as a tuple in the ”Accepted Paper” instance relations, it is automatically a tuple of the ”Paper” instance relations as well.

The relations for storing the instances of a class are named ”tc<id>”, where ”<id>” is the id of the class of which the instances are stored. Similarly, the property relations are named ”tp<id>”, with ”id” being the id of the corresponding property.

4.2.3 Schema specific no-IsA representation

The only difference of the schema specific no-IsA representation is that the inheritance between relations is not used, and therefore this representation can be used with relational DBMSs. Applications using this representation can aquire the IsA relations between classes or properties by querying on the schema relations presented in the schema section. In order to avoid duplication of information, each resource is stored only in the instance relation of the class of which it is a direct instance. For example, ”&RULpaper” is a direct instance of ”Accepted Paper” and also a indirect instance of ”Paper”, but it is only stored in the former.

4.2.4 Hybrid representation

The hybrid representation uses one relation for all class instances and one relation for the property instances of the same type (fig 4.5).

These relations contain the ”id” of each class or property of which an instance is stored. The class instance relation contains also a column for storing the class instance URI (resource).

Properties are grouped by domain and range type. According to this type, the property relations contain two columns for storing the source and the target of each property instance. For example, properties with a class as domain and a floating point number as range are stored in one relation with a ”varchar” and a ”float” attribute.

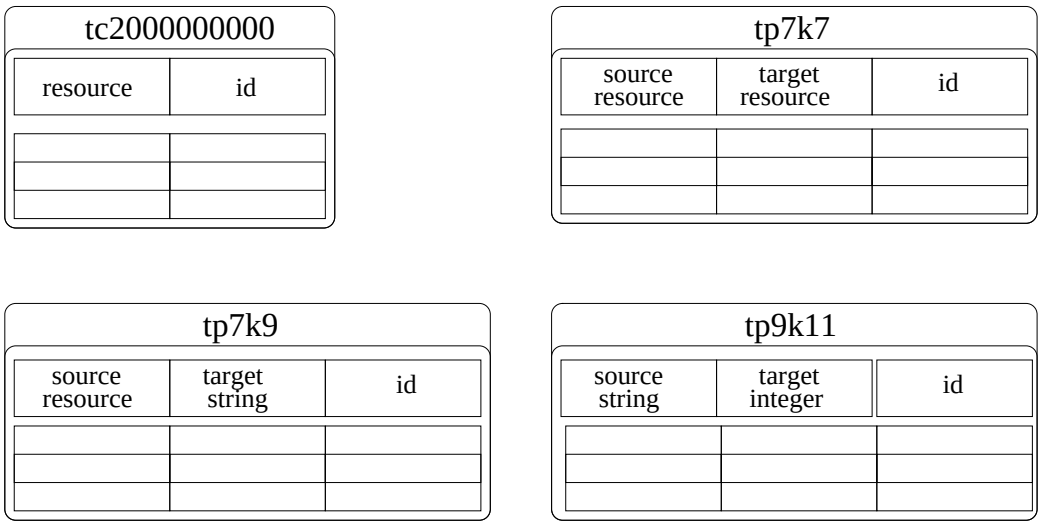

Figure 4.5: *The tc2000000000 is used to store the class instances. The resource attribute is the URI of the class instance, while the id is the id of the most specific (direct) class that this URI is instance of. The instances of the properties with a class as domain and range are stored in tp7k7. If the domain and/or range is a literal, they are stored in a different relation, depending on the type of the literal. For example, the instances of the properties with class domain and string range are stored in tp7k9. The instances of the properties with string domain and integer range are stored in tp9k11. There probably exist other relations for property instances as well, depending on the schema definition of the stored namespace.*

4.3 Translating from RUL to WL

RUL is used to update an RDF description, but it is implemented over a database, so the RUL statements have to be translated in a database update language. We will use WL to describe the database update operations used by RUL, from the point of view of a graph representation. In chapter 3 we described the formal semantics of RUL in a declarative way. We used these formal semantics to describe what are the preconditions, the effects and the side effects of its RUL operation. In this section we will describe the RUL operations with WL in a more precedural way. The WL translations are used to describe how these preconditions, effects and side effects are implemented over specific, real world database representations. Obviously, the formal semantic of chapter 3 are consistent with the semantics derived by the WL translations given here.

At the schema level, there is the class graph and the property graph. In the data level, there are nodes, representing resources, and property instances that are arcs between the nodes. There are also arcs connecting the nodes and the property arcs with the schema. The RUL operations have already been described with this model in mind, in chapter 2. In this chapter we will show the arc modification procedures that are used by RUL, as they are expressed in WL operating over any of the database representations of RDF Suite. Later on we will give more detailed translations of the RUL atomic operations in WL. The relations that are involved in RUL translations, including the temporary relation, of the retrieved results, have already been analyzed in section 4.2.

a. Schema-specific representation and Schema specific no-IsA representation

- Removing an instantiation link between a class C with id cid and a resource $\&r$:

$delete_{tc<cid>}(\&r)$

- Adding an instantiation link between a class C with id cid and a resource $\&r$:

$insert_{tc<cid>}(\&r)$

- Removing a property instance of P with id pid between a resource $\&r1$ and a resource $\&r2$:

$delete_{tp<pid>}(\&r1, \&r2)$

- Adding a property instance of P with id pid between a resource $\&r1$ and a resource $\&r2$:
$insert_{tp<pid>}(\&r1, \&r2)$

b. Hybrid representation - Removing an instantiation link between a class C with id cid and a resource $\&r$:

$delete_{tc2000000000}(\&r, cid)$
- Adding an instantiation link between a class C with id cid and a resource $\&r$:
$insert_{tc2000000000}(\&r, cid)$
- Removing a property instance of P with id pid between a resource $\&r1$ and a resource $\&r2$:
$delete_{tp7k7}(\&r1, \&r2, pid)$
- Adding a property instance of P with id pid between a resource $\&r1$ and a resource $\&r2$: $insert_{tp7k7}(\&r1, \&r2, pid)$

If the property P has a literal as domain and/or range, then instead of the relation $tp7k7$, we use the relation that is used to store this kind of properties. For example, for properties with a class as domain and an integer as a range, we use $tp7k11$, because 11 is the code meaning ”integer” in this database representation.

The above operations add or remove instantiation links between classes and class instances or properties and property instances. However the RUL semantics of these operations include various side-effects. The RUL semantics is implemented by using WL foreach and combining it with the corresponding RQL query translations.

