University of Crete
Computer Science Department

Belief Change in Semantic Web Environments

George Konstantinidis
Master's Thesis

Heraklion, January 2008

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Αναθεώρηση Γνώσης σε Περιβάλλοντα Σημασιολογικού Ιστού

Εργασία που υποβλήθηκε απο τον
Γιώργο Κωνσταντινίδη
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

_____
Γιώργος Κωνσταντινίδης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

_____
Γρηγόρης Αντωνίου, Καθηγητής, Επόπτης

_____
Δημήτρης Πλεξουσάκης, Καθηγητής, Μέλος

_____
Γιάννης Τζίτζικας, Επίκουρος καθηγητής, Μέλος

Δεκτή:

_____
Πάνος Τραχανιάς, Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Απρίλιος 2008

Abstract

Towards the realization of the vision of the Semantic Web, one of the most significant tasks to be performed is the transformation of current human-oriented Web information into machine-processable Web information. In this direction, standards have been adopted in order to structure the data (XML) and to describe the semantics of the data (meta-data expressed in RDF).

RDF is a data model and along with the RDF Schema, which defines the vocabulary of this model, they form a mechanism which provides a formal, machine processable representation of knowledge. However the nature of world is dynamic and as the world changes, the knowledge itself, or our view of it, is subject to changes. Consequently, modeling a dynamic world means encapsulating a mechanism for updating knowledge.

The algorithms dealing with the incorporation of new knowledge in an ontology (ontology evolution) often share a rather standard process of dealing with changes. We acknowledge that this process consists of the specification of the language, the determination of the allowed update operations, the identification of the invalidities that could be caused by each such operation, the determination of the various alternatives to deal with each such invalidity, and, finally, some (manual or automatic) selection mechanism that allows singling out the "best" of these alternatives. Unfortunately, most ontology evolution algorithms implement these steps using a case-based, ad-hoc methodology, which is cumbersome and error-prone.

Knowledge updating is a problem which has been thoroughly examined in the field of Artificial Intelligence under the term belief change. One key idea in the belief change field is that an update operation should produce an updated belief which is as close as possible to the original belief. This approach is often described as minimal change approach. Trying to define "minimal" a lot of propositions have been made, among which is the definition of an ordering of the possible update results.

This work presents a framework for updating knowledge, where both the initial knowledge and the update are expressed in a special subset of First Order Logic. Updating is based on a well-formed set of Integrity Constraints on this logic and

a predefined ordering between the possible update results. Using this framework we apply this updating mechanism to a specific application: the RDF/S language. We define a model to express RDF language in terms of First Order Logic; an ordering between possible update results and build optimizations of the framework's updating techniques based on RDF's particular set of integrity constraints.

Through the application of our framework's techniques on RDF/S we express how the peculiarities of a specific language (which can be expressed with First Order Logic) could be used to optimize the proposed framework for the specific case. On the practical side, we speedup our general-algorithm by developing several, special per-operation, versions of it, which are also formally equivalent to it. We also discuss a number of issues raised during the implementation of the algorithm in a real-world environment.

Αναθεώρηση Γνώσης σε Περιβάλλοντα Σημασιολογικού Ιστού

Γιώργος Κωνσταντινίδης
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Αναθεώρηση Γνώσης σε Περιβάλλοντα Σημασιολογικού Ιστού

Ένα από τα σημαντικότερα έργα που πρέπει να επιτελεσθούν προς την πραγματοποίηση του οράματος του Σημασιολογικού Ιστού, είναι ο μετασχηματισμός των ανθρωπίνως-προσανατολισμένων ιστιακών πληροφοριών σε μηχανικά επεξεργάσιμες πληροφορίες. Σε αυτήν την κατεύθυνση, πρότυπα έχουν υιοθετηθεί προκειμένου να δομήσουν τα δεδομένα (XML) αλλά και να περιγράψουν τη σημασιολογία των δεδομένων (μεταδεδομένα που εκφράζονται σε RDF).

RDF ειναι ενα μοντέλο δεδομένων και μαζί με το RDF Schema, το οποίο καθορίζει το λεξιλόγιο αυτού του μοντέλου, αποτελούν ένα μηχανισμό που παρέχει μια τυπική, μηχανικά επεξεργάσιμη αναπαράσταση της γνώσης. Ωστόσο, η φύση του κόσμου είναι δυναμική και καθώς ο κόσμος αλλάζει, η ίδια η γνώση, ή η άποψή μας για αυτήν, υπόκειται σε αλλαγές. Κατά συνεπεια, η μοντελοποίηση ενός δυναμικού κόσμου πρέπει να συνοδεύεται από ένα μηχανισμό για την αναθεώρηση/επικαιροποίηση της γνώσης.

Οι αλγόριθμοι που εξετάζουν την ενσωμάτωση της νέας γνώσης σε μια οντολογία (εξέλιξη οντολογίας) συχνά μοιράζονται μια μάλλον τυποποιημένη διαδικασία για να αντιμετωπίσουν τις αλλαγές (ή ενημερώσεις). Αναγνωρίζουμε αυτή τη διαδικασία η οποία αποτελείται από την προδιαγραφή της γλώσσας, τον προσδιορισμό των τελεστών/πράξεων αναθεώρησης, των προσδιορισμό ασυνεπειών που θα μπορούσαν να προκληθούν από κάθε τέτοια λειτουργία, τον καθορισμό των διάφορων εναλλακτικών λύσεων για την αντιμετώπιση κάθε τέτοιας ασυνέπειας, και, τελικά, κάποιο (χειρωνακτικό ή αυτόματο) μηχανισμό επιλογής που θα επιτρέπει να ξεχωρίζει η "καλύτερη" αυτών των εναλλακτικών λύσεων. Δυστυχώς, οι περισσότεροι αλγόριθμοι εξέλιξης οντολογιών εφαρμόζουν αυτά τα βήματα χρησιμοποιώντας μια "ad-hoc", ειδική μεθοδολογία, η οποία είναι δύσχρηστη και επιρρεπής σε λάθη.

Η ενημέρωση (ή αναθεώρηση) γνώσης είναι ένα πρόβλημα που έχει εξεταστεί λεπτομερώς στον τομέα της Τεχνητής Νοημοσύνης υπό τον όρο αναθεώρηση/αλλαγή πεποιθήσεων. Μια βασική ιδέα στον τομέα αλλαγής πεποιθήσεων είναι ότι μια λειτουργία ενημερώσεων πρέπει να παράγει μια ενημερωμένη πεποίθηση που είναι όσο το δυνατόν πιό κοντά στην αρχική πεποίθηση. Αυτή η προσέγγιση περιγράφεται συχνά ως προσέγγιση ελάχιστης αλλαγής (minimal change approach). Έχουν γίνει πολλές προτάσεις προσπαθώντας να καθοριστεί

αυτό το “ελάχιστο”, μεταξύ των οποίων είναι και ο καθορισμός μιας διάταξης των πιθανών αποτελεσμάτων των ενημερώσεων.

Η παρούσα εργασία παρουσιάζει ένα πλαίσιο για αναθεώρηση γνώσεων, όπου και η αρχική γνώση και η ενημέρωση εκφράζονται σε ένα ειδικό υποσύνολο της πρώτης τάξεως λογικής (First Order Logic–FOL). Η ενημέρωση είναι βασισμένη σε ένα καλά διατυπωμένο σύνολο περιορισμών ακεραιότητας πάνω σε αυτή τη λογική, και σε μια προκαθορισμένη διάταξη μεταξύ των πιθανών αποτελεσμάτων των ενημερώσεων. Χρησιμοποιώντας αυτό το πλαίσιο εφαρμόζουμε τον μηχανισμό ενημερώσεων σε μια συγκεκριμένη εφαρμογή: τη γλώσσα RDF/S. Ορίζουμε ένα μοντέλο για να εκφράσουμε τη γλώσσα RDF με όρους πρώτης τάξεως λογικής, μια διάταξη μεταξύ των πιθανών αποτελεσμάτων αναπροσαρμογών και χτίζουμε βελτιστοποιήσεις των τεχνικών ενημέρωσης του πλαισίου μας, βασισμένοι στο ιδιαίτερο σύνολο περιορισμών ακεραιότητας της RDF/S.

Με με την εφαρμογή των τεχνικών του πλαισίου μας σε RDF/S επιδεικνύουμε τον τρόπο με τον οποίο οι ιδιαιτερότητες μιας συγκεκριμένης γλώσσας (που μπορεί να εκφραστεί με FOL) θα μπορούσαν να χρησιμοποιηθούν για να βελτιστοποιήσουν το προτεινόμενο πλαίσιο για ειδικές περιπτώσεις. Σε πρακτικό επίπεδο, επιταχύνουμε τον γενικό αλγόριθμό μας με την ανάπτυξη αρκετών πρόσθετων, συγκεκριμένων ανά-λειτουργία ενημέρωσης, εκδόσεων αυτού, οι οποίες είναι επίσης τυπικά ισοδύναμες με αυτόν. Ακόμα, συζητάμε διάφορα ζητήματα που τίθενται κατά τη διάρκεια υλοποίησης του αλγορίθμου σε ένα περιβάλλον πραγματικού χρόνου.

Επόπτης Καθηγητής: Γρηγόρης Αντωνίου
Καθηγητής Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

To my parents, Theo and Katia,

for their endless support

Acknowledgements

There are too many people, directly or indirectly, connected to this long and difficult effort; I would like to thank Professor Grigoris Antoniou for supervising, guiding and having trust in this work. I would like to deeply thank Associate Professor Vassilis Christophides for our turbulent but most productive discussions. I would also like to thank Assistant Professor Yiannis Tzitzikas and Professor Dimitris Plexousakis for their comments on this dissertation.

A deep acknowledgement and many thanks to Giorgos Flouris; many initial theorems and proofs of this thesis are his intellectual property. Apart from starting this work, he was always watching, helping and producing with me.

An effort to list friends and beloved persons who supported me during my work, would be fruitless as they are too many; thanks to all my friends, from early university years to the M.Sc. program. Special thanks to my girlfriend Katerina.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Everything that exists, it is only change.

Heraclitus

1.1 Formalizing Ontology Evolution

An indispensable part of any knowledge-based application is the ability to change its corpus of knowledge. Stored knowledge may need to change due to various reasons, including changes in the modeled world, new information on the domain, newly-gained access to information previously classified, unknown, or otherwise unavailable, changes in the usage pattern, and other eventualities [20]. Therefore as change management is a key component of any knowledge-intensive application, the same is true for the Semantic Web, where knowledge is usually expressed in terms of ontologies and refined through various methodologies using *ontology evolution* techniques. In this work, we consider the case of ontologies and introduce a formal framework to handle the evolution (change) of an ontology given a change operation.

1.1.1 Success and Validity

The problems that arise during knowledge evolution are related with how to respond to the requested update in a consistent manner. This basically amounts to two requirements. The first is making sure that the update is *successful*, i.e., that the update request is actually implemented in the ontology. This requirement corresponds to the Principle of Success in the standard *belief revision* literature [34]. The second requirement amounts to guaranteeing that the resulting ontology is

valid (Principle of Validity [34]). Validity in this context may take several different forms, depending on the application and the ontology language considered.

In order to explain these notions, we will provide an example: Consider modeling a domain where any property in the ontology representing that world has exactly one domain concept. In effect we could enforce this restriction in our ontology by naming valid the ontologies whose properties follow the above rule. Then, in the face of a change, one major aspect of the problem of revising our ontology is to find a different ontology that includes our change and still obeys the aforementioned rule, i.e. an ontology, all of whose properties, have exactly one domain concept. If we didn't have to obey to the above constraint, we wouldn't have a problem; we would apply directly any change, e.g. delete any class without caring if this was the domain of any property.

The above requirements are not always compatible. In many cases, the raw (naive) execution of a change request upon an ontology (in order to satisfy success) may cause problems leading the ontology to invalidity. On the other hand rejecting the operation would violate success, so satisfying both these requirements is not trivial. In such cases certain additional actions should be executed (more change operations) in order to restore validity (and retain success satisfaction). These additional actions are usually referred to as *side-effects*).

Returning to our example, consider that the desired change which we would like to apply to our ontology is to delete a class concept. Due to the existence of the integrity constraints on our knowledge, such as the "exactly one domain concept" rule, we cannot just apply a reckless deletion of a class; yet we want to end up in a result where the following two conditions hold:

(a) The class we have deleted is missing

(b) Every property has exactly one domain

This gives rise to the problem of identifying the side-effects that would restore validity; these shall be somehow dictated by the ontology modeling constraints and can, in their turn, trigger more side effects in order for the result to be valid. In fact, there may be more than one ways to restore validity. For example, while implementing the above change we should take care of any properties that turn up without domains; in these cases we conclude that either we have to delete them or attach them to an alternative domain. This is the only way for our update (the removal of a concept) to be successful, still adhering to the specified constraint.

1.1.2 Minimal Change Approach: Choosing how to Evolve

As we can see an update operation can have some alternative side-effects in order to be implemented. In such a case, an ontology evolution algorithm could either detect every alternative and choose one, or it could choose all along to enforce one inconsistency resolution, without estimating all possible; by rejecting, for example, other potential solutions as *provably* worse. In either case, it is counter intuitive that a selection mechanism should be in place to determine the "best" option. Note that as we will see later, most implemented ontology evolution attempts, unfortunately, choose one way to implement a change without any formally sustained excuse (i.e., they reject other possibilities inconsiderately). Also, note that as side-effects can trigger side-effects of their own there are in fact many different alternative *sequences* of side-effects that would result to a successful and valid update; moreover, in some cases, it is possible that there is no way to perform an operation that satisfies both the Principles of Validity and Success (in which case our technique will be able detect the problem and report it).