For example, the INSERT class instance RUL operation is implemented by deleting any classification links between the ancestors of the specified class and the specified instance, and then inserting a new one between the instance and the class. This effects where also described in 3.1.1, where the classification links are deleted. In this formal description we suppose that if a resource is direct or indirect instance of a class, there is classification link between the class and the resource, while in the actual database rep-

resentations we store only the direct classification links. In the formal desctiption of the operation, the side effect of the operation, which is the insertion of the classification links between the resource and ancestors of class C, is not needed. In the schema-specific representation the operation is implemented like this:

INSERT C(&r) in WL (schema-specific):

$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$

$\{\ delete_{tc<superCid>}(\&r)\ \}$

$insert_{tc<cid>}(\&r)$

where cid is the id of class C, $\&r$ is the inserted instance and $subClassOf$ is a query returning the ids of class pairs sharing the ancestor-descendant relationship. The $subClassOf$ query for class instances:

$$subClassOf(id, superId) \leftarrow t1000000000(id, K1, K2, K3),\ subclass(superId, P, superIndex),\ superId > id, superIndex \leq id$$

In case the class graph is a DAG instead of a tree, the non-tree descendant-ancestor relationships are given by the follwing query:

$nonTree(id, superId) \leftarrow class\_anc(id, superId, index, direct\_flag)$

In the following translations, we omit the detailed explanation of the queries used in the foreach clauses.

INSERT C(&r) in WL (schema-specific no-IsA):

$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$

$\{\ delete_{tc<superCid>}(\&r)\ \}$

$insert_{tc<cid>}(\&r)$

$foreach\ subId : ans(subId) \leftarrow subClassOf(subId, id), id = cid$

$\{\ delete_{tc<cid>}(\&r)\ \}$

The last foreach is used to eliminate duplications, in the case of $\&r$ being an instance of some sub-class of C.

INSERT C(&r) in WL (Hybrid):

$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$

$\{\ delete_{tc2000000000}(\&r, superCid)\ \}$

$insert_{tc2000000000}(\&r, cid)$

$foreach\ subId : ans(subId) \leftarrow subClassOf(subId, id), id = cid$

$\{\ delete_{tc2000000000}(\&r, cid)\ \}$

The INSERT property is similar, with the exception that the class instances and/or literals ($\&r1$, and $\&r2$) forming the inserted property instance are checked for domain/range type consistency with the property P. If the domain/range check shows invalid values, the operation is aborted. Details about when and why a RUL operation might be aborted will be presented in section 4.4.

INSERT P(&r1, &r2) in WL (schema-specific no-IsA):

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ delete_{tp<superPid>}(\&r1, \&r2)\ \}$

$insert_{tp<pid>}(\&r1, \&r2)$

INSERT P(&r1, &r2) in WL (schema-specific no-IsA):

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ delete_{tp<superPid>}(\&r1, \&r2)\ \}$

$insert_{tp<pid>}(\&r1, \&r2)$

$foreach\ subId : ans(subId) \leftarrow subPropertyOf(subId, id), id = pid$

$\{\ delete_{tp<pid>}(\&r1, \&r2)\ \}$

INSERT P(&r1, &r2) in WL (Hybrid):

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ delete_{tp2000000000}(\&r1, \&r2, superPid)\ \}$

$insert_{tp2000000000}(\&r1, \&r2, pid)$

$foreach\ subId : ans(subId) \leftarrow subPropertyOf(subId, id), id = pid$

$\{\ delete_{tp2000000000}(\&r1, \&r2, pid)\ \}$

The DELETE class instance operation side effects are the insertion of the deleted value as instances of the immediate super-classes of C. Here $RUL-INSERT(cid, \&r)$ is a method executing a RUL INSERT operation, as explained previously. We omit here the check for the existence of $\&r$ as an instance of C, which can lead to the abortion of the operation.

DELETE C(&r) in WL (schema-specific):

$delete_{tc<cid>}(\&r)$

$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$

$\{\ RUL-INSERT(superCid, \&r)\ \}$

DELETE C(&r) in WL (schema-specific no IsA):

$delete_{tc<cid>}(\&r)$

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

$\{\ delete_{tc<subCid>}(\&r)\ \}$

$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$

$\{\ RUL-INSERT(superCid, \&r)\ \}$

The first foreach ensures that $\&r$ is removed from all subclasses of C. In this representation this has to be done by traversing through the schema, while in the schema-specific with IsA representation the deletion from the sub-class relations is ensured by the inheritance feature supported by the underlying ORDBMS.

DELETE C(&r) in WL (Hybrid):

$delete_{tc2000000000}(\&r, cid)$

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

$\{\ delete_{tc2000000000}(\&r, subCid)\ \}$

$$foreach\ superCid : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid$$
$$\{\ RUL - INSERT(superCid, \&r)\ \}$$

If the class graph is not a tree but a DAG, we also execute an RUL INSERT operation for the classes that are ancestors of the sub-classes of C that have $\&r$ as an instance. These ancestors are not nessecarily related through subsumption with C. This is achieved by executing in advance a statement that stores the required classes in a temporary relation T:

$$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$$
$$\{$$
$$foreach\ anc : ans(anc) \leftarrow subClassOf(id, anc),$$
$$tc < subCid > (\&r), id = subCid$$
$$\{\ insert_T(anc)\ \}$$
$$\}$$
$$foreach\ anc : ans(anc) \leftarrow subClassOf(anc, id), T(anc), id = cid$$
$$\{\ delete_T(anc)\ \}$$

The last $foreach$ eliminates from the temporary relation the ancestors of sub-classes of C that are also sub-classes of C, so that they won't be affected by the rest of the operation.

The above procedure retrieves in advance the classes that should keep $\&r$ as an instance, after the DELETE operation is completed, in the case of a DAG class hierarchy. The last foreach of the main DELETE translation is, now, modified in the following form:

$$foreach\ superCid : ans(superCid) \leftarrow T(superCid)$$
$$\{\ RUL - INSERT(superCid, \&r)\ \}$$

If exist property instances emanating from or ending to the deleted class instance, they are also affected. If the property's domain or range still contains the deleted class as an instance, the property is not modified. Otherwise, there must be a super-property

that has a domain/range with $\&r$ as an instance. If this is the case, a RUL-DELETE is executed over the sub-property through which the modified property is accessible. The result of this DELETE property instance operation is either the re-instantion of the original instance as an instance of a super-property with compatible domain/range, or the removal of the property instance.

The DELETE property instance operation is implemented in a very similar way. Again, we omit the, now familiar, domain/range checks as well as the check for the existence of the instance. We also omit the handling of the case when the property graph is not a tree. It is exactly the same as in the case of DELETE class instances with the obvious difference that the schema queries traverse through the property graph.

DELETE P(&r1, &r2) in WL (schema-specific no IsA):

$delete_{tp<pid>}(\&r1, \&r2)$

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ RUL - INSERT(superPid, \&r1, \&r2)\ \}$

DELETE P(&r1, &r2) in WL (schema-specific no IsA):

$delete_{tp<pid>}(\&r1, \&r2)$

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

$\{\ delete_{tp<subPid>}(\&r1, \&r2)\ \}$

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ RUL - INSERT(superPid, \&r1, \&r2)\ \}$

DELETE P(&r1, &r2) in WL (schema-specific no IsA):

$delete_{tp2000000000}(\&r1, \&r2, pid)$

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

$\{\ delete_{tp2000000000}(\&r1, \&r2, pid)\ \}$

$foreach\ superPid : ans(superPid) \leftarrow subPropertyOf(id, superPid), id = pid$

$\{\ RUL - INSERT(superPid, \&r1, \&r2)\ \}$

The REPLACE class instance operation is more complicated, as it can be viewed as a sequence of two operations (we call them erasure and addition). The main effect of the

first is to erase the instance $\&r$ while the main effect of the second is to insert the new instance $\&r'$. Another complication with REPLACE is that it has to replace not only the direct instances of C, but also the instances $\&r$ of the sub-classes of C, if any. The new values should be inserted exactly where the old, removed ones where, meaning that the new values should be instances of the sub-class of C that the old values where instances of. This affects the instances superclasses of C (or even some other classes in the case of a non-tree class graph), as the $\&r$ instances of these super-classes must be removed (side effect).

Property instances emanating from or ending to $\&r$, are modified so that they now emanate from or end at $\&r'$.

REPLACE C(&r $\leftarrow$ &r') (Schema-specific):

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

{

$foreach\ id : ans(superCid) \leftarrow tc < id > (id, \&r), id = subCid$

$\{\ insert_T(id)\ \}$

}

$delete_{tc<cid>}(\&r)$

$RUL - INSERT(cid, \&r')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

$\{\ RUL - INSERT(subId, \&r')\ \}$

$foreach\ P : ans(P) \leftarrow emanatingFrom(\&r, P)$

{

$foreach\ target : ans(target) \leftarrow tp < P > (source, target), source = \&r$

$\{\ replace_{tp<P>}((\&r, target), (\&r', target))\ \}$

}

$foreach\ P : ans(P) \leftarrow endingTo(\&r, P)$

{

$foreach\ source : ans(target) \leftarrow tp < P > (source, target), target = \&r$

$\{\ replace_{tp<P>}((source, \&r), (source, \&r'))\ \}$

}

REPLACE C(&r ← &r') (Schema-specific no IsA):

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

{

$foreach\ id : ans(superCid) \leftarrow tc < id > (id, \&r), id = subCid$

{

$insert_T(id)$

$delete_{tc<id>}(\&r)$

}

}

$delete_{tc<cid>}(\&r)$

$RUL - INSERT(cid, \&r')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

{ $RUL - INSERT(subId, \&r')$ }

$foreach\ P : ans(P) \leftarrow emanatingFrom(\&r, P)$

{

$foreach\ target : ans(target) \leftarrow tp < P > (source, target), source = \&r$

{ $replace_{tp<P>}((\&r, target), (\&r', target))$ }

}

$foreach\ P : ans(P) \leftarrow endingTo(\&r, P)$

{

$foreach\ source : ans(target) \leftarrow tp < P > (source, target), target = \&r$

{ $replace_{tp<P>}((source, \&r), (source, \&r'))$ }

}

REPLACE C(&r ← &r') (Hybrid):

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

{

$foreach\ id : ans(superCid) \leftarrow tc < id > (id, \&r), id = subCid$

{

$insert_T(id)$

$delete_{tc2000000000}(\&r, id)$

}

}

$delete_{tc2000000000}(\&r, cid)$

$RUL - INSERT(cid, \&r')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

{ $RUL - INSERT(subId, \&r')$ }

$foreach\ P : ans(P) \leftarrow emanatingFrom(\&r, P)$

{

$foreach\ target : ans(target) \leftarrow tp < P > (source, target), source = \&r$

{ $replace_{tp2000000000}((\&r, target, P), (\&r', target, P))$ }

}

$foreach\ P : ans(P) \leftarrow endingTo(\&r, P)$

{

$foreach\ source : ans(target) \leftarrow tp < P > (source, target), target = \&r$

{ $replace_{tp2000000000}((source, \&r, P), (source, \&r', P))$ }

}

An idea would be to implement RUL-REPLACE by using the WL replace operation and then applying the side effect by deleting the values of the instances of the ancestors from the corresponding database relations. Strangely enough, this approach is in every way equivalent to the one presented above. A careful observation would reveal that the combination of foreach, insert and delete statements used above, is actually the explanation of WL deterministic replace given in chapter 3.

The REPLACE property instance operation is, as well, symmetrical to the one for class instances. All values are checked for consistency with the domain and range of the property P, but this part is omitted here.

REPLACE P(&s ← &s', &t ← &t') (Schema-specific):

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

{

$foreach\ id : ans(superPid) \leftarrow tp < id > (id, \&s, \&t), id = subCid$

{ $insert_T(id)$ }

}

$delete_{tp<pid>}(\&s, \&t)$

$RUL - INSERT(pid, \&s', \&t')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

{ $RUL - INSERT(subId, \&s', \&t')$ }

REPLACE P(&s ← &s', &t ¡- &t') (Schema-specific no IsA):

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

{

$foreach\ id : ans(superPid) \leftarrow tp < d > (id, \&s, \&t), id = subPid$

{

$insert_T(id)$

$delete_{tp<id>}(\&s, \&t)$

}

}

$delete_{tp<cid>}(\&s, \&t)$

$RUL - INSERT(cid, \&s', \&t')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

{ $RUL - INSERT(subId, \&s', \&t')$ }

REPLACE P(&s ← &s', &t ← &t') (Hybrid):

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

{

$foreach\ id : ans(superPid) \leftarrow tp < d > (id, \&s, \&t), id = subPid$

{

$insert_T(id)$

$delete_{tc2000000000}(\&s, \&t, id)$

}

}

$delete_{tc2000000000}(\&s, \&t, cid)$

$RUL - INSERT(cid, \&s', \&t')$

$foreach\ subId : ans(subId) \leftarrow T(subId)$

{ $RUL - INSERT(subId, \&s', \&t')$ }

Finally, the REPLACE classification operation deletes the classification arc between the class C and resource $\&r$ and replaces it with a new one between C' and $\&r$. Under the light of the database representations used in RDF Suite, this means that the value representing the class instance should be moved from the relation storing the instances of C to the relation storing the instances of C'. The sub-classes of C will also lose this instance. The super-classes of C will lose this instance if it is accessible to them only through C: If there is a super-class of C that has $\&r$ as an instance through any other class irrelevant to C, then $\&r$ will continue to be instance of this super-class. Again, if $\&r$ is not an instance of C, the operation is aborted, but that part is omitted here.

REPLACE C ← C'(&r) (schema specific):

$delete_{tc<cid>}(\&r)$

$RUL - INSERT(cid', \&r)$

In the case of a non-tree class graph, the operation for the schema specific representation is like the one for schema-specific with no IsA.

REPLACE C ← C'(&r) (schema specific no IsA):

$delete_{tc<cid>}(\&r)$

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

{ $delete_{tc<subCid>}(\&r)$ }

$RUL - INSERT(cid', \&r)$

REPLACE C ← C'(&r) (Hybrid):

$delete_{tc2000000000}(\&r, cid)$

$foreach\ subCid : ans(subCid) \leftarrow subClassOf(subCid, id), id = cid$

$\{\ delete_{tc2000000000}(\&r, subCid)\ \}$

$RUL - INSERT(cid', \&r)$

The existence of property instances emanating from or ending to $\&r$ usually causes the operation to be aborted. An exception is when these property instances can also be instances of $\&r$ after the execution of the operation. This happens when C is irrelevant to the domain/range, or C' is a subclass of the domain/range of the property. The details of the property check are omitted here, because these property instances are never modified by this kind of REPLACE operation.