This element of choice among our various options on satisfying success and validity, calls indeed for formal confrontation. A key idea that this work employs towards addressing this problem uses the *Principle of Minimal Change* [14, 34]. This principle is used in Belief Revision and the current work is indeed inspired by the field of belief revision, exploiting techniques whose origins lie in the Artificial Intelligence area. The Principle of Minimal Change states that the appropriate side-effects of an updating action should be such that the resulting ontology (or corpus of knowledge in general) should be as "close" to the original as possible, i.e., this corpus of knowledge should undergo the least possible changes. By "closest" we imply a function that computes a "distance" between two valid ontologies.

In an effort to define "minimal" a lot of belief change propositions have been made, among which is the definition of an ordering of the possible valid update results [36]. This ordering implies a "distance"' or else a "cost" of the results. In the current work, the problem of determining the "distance" between two ontologies, or the "cost" of the resulting one is similarly introduced and dealt with. This study, provides customization of the evolution strategy using such an ordering, so as to select the side-effects that lead to the minimal (with respect to the ordering) changes to the original ontology; this way we establish a formal way of choosing the "cheapest" side-effect and therefore "cheapest" ontology among the set of alternatives that arise due to an update operation.

Although cost is defined by the proposal of a specific ordering, the framework designed is independent and parameterizable to any possible ordering that retains certain properties. More specifically, our ordering is defined among the constructs of an ontology, reflecting in an ordering among the resulting ontologies. The selec-

tion of the "cheapest" side-effects, the evolution strategies, the representation of the knowledge and of the change operations are all described using formal tools that allow us to handle changes in a more consistent,transparent and fully customizable way, as well as to handle any type of change, including yet undefined operations or bulk updates.

1.1.3 RDF/S Ontology Evolution

For presentation and implementation purposes, we chose RDF as a platform upon which to apply our framework. Note that, quite often, modeling different domains of discourse is reflected upon customizing RDF's loose semantics in tighter formations. For example, RDF allows membership loops, but, it has been acknowledged as a reasonable possibility to restrict oneself to a subset of RDF graphs which do not contain any such 'loops' of class membership[1]. In effect, semantic extensions may impose syntactic conditions which forbid such looped or other per-case undesired constructions. Known efforts towards these directions can be found in [51, 68, 74].

It seems therefore legitimate to use a concretization of RDF with respect to one's particular needs and characteristics. In this work, we make use of such RDF concretizations. This is done, firstly, for presentation purposes, as it is much easier to visualize the process with a particular language in mind, secondly, to reveal the way our proposed framework is established upon the notion of validity and therefore to point out how such ontology validity contexts can be formed, and, thirdly, to exhibit practical feasibility by implementing our approach for a particular RDF variant, namely the one presented in [51].

1.2 Identifying the Evolution Process

1.2.1 Acknowledging the Pattern

Although the intuition of Minimal Change lies silently underneath most of the current systems and tools supporting ontology evolution, the lack of a formal theoretical model creates a series of inconsistencies in defining and selecting the side-effects of an update operation or in defining the evolution strategies within these systems. Apart from the Principle of Minimal Change, manifestations of almost any idea presented in this chapter can be found in the implementations of standard ontology evolution systems, nevertheless this approach is the first to handle the problem in a formal manner.

[1] http://www.w3.org/TR/2004/REC-rdf-mt-20040210/#technote

These notions, ideas and principles that we are referring to are in fact the most critical part of an ontology evolution algorithm and they regard the determination of *what* can be changed and *how* each change should be implemented. One of the arguments of this study is that this determination can be split into the following 5 steps, which, although not explicitly stated, are shared by many ontology evolution frameworks:

1. *Model Selection.* The allowed changes are constrained by the expressive power of the ontology representation model. Thus, the selection of the model may have profound effects on what can be changed and constitutes an important parameter of the evolution algorithm.

2. *Supported Operations.* In step 2, the allowed change operations upon the ontology are specified.

3. *Validity Model.* Problems related to the validity of the resulting ontology may arise whenever a change operation is executed; such problems depend on the validity model assumed for ontologies.

4. *Invalidity Resolution.* This step determines, for each supported operation and possible invalidity problem, the different (alternative) actions that can be performed to restore the validity of the ontology.

5. *Action Selection.* During this step, a selection process is used to determine the most preferable among the various potential actions (that were identified in the previous step) for execution.

Unfortunately, most of the existing frameworks (e.g., [33, 6, 71, 85]) address ontology evolution issues related to the above 5 steps in an ad-hoc way. As we will see, this approach causes a number of problems (e.g., reduced flexibility, limited evolution primitives, non-faithful behavior etc), so evolution algorithms could benefit a lot from the formalization of the aforementioned change management process. In fact the above pattern is a pattern that could express any ontology evolution process, yet current tools often miss entirely addressing some steps of this pattern.

1.2.2 Ad-hoc Nature

An ad-hoc solution is an implementation of a specific change operation (or set of them), so as to end up in a resulting ontology, enforcing these operations upon an initial ontology. When considering the result of one ad-hoc change operation (or a set of them), this lies among every valid ontology that implements the operation

requested. Hence, we could say that an ad-hoc operation selects to implement one (or even a few) of the many valid ontologies that contain the initial update request. It is obvious at first that these resulting ontologies could be infinite in the general case.

Therefore,it is also obvious that, in order to implement an ad-hoc evolution system, one should decide in advance the change operations to be supported, so that the proper side-effects and selection mechanisms (or parameterizations) can be determined and inserted into the program code. This decision usually involves the determination of a set of fine-grained elementary updates, as well as a set of more coarse-grained composite updates.

The popular ad-hoc methodology of building change algorithms relies on, and therefore suffers from, the necessity of pre-determining the supported evolution operations. Unforeseen change operations can be supported only by adding more code to the system, but this is a non-ending process, as the number of operations is potentially infinite. Ergo, we are faced to the problem of confronting any different change operation (a very difficult problem for complex operations) and also any different initial ontology. Our only recourse is to try to encode our preference in such a way that comparing of new side-effects of unforseen update operations in any ontology, would be evident.

1.2.3 Need of A Framework

We argue that many of the above problems could be resolved by introducing an adequate evolution framework that would allow the description of an algorithm in more formal terms, as a modular sequence of selections in terms of the model used, supported operations, identification of plausible side-effects and selection mechanism. Such a framework would allow justified reasoning on the system's behavior.

Such a framework should use fine-grained modeling of ontologies, support evolution operations regarding all elements of the model and address all side-effects of every operation. To be transparent and fully customizable it should be parameterizable with respect to an evolution strategy which would be applied to every change operation. In addition, change operations supported should not be limited to the elementary but be any arbitrary combination of them.

1.3 This Thesis

1.3.1 Contribution of this Study

A general description of our major ideas mostly directed to the Ontology Evolution process we acknowledge was introduced in [61], where we criticize the informality of current semantic web tools' approaches to ontology evolution. We thereby presented a sketch of our formal ontology evolution framework found in this work.

The main advantage of our approach lies in its formal foundations, which allow us to avoid most of the problems associated with standard ontology evolution algorithms, which are usually designed using an ad-hoc approach.

In particular, our formal approach allows us to deal with arbitrary change operations (rather than a predetermined set). In addition, it considers all the invalidities related to each change and all the possible ways to deal with them. Moreover, it provides a parameterizable method to select the "best" option to deal with an invalidity, according to some metric. it allows us to deal with implicit knowledge, as well as to prove results related to our method. In addition, it is general, so it can be applied to a variety of ontology representation languages, and exhibits consistent behavior, in the sense that it is faithful to the choices and policies adopted.

The formal nature of the process allows us to avoid resorting to the tedious and error-prone case-based reasoning that is necessary in other frameworks for determining invalidities and solutions to them, and provides a uniform way to select the "best" option out of the list of available ones, using some total ordering. Our framework can be used for several different declarative ontological languages and semantics, despite that for implementation and visualization purposes, we instantiated it for the case of RDF(S), under the semantics described in [51].

Another view of this work reveals an attempt to model the nature of the preference we might have, in the "way" a change operation is carried out. The rationale is that the arguments that drive us to implement one solution instead of another, can be gathered and force an evolution strategy or politics. Such a strategy (which lies underneath the choices taken when designing any ad-hoc change operation) will help us slough off the necessity of coping with evolution cases one by one, permitting us to deploy a mechanism that counters evolution "automatically" with respect to this strategy.

We, thereby, enforce predictability and so transparency between different update requests, as a system that is consistent to its strategy will always respond to any update requested in a transparent manner.

1.3.2 Structure of the Thesis

We organize the rest of the dissertation as following. Chapter 2 provides surveys on the areas that this study steps in. The first two sections deal with Artificial Intelligence, as this area's basic ideas and techniques are followed by our ontology evolution approach. Section 2.1 presents an introduction to the Tarski's Logical Framework, a logic-based mechanism whose formalisms and definitions will be very useful to the continuing study. Following, section 2.2 constitutes an informal survey on the Belief Change literature. This survey will be kept at a rather informal level for presentation purposes and cite relevant references for additional studying.

The aim of this chapter 2 is to make the current dissertation self-contained. However, a survey of the Semantic Web area is not provided. Given that the content of this dissertation addresses mostly the Semantic Web community, several issues are taken for granted; readers not familiar with the semantic web concepts and techniques should first get acquainted with this area before reading this dissertation. Chapter 2 continues with sections 2.3 and 2.4, where we define ontology evolution and give a general overview of the principles, attempts and studies in the field; this allows us to place our work in its correct context. We will make a short review of the theoretical works on Ontology Evolution both related and non-related to Belief Change theories.

Next, in Chapter 3 we will we summarize some of the most important problems related to existing ontology systems and briefly explain how these problems can be overcome using a formal approach (see section 3.1). This chapter also studies the most dominant implementations of ontology evolution algorithms and formulates our motivation towards formalizing the evolution attempt.

In section 3.2 we present and describe some typical ontology evolution systems, having some update operations submitted to the most prevalent of them and analyzing their behavior. Thus we present some motivating examples for the current work. We show that some of these systems lack an underlying overall "cost" model (with respect to the afore-mentioned ordering for minimal change) and others make inconsistent underlying assumptions in update operations, i.e. they don't react with the same "evolution strategy" to every update. Basically, we show how these systems fit on a five-step evolution process, which we thereby recognize and propose. We criticize the ad-hoc methodology that other systems employ to tackle the problems related to certain steps. Moreover, we motivate the general framework that we employ in order to model the various steps of the evolution process recognized and to deal with the problems determined.

In chapter 4 we materialize the previously formulated evolution process, proposing a theoretical framework for updating knowledge, where both the initial knowledge and the update are expressed in a special subset of First Order Logic. Up-

dating is based on a well-formed set of Integrity Constraints on this logic and a predefined ordering between the possible update results. This mechanism is introduced in section 4.1, where we we present our modeling of ontologies and updates while we also emphasize the main properties of our framework and show how it can be applied to several representation languages.

In sections 4.2 and 4.3 we exhibit the merits of our framework via the development of a general-purpose algorithm for ontology evolution. Such an algorithm has general applicability, but we demonstrate how this algorithm can be employed for the RDF/S case in Chapter 5; using this framework we apply our updating mechanism to the RDF/S language .We define a model to express RDF/S language (section 5.1) and the validity constraints of our RDF Fragment (section 5.2) in terms of First Order Logic. In subsequent sections, we define an ordering between the possible update results and we also build optimizations of the framework's updating techniques based on RDF's particular set of integrity constraints. This is done by exploiting the peculiarities of RDF/S in order to optimize our general framework for the RDF/S case. We elaborate on change operations, stating that our initial framework is tunable and can support any operation (even those not presented here).

Moreover, in section 5.5, we specialize our approach for devising a number of special-purpose algorithms for coping with RDF/S changes (similar to the existing ad-hoc ontology evolution algorithms), which sacrifice generality for efficiency; although these instantiations are similar to ad-hoc approaches, predetermining all possible eventualities and building them into the system, they are provably exhibit-equivalent to the general-purpose algorithm, thus having the desired behavior, and guarantying that no special case has been left out of our consideration. Therefore, we illustrate how our algorithm can be sped up using special-purpose instantiations of it for most of the common operations.

The above algorithms have been implemented as part of the FORTH-ICS Semantic Web Knowledge Middleware (SWKM), which provides generic services for acquiring, refining, developing, accessing and distributing community knowledge. The SWKM is composed of four main services, namely the Comparison Service (which compares two RDF graphs, reporting their differences), the Versioning Service (which handles and stores different versions of RDF graphs), the Registry Service (which is used to manipulate metadata information related to the stored RDF graphs) and the Evolution Service (which deals with the evolution of RDF graphs). Chapter 6 describes the implementation of our algorithms for the Evolution Service; it provides architecture of our software, a design documentation for the implemented system and also discusses over a number of technical issues that arose during this process of implementation. To the best of our knowledge, our implementation is the first one that allows the handling of changes to

ontologies in a fully automatic way, as well as the first system that allows any type of update request to be processed.

Finally, this dissertation concludes in chapter 7 by summarizing our main results and proposing ideas for future work.

Chapter 2

Background

Progress is a nice word. But change is its motivator.

Robert Francis Kennedy

This chapter aims at providing the general background knowledge needed so as to make this dissertation as much self-contained as possible. We provide a brief overview of the necessary main concepts, fields and ideas expressed in the relevant fields as well as the appropriate terminology.

This chapter is spreading over two directions: Firstly we will be studying some very useful logic-based mechanisms, and we thereinafter, will make an informal survey on the belief change literature as this is our main means towards our effort to develop an automatic, rational and effective ontology evolution algorithm. This survey will be kept at a rather informal level for presentation purposes and cite relevant references for additional studying. Secondly, we will shortly present the Ontology Evolution field within the Semantic Web. We will make a short review of the theoretical works on Ontology Evolution both related and non-related to Belief Change theories.

2.1 Tarski's Logical Framework

In 1928 Tarski introduced a general logical framework. This framework currently engulfs most prevalent knowledge representation languages, as well as most of the logics considered by belief change approaches. Therefore, Tarski's framework is not only very important but it can (and will in this work) serve as a useful common ground for interaction between a belief change algorithm and an ontology representation language.

Tarski's logical framework defines a logic as a pair $< L, Cn >$, where L is the set of expressions or propositions of the underlying language and Cn is a unary *consequence operator*; it basically is a function which maps sets of propositions to sets of propositions. The intuitive meaning of Cn is that a set X implies exactly the propositions contained in $Cn(X)$. In other words, L represents what is expressible in the logic (i.e., the language), while Cn determines what is implied by each set of propositions. In this framework, a set of propositions of the underlying language L (i.e., a subset of L) defines a *belief*, which is the basic block of knowledge of the framework.

Some intuitive restrictions on the Cn operator were introduced in order to constrain the inference mechanism of the logic. These restrictions force the Cn operator to behave in a rational manner and are summarized in the following conditions:

$$\forall X \subseteq L, X \subseteq Cn(X) \quad \text{(Inclusion)}$$

$$\forall X \subseteq L, Cn(X) = Cn(Cn(X) \quad \text{(Iteration)}$$

$$\forall X, Y \subseteq L, if\ X \subseteq Y then\ Cn(X) \subseteq Cn(Y) \quad \text{(Monotonicity)}$$

Now, an inference relation ($\vdash$) between sets (beliefs) is defined as follows:

$$X \vdash Y \Leftrightarrow Y \subseteq Cn(X)$$

We say that a belief Y is *implied* by a belief X iff $Y \subseteq Cn(X)$, or, equivalently, iff $X \vdash Y$. The inference relation, can be easily shown to satisfy the following properties:

$$\forall X, Y \subseteq L, if\ X \subseteq Y, then\ Y \vdash X \quad \text{(Reflexivity)}$$

$$\forall X, Y, Z \subseteq L, if\ X \vdash Y and\ X \cup Y \vdash Z, then\ X \vdash Z \quad \text{(Transitivity)}$$

$$\forall X, Y, Z \subseteq L, if\ X \vdash Y then\ X \cup Z \vdash Y \quad \text{(Weakening)}$$

The opposite can also be shown as given a relation which satisfies the properties of reflexivity, transitivity and weakening, the function: $Cn(X) = \{y \in L | X \vdash \{y\}\}$ satisfies inclusion, iteration and monotony (i.e., it is a consequence operation). Therefore, there is a tight correspondence between inference relations and consequence operations. From this aspect, a logic can also be defined as a pair $< L, \vdash >$ [31], [65].

Two beliefs are called equivalent, denoted by $X \equiv Y$, iff $X \vdash Y$ and $Y \vdash X$, or, equivalently, iff $Cn(X) = Cn(Y)$. Tarski's consequence operator has the properties of an algebraic closure operator, or an abstract mathematical closure operator. Hence, calculating the consequences of a set corresponds to taking the

closure of this set. Note that later on, we will often use the term closure to refer to the process of calculating the consequences of a set. Based on the aforementioned properties, several interesting results have been gathered in the following lemma.

Lemma 1. Consider a logic $< L, Cn >$, under Tarski's model. The following hold:

$$Cn(Cn(X) \cup Cn(Y)) = Cn(Cn(X) \cup Y) = Cn(X \cup Y)$$

$$Cn(Cn(X) \cap Cn(Y)) = Cn(X) \cap Cn(Y)$$

$$If\ X \subseteq Cn(Y) then\ Cn(X \cup Y) = Cn(Y)$$

$$If\ X \subseteq Cn(Y) then\ Cn(X) \subseteq Cn(Y)$$

$$Cn(X \cap Y) \subseteq Cn(X) \cap Cn(Y) \subseteq Cn(Cn(X) \cap Cn(Y))$$

$$Cn(X \cup Y) \supseteq Cn(X) \cup Cn(Y)$$

2.2 Belief Change

2.2.1 Basic Concepts

A certain set of beliefs or knowledge, as the world itself is generally not static, evolves over time. Possible causes of the alteration of one's beliefs could be that new, previously unknown, classified, or otherwise unavailable information may have become known; revealment of a new fact through a new observation or experiment may have happened; or the actual world (e.g., domain of interest) may change, as, in principle, domains are not static themselves. *Change Adaptation* refers to the process the beliefs should be somehow subjected to, in all the above cases. Alternatively, the new information could be ignored.

Any structure, in effect Knowledge Base (i.e., KB) that holds beliefs or other information like facts, rules, data etc regarding a domain of interest, adheres to the discussed thoughts. The research area of *belief change* deals with the *adaptation* of a KB to *new information* [34]. This research has found applications in diverse areas like learning [54], software engineering [88], marketing research [88], ontology evolution [26, 27] and others. While, the field of *knowledge representation* deals with static knowledge and how this can be represented, *belief change* deals with the dynamic aspects of knowledge, i.e., how this knowledge evolves. The purpose of the rest of this brief review is to expose the diversity of the change problem, as well as the variety of approaches that have been considered. In times, several philosophical and practical issues related to belief change have been identified. We will not be concerned with a detailed description of such issues; we will

rather briefly scratch the surface of the field, by mentioning the main tradeoffs and problems that have been dealt within the field and point to certain references to further studying.

It is due to the amount of diverse work performed in the area, as well as the maturity of many approaches, that allowed researchers to hope that several belief change techniques, ideas or intuitions will prove useful in the field of ontology evolution [20].

2.2.2 Knowledge Representation

A prior step to that of approaching the practical subject of knowledge changing is to decide on the formalisms that will be employed for the representation of knowledge. A few rather general assumptions should be satisfied by any mechanism used for knowledge representation and these requirements have been realized in a family of logics which most approaches in the literature deal with (e.g., [1, 2, 3, 9, 10, 16, 36, 37, 65, 70]; this family includes most classical logics, e.g., Propositional Logic and First-Order Logic (FOL) [11, 18, 66]. However, the most studied logic in the belief change area is Propositional Logic: [13, 14, 24, 23, 25, 52, 87]. There are few exceptions that deal with non-classical logics, most notably [8], dealing with non-monotonic theories and [19], which focuses on the database paradigm.

Recently, there have been some belief change approaches dealing with Description Logics (DLs) or related languages; however the small number of attempts shows that the issue of ontology evolution has not been addressed by the belief change community. Some of these attempts try to apply existing belief change approaches to DLs (e.g., [50, 62, 67, 20, 77, 41]), while others apply belief change techniques to the evolution of concepts [30, 86].

Belief Bases versus Belief Sets

A fundamental question regarding the representation of knowledge should be resolved before deciding which underlying language (i.e, logic) to select for constructing a changing mechanism; this involves the decision of whether one should consider knowledge being represented in the form of a *belief base* or a *belief set*. Belief sets are (large) infinite structures, closed under their logical consequences, in effect, they contain explicitly all the knowledge deducible from the KB. On the other hand, belief bases are (small) finite sets of expressions of the underlying language which contain some of the information of a KB explicitly, the rest being deducible via the inference mechanism (or the standard reasoning) of the underlying logic.

As one can calculate the logical consequences of a belief base, or in other words

closing it with respect to its inference, we can reveal the full knowledge of a KB, whenever necessary. Therefore we can easily transform a belief base to a belief set (and vice-versa). Hence, the two views of a knowledge set are equivalent, and in fact as far as knowledge representation is concerned, they are the same; one *states explicitly* what the other *implies*.

However, things differentiate in belief change. When changes are performed upon a belief base, we temporarily have to ignore the logical consequences of the base; one can only apply direct changes to knowledge that is stored explicitly. On the other hand, implicit knowledge cannot be changed directly (though it could be indirectly affected by the changes in the explicit knowledge). In effect,in belief change there is a clear distinction between the two alternatives for storing knowledge.

Both the approaches have pros and cons in the extend that belief change is concerned. In the belief base approach, our options for the change are limited to affect the base itself, so the changes are generally more coarse-grained; this restricts our options for belief change, opposing a common principle which desires fine-grained changes. In the belief set approach, all the information is stored explicitly, the changes are fine-grained and there is no distinction between implicit and explicit knowledge. Nevertheless, this could mean that belief sets are infinite structures. Therefore, it is not possible to deal with them directly in a practical setting, so, in application level, only a small subset of a belief set is explicitly stored; in contrast to the belief base approach however, the implicit part of our knowledge is assumed to be of equal value to the explicitly stored one and treated similarly. An extensive discussion on a knowledge-level and motivation on belief base operations is presented in [44]. Studies have been done on comparison of the approaches[42, 45] and thoughts on their connection can be found in in [31, 43].

Foundational versus Coherence Theories

Knowledge Representation in literature has also been considered by several related philosophical studies. Mainly, two viewpoints have been discussed: foundational theories and coherence theories. Under the foundational viewpoint, each piece of our knowledge serves as a justification for other beliefs; our knowledge is like a pyramid, in which *every belief rests on stable and secure foundations whose identity and security does not derive from the upper stories or sections* [79].

On the other hand, according to the coherence theory, our beliefs do not require any justification. A belief is justified by how well it fits with the rest of the beliefs, in forming a coherent and mutually supporting set; thus, every piece of knowledge helps directly or indirectly to support any other. In this sense, knowledge is like a raft, *every plank of which helps directly or indirectly to keep all the others in*

place, and no plank of which would retain its status with no help from the others [79]. Note that the foundational viewpoint is closer to the belief base approach, while coherence theories match with the use of belief sets as a proper knowledge representation format. A more detailed discussion on these issues can be found in [35].

Change Representation

The format (and semantics) of the new information may vary. One common approach is to regard the "new information" as a single expression of the underlying logic, which is received and rendered to our knowledge in an autonomous, independent manner. In other occasions, belief change is encountered as a continuous process, so each change is related to previous (and future) changes, in effect a change can be a set of more elementary changes occurring in a sequence rather than a one-off, standalone process. *Iterated* belief change is a field which deals with the kind of semantics for this type of belief change [10, 15, 49, 63, 69]. Apart from the above we could regard the new information to be a whole new KB; in this case, we are dealing with belief *merging* [58, 59].

2.2.3 Change Operations

In the following we turn our focus away from stored knowledge towards issues regarding the nature of new information, and the way to handle it. In several cases, the new information is something to be added to our knowledge. In other occasions, the new information represents or opposes to something which is "wrong" in our beliefs and should be retracted. These two basic actions, however do not have always the same semantics.

Changing a static world: Revision

In [1] the authors identify three different types of belief change. The simplest one, though not an interesting one, is *expansion*. Expansion refers to the naive addition of information to our knowledge; reckless application without taking any special provisions to ensure the quality (i.e., consistency) of the KB after this addition. This operation is implemented trivially as a set-theoretical union of the new information (i.e., the change) and our beliefs (old KB). Therefore, it is not a worth-studying operation; if the new information contradicts the currently held beliefs then the result will be an inconsistent KB and hence useless.

Secondly, *revision* operation is defined, which is the most useful operation in terms of practical applicability. While similar to expansion, revision has the very

important difference that demands a certain quality on the results; the result should be a consistent set of beliefs. There are several ways to achieve this property on the resulting beliefs. When contradicting old knowledge one could choose to reject the change, although this would object to the well-known and used principle of Primacy of New Information [13], which is discussed later on this chapter. Hence most of the times, one chooses to apply the changes upon the beliefs, facing however, possible inconsistency problems. These problems could be overcome with potential additional application of consequential changes, in order to retain the quality of the result. In some cases for example, one may need to abandon a part of the currently held beliefs while adding the new information. The difficult and interesting part of revision is how to select the beliefs that should be abandoned.

The most fundamental operation and, consequently, the most important operation for theoretical purposes is *contraction* [34]. Contraction corresponds to the removal of information from a KB, but in a consistent way. For example, when a piece of information becomes unreliable and we would like to stop believing it, contraction may be necessary. A property of a contraction operator is that it should retract the unreliable information from both explicit and implicit knowledge; the latter could re-emerge as a consequence of the remaining beliefs, so simply removing the information from explicit knowledge may not be enough. Hence, a contraction operator may need to also remove beliefs which at first seem unrelated to the retracted piece of knowledge.

Throughout some approaches, several semantics was introduced for contraction. For example, some regard the new information (change) to be a whole set of expressions and to have *package* or *choice semantics* [32]; under package semantics, all expressions in the set must be retracted from the KB, while under choice semantics, it is sufficient that at least one of them is retracted.

The above operations regard a static world; in effect, as far as revision and contraction are concerned, the world itself does not change, but our perception of the world does. Thus, these operations are used when some new information about the real world has been disclosed, driving us to alter our conceptualization of the world in order to represent it more accurately.