The REPLACE classification for properties is very similar:

REPLACE P ← P'(&s, &t) (schema specific): $delete_{tp<pid>}(\&s, \&t)$

$RUL - INSERT(pid', \&s, \&t)$

In the case of a non-tree class graph, the operation for schema specific representation is like the one for schema-specific with no IsA.

REPLACE P ← P'(&s, &t) (schema specific no IsA):

$delete_{tp<pid>}(\&s, \&t)$

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

$\{\ delete_{tp<subPid>}(\&s, \&t)\ \}$

$RUL - INSERT(pid', \&s, \&t)$

REPLACE P ← P'(&r) (Hybrid):

$delete_{tp2000000000}(\&r, \&s, cid)$

$foreach\ subPid : ans(subPid) \leftarrow subPropertyOf(subPid, id), id = pid$

$\{\ delete_{tp2000000000}(\&r, subPid)\ \}$

$RUL - INSERT(pid', \&r, \&s)$

4.4 Safety

The concept of safety is related to the presence of variables in the RUL statements and the ability to insert a new value, meaning a value that does not exist in the initial description base. We have already note that any variable appearing in the head of an update statement must also appear in the FROM clause. Therefore, a statement with variables but no FROM clause is invalid. The only way to insert new values in the description base is through constant values in the update statement head.

The following statement is invalid, because variable X does not apear in the FROM clause:

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

RUL interpreter produces a syntax error in the case of an unsafe statement. In some cases, though, it is possible to handle unsafe variables like wildcards. In the previous example, we know that X must be evaluated with instances of the domain of the property $keyword$ (which is the class $Paper$). Therefore, we could treat the statement like the following:

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

In the current implementation of RUL, this feature is not supported and every variable must apear in the FROM clause.

For example, the following statement inserts a new value $\&RULpaper$ in the class $Paper$:

```
INSERT Paper(&RULpaper)
```

A constant variable in the head is not necessarily a new value for the description base. For example, if the $\&RULpaper$ is already an instance of the class $Paper$, the above statement is still valid. Another case is when we use the constants to explicitly specify an

already existing value, e.g. when multi-classifying a class instance ($\&RULpaper$ might already be an instance of a class with no subsumption relation with the class $Paper$).

A constant appearing in a RUL statement is a class name, a property name, a class instance or a literal value. If it is a class or property name, the constant cannot be a new value, as RUL does not support schema updates. For example, the class $Paper$ should exist in the loaded RDF schema, otherwise the statement execution will return $false$. The only new values that can be inserted are class instances. Obviously, a new property instance is represented as a couple of class instances and/or literal values.

RUL implementation treats the insertion of new and existing values in the same way. It is always checked if the value is already an instance of the specified class or property. If it is not, it is inserted in the corresponding database relation. In case this is an already existing value of another class, then the side effects of the operation remove any duplicates from the database. For example:

```
INSERT AcceptedPaper(&RULpaper)
```

If $\&RULpaper$ is already an instance of $Paper$, which is a super-class of $AcceptedPaper$, RUL performs the following WL operations in the schema-specific representations:

$delete_{tc<Paper-id>}(\&RULpaper)$
$insert_{tc<AcceptedPaper-id>}(\&RULpaper)$

or the following WL operations in the hybrid representation:

$delete_{tc2000000000}(\&RULpaper, Paper - id)$
$insert_{tc2000000000}(\&RULpaper, AcceptedPaper - id)$

If $\&RULpaper$ is not an instance of any super-class of $AcceptedPaper$ (or of any class, for that mater), the delete operations do not modify the database, but they are executed nevertheless. This is not a performance drawback, because the WL delete operations do not cost more than the necessary queries used to determine if there are any ancestors of $AcceptedPaper$ with $\&RULpaper$ as an instance.

In the case of RUL DELETE operations, the constant values should not be new, but the user is allowed to use new values here as well. A user may ask to DELETE a non-existing class or property instance, if, for instance, it is unknown if this instance exists in the database.

Again, RUL treats new and existing instances in the same manner. It checks if this is an instance of the specified class, and if it is not, the operation does not go further. If this instance exists under a class or property not related to the class or property specified in the RUL update statement, it does not affect the operation.

The RUL DELETE operation effect is to remove the tuple representing the specified instance from the corresponding database relation. The side effect following, inserts the instance under the immediate ancestors of the specified class or property. Therefore, there is the danger of inserting an instance that did not originally existed in the initial description.

For example, if $\&RULPaper$ does not exist in the description base at all, then the following operation:

```
DELETE AcceptedPaper(&RULPaper)
```

produces the following WL operations in the schema-specific representation:

$delete_{tc<AcceptedPaper-id>}(\&RULPaper)$
$insert_{tc<Paper>}(\&RULPaper)$

the first WL operation has no effect, but the second inserts a new value as an instance of Paper.

RUL is safeguarded from this undesired effect by aborting the DELETE operation if the deleted value is not an instance of the specified class (in our example, if $\&RULPaper$ is not an instance of $AcceptedPaper$), so that the side effect operation is never executed.

It should be stressed out that the danger of inserting an undesired value as a side effect of the DELETE operation entails even when there are only variables in the class instance fields of the operation. For example:

```
DELETE AcceptedPaper(X) FROM RejectedPaper(X)
```

Obviously, there will be some instances of $RejectedPaper$ that are not instances of $AcceptedPaper$ (actually, we expect this condition to hold for all of them), and the operation will be aborted. Note that if there are some instances common to both classes, they will not be removed. A RUL statement is either executed in the whole, or not at all.

In the case of RUL REPLACE, some constant values may be new and some may not. Recall that the structure of the REPLACE operation for class instances is the following:

```
REPLACE ClassName(oldInstance <- newInstance)
```

REPLACE is translated as a removal of $oldInstance$, followed by the insertion of the $newInstance$. For example:

```
REPLACE Paper(&RULPaper <- &RULFinalEdition)
```

If $\&RULPaper$ is an instance of $AcceptedPaper$ (a sub-class of $Paper$), then this is the WL translation for the schema-specific representation:

$delete_{tc<AcceptedPaper-id>}(\&RULPaper)$
$insert_{tc<AcceptedPaper-id>}(\&RULFinalEdition)$

But if it is not an instance of $Paper$ at all, the operation is aborted.

The abortion of a REPLACE operation happens for exactly the same reasons as in the abortion of a DELETE operation, which is to safeguard the description base from new values that should not be inserted. The $newInstance$ value, on the other hand, can be a completely new value. On the above example, it is not necessary for RULFinalEdition to exist. In fact, this is the most expected case for the REPLACE operation: the replacement of an existing instance with a new one. Obviously, a REPLACE operation can be aborted even if there are no constants, if the variable evaluation results to the removal of non-existing values.

The REPLACE for property instances:

```
REPLACE PropertyName(oldSource<-newSource, oldTarget<-newTarget)
```

The $oldSource$ and $oldTarget$ values cannot be new, and the $(oldSource, newSource)$ couple should be an instance of $PropertyName$, otherwise the operation is aborted. The $newSource$ and $newTarget$ values can be new, as long as they are of the corresponding literal type or instances of the domain/range of the property (which is a REPLACE property precondition).