Changing a dynamic world: Update Vs. Revision

In reality, the world is not indeed static and it might change as well. Therefore, we should realize another type of change and the KB should be adapted to the new reality. The semantics of this kind of change is quite different from those mentioned above, calling for introduction of a new pair of operations, namely *update* and *erasure* [53]. Update refers to addition of information and is similar to revision, while erasure refers to removal of information and is similar to contraction.

These operations are substantially different from their "static" correspondents as they both apply when the world dynamically changes.

In order to clarify the situation, we will consider the useful *switch example* [20]. In this example our world consists of three independent switches S1, S2 and S3. Suppose that our knowledge or belief of this world is that exactly one of S1, S2 or S3 is on. Our knowledge could be represented by the Propositional Logic KB:

$$(S1 \wedge \neg S2 \wedge \neg S3) \vee (\neg S1 \wedge S2 \wedge \neg S3) \vee (\neg S1 \wedge \neg S2 \wedge S3)$$

Now, suppose that somehow we learn or observe that S1 is on. The proper reaction to this observation is to assume that S2 and S3 are off, as this coincides with both the fact that exactly one switch is on (old KB) and with the fact that S1 is on (change). Therefore, the new KB should be:

$$(S1 \wedge \neg S2 \wedge \neg S3)$$

Notice that initially our knowledge was a disjunction of the several possible worlds that could exist. One switch was on for sure, and there was only one reality, however we did not know the true reality and we kept all the possibilities. Later on, among the three possible states of the world, we chose to keep the one that agrees with the new information. This is the case of a revision operator. Nevertheless, things are very different when coming to deal with a dynamically changing world. In this latter case the world itself alters rather than being disclosed.

Consider for example that we send a robot into the room with the intention to turn S1 on; after that we have no reason to assume that S1 was on before the robot's intrusion. We now know that the robot changed the real world by switching S1 on; so S1 should be on in the new KB. However, we cannot know if S1 was indeed the switch that was formerly on or wether we now have two switches on. Concluding we should assume that at the end of the robot's action, either S1 alone, or S1 and S2, or S1 and S3 are on. The new KB should be:

$$(S1 \wedge \neg S2 \wedge \neg S3) \vee (S1 \wedge S2 \wedge \neg S3) \vee (S1 \wedge \neg S2 \wedge S3)$$

In this case the expected result is different than in the case of revision. We don't choose among the possible worlds; we rather change each of the possible worlds independently so as to coincide with the new fact. This is the case of an update operator. Notice that in our example, in both operations the formal representation of the original KB and the information that initiated the change (S1) is the same. Nevertheless, the outcome of each operation is different. For an extensive discussion on the differences between the various change operations, see [53].

2.2.4 Change Process

Apart from the aforementioned technicalities, there are several philosophical questions related to belief change for which, in general, any attempt to give an answer depends on the application at hand. Several such issues, mostly related to the representation of the knowledge and change have been presented; we now come to discuss about how to decide on the realization of a change, i.e., how the change itself should be performed.

Change Methodology

In general, there are two methodology approaches: *postulation* and *explicit construction*[65]. The postulation approach dictates the formulation of a set of formal conditions (usually in the form of postulates) that a belief change algorithm should satisfy in a given context. On the other hand under the prism of explicit construction, one seeks explicit algorithms or constructions leading to algorithms, which are appropriate for the given context.

The two approaches are not contrarious but complementary [65]. While developing an explicit construction (or algorithm), one may identify certain wanted conditions for this construction, something that could be seen as a "request specification", that could drive to or form a postulation. On the other hand sometimes, in order for postulation to produce better results, it's initial findings are checked with an explicit construction that satisfies (or not) the postulates; this algorithm may help in determining the applicability and rationality of the proposed postulation(s) and lead to refinements and improvements.

Belief change area has took advantage of both methods and both have given interesting results. Explicit belief change algorithms are presented in [13, 14, 16, 87],and constructions that lead to algorithms or families of algorithms are given in [1, 2, 3, 9, 10, 31, 36, 37, 49, 52, 70] whereas potential postulations of the process are provided in [1, 16, 17, 44, 49, 53, 52]. The overlapping in this lists of references for the methods presented, exposes that such works have been developed in a parallel and complementary manner [65].

Change Implementation

Subsequently we will briefly discuss issues related to the determination and implementation of the actual change, which is the main focus of belief change. Any given approach to belief change has its properties determined by the stance it takes towards these concerns. Hence, each work is more suitable for a certain context and class of applications.

One issue that concerns the determination of a change early in the process of a specification of a change approach is related to the *acceptance* of the new information. It is a very common strategy to accept unconditionally the new information, so the resulting KB should contain the new information (in case of a revision, for example) or should not contain it(e.g, in case of a contraction). This implies a complete entrustment to the incoming data, and is often referred to as the *Principle of Primacy of New Information* [13]. Primacy of New Information coincides with the common intuition that newer information generally reflects a newer and more accurate view of the world. An alternative and less employed approach is to review the changes under a more critical scope and possibly reject them or a part of them. This approach is most useful in an agent communication context, or when a possible unreliable or untrustworthy source provides the information. This effort can be found in literature under the term *non-prioritized* belief change [46].

Principle of Irrelevance of Syntax [13] argues that the result of a change should not be affected by the syntactical representation of the KB nor of the change. While the principle applies in both the KB and the change it is possible and happens to apply it partially, only on the KB or the new information. Clearly this principle finds strong grounds in coherence theories [1, 13, 34]. This is generally true as in approaches that use belief sets the intuitively expected result of a change is determined by the semantics of the operands (KB and change), rather than their syntactical formulation. Under the foundational viewpoint, however, Principle of Irrelevance of Syntax generally fails [44].

According to the *Principle of Consistency Maintenance* [13] the result of a change should be a consistent KB. This is generally accepted as we have already stated that inconsistent (under classical logic) KBs do not carry any interesting information and, therefore, should be avoided. Whereas for classical logic this principle is valid, there are frameworks in which the underlying logic itself has an internal mechanism to deal with inconsistencies, such as nonmonotonic and paraconsistent KBs [5]. One thing that remains to be settled is the exact meaning of the "consistency" term; in terms of FOL we consider inconsistent any set that contains or implies both a proposition and its negation. More generally in the belief change literature, the meaning of a *consistent* KB is one which does not imply any tautologically false propositions.

Principles that are intuitively obvious, as well as essential for technical reasons, have often been implicitly used in the literature without explicit referral. Two such principles that stand out as commonly but silently used are the *Principle of Fairness* which guarantees determinism and reproducibility of the result of a change and the *Principle of Adequacy of Representation* which ensures that the resulting, altered KB will be represented using the same underlying formalisms as the initial one [13].

Undebatably, the most important, and by so far the most influential principle related to the implementation of a change is the *Principle of Minimal Change* [52]; other names in the literature are the *Principle of Persistence of Prior Knowledge* [13] or the *Principle of Conservation* [35]. This principle argues that the resulting KB should be as "close" as possible to the original KB; in other words, from all the possible results that satisfy the other principles, one should choose the one that retains most of the information from the old KB. Although the validation of this principle for change operators is taken for granted in most works, there is no consensus on how this principle should be formally expressed. The exact meaning of "closeness of KBs" or the measurement of "loss of information" needs to be defined.

Towards these directions there have been several attempts; proposals on metrics that count loss information in several ways, being used in different algorithms can be found in [2, 13, 14, 31, 36, 37, 52, 65, 87], different postulations that capture the same principle in a different manner are presented in [1, 44] and long debates on the pros and cons of each postulation are discussed in [34, 42]. The major reason for this diversity of is that when it comes to choosing what has to be given up in a certain change, extra-logical information needs to be considered; logical considerations alone cannot point this out [34]. The instantiation of the Principle of Minimal Change in a formal realization is in the core of each belief change algorithm or postulation attempt. In fact this principle is the very essence of belief change and the way this principle is being formulated, determines the properties of any given approach to a large extent.

2.2.5 The AGM theory

The most influential approach in the field of belief change is with no doubt the *AGM* theory [1], developed by Alchourron, Gärdenfors and Makinson (AGM is an acronym of their last names initials). AGM used the postulation method, although the general trend at the time was the explicit construction. In effect, in this work an attempt is made to introduce formalisms to the field of belief change by addressing the general problem of finding the properties that a "rational" belief change operator should adhere to. AGM dealt with the operators revision, contraction and expansion, providing one set of postulates, namely the *AGM postulates*, for the revision and contraction operations.

When formulating their theory the authors made several, quite general assumptions. They followed Tarski's viewpoint on the definition of a logic: a logic in the AGM theory is a pair <L,Cn>. Nevertheless, they made some additional assumptions: they assumed (a) that the underlying logic is closed under the standard operators of PL ($\neg$,$\wedge$, $\vee$,$\rightarrow$ etc) and (b) that the consequence operator includes classical

tautological implication, (c) is compact (i.e., closed and bounded) and (d) satisfies the rule of introduction of disjunctions in the premises (IDP rule).

The formal expression of IDP is:

$$Cn(X \cup (Cn(Y) \cap Cn(Z))) = Cn(X \cup Y) \cap Cn(X \cup Z), \forall X, Y, Z \subseteq L$$

AGM theory assumes a KB to be a set of propositions of the underlying logic (say $K \subseteq L$) which is closed under logical consequence (i.e., $K = Cn(K)$). A KB is also called a *theory* and we can see that it is indeed a belief set, i.e., AGM adopts the coherence model. The KB can be revised or contracted with any single expression $x \in L$ of the underlying logic. Therefore, the operations can be formalized as follows: revision can be regarded as a function mapping the pair (K,x) to a new KB $K\prime = K + x$ and contraction as a function mapping the pair (K,x) to the new KB $K\prime = K - x$.

The AGM Postulates

By the aforementioned assumptions and only, any binary operator is allowed to be or "revision" or a "contraction" operator, which, of course, should not be the case; AGM introduced several constraints in the form of rationality postulates on the result of such operators. These postulates are only enumerated here; for a discussion on the AGM theory and a more detailed presentation see [1].

For K a KB and x a change expression, the AGM postulates for revision are the following:

$$K + x \; is \; a \; theory \tag{K+1}$$

$$x \in K + x \tag{K+2}$$

$$If \; \neg x \notin Cn(K), then \; K + x = Cn(K \cup \{x\}) \tag{K+3}$$

$$If \; \neg x \notin Cn(K), then \; K + x = Cn(K \cup \{x\}) \tag{K+4}$$

$$If \; Cn(\{x\}) = Cn(\{y\}), then \; K + x = K + y \tag{K+5}$$

$$(K + x) \cap K = K - (\neg x) \tag{K+6}$$

$$K + (x \wedge y) \subseteq Cn((K + x) \cup y) \tag{K+7}$$

$$Cn((K + x) \cup \{y\}) \subseteq K + (x \wedge y), provided \; that \; \neg y \in K + x \tag{K+8}$$

For K a KB and x a change expression, the AGM postulates for contraction are the following:

$$K - x \; is \; a \; theory \tag{K-1}$$

$$K - x \subseteq K \tag{K-2}$$

$$If\ x \notin Cn(K), then\ K - x = K \tag{K-3}$$

$$If\ x \notin Cn(\emptyset), then\ x \notin Cn(K - x) \tag{K-4}$$

$$If\ Cn(\{x\}) = Cn(\{y\}), then\ K - x = K - y \tag{K-5}$$

$$K \subseteq Cn((K - x) \cup \{x\}) \tag{K-6}$$

$$(K - x) \cup (K - y) \subseteq K - (x \wedge y) \tag{K-7}$$

$$K - (x \wedge y) \subseteq K - x, provided\ that\ x \notin K - (x \wedge y) \tag{K-8}$$

2.3 Ontology Change in Literature

The development and maintenance of ontologies, which are generally large and complex structures, rises a number of interesting research issues such as the very important problem of *ontology change*. Ontology change refers to the problem of changing an ontology in response to a certain need.

2.3.1 Why do ontologies change?

In literature, the term ontology change came to be used in a broad sense, covering any aspect of change or ontology modification, including changes to the ontology in response to external events, changes dictated by the ontology engineer, changes forced by the need to translate the ontology in a different language or using different terminology and so on. Apart of such change implementations the term Ontology Change has often in fact been met to refer to the problem of deciding the modifications to perform upon an ontology in response to a certain need for change, as well as the problems that are indirectly related to the change procedure such as the maintenance of an ontology's different versions. Some times the term has broaden even further to cover the management of the changes effects in depending data, services, applications or agents.

In this context of changing an ontology can change for several reasons the simplest of which simply because the domain of interest has changed [81]. Independently of the changing world or domain, we may need to alter the perspective under which the domain is viewed [72]. In other cases we may wish to embody additional functionality, according to a change in a client's needs [39]; we may for example want to serve communication needs between heterogeneous sources of information or to fuse information from different ontologies.

Furthermore, we may also discover a design flaw in the original conceptualization of the world [76]. Other reasons of change: a sub-domain or different features of the domain become important [47]; new, previously unknown, classified or otherwise unavailable information, becomes available (could be discovery of some instance data, another ontology or a new observation) [47]. More examples of reasons initiating changes can be found in [55, 72].

2.3.2 Phases of Changing

The definition of change so far involves the decision on the modifications to perform; these may be made automatically, semi-automatically or manually. In order to tame the complexity of the problem, six phases of ontology evolution have been identified, occurring in a cyclic loop [80]. Initially, we have the *change capturing* phase, where the changes to be performed are identified. Three types of change capturing have been identified: structure-driven, usage-driven and data-driven [40].

Once the changes have been determined, they have to be properly represented in a suitable format during the *change representation* phase. The third phase is the *semantics of change* phase, in which the effects of the change(s) to the ontology itself are identified; during this phase, possible problems that might be caused in the ontology by these changes are also determined and resolved. The *change implementation* phase follows, where the changes are physically applied to the ontology, the ontology engineer is informed on the changes and the performed changes are logged.

These changes need to be propagated to dependent elements; this is the role of the *change propagation* phase. Finally, the *change validation* phase allows the ontology engineer to review the changes and possibly undo them, if desired. This phase may uncover further problems with the ontology, thus initiating new changes that need to be performed to improve the conceptualization; in this case, we need to start over by applying the change capturing phase of a new evolution process, closing the cyclic loop. An alternative, but similar, approach which identifies five phases, can be found in [76].

In order to illustrate the role of the six phases consider the example of Fig. 2.1. In this example the intention of the change is to remove concept A from our ontology. During the change capturing phase we identify the need to remove A and initiate the six-phase process of ontology evolution. During the change representation change we determine the kind of change(s) that must be performed in order to remove A. Once the required change is identified (i.e. “Remove_Concept” for the concept A in our example), we proceed to the semantics of change phase.

At this phase one should identify any problems that will be caused when the chosen action is actually implemented, thus guaranteeing the (correctness) validity

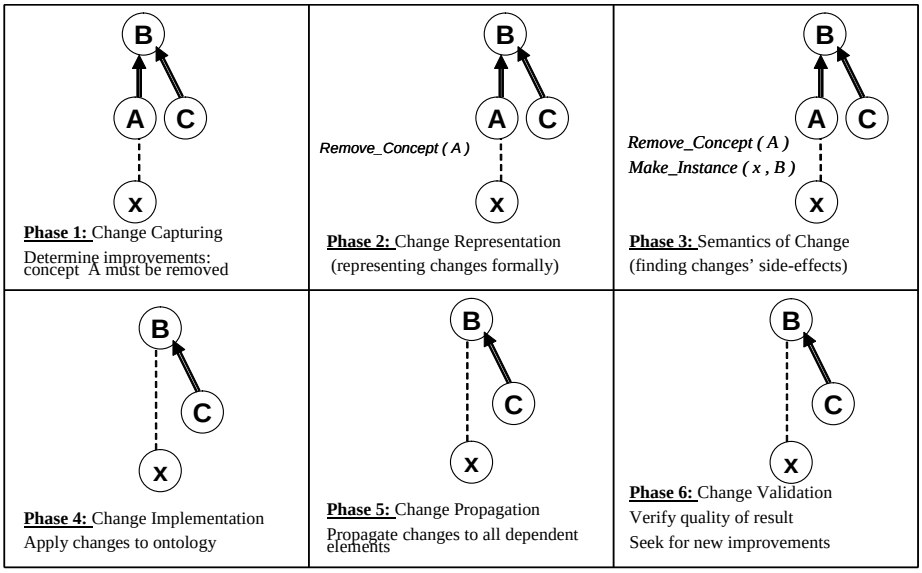

Figure 2.1: The 6 Ontology Evolution Phases

of the ontology at the end of the process. In our example, we need to determine what to do with A's instances; for example, we could delete them or re-classify them to one of A's superconcepts. Several proposals have been made in order to implement this step. This phase is the most crucial of ontology evolution; during that phase the direct or indirect changes caused by a given change request are determined.This is also the phase which our work mainly focus upon.

Some have stated that the final decision should be made directly by the ontology engineer, or by the system's designers at design time. Towards this direction, the authors of [80] suggest a change designment through the selection of certain pre-determined "evolution strategies", which indicate the appropriate action per-case. Other manual or semi-automatic approaches are also possible (see [40]).

Next, during the implementation phase, the changes identified in the two previous phases are actually implemented in the ontology. This could be done using an appropriate tool, like, for example, the KAON API [80] or the Versioning Service of SWKM [4]. Such a tool should have transactional properties, based on the ACID model, i.e., guaranteeing Atomicity, Consistency, Isolation and Durability of changes [40]. It should also present the changes to the ontology engineer for final verification and keep a log of the implemented changes [40].

The change propagation phase guarantees that all induced changes will be propagated to the interested parties or ontologies. This problem has been addressed

in [64], where two different methods to address the problem are presented, namely push-based and pull-based approaches.

The first approach (used by [64] and [80]), propagates the changes to the dependent ontologies as they happen, whereas in the second (pull-based approach), the propagation is initiated after the explicit request of each of the dependent ontologies. Alternatively, an ontology versioning algorithm could be used, allowing the interested parties to work with the original version of the ontology and update to the newer version at their own pace, if at all [56]. Given the decentralized and distributed nature of the Semantic Web, this alternative is considered more realistic for practical purposes [47].

Finally, the change validation phase should allow the ontology engineer to review and possibly undo the changes performed. At this point the engineer can start a new sequence of changes to further improve the conceptualization of the domain as represented by the ontology.

2.3.3 Fields of Change

Change as defined above intersects with several related research areas which are studied both separately and in combination with each other in the literature. In [20, 22], the authors identify nine such areas, namely ontology mapping, morphism, alignment, articulation, translation, evolution, versioning, integration and merging. Each one of these fields exposes a certain facet of the complex problem of ontology change from a different view or perspective, dealing with different application needs, change scenarios or "needs for change".

Translation, alignment, articulation, mapping, morphism, integration, merging

A change in the specification of a domain refers to a change in the way the conceptualization is formally recorded, i.e., a change in the representation language. This type of change is dealt with in the field of ontology translation, and so it is not regarded as an "immiscible" type of change. In addition, ontology development is becoming more and more a collaborative and parallelized process, whose parts (of the ontology) need to be combined to produce the final ontology [57]; this process would require changes in each subontology to reach a consistent final state.

However in ontologies a "final" state is rarely final, as ontology development is usually an ongoing process [47]. Ontology changes are also related to the distributed nature of the Semantic Web; ontologies are usually depending on other ontologies, (over which the ontology engineer may indeed have no control), so if the remote ontology is changed for any reason, the dependent ontology might also

need to be modified to reflect possible changes in terminology or representation [47].

For example, a certain agent, service or application may need to use an ontology whose terminology or representation is different from the one it can understand[1], so the change she needs to perform is some kind of translation between ontologies. Furthermore, we may also need to merge or integrate information from two or more ontologies as a for a certain application's necessities [75]. Some times there is a need to support and maintain different interoperable versions of the same ontology [47, 56, 48] a problem greatly interwoven with ontology change [55].

Ontology Versioning

Ontology versioning refers to the ability to handle an evolving ontology by creating and managing different variants (versions) of it [55]. After performing the actual changes on an ontology, ontology versioning algorithms come into play. Ontology versioning typically involves the storage of both the old and the new version of the ontology in a way, thought, that the different versions of the ontology are identifiable. Versioning deals with the relation between different versions (e.g., a tree of versions) as well as some compatibility information, like information regarding the compatibility of any pair of versions of the ontology. In effect, it deals with the process of managing different versions of an evolving ontology, maintaining interoperability between versions and providing transparent access to each version as required by the accessing element (data, service, application or other ontology).

Several problems are associated with this task. Some of these problems are not at all trivial as for example, the fact that any ontology versioning algorithm should be based on some type of identification mechanism (to differentiate between various versions of an ontology) is not as easy as it may seem; for example, it is not clear when two ontologies constitute different "versions". Should any change in the file that stores the ontology specification constitute the creation of a new version? When a concept specification changes, but the new specification is semantically equivalent to the original one, should this constitute a new version? More generally, how do syntactical changes compare with semantical ones? Should any change constitute a new version?

These and similar problems are discussed with in [47] and [56]. A desirable ontology versioning system should also have the ability to allow transparent access to different versions of the ontology, by automatically relating versions with data sources, applications and other dependent elements [55].

[1] http://www.starlab.vub.ac.be/research/projects/knowledgeweb/kweb-223.pdf

Ontology Evolution in Ontology Change

In the previous paragraphs, we noted that the term ontology evolution has in fact been used used as encapsulated in the ontology change field. The term ontology evolution has not always been used to refer to the same ideas or processes. The greatest confusion that prevails is between evolution and versioning.

For example a recent survey which counterpoises ontology schema evolution and versioning to database schema evolution and versioning [72], argues that evolution and versioning in ontologies are in fact indistinguishable. Moreover it is the distributed and decentralized nature of the Semantic Web which bounds existence of multiple versions of ontologies; interdependent ontologies are likely to be owned by different parties and as a result, some parties may be unprepared to change and others may even be opposed to it [47]. All these facts enforce the maintenance of different versions of ontologies, making ontology evolution (under this understanding) useless in practice.

In other works, ontology versioning is considered a stronger variant or a superset of ontology evolution [40]; ontology evolution is concerned with the ability to change the ontology without losing data or invalidating the ontology, whereas ontology versioning should additionally allow access to different variants (versions) of the ontology. However, while ontology evolution is concerned with the validity of the newest version, ontology versioning additionally deals with the validity, interoperability and management of all previous versions, including the current (newest) one. This viewpoint is also influenced by related, relational and object-oriented database schema evolution and versioning research.

Nevertheless, apart from versioning, in literature there other areas claiming also a strong interleaving with ontology evolution. Propagation of changes to dependent elements is acknowledged as a part of ontology evolution, because the extensive web of interrelationships that is usually formed around an ontology forces us to consider such issues [64]. Therefore there is a tight coupling of ontology evolution algorithms (and systems) with these issues as well.

Another example claiming the definition of evolution, is in [80], where ontology evolution is defined as the timely adaptation of an ontology to changed business requirements, to trends in ontology instances and patterns of usage of the ontology-based application, as well as the consistent management and propagation of these changes to dependent parts. Again, this definition covers both ontology evolution and versioning. Under this prism, however, ontology versioning and change propagation are considered internal parts of the process of ontology evolution.

The dominant confusion is partly justified by the fact that ontology evolution and ontology versioning are indeed closely related areas; despite that, in this thesis

we will consider the definitions of works where the two areas have discrete and well-defined boundaries [20, 60]. The most disambiguating description of ontology evolution is *the process of modifying an ontology in response to a certain change in the domain or its conceptualization*[20]. The diversity of the different forms that the problem of ontology change may take disallows any attempt for a unifying formal specification, because the purpose, the scope, the input, the output and the properties of the various fields are different. Ontology evolution in this context is further discussed in the next subsection.

Ontology evolution could be considered as the "purest" type of ontology change, in the sense that it deals with the changes themselves. As already mentioned, this area will be the main focus of this thesis. Ontology evolution is a very important problem, as the effectiveness of an ontology based application heavily depends on the quality of the conceptualization of the domain by the underlying ontology [81], which is directly affected by the ability of an evolution algorithm to properly adapt the ontology to changes in the domain and to new necessities in the conceptualization. Its significance is further emphasized by the fact that ontologies often model dynamic environments [80].

As we have already stated, an ontology is a specification of a shared conceptualization of a domain [38]. Thus, a change may be caused by either a change in the domain, a change in the conceptualization or a change in the specification [55]. The third type of change (specification change), as mentioned in the previous subsection, is in fact a translation problem. Thus, the definition of ontology evolution covers the first two types of change only; changes in the domain and changes in the conceptualization.

2.4 Change and Evolution: Current Context

For the scopes of this dissertation note that we will be narrowing down the meaning ontology evolution. For the rest of this work we will be focusing upon the two "major" phases of ontology change, namely the change representation and the semantics of change phase [80]. That is what we will be referring to as evolution and there lies also the practical interest of evolution. In effect, this is where the "problematic" part of ontology evolution resides and what justifies the existence of this work and our contribution.

Before these two phases, the necessity for a change is identified; during these phases, we represent the change in a suitable format and determine the modifications that must be made to the ontology in response to this need, as well as the indirect effects of these modifications on the validity, consistency and quality of the information stored in the ontology. In case that our initial modifications cause

problems that we detect (e.g., the removal of a class renders a certain instance unclassified and thus invalid, as in our example), further actions (changes) may be required to restore the ontology to an "acceptable" state. Later, during the fourth, fifth and sixth phase all the changes that have been determined, are to be applied to the ontology, propagated to dependent elements and validated respectively.

2.4.1 Changes: Elementary and Composite

According to [80, 82], change operations can be classified into elementary (involving a change in a single ontology construct) and composite ones (involving changes in multiple constructs), also called atomic and complex in [83]. Elementary changes represent simple, fine-grained changes; composite changes represent more coarse-grained changes and can be replaced by a series of elementary changes. Even though possible, it is not generally appropriate to use a series of elementary changes to replace a composite one, as this might cause undesirable side-effects [80]; the proper level of granularity should be identified in each case.

Examples of elementary changes are the addition and deletion of elements (concepts, properties etc) from the ontology. In our particular example of section 2.3.2, a simple "Remove_Concept" operation should be enough to perform the required change, executed as an elementary operation. Alternatively, we might have wished to move the concept A to some other point in the concept hierarchy. Intuitively we would characterize any operation representing this change as a composite change (or by a series of elementary changes).

There is no general consensus in the literature on the type and number of composite changes that are necessary. In [80], 12 different composite changes are identified; in [72], 22 such operations are listed; in [83] however, the authors mention that they have identified 120 different interesting composite operations and that the list is still growing! In fact, since composite operations can involve changes in an arbitrary number of constructs, there is an infinite number of them. Although composite operations can, in general, be decomposed into a series of elementary ones, for ad-hoc systems this is not of much help, as (as we will see later) the decomposition of a non-supported operation into a series of supported ones (even if possible) should be done manually.