Therefore, the INSERT and REPLACE operations can be used to insert new values to the description base.

The REPLACE classification operation does not accept any new values, and like the other kinds of REPLACE operations, it is aborted if the modified class or property instance is not an instance of the specified class. Recall that:

```
REPLACE oldClass<-newClass(&classInstance)
```

If $classInstance$ is not an instance of $oldClass$ or, even worse, does not exist at all, the operation is aborted for the same safety reasons as the DELETE and the other kinds of REPLACE operations.

4.5 Determinism

It is a design choice for RUL to have deterministic semantics. By the notion of determinism we mean that the application of the same RUL statement over the same initial database instance will always results in the same output database instance.

We have already seen how atomic update statements are expressed in WL, and why WL is deterministic. We have to show that RUL is still deterministic in the case of variables included in the statement as well as when the statement contains any arbitrary sequence of RUL operations, some of them with variables.

RUL implementation can be described with WL, therefore any sequence of WL statements produced by RUL is a deterministic program, because WL is deterministic. It is enough to show that a RUL statement produces always the same WL program if applied over the same database instance.

Recall that the query part of RUL is evaluated before any update operations are fired. The retrieved results are put in a temporary relation to be used during the update part of the statement evaluation. The order of these results, for the same update operation, is not always the same for the same query. This is not a drawback of RQL neither does it mean that RQL is not deterministic. The results of an RQL query over the same description base will always be the same, but not necessarily their order.

Another observation we have to recall from the previous chapters is that WL entails the danger of non-determinism if an insert and a delete over the same relation are executed as part of the same foreach clause. This problem was resolved by specifying the semantics of foreach so that its operation is executed for all retrieved results, and the next operation is executed for the same results afterwards. The same idea is used in RUL implementation: If there are multiple insert, delete and/or modify WL operations in the translation of some RUL statement, they are never mixed up (especialy if it is possible to operate over the same relation).

All the WL translations provided in the corresponding chapter are consistent to that principle. The only part of these translations that needs clarification is the following kind of WL statement:

$$foreach\ X : Q(X)$$
$$\{\ RUL - INSERT(C, X)\ \}$$

We have seen that RUL-INSERT might contain a number of WL insert and delete operations. For that reason, the insert and delete operations contained as part of the traslation of RUL-INSERT are grouped and executed together, so that the above WL translation is equivalent to the following RUL statement:

```
INSERT C(X) FROM Q(X)
```

which is translated as follows:

INSERT C(X) FROM Q(X) in WL (schema-specific):

$$foreach\ superCid, r : ans(superCid) \leftarrow subClassOf(id, superCid), id = cid,$$
$$tempUpdate(oid, id, K1, r, K2, K3, K4),$$
$$oid = opId$$
$$\{\ delete_{tc<superCid>}(r)\ \}$$
$$foreach\ r : ans(r) \leftarrow tempUpdate(oid, id, K1, r, K2, K3, K4),$$
$$oid = opId, id = cid$$
$$\{\ insert_{tc<cid>}(r)\ \}$$

where $opId$ is the operation id and cid the class id.

All atomic update translations are modified in a similar manner for the case of instance variables in the RUL statements. The modification is that each WL insert, delete or replace operation is wrapped with a foreach clause of the following form:

$$foreach\ r1, ... : ans(r1, ...) \leftarrow tempUpdate(oid, id, K1, r1, ...), oid = opId, id = cid$$

We now have to deal with statements containing schema variables, like the following:

```
INSERT $C(X) FROM Q($C, X)
```

The tempUpdate temporary relation is again used here, so that schema and data variables can be deal by RUL in a uniform way. The tempUpdate relation contains two columns for storing schema variables. It is trivial to modify the translation so that schema variables are taken into account:

INSERT C(X) FROM Q(X) in WL (schema-specific):

$$foreach\ superCid, r, cid : ans(superCid) \leftarrow subClassOf(cid, superCid), oid = opId,$$
$$tempUpdate(oid, cid, K1, r, K2, K3, K4),$$
$$\{\ delete_{tc<superCid>}(r)\ \}$$
$$foreach\ r, cid : ans(r, cid) \leftarrow tempUpdate(oid, cid, K1, r, K2, K3, K4), oid = opId$$
$$\{\ insert_{tc<cid>}(r)\ \}$$

All update operation translations can be modified in the same fashion, by wrapping each WL insert, or delete operation with a foreach clause of the following form:

$$foreach\ id1, ..., r1, ... : ans(id1, ..., r1, ...) \leftarrow$$
$$tempUpdate(oid, id1, ..., r1, ...), oid = opId, id = cid$$

To conclude, the only danger for the deterministic semantics of RUL is that the produced database update translation might not always be the same for the same RUL statement over the same initial description. This problem is resolved by executing all database insert operations over the same relation together and separated by the database delete operations. To achieve this, we make use of the tempUpdate temporary relation, where the values of the evaluated variables are stored.

4.6 Translating to SQL

It is not difficult to retrieve SQL statements from the WL translations provided in section 4.3. The insert and delete statements of WL are equivalent to the insert and delete clauses of SQL. The SQL MODIFY clause, though, has different semantics than the replace of WL. This is another reason for our WL translations avoiding the WL replace statement.

SQL INSERT clause can be combined with a SELECT-FROM-WHERE SQL statement, e.g.

```
INSERT tc<cid> SELECT TU.resource1
FROM tempUpdate TU WHERE TU.oid = <opId>
```

while the SQL DELETE clause can be followed by FROM-WHERE clauses, e.g.:

```
DELETE FROM tc<cid> WHERE resource=tempUpdate.resource1 AND
        tempUpdate.oid = <opId> AND tempUpdate.id1 IN (...)
```

We can express all of our WL translations as long as SQL can express SPJ queries. In reality, we prefer to follow a hybrid approach, by implementing some of the iteration functionality with PL/SQL methods loaded into the database. PL/SQL is the procedural extension of SQL99.

Both schema specific representations store the various instances in one relation per class or property. Therefore, updates and queries have to be performed over database relations the names of which are produced at run-time. E.g., when the class or property is part of the iteration, the name of the relation affected by an update must be produced dynamically by the program. In the following example

$$foreach\ r, cid : ans(r, cid) \leftarrow tempUpdate(oid, cid, K1, r, K2, K3, K4), \\ oid = opId$$
$$\{\ insert_{tc<cid>}(r)\ \}$$

the $tc < cid >$ relation changes according to the values bound to cid. PL/SQL can use the query in the $foreach$ clause as an iteration condition and the corresponding SQL update statement as the body of the iteration. This functionality can also be achieved in the main memory of the RUL application, but the PL/SQL functions are faster. What's more, RUL can take advantage of future improvements in the implementation of PL/SQL by various DBMS.

If the database relations that are affected or queried are known in advance, we avoid the PL/SQL functions, as the foreach clauses can be expressed in a declarative style. A foreach condition containing an update statement, like this:

$$foreach\ x : Q(x)\ \{\ insert_T(x)\ \}$$

is expressed with the condition pushed in the SQL statement:

```
INSERT INTO T SFW(X)
```

where SFW(X) is a SELECT-FROM-WHERE query equivalent to $Q(X)$

Similarly, a foreach containing a WL delete:

$$foreach\ x : Q(x)\ \{\ delete_T(x)\ \}$$

is translated as

```
DELETE FROM T WHERE tuples-of-T IN SFW(X)
```

Finally, a foreach clause containing more than one update statements is handled by storing the results of the foreach condition in advance, and executing the corresponding SQL update statements over the stored results. The semantics of foreach is perfectly compatible when following this approach, in all cases. What's more, we prefer this technique for performance reasons, because we avoid to repeat costly join operation. For example:

$foreach\ x, y : Q(x, y)\ d\{\ insert_{T1}(x), delete_{T2}(y)\ \}$

is translated as:

```
INSERT INTO temporaryTable SFW(x, y)
INSERT INTO T1 SELECT x FROM temporaryTable
DELETE FROM T2 WHERE tuples-of-T2 IN (
       SELECT y FROM temporaryTable
       )
```

If the SELECT-FROM-WHERE query is the translation of an RQL schema query, the RQL methods that execute this query are called and the result is stored in one database relation used by RQL for that purpose. RUL makes use of this relation. If the SELECT-FROM-WHERE is not a schema-only query, the results are stored in the already existing tempUpdate relation, so that we avoid the creation of an unspecified number of temporary relations.

Each RUL statement is handled in one SQL transaction. In RQL, each RQL statement is also handled as one SQL transaction which is aborted after the completion of the query. In RUL we need the updates to actually affect the database, so if the statement is valid and the preconditions of the operations hold, we commit the transaction. If the preconditions do not hold, though, it is aborted. When the RUL statement is successfuly executed and the transaction is going to be committed, all temporary relations are dropped.

4.7 Optimizations

The RUL operations are implemented as a combination of main memory operations, queries against the database and database update operations. When optimizing RUL, we adopt the techniques used in RQL. In addition, we optimize the translation from WL to SQL whenever possible, we limit the number of temporary relations created and erased and reduce the number of SQL update statements produced during translation.

4.7.1 Minimizing the use of main memory operations

The main memory operations are used (a) to produce the various SQL statements (for querying or modifying) and (b) to implement the WL foreach clauses that are not expressible in SQL. For example, in order to erase a resource from being instance of a class, in schema-specific with no-IsA we have to traverse through the subclasses of that class. This is implementing by iterating in the set of subclasses in main memory. In each step of the iteration, an SQL DELETE statement is produced. No optimization techniques are used in this part of RUL operations. In general, we avoid main memory operation while translating from WL to SQL, whenever the WL programs are expressible in sequences of SQL statements.

The queries against the database take place (a) while evaluating the query part of the RUL statement, (b) whenever we want to check for the existence of a class or property instance and (c) whenever we evaluate various schema queries. The query part of the RUL statement is evaluated by RQL. In the other cases, if the query is part of a WL foreach clause with update statements, we express it inside the SQL update statement whenever possible. If it is not possible, the query is evaluated by the RQL code, and the iteration is performed by RUL in main memory. If the query is not part of a WL foreach clause, therefore not directly related to database update operations, it is also evaluated by RQL code. The query conditions pushed in RUL update statements are the only SQL statements produced directly by RUL, but they are expressed exactly as they would in RQL, taking benefit of all optimization techniques used there.

When we say that a resource is an instance of class, we mean that it could be an instance of some sub-class of this class, so it could be stored in the instance relation of this sub-class instead. In schema specific with no IsA, the check of the existence of an instantiation link between a class/property and an instance, requires a traversal through the sub-classes/sub-properties of this class/property, and a query on every relation used to store the instances of the sub-classes/sub-properties. For example, to check if $\&RULPaper$ is an instance of $Paper$, we have to seek for it in the relation where $Paper$ instances are stored, as well as in the relations containing the instances of $AcceptedPaper$ and $RejectedPaper$.

In schema specific with IsA, we can avoid this traversal by seeking only in the instances relations of the top class (in the example, $Paper$). The instance relations of the sub-classes/sub-properties are also included in this query through inheritance.

In the hybrid representation, we observe that there is a unique relation for storing the class instances, and a unique relation for the property instances of the same type of domain and range. Following the example of RQL, we use the id and index codes. The traversal through the class or property graph is replaced by a simple condition over the values of the ids of the sub-classes. A class or property $subC$ with $sub-id$ as id is a sub-class of another class or property C with cid and $cindex$ as id and index, if $sub-id < cid$ and $id \geq cindex$. We use this condition when joining the relation of class/property instances with the tempUpdate relation to check if a future instance of some class or property is already an instance of it. Other similar checks, like domain and range checks in property updates, are also handled this way, because they imply containment queries.

When translating to SQL, the hybrid representation allows the use of an SQL condition instead of an iteration over the retrieved class or properties. For example, the following WL program:

INSERT C(&r) in WL (Hybrid):
//side effects

$$foreach\ superCid, \&r : ans(superCid, \&r) \leftarrow subClassOf(cid, superCid),$$
$$tempUpdate(oid, cid, K1, \&r, K2, K3, K4), oid = opId$$

$\{\ delete_{tc2000000000}(\&r, superCid)\ \}$

//effects

$foreach\ \&r, cid : ans(\&r, cid) \leftarrow$

$tempUpdate(oid, cid, K1, \&r, K2, K3, K4), oid = opId$

$\{\ insert_{tc2000000000}(\&r, cid)\ \}$

//duplicate elimination

$foreach\ subId, \&r : ans(subId, \&r) \leftarrow subClassOf(subId, cid),$

$tempUpdate(oid, cid, K1, \&r, K2, K3, K4), oid = opId$

$\{\ delete_{tc2000000000}(\&r, cid)\ \}$

is translated in SQL as:

```
//side effect
DELETE FROM tc2000000000 WHERE (resource, id) IN
SELECT inst.resource, sc.superCid FROM subclass sc, tc2000000000 inst
WHERE (sc.id > cid AND sc.index >= cid) // subClassOf

//effects
INSERT INTO tc2000000000
SELECT res.resource1a, res.id FROM tempUpdate res

//duplicates elimination
DELETE FROM tc2000000000 WHERE (resource, id) IN
SELECT inst.resource, inst.id FROM subclass sc, tc2000000000 inst
WHERE sc.id = res.id AND inst.resource = res.resource
AND sc.id >= inst.id sc.index >= inst.id
```

The last WL foreach, that does the duplicates elimination, is used to counter some of the modifications applied by the ”effects” foreach statement. We can push this condition

to the effects statement, by using the SQL NOT IN construct. The effects statement is now expressed as:

```
//effects
INSERT INTO tc2000000000
SELECT res.resource1a, res.id FROM tempUpdate res
WHERE (res.resource1a, res.id) NOT IN
(//instanceOf
SELECT inst.resource, inst.id FROM subclass sc, tc2000000000 inst
WHERE sc.id = res.id AND inst.resource = res.resource
AND sc.id >= inst.id sc.index >= inst.id
)
```

We have seen in the WL translations chapter that this kind of expressions that counter the effects applied in a previous step of a program are very common. In RUL implementation all these cases are expressed by using ”NOT IN”. Obviously, this trick is applied in the schema specific with IsA representation as well, because it is possible to express the instanceOf query with one SQL condition.