Thus, creating a complete list of composite operations is not possible, but, fortunately, as we will see it is not necessary either, since a composite operation can always be defined as a series of elementary operations. What remains to be done is to develop an ontology evolution framework which exploits this feasibility.

2.4.2 Representing changes

A *change specification* in [55], is presented as "version relation" between ontological elements (such as classes) appearing in different versions of the ontology. The introduction of such a relation and the properties that such a relation should have form an interesting problem. Change specification's role is to make the relationship between different versions of ontological elements explicit. Other issues involved is the so-called "packaging of changes" [56] as well as the different types of compatibility and how these are identified [55].

A survey on the different ways that can be used to represent a set of changes, as well as the relation and possible interactions between such representations is provided in [57]; furthermore, a standard ontology of changes is proposed, containing both elementary and composite operations. A similar ontology of changes is proposed in [76], where the changes are identified through a version log stored in this ontology which in fact implements the pre-mentioned version relation in order to store the different versions of each element, as well as the relation between them and some related meta-data regarding these changes. The latter is in fact a technique proposed in [56].

2.5 Belief Change In Ontology Evolution

2.5.1 Why belief change

The considerations of section 2.2 form only a partial list of the issues that have been addressed by the belief change literature. The determination of the changes to be made in reaction to some new data is a complex and multifaceted issue. Apart from that several considerations need to be taken into account before choosing the modifications to be performed upon a KB.

Ontology change is a type of knowledge change, and therefore the pre-mentioned considerations hold also there, including ontology evolution. However, current ontology evolution works handle the problem in an inadequate manner, as they are not even considering most of these issues in the determination of their evolution algorithms [20, 27, 26]. For example, note that unlike the ontology evolution literature, in the belief change there is no human involved in the process of change.

Some works ([27, 26]) have recognized that the need for automatization in the evolution process calls for a belief change confrontation. Belief change deals with the problem in a fully automatic manner. In fact, the option of using a human in the loop of belief change was never even considered as an option, despite the complexity of the problem. This fact forms an additional argument in favor of the use of belief change techniques in ontology evolution, as such techniques will

arguably lead to automatic methods for dealing with such problems.

2.5.2 Relevant works

It is only recently, that the idea of recruiting belief change techniques for changing ontologies have come into use. Basically a few applications and works which are at a preliminary stage have taken exploitation of certain belief change techniques[50, 62, 67, 77, 41]. These works were developed independently, and they more or less agree to our line of thought as described in this thesis. We will briefly present these approaches, as they are somehow complementary to our view and our research direction regarding ontology evolution.

AGM

In AGM [1] theory, while authors where formulating their theory they made several, quite general assumptions. They followed Tarski's viewpoint on the definition of a logic: a logic in the AGM theory is a pair <L,Cn>. This assumption provides the link between their theory and the world of ontologies, as ontological representation languages like DLs and OWL can also be viewed as <L,Cn> pairs, as already mentioned.

This standpoint gave rises to several works which proposed the use of the AGM theory for ontology evolution. One such work, at a preliminary stage, is [50]. In this work some informal ideas regarding the connection of the AGM theory with ontology evolution are provided. The focus lies on the operations of contraction and revision and, following the lead of the AGM postulates, certain properties that should hold in a rational contraction and revision operation are presented.

However, these properties are not directly applicable to many DLs, for the same reasons that the AGM theory itself is not directly applicable to such DLs [28]. In [20, 29] the authors perform a study on the AGM theory and explain why the AGM postulates cannot be applied in the ontology evolution context as-is; they also provide a generalization of the AGM theory, which is one possible solution to the problems faced by the authors of [50] during their preliminary attempt with the AGM theory.

An other interesting feature of our study is the verification that current research on ontology evolution which steps on belief change uses only the explicit construction method (see section 2.2); one interesting side-effect of our approach is that it allows the development of a postulation method for this problem.

Epistemic Entrenchment

Classical belief merging has inspired works on ontology merging and has triggered further discussion upon these [58, 59]. However initial results that initiated such discussion were not examining ontology merging, but evolution. Based on epistemic entrenchment (another work of belief change literature), the authors of [62], the focus on a certain preference ordering. This approach is very similar to ours as we also model the minimal change principle with a certain ordering as we will see in chapter 4. However there are significant differences; unlike our approach, the authors try to enforce the updating upon DL's and in fact the only DL considered is ALU. Nevertheless, we focus on RDF/S as we will see.

Moreover in the above work the new knowledge is constrained to be a single DL axiom (however in literature there exist proposals on how to address the family of DLs as a whole, as well as OWL [20]). Apart form this furher simplifications are also made; the interrelationship between the Abox and the Tbox of the DL KB in the context of ontology evolution is studied and the distinction between the two parts of the KB is taken seriously. Under their understanding, the term ontology revision refers to the addition of an Abox assertion to the DL KB (i.e., they deal mostly with changes in the data level of abstraction rather than the schema; however, there have been discussions on applying belief change techniques to the evolution of concepts [30, 86]). Furthermore this "revision" is the only change operation considered, studying how new knowledge can be incorporated in a DL KB without introducing inconsistencies.

Maxi-adjustment

The maxi-adjustment algorithm [7], originally introduced for propositional knowledge integration, is re-cast to the context of DLs, in [67]. This algorithm allows the elimination of any inconsistencies that could arise in a stratified KB after its expansion with a new proposition. The authors propose that this process can, in turn, lead to the development of a revision algorithm. However, to overcome certain problem the authors depart from the classical DL model, limiting the applicability of their approach.

Chapter 3

Related Work And Motivation

Everyone thinks of changing the world,
but no one thinks of changing himself.

Leo Tolstoy

3.1 Problems in current approaches

3.1.1 Ad-hoc nature

Human driven change

The field of Ontology Evolution is very active, yet immature [72], however more and more works are dealing with it, as it is a research area of great importance for the Semantic Web. Similarly [73] states that the current situation of the research on ontology evolution is still in its early stages.The current state of the art in ontology evolution, as well as a list of existing tools that help the process can be found in [40]. Some of these tools are simple ontology editors, while others provide more specialized features to the user. Attempts to address this problem have found a lot of obstacles and problems. In literature there is a respectable amount of problematic approaches recognized.

For example, one problem with current approaches is that they require a varying level of human intervention in order to work properly. Some researchers consider this a necessary feature of the ontology evolution process [39], [40], considering unrealistic the expectation that changes upon ontologies could be handled automatically [73]. This conclusion is based on the argument that any given change can be resolved in several ways; moreover, the user requirements may be differing, even for the same change [81].

Thus, it is difficult for a computer system to decide on the best way to resolve a given change. Despite the validity of this claim, we believe that it is unrealistic to always rely on human participation during ontology evolution [80], for several reasons. First of all, the human user that intervenes in the process should be an ontology engineer and have certain knowledge on the domain. Very few people can be both domain and ontology experts. But even for these specialized experts, it is very hard to perform ontology evolution manually [39], [80].

So, in domains where changes occur often, it is simply not practical to rely on humans; the same holds in applications where it is difficult, impossible or undesirable for an ontology engineer to handle the change himself (as in autonomous robots, software agents or time-critical applications). A manual evolution approach will, in many cases, perform better, in the sense of rationality, than any computer system devisable. However, there are exceptions to this rule. Different ontology engineers may have different views on how a certain change should be implemented [80].

Whenever necessary, the ontology engineer should have the option to undo some, or all, the changes performed and/or perform changes of his own. In short, the role of the ontology engineer is not to perform the changes himself but to verify the quality of the system's output and possibly parameterize it, if she is not satisfied with the results. The most sophisticated current ontology evolution systems provide all these supporting features [40], but cannot perform the changes automatically. Therefore, the main gap that remains to be resolved is the automation of the process; this problem will be one of the main concerns of this thesis.

A list of systems related to the task of ontology evolution can be found in [40]. Some of these systems are simple ontology editors, but the most sophisticated of them provide features that greatly aid the user in the task of changing an ontology.For more details on such systems refer to [DSWKB00], [40].

Lack of adequate formalization

In existing attempts there is a general lack of adequate formalizations. A change request is an explicit statement of the modifications that are to be performed on the ontology. Nonetheless, the knowledge engineer should a-priori define the request in response to a more abstract need (e.g., an observation or a new fact). Therefore, current systems do not determine the actual modifications to be made upon the ontology when confronted with a different fact, but actually they more or less help the user determine the changes and import them to the system for implementation.

Furthermore, the fraction of the systems that do materialize some of the tasks of evolution are usually developed using ad-hoc heuristics that are based on the experience and expertise of their designers, ontology engineers or domain experts.

Practically, these tools attempt to emulate human behavior [84]. They are not theoretically founded nor have they specified any theoretical properties, formal methods or evaluation. Consequently, given an ontology and a change, a result is characterized as "good" or "best" based solely on intuition. Therefore we are not able to compare two ontology evolution algorithms with respect to effectiveness or rationality. In the majority of the current tools or systems dealing with evolution, such issues have been disregarded, as the decisions on the modifications to be made are left either on humans or are ad-hoc predesigned.

Empirical decisions based on the expertise of the person who represents the modification are taken when evolution is performed: when the ontology engineer encounters a change, she takes the decision based on his alternatives and chooses the best one, which is then fed to the system for implementation. All these decisions rely on her expertise on the subject, instead of a step-by-step, exhaustive method of evaluation. Nevertheless, in order to develop fully automatic ontology evolution algorithms, considerable issues need to have a clear, formal confrontation. In effect, in order to construct an ideal system, one should analyze and set specific properties to be satisfied by a "proper" ontology evolution algorithm, like:

- Track down (provably) all the alternative ways to address a given change, using a formal and exhaustive process and guarantee that there are no further alternatives for a given change, except the ones found.
- Acknowledge the features and properties that make a certain result of a change "better" than another, therefore maintaining a clear definition of best in this context, so it is able to:
- Decide on the "best" of the different alternatives.

Unfortunately, addressing these issues in a general frame is not easy using the current research direction as each type of change is treated in a different way, using a stand-alone, specialized and informal procedure. If a more formal path is not taken, the ontology evolution research is doomed to never find answers to these questions, and as a result to never escape the need for an ontology engineer supervising and participating in the procedure of ontology evolution. The opposite also is true: the participation of the ontology engineer in the process of ontology evolution threatens any formalization attempt, because it suggests non-determinism and non-reproducibility of the outcome.

More changes, more designing

A further problem with the so-far research is related to the representation of changes (second phase of ontology change). In current tools, which are mainly ontology

editors, there is usually little or no support for any kind of composite (complex) changes to the ontology. Ontological elements are simply deleted and added by the user, or, can be moved or copied [40]. In a little more sophisticated tools for ontology evolution, there is a pre-defined set of elementary and/or composite operations that are predesigned and stored onto the system in order to be supported, providing a somewhat greater flexibility to the engineer.

For each operation, there is a associated process that manipulates the change and its semantics. This procedure can sometimes be parameterized to comprise different necessities. However, there is no guarantee that the provided parameterization is enough to cover any possible need of the ontology engineer. Unpredictable needs may require unpredictable reactions to a certain change. Moreover, the number of complex operations that can be considered has no limit, as already discussed. Even in the case that we restrict ourselves to the most ordinary types, as we already mentioned there is usually a large number of them [83].

As a consequence the number of complex different operations to consider is extremely high; creating a new procedure for each such change is a non-scalable approach which increases the complexity of current ontology evolution tools. In effect, as also mentioned and will also be seen later the number of different composite changes is indeed infinite. Therefore any attempt to number them is doomed to always be insufficient.

Furthermore not all the different types of change are readily available at design-time; the problem becomes more complicated. New needs may require new operations. To manipulate operations that are not in the supported list, the ontology engineer must select a sequence of two or more simpler (more elementary) operations to perform. Such a selection might affect the quality of the change and could lead to unpredictable problems [80].

For example, deleting and then re-adding a concept in a different position in the hierarchy is not the same as moving the concept; in the former case, the instances of the concept will have to be removed or re-instantiated to a different concept; in the latter, no changes to the individuals will be required. In this example, the effects of the dividing of the operation in two more introductory operations may be easy to determine and/or undo. Yet, in more complex cases, this is far from trivial. In any case, this process cannot be performed without human involvement as it is unfeasible for a computer system to manipulate automatically a change that is not in its list of supported operations, even though the unknown operation can be perfectly split in more elementary, known operations.

An unforeseeable to the system operation is one that the former does not know its semantics. So there is a major problem on how could the system manipulate an operation it does not know and whether it could choose to use some of the supported operations for substituting the unknown operation. This kind of problem

is very likely to occur (as the creation of a complete list of composite operations is not possible [57]). In section 5.6 construct a paradigm showing that:

(a) Not any change operation can be decomposed to the existing elementary changes, in fact

(b) there is not a finite number of elementary, *modular with respect to validity*, changes in which any complex change can be decomposed, as in some occasions

(c) the estimated effects of the split series of changes, on the final result, are not the same with the intended effects of the initial complex change.

(d) Moreover different composite changes are of an infinite amount.

As a result, the current situation as far as the representation of changes is concerned practically overrules the possibility for automatic ontology evolution, which is a significant drawback of current systems.