Finally, this idea is also applied in the schema specific with no IsA, although the query that checks the existence of an instance of a class requires seeking in many dynamically acquired relations. In this case, there is a statement that removes in advance from the tempUpdate relation the values that are going to be countered, so they won’t be inserted and removed from the instance relations later.

The optimized SQL translation is still expressively equivalent to the initial WL one, but it performs better.

4.7.2 Optimizing according to the variables in RUL statement head

We have seen that RUL support eight kinds of update operations: the INSERT for class instances, the INSERT for property instances, the DELETE for class instances, the DELETE

for property instances, the REPLACE for class instances, the REPLACE for property instances, the REPLACE for class classification and the REPLACE for property classification. We group the operations of the same kind whenever they contain schema variables or if they contain instance and literal variables.

If an operation statement contains only constants, it is executed without making use of the temporary relation tempUpdate. Constant operations are not affected by queries and therefore there are no results to be stored. The WL translations of these operations have been presented in the WL translation chapter.

If an operation statement contains constant schema names and at least one instance variable, the query results are stored in the tempUpdate relation, but the schema fields of the relation contain the same value in all tuples. Recall the elimination of some tuples from this relation in case their schema fields contain classes or properties related through subsumption. If RUL is aware that the operation statement contains no schema variables, it skips the elimination procedure.

Finally, if the operation statement contains schema variables, all techniques presented here are applied. In this case, RUL does not distinguish between operation statements with constant or variable instances. The retrieved results are stored in the tempUpdate relation, even if the instance names are constant (and trerefore the same in all tuples). The temporary relation tempUpdate was proven useful in the case of updates with schema variables. The retrieved results stored there can be processed so that some values are eliminated before the update process is fired.

We observe that an RUL INSERT operation aims to specialize class or property instances by making them instances of more specific classes. RUL DELETE aims to generalize the instances by making them instances of more general classes or properties. In the case of an update with schema variables, the retrieved classes or properties may be related with subsumption relations. If this is the case, it might be possible that a resource is going to be inserted as an instance of two different classes related through subsumption.

For example (fig 4.6):

```
INSERT $C(X) FROM Author{X}, $C.hasCommittee{Y} WHERE Y=...
```

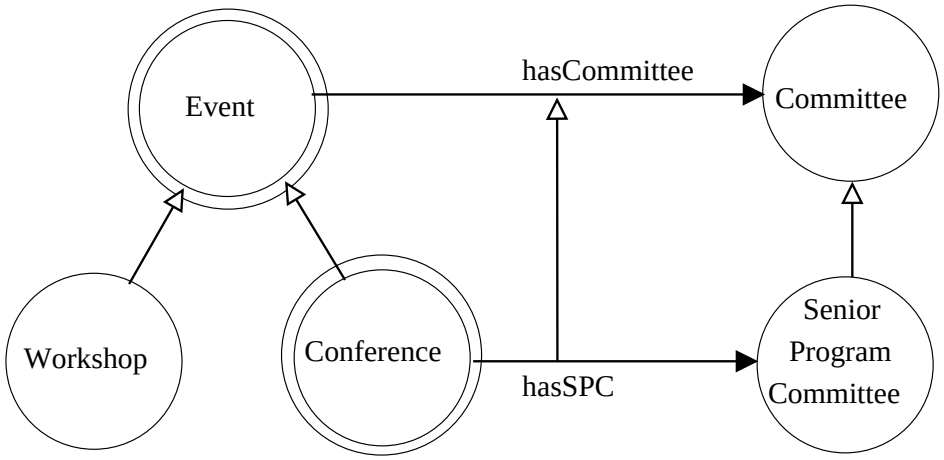

Figure 4.6: *The double cycles denote the classes evaluated as C in the folowing RUL statement:* $INSERT\ \$C(X)\ FROM\ Author\{X\},\ \$C.hasCommittee\{Y\}\ WHERE\ Y = ...$

The tempUpdate relation will look like table 4.6.

Table 4.6: tempUpdate temporary relation

oid	id1	id2	resource1a	resource2a	resource1b	resource2b
3	Event-id	null	MorningMeeting	null	null	null
3	Event-id	null	VisitingTheSights	null	null	null
3	Event-id	null	ReviewersParty	null	null	null
3	Event-id	null	Presentations	null	null	null
3	Conference-id	null	MorningMeeting	null	null	null
3	Conference-id	null	ReviewersParty	null	null	null
3	Conference-id	null	Presentations	null	null	null

We can see that some resources will be instances of $Conference$ as well as $Event$. According to the semantics of RUL INSERT, this is equivalent to the insertion of the resources only under $Conference$, because it is a sub-class of $Event$, as shown in figure 4.6. It is a good idea to remove from the common tuples the ones containing the $Event - id$.

In general, if we are going to execute an INSERT operation with schema variables, we eliminate some of the tuples with equal class instance values, so that each set of class instance values is going to be inserted only to the most specific of the related classes. The WL program that performs the elimination:

$$
\begin{aligned}
&foreach\ id, resource : ans(id) \leftarrow \\
&\qquad tempUpdate(oid, id, K1, resource, K2, K3, K4), oid = opId \\
&\{ \\
&\quad foreach\ oid, superCid, K1, r, K2, K3, K4 : ans(superCid) \leftarrow \\
&\qquad\qquad tempUpdate(oid, superCid, K1, r, K2, K3, K4), oid = opId, \\
&\qquad\qquad r = resource, subClassOf(id, superCid) \\
&\qquad \{\ delete_{tempUpdate}(oid, superCid, K1, r, K2, K3, K4)\ \}\ \}
\end{aligned}
$$

For properties, the program is the following

$$
\begin{aligned}
&foreach\ id, source, target : ans(id) \leftarrow \\
&\qquad tempUpdate(oid, id, K1, source, target, K3, K4), oid = opId \\
&\{ \\
&\quad foreach\ oid, superCid, K1, s, t, K3, K4 : ans(superPid) \leftarrow \\
&\qquad\qquad tempUpdate(oid, superPid, K1, r, t, K3, K4), oid = opId, \\
&\qquad\qquad s = source, t = target, subPropertyOf(id, superPid) \\
&\qquad \{\ delete_{tempUpdate}(oid, superPid, K1, s, t, K3, K4)\ \}\ \}
\end{aligned}
$$

In the example, the tempUpdate relation will have the form of table 4.7 after the completion of the elimination process.

In DELETE, we remove some tuples so that for each set of class instances, the classes or properties that will remain in the relation are the most general of the related classes or properties.

A symetrical example is this

Table 4.7: tempUpdate temporary relation after the elimination process for INSERT

oid	id1	id2	resource1a	resource2a	resource1b	resource2b
3	Event-id	null	VisitingTheSights	null	null	null
3	Conference-id	null	MorningMeeting	null	null	null
3	Conference-id	null	ReviewersParty	null	null	null
3	Conference-id	null	Presentations	null	null	null

```
DELETE $C(X) FROM Author{X}, $C.hasCommittee{Y} WHERE Y=...
```

where the tempUpdate relation is the same as in the previous example (4.7.2). Here, the elimination process will have the effects presented in table 4.8.