3.1.2 Towards Formality and Practicality

Automated ontology evolution approaches seem, in general, detached from real problems and are not easily adaptable for use in an ontology evolution tool; to our knowledge, there is no implemented tool that uses one of the algorithms developed by formal approaches. On the other hand, editor-like tools do not provide enough automation and employ ad-hoc methodologies to deal with the problems raised during an update operation; such ad-hoc methodologies cause several problems that are thoroughly discussed in Section 3.2.

Our approach is motivated by the need to develop a formal framework that will lead to an easily implementable ontology evolution algorithm. We would like our approach to enjoy the formality of the second class of tools, and use this formality as a basis that will provide formal guarantees related to the behavior of the implemented system, thus avoiding the problems related to the ad-hoc nature of existing practical methodologies.

More specifically, our approach could be viewed as belonging to the class of works that it result to a formal, theoretical model to address changes. This model is based on a formal framework that is used to describe the process of ontology evolution as addressed by current editor-like tools (so it is also related to the first class of works), and allows us to develop an abstract, general-purpose algorithm that provably performs changes in an automated and rational way for a variety of languages, under different parameters (validity model and ordering). As discussed,

Our work is focused on the "core" of the ontology evolution problem, namely the change representation and semantics of change phases. Issues related to change capturing, implementation of changes, transactional issues, change propagation, visualization, interfaces, validation of the resulting ontology etc are not considered in this study.

On the other hand, our approach also results to an implemented tool, namely the Evolution Service of the SWKM. Our general-purpose algorithm can be applied for any particular language and set of parameters that is useful for practical purposes; for the purposes of SWKM we set these parameters so as to correspond to the RDF language under the semantics described in [78]. Fixing these parameters also allows us to better present our approach, as well as to evaluate and verify its usefulness towards the aim of implementing an ontology evolution tool. In addition to the implementation of the general-purpose algorithm, our formal framework allows the development (and implementation) of special-purpose algorithms which are more suited for practical purposes; such algorithms provably exhibit the same behavior as the general-purpose one, so we can have formal guarantees as to their expected output. Both the general-purpose and the special-purpose algorithms are implemented for the Evolution Service of SWKM.

3.2 Evolution Process

In this section, we elaborate on the five steps we described in section 1.2 and describe how some typical ontology evolution tools ([6, 33, 71, 85]) fit into this five-step process. In addition, we point out the problems that the ad-hoc implementation of these tools causes, and show how such problems could be overcome through the use of a formal framework, like the one described in Section 4.1.

3.2.1 Model Selection and Supported Operations

Obviously, the first step towards developing an evolution algorithm is the determination of the underlying representation model for the evolving ontology; this is what we capture in the first step of our 5-step process. Most systems assume a language supporting the basic constructs used in ontology development, like class and property subsumption relationships, instantiation relationships and domain and range restrictions for properties.

The selection of the representation model obviously affects (among other things) the operations that can be supported; for example, OntoStudio [85] does not support property subsumption relations so all related changes are similarly overruled. Further restrictions to the allowable changes may be introduced by various design

decisions, which may disallow certain operations despite the fact that they could, potentially, be supported by the underlying ontology model. For example, OntoStudio does not allow the manipulation of implicit knowledge, whereas OilED [6] does not support any operation that would render the ontology invalid (i.e., it does not take any actions to restore validity, but rejects the entire operation instead). The determination of the allowed (supported) update operations constitutes the second step of our 5-step process.

According to [80, 82], change operations can be classified into elementary (involving a change in a single ontology construct) and composite ones (involving changes in multiple constructs), also called atomic and complex in [83]. Elementary changes represent simple, fine-grained changes; composite changes represent more coarse-grained changes and can be replaced by a series of elementary changes. Even though possible, it is not generally appropriate to use a series of elementary changes to replace a composite one, as this might cause undesirable side-effects [80]; the proper level of granularity should be identified in each case. Examples of elementary changes are the addition and deletion of elements (concepts, properties etc) from the ontology. There is no general consensus in the literature on the type and number of composite changes that are necessary. In [80], 12 different composite changes are identified; in [72], 22 such operations are listed; in [83] however, the authors mention that they have identified 120 different interesting composite operations and that the list is still growing! In fact, since composite operations can involve changes in an arbitrary number of constructs, there is an infinite number of them. Although composite operations can, in general, be decomposed into a series of elementary ones, for ad-hoc systems this is not of much help, as the decomposition of a non-supported operation into a series of supported ones (even if possible) should be done manually.

The above observations indicate an important inherent problem with ad-hoc algorithms, namely that they can only deal with a predefined (and finite) set of supported operations, determined at design time. Therefore, any such algorithm is limited, because it can only support some of the potential changes upon an ontology, namely the ones that are considered more useful (at design time) for practical purposes, and, thus, supported.

3.2.2 Validity Model and Invalidity Resolution

It is obvious that a user expects his update request to be executed upon the ontology. Thus, it is necessary for the resulting ontology to actually implement the change operation originally requested, i.e., that the actual changes performed upon the ontology are a superset of the requested ones; this requirement will be called *success*.

The naive way to implement an update request upon an ontology would be to simply execute the request in a set-theoretic way. That would guarantee the satisfaction of the above principle (success); nevertheless, this would not be acceptable in most cases, because the resulting ontology could be invalid in some sense (e.g., if a class is removed, it does not make sense to retain subsumption relationships involving that class). Thus, another basic requirement for a change operation is that the result of its application should be a *valid* ontology, according to some validity model. This requirement is necessary in order for the resulting ontology to make sense.

Both the above principles are inspired by research on the related field of belief revision [13, 34], in which they are known as the Principle of Validity and Principle of Success respectively. The Principle of Success is well-defined, in the sense that we can always verify whether it is satisfied or not. The Principle of Validity however, depends on some underlying validity model, which is not necessarily the same for all languages and/or ontology evolution systems. Thus, each system should define the validity model that it uses. For example, do we accept cycles in the IsA hierarchy? Do we allow properties without a range/domain, or with multiple ranges/domains? Such decisions are included in the validity model determined in step 3 of our 5-step process.

Determining how to satisfy the Principles of Success and Validity during a change operation is not trivial. The standard process in this respect is to execute the original update request in a naive way (i.e., by executing plain set-theoretic additions and deletions), followed by the initiation of additional change operations (side-effects) that would guarantee validity. In principle, there is no unique set of side-effects that could be used for this purpose: in some cases, there is more than one alternatives, whereas in others there is none. The latter type of updates (i.e., updates for which it is not possible for both Success and Validity to be satisfied) are called *infeasible* and should be rejected altogether. For example, the request to remove a class, say C, and add a subsumption relationship between C and D at the same time would be infeasible, because executing both operations of the composite update would lead the ontology to an invalid state (because a removed class C cannot be subsumed by another class) and it can be easily shown that there is no way (i.e., side-effects) to restore validity without violating success for this update. The determination of whether an update is infeasible or not, as well as of the various alternative options (for side-effects) that we have for guaranteeing success and validity (for feasible updates) constitutes the fourth step of our 5-step process.

Let us consider the change operation depicted in Figure 3.1(a), where the ontology engineer expresses the desire to delete a class (B) which happens to subsume another class (C). It is obvious that, once class B is deleted, the IsAs relating B

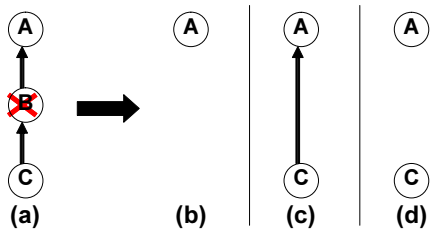

Figure 3.1: Three alternatives for deleting a class

with A and C would refer to a non-existent class (B), so they should be removed; the validity model should capture this case, and attempt to resolve it. One possible result of this process, employed by Protégé [71], is shown in Figure 3.1(b); in that evolution context, a class deletion causes the deletion of its subclasses as well. This is not the only possibility though; Figures 3.1 (c) and (d), present other potential results of this operation, where in (c), B's subclasses are re-connected to its father, while in (d), the implicit IsA from C to A is not taken into account. KAON [33], for example, would give either of the three as a result, depending on a user-selected parameter.

In this particular example, both KAON and Protégé detect the invalidity caused by the operation and actively take action against it; however, the validity model employed by different systems may be different in general. Moreover, notice that an invalidity is not caused by the operation itself, but by the combination of the current ontology state and the operation (e.g., if B was not in any way connected to A and C, its deletion would cause no problems). Therefore, in order for a mechanism to propose solutions against invalidities, both the ontology and the update should be taken into account. Notice that the mechanism employed by Protégé, in Figure 3.1, identifies only a single set of side-effects, while KAON identifies three different reactions. This is not a peculiarity of this example; the invalidity resolution mechanism employed by Protégé identifies only a single solution per invalidity; this is not true for KAON and OntoStudio.

3.2.3 Action Selection

Since, in the general case, there are several alternative ways (i.e., sets of side-effects) to guarantee success and validity, we need a mechanism that would allow us to select one of the alternatives for implementation (execution). This constitutes the last component of an evolution algorithm (step 5). Such a mechanism is "pre-built" into systems that identify only a single possible action, like Protégé, but can be also parameterizable. KAON, for example, provides a set of options (called

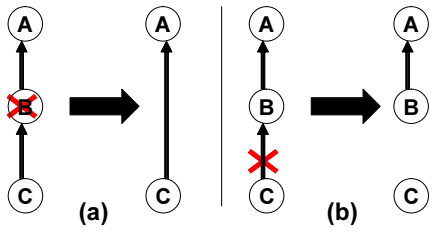

Figure 3.2: Implicit knowledge handling in KAON

evolution strategies) which allow the ontology engineer to tune the system's behavior and, implicitly, indicate what is the appropriate invalidity resolution action for implementation per case. OntoStudio provides a similar customization over its change strategies.

Notice that our preference among resulting ontologies reflects in a preference among side-effects of the corresponding update operations. For instance, if we prefer the result of Figure 3.1 (c), we can equivalently say that we prefer the (explicit) addition of the (implicit) subsumption relation shown in (c) together with the deletion of the two initial IsAs as a side-effect to this operation, over the deletion of the two initial IsAs and class C, shown in (b), or just the deletion of the two IsAs, as in (d). Therefore, the evolution process can be tuned by introducing a preference ordering upon the operations' side-effects that would dictate the related choice (evolution strategy). Given that the determination of the alternative side-effects depends on both the update and the ontology, there is an infinite number of different potential side-effects that may have to be compared. Thus, we are faced with the challenge of introducing a preference mechanism that will be able to compare any imaginable pair of side-effects.

It is worth noting here the connection of this preference ordering with the well-known belief revision Principle of Minimal change [13] which states that the resulting ontology should be as "close" as possible to the original one. In this sense, the preference ordering could be viewed as implying some notion of relative distance between different results and the original ontology, as identified by the preference between these results' corresponding side-effects.

3.3 Discussion

To the best of the author's knowledge, all currently implemented systems employ ad-hoc mechanisms to resolve the issues described above. The designers of these systems have determined, in advance, the supported operations, the possible in-

Table 3.1: Summary of ontology evolution tools

			Protégé	KAON	OntoStudio	OilED	SWKM
Change Representation	Fine-grained Model (Step 1)		✓	✓	×	✓	✓
	Supported Operations (Step 2)	Elementary	✓	✓	✓	×	✓
		Composite	×	×	×	×	✓
Semantics of Change	Validity Model (Step 3)	Faithful	×	×	✓	✓	✓
		Complete	×	×	✓	×	✓
	Invalidity Resolution (Step 4)	No alternatives				✓	
		One alternative	✓				
		Many alternatives		✓			
		All alternatives			✓		✓
	Selection Mechanism (Step 5)	None	✓			✓	
		Per-case		✓	✓		
		Globally					✓

validities that could occur per operation, the various alternatives for handling any such possible invalidity, and have already pre-selected the preferable option (or options, for flexible systems like KAON) for implementation per case; this selection (or selections) is hard-coded into the systems' implementations.

This approach causes a number of problems. First of all, each invalidity, as well as each of the possible solutions to each one, needs to be considered individually, using a highly tedious, case-based reasoning which is error-prone and gives no formal guarantee that the cases and options considered are exhaustive. Similarly, the nature of the selection mechanisms cannot guarantee that the selections (regarding the proper side-effects) that are made for different operations exhibit a faithful overall behavior. This is necessary in the sense that the side-effect selections made in different operations (and on different ontologies) should be based on an operation-independent "global policy" regarding changes. Such a global policy is difficult to implement and enforce in an ad-hoc system.

Such systems face a lot of limitations due to the above problems. For example, OilED deals only with a very small fraction of the operations that could be defined upon its modeling, as any change operation that would be triggering side-effects is unsupported (e.g., the operation of Figure 3.1 is rejected). In Protégé, the design choice to support a large number of operations has forced its designers to limit the flexibility of the system by offering only one way of realizing a change; in OntoStudio, they are relieved of dealing with (part of) the complexity of the aforementioned case-based reasoning as the severe limitations on the expressiveness of the underlying model constrain drastically the number of supported operations and cases to consider. Finally, in KAON, some possible side-effects are missing (ignored) for certain operations, while the selection process implied by KAON's parameterization may exhibit invalid or non-uniform behavior in some cases. As an example, consider Figure 3.2, in which the same evolution strategy was set in both (a) and

(b); despite this, the implicit IsA from C to A is only considered/retained in case (a).

Table 3.1 summarizes some of the key features of ontology evolution systems, categorized according to the 5-step process introduced in this work, and shows how each step is realized in each of the four systems discussed here, as well as in the Evolution Service of SWKM, described in Sections 4.1, 4.3 below.

We argue that many of the problems identified in this section could be resolved by introducing an adequate evolution framework that would allow the description of an algorithm in more formal terms, as a modular sequence of choices regarding the model used, the supported operations, the validity model, the identification of plausible side-effects and the selection mechanism. Such a framework would allow justified reasoning on the system's behavior, without having to resort to a case-by-case study of the various possibilities. To the best of the authors' knowledge, there is no implemented system that follows this policy. In Section 4.1, we describe such a framework and specialize it for RDF ontologies.

Chapter 4

Evolution Framework

Change begets change.

Charles Dickens

4.1 A Framework For Updating Knowledge

Our first work is to define the logic that will be working with. We will then define our framework with respect to the evolution process we acknowledged in the previous chapter.

4.1.1 Model and Operations

We consider a First-Order Language which allows the following:

- Parentheses: $(,)$
- Sentential connective symbols: $\wedge$, $\neg$, $\rightarrow$, ...
- A countably infinite number of variables: u_1, u_2, ...
- The equality symbol: $=$
- Quantifier symbols: $\forall$, $\exists$
- A finite number of predicate symbols: P_1, P_2, ..., P_n
- A countably infinite number of constant symbols: x_1, x_2,...
- No function symbols

We will denote with L the set of all well-formed formulae that can be formed in this FOL. We equip our FOL with closed semantics, i.e., CWA (*closed world assumption*). This means that, for two formulas p, q, if $p \nvdash q$ then $p \vdash \neg q$. Any expression of the form $P(x_{j1}, ..., x_{jk})$ will be called a *positive ground fact* where P is a predicate of arity k and $x_{j1}, ..., x_{jk}$ are constant symbols. Any expression of the form $\neg P(x_{j1}, ..., x_{jk})$ is called a *negative ground fact* iff $P(x_{j1}, ..., x_{jk})$ is a positive ground fact.

We denote by L^+ the set of positive ground facts, L^- the set of negative ground facts and set $L^0 = L^+ \cup L^-$, called the set of ground facts of the language. The major restriction of our model is that knowledge is represented using positive ground facts only. On the other hand, updates can contain only ground facts, which can be either positive or negative. Positive ground facts encode the information that should be added in the knowledge, while negative ground facts encode the information that should be removed from our knowledge. Formally:

- A *belief set* is a set $K \subseteq L^+$
- An *update* is a set $U \subseteq L^0$

In simple words, belief set is any set of positive ground facts whereas update is any set of positive or negative ground facts. By definition, belief sets have two properties:

(a) they are always consistent and

(b) they imply only the ground facts that are already in the belief set.

Note that in this chapter the term *consistent* has the very specific meaning of FOL consistency, which dictates that a belief set is consistent iff it does not imply both a proposition and its negation. Notice that, abusing notation, for two sets of ground facts U, V, we will say that U implies V ($U \vdash V$) to denote that $U \vdash p$ for all $p \in V$. Combining the above two properties with the CWA semantics, we can state that if a ground fact isn't contained in a belief set, then the belief set implies its negation. In effect it holds:

- iff $P(x) \in K$ then $K \vdash P(x)$
- iff $P(x) \in K$ then $K \nvdash \neg P(x)$
- iff $P(x) \notin K$ then $K \vdash \neg P(x)$
- iff $P(x) \notin K$ then $K \nvdash P(x)$

- iff $K \vdash P(x)$ then $K \nvdash \neg P(x)$
- iff $K \nvdash P(x)$ then $K \vdash \neg P(x)$‘

One first application of these properties is that the addition of $\neg P(x)$ in K is the same as the contraction of $P(x)$ from K, because being unable to infer $P(x)$ implies (due to CWA) being able to infer $\neg P(x)$ (and vice-versa). Therefore, the addition of negative ground facts to our belief sets corresponds to contraction/erasure in the standard terminology, while the addition of positive ground facts corresponds to revision/update in the standard terminology. Note that under closed semantics revision and update do not differentiate as in our example of section 2.2.3. In that example, we had partial knowledge of the world, but under CWA, what we don't know simply doesn't hold. In fact, in a closed world, update is degenerating to revision. Although this might be an obstacle for the applicability of our framework in richer languages, CWA is sufficient, useful and desirable for the scopes of our work.

4.1.2 Consistency and Validity

We assume that, in general, not every belief set is a valid representation of our knowledge. To discriminate between "valid" and and "non-valid" representations, we allow the introduction of Integrity Constraints. For technical reasons, we assume that all constraints can be encoded in the form of a DED (disjunctive embedded dependencies), which have the following general form:

$$\forall u P(u) \rightarrow \vee_{i=1,...,n} \exists v_i Q_i(u, v_i) \qquad \text{(DED)}$$

where:

- u,v_i are tuples of variables
- P, Q_i are conjunctions of relational atoms of the form $R(w_1, ..., w_n)$ and equality atoms of the form $(w = w')$,
- where $w_1, ..., w_n$, w, w' are variables or constants
- P may be the empty conjunction

An *integrity constraint* (or rule) is a FOL formula of the above form (i.e., a DED). Integrity constraints (DEDs) are FOL formulas. Notice however, that not all FOL formulas can be written in this form. Yet, DEDs are expressive enough for our case. We say that a belief set K *satisfies* an integrity constraint c, if $K \vdash c$.

Obviously for a set C of Integrity Constraints, K satisfies C ($K \vdash C$) iff K satisfies c, for all $c \in C$.

Let us see what it means for K to satisfy an integrity constraint c. Let's start with a simple case; suppose that $c = \forall u P(u) \rightarrow Q(u)$, where P, Q are simple positive predicates and u is a variable. Take some constant x. It holds:

$$If\ P(x) \notin K\ then\ K \vdash \{\neg P(x)\} \vdash \{P(x) \rightarrow Q(x)\}$$

and

$$If\ Q(x) \in K\ then\ K \vdash \{Q(x)\} \vdash \{P(x) \rightarrow Q(x)\}.$$

Also

$$\begin{array}{l} If\ P(x) \in K\ and\ Q(x) \notin K \\ then\ K \vdash \{P(x)\}\ and\ K \vdash \{\neg Q(x)\}, \\ so\ K \vdash \{P(x) \wedge \neg Q(x)\}. \end{array}$$

Equivalently $K \vdash \{\neg(P(x) \rightarrow Q(x))\}$. Since K is consistent, we conclude that $K \nvdash \{P(x) \rightarrow Q(x)\}$. The above thoughts lead to the following conclusion:

$$K \text{ satisfies } c \text{ iff for all constants } x : P(x) \notin K \text{ or } Q(x) \in K.$$

Or

$$K \text{ satisfies } c \text{ iff for all constants } x : K \vdash \{\neg P(x)\} \text{ or } K \vdash \{Q(x)\}.$$

This can be easily extended to the general case: Suppose that $c = \forall u P(u) \rightarrow \vee_{i=1,...,n} \exists v_i Q_i(u, v_i)$. Suppose also that: $P(u) = P_1(u) \wedge P_2(u) \wedge ... \wedge P_k(u)$ for some $k \geq 0$ and that $Q_i(u, v_i) = Q_{i1}(u, v_i) \wedge Q_{i2}(u, v_i) \wedge ... \wedge Q_{im}(u, v_i)$ for some $m > 0$ depending on i. Then K satisfies c iff for all tuples of constants x at least one of the following is true:

- There is some $j : 0 < j \leq k$ such that $K \vdash \{\neg P_j(x)\}$.
- There is some $i : 1 \leq i \leq n$ and some tuple of constants z such that for all $j = 1, 2, ..., m\ K \vdash \{Q_{ij}(x, z)\}$.

We can conclude that $K \vdash c$ iff for all tuples of constants x at least one of the following sets is implied by K:

$$\begin{array}{l} \{\neg P_j(x)\}, 0 < j \leq k \\ \{Q_{i1}(x, z) \wedge Q_{i2}(x, z) \wedge ... \wedge Q_{im}(x, z)\}, 1 \leq i \leq n, z : constant \end{array}$$

Based on the above observation, we define the component set of c with respect to some tuple of constants x as follows:

$$Comp(c,x) = \{\{\neg P_j(x)\}|0 < j \leq k\} \cup \{\{Q_{i1}(x,z) \wedge Q_{i2}(x,z) \wedge ... \\ \wedge Q_{im}(x,z)\} \,|1 \leq i \leq n, z : constant\}$$

Using Comp(c,x), the above facts can be rewritten by saying that:

K satisfies c iff for all constants x there is some $U \in Comp(c,x)$ such that $K \vdash U$.

Our definitions are also suitable also in the limit cases. For example, if $V = \emptyset$, then the empty set is perfectly acceptable in $Comp(c,x)$, and it is trivially satisfied by all K (i.e., $K \vdash \emptyset$, so $K \vdash c$). If, on the other hand, $Comp(c,x)$ turns out to be $\emptyset$, then K does not satisfy c, because there is no $U \in Comp(c,x) = \emptyset$ for which $K \vdash U$. Thus, the above process is embedded neatly into our framework.

Elimination of equality atoms

Some special consideration should be taken for the case where a $Q_{ij}(x,z)$ is an equality of the form $w = w\prime$. Obviously, during updates, we cannot change the value of a constant. For example, if x, y are distinct constants, then it is assumed that $x \neq y$. This is known in the literature as the "unique name assumption". This assumption is appropriate for our purposes; for example, as we will see later two RDF classes A, B are by definition distinct. Now take any $Q_{ij}(x,z)$in some set $V \in Comp(c,x)$. If this is an equality of the form $w = w\prime$, then the truth value of this equality is known from the very beginning. Since there are no free variables in $Q_{ij}(x,z)$ and given the unique name assumption, the truth value of $w = w\prime$ is independent of any particular KB K. In other words $w = w\prime$ is either a tautology or a contradiction, for the given assignment of variables to constants. This allows us to simplify the definition of $Comp(c,x)$ as follows: Take any $V \in Comp(c,x)$ and some $Q_{ij}(x,z) \in V$ such that $Q_{ij}(x,z)$ is an equality of the form $w = w\prime$. If $w = w\prime$ is a tautology, then remove it from V, i.e., set $V := V \setminus \{Qij(x,z)\}$. If $w = w\prime$ is an antinomy, then remove V from $Comp(c,x)$, i.e., set $Comp(c,x) := Comp(c,x) \setminus \{V\}$. For the rest of this thesis we will not be discussing about equalities, as we consider these eliminated at run-time of an update, as described.

The intuition behind these decisions is obvious (it preserves the definition of "K satisfies c"). This process does not add any extra complexity to the problem, even in the limit cases. Moreover, the above process is being undertaken for two reasons:

(a) it reduces the size of $Comp(c, x)$ and speeds up the relevant checks, and

(b) under the above assumptions, the elements of $Comp(c, x)$ contain only positive and negative ground facts, so they are updates in our terminology; this way, $Comp(c, x)$ is a *family of updates* set.

Def. 1. *Consider a FOL language $< L, Cn >$ and a set of Integrity Constraints C. A belief set K will be called valid with respect to $< L, Cn >$ and C iff K is consistent and it satisfies the integrity constraints C. Valid belief sets will also called Knowledge Bases (KBs). Using our terminology, a belief set K is valid iff for all $c \in C$ and all constants x, there is some $U \in Comp(c, x)$ such that $K \vdash U$.*

4.1.3 Invalidities Resolution

After deciding on the logic and defining our terminology we will dive into the detailed process of an update operation. As an initial simple case, consider that there are no integrity constraints or that we are not interested in the result being a KB. In this case, the most rational choice would be to simply apply the changes in U upon K; this means that the positive ground facts that appear in U should be added to K (if not already present) and the positive ground facts that appear in U as negative ground facts should be removed from K (if present). The other ground facts that appear in K should stay unaffected. These ideas can be formalized as follows:

Def. 2. *The raw application of an update U upon a belief set K is denoted by $K + U$ and is the following belief set: $K + U$=$\{P(x) \in L^+ \mid P(x) \in U$ or $(P(x) \in K$ and $\neg P(x) \notin U)\}$*

If we assume that U is consistent (as we will also see later we are interested only in consistent updates) then it is trivial to show that:

$$K + U = \{P(x) \in L^+ | P(x) \in K \cup U and \neg P(x) \notin U\}$$

An *elementary* or *singular* operation or update is the addition of a positive or negative ground fact in a KB. Every ground fact in our FOL can form an elementary operation when contracting it from a KB or revising the latter with this. However, raw application of even one singular operation to a KB may drive to an non-valid belief set.

For example: consider the rule $c = P(x) \rightarrow Q(x)$, where P,Q are ground facts,and x a constant. Component set of c, is $Comp(c, x)$=$\{\{\neg P(x)\}, \{Q(x)\}\}$. Let K be our KB before the update. From the definition of a KB we have that for all constants x, there is some $U \in Comp(c, x)$ such that $K \vdash U$. Let $P(x)$ and $Q(x)$

miss from K. Then it holds that $K \nvdash \neg P(x)$. Suppose and that $U = \{\neg P(x)\}$. The absence of $P(x)$ is exactly why K respects our rule. Now suppose we want to add $P(x)$ in K and get a resulting belief set K'. K' is not any more a KB (is not valid), because $K' \vdash P(x)$, and so there is not a $U \in Comp(c, x)$ that $K' \vdash U$.

In order to restore the validity, we have to add also to our belief set one set $U \in Comp(c, x)$, so then our belief set will imply U,satisfying the above rule. In simple words, under the specific constraint when we add $P(x)$ to a KB, we have to add also $Q