Table 4.8: tempUpdate temporary relation after the elimination process for DELETE

oid	id1	id2	resource1a	resource2a	resource1b	resource2b
3	Event-id	null	MorningMeeting	null	null	null
3	Event-id	null	VisitingTheSights	null	null	null
3	Event-id	null	ReviewersParty	null	null	null
3	Event-id	null	Presentations	null	null	null

The elimination WL program for DELETE class instances:

$$
\begin{aligned}
&foreach\ id, resource : ans(id) \leftarrow \\
&\qquad tempUpdate(oid, id, K1, resource, K2, K3, K4),\ oid = opId \\
&\{ \\
&\quad foreach\ oid, subCid, K1, r, K2, K3, K4 : ans(subCid) \leftarrow \\
&\qquad\quad tempUpdate(oid, subCid, K1, r, K2, K3, K4),\ oid = opId, \\
&\qquad\quad r = resource,\ subClassOf(subCid, id) \\
&\qquad \{\ delete_{tempUpdate}(oid, subCid, K1, r, K2, K3, K4)\ \}\ \}
\end{aligned}
$$

For properties, the program is the following

$$
\begin{aligned}
&foreach\ id, source, target : ans(id) \leftarrow \\
&\qquad tempUpdate(oid, id, K1, source, target, K3, K4),\ oid = opId \\
&\{ \\
&\quad foreach\ oid, subCid, K1, s, t, K3, K4 : ans(subPid) \leftarrow \\
&\qquad\quad tempUpdate(oid, subPid, K1, r, t, K3, K4),\ oid = opId, \\
&\qquad\quad s = source,\ t = target,\ subPropertyOf(subPid, id) \\
&\qquad \{\ delete_{tempUpdate}(oid, subPid, K1, s, t, K3, K4)\ \}\ \}
\end{aligned}
$$

The REPLACE operation does make use of this elimination trick as well. Recall that the first phase of REPLACE is the removal of the instance. The removal of an instance of some class is equally effective with the removal of the same instance from a super-class of it. Also recall that in the second internal phase of the execution of a REPLACE, the RUL INSERT operation is used, so the elimination is also applied there. What's more, the REPLACE statements with constant schema names in the head might produce translations equivalent to an INSERT with a schema variable. Therefore the elimination procedure is useful even for some RUL statements with no schema variables.

5

Conclusions and future work

An expressive declarative language for updating RDF graphs has been presented while ensuring that insertion/deletion/replacement of nodes and arcs does not violate 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. We compared the semantics of RUL operations with other RDFS update languages, as well as with the knowledge base update operations as well as database update languages. The architecture of RUL was then illustrated, by presenting the design principles, the integration with RQL and the translations to WL and SQL.

In future work, we plan to benchmark the performance of the implemented RUL op-

erations for various schemata, descriptions and database representations. We should also consider the definition of an update language for managing RDFS schema updates, based on RUL. Further improvements can me made to the existing RUL implemenation, like the implementation of a rollback and transaction control mechanism to both RUL and RQL.

Bibliography

[1] The ICS-FORTH RDF Suite. `http://athena.ics.forth.gr:9090/RDF`.

[2] A. G. Perez. A Survey on Ontology Tools. Deliverable 1.3 IST Project OntoWeb, May 2002.

[3] A. Magkanaraki and G. Karvounarakis and V. Christophides and D. Plexousakis and T. Anh. Ontology Storage and Querying. ICS-FORTH Technical Report No 308, April 2002.

[4] A. Seaborn. RDQL - A Query Language for RDF. http://www.w3.org/Submission/RDQL.

[5] S. Abiteboul and V. Vianu. A Transcation Language Complete for Database Update and Specification. In *Proc. 6th PODS*, pages 260–268, 1987.

[6] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *Proc. 7th PODS*, pages 240–250, 1988.

[7] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Proc. 1st ISWC*, 2002.

[8] D. Oberle and R. Volz and B. Motik and S. Staab. KAON Server Prototype. Deliverable 6, IST Project WonderWeb, January 2002.

[9] A. Das, W. Wu, and D. McGuinness. Industrial Strength Ontology Management. In *The Emerging Semantic Web*. IOS Press, 2002.

[10] G. Flouris. On Belief Change and Ontology Evolution. Doctoral Dissertation, Department of Computer Science, University of Crete, February 2006.

[11] A. Fuhrmann and S.O.Hansson. A survey on multiple contractions. In *Journal of Logic, Language, and Information*, pages 39–76, 1994.

[12] R. V. Guha. Rdfdb ql. http://www.guha.com/rdfdb/query.html.

[13] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *Proc. 3rd ISWC*, pages 502–517, 2004.

[14] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A declarative query language for RDF. In *Proc. 11th WWW*, 2002.

[15] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. RQL: A Functional Query Language for RDF. In *The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data*. Springer.

[16] H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In *In Peter Gardenfors, editor, Belief Revision, Cambridge University Press*, pages 183–203, 1992.

[17] K.G. Clark. SPARQL Protocol for RDF. http://monkeyfist.com/kendall/sparql-protocol/, 2004.

[18] M. Lawley. On the Power of Database Update Languages. In *Proc. 15th Australian Computer Science Conference*, January 1992.

[19] V. C. M. Magiridou, S. Sahtouris and M. Koubarakis. RUL: A Declarative Update Language for RDF. In *Fourth International Semantic Web Conference (ISWC'05), Galway, Ireland*, November 2005.

[20] M. Wallace. Compiling Integrity Checking Into Update Procedures. In *Proc. 12th IJCAI*, pages 903–908, August 1991.

[21] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proc. 2nd ISWC*, 2003.

[22] W. Nejdl, W. Siberski, B. Simon, and J. Tane. Towards a Modification Exchange Language for Distributed RDF Repositories. In *Proc. 1st ISWC*, pages 236–249, 2002.

[23] P. Hayes. RDF Semantics. http://www.w3.org/TR/rdf-mt/, February 2004.

[24] S. Sarkar and H.J.C. Ellis. Five Update Operations for RDF. Rensselaer at Hartford Technical Report, RH-DOES-TR 03-04, September 2003.

[25] S.Abiteboul and V.Vianu and R. Hull. *Foundation of databases*. 1995.

[26] A. Seaborne. An RDF NetAPI. In *Proc. 1st ISWC*, pages 399–403, 2002.

[27] V. Christophides, D. Plexousakis, M. Scholl, S. Tourtounis. On Labeling Schemes for the Semantic Web. In *Proc. 12th International World Wide Web Conference (WWW'03)*, May 2003.

[28] W. May and J. Alferes, F. Bry. Towards Generic Query, Update, and Event Languages for the Semantic Web. In *Proc. 2nd PPSWR*, 2004.

[29] V. C. Y. Theoharis and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Fourth International Semantic Web Conference (ISWC'05), Galway, Ireland*, November 2005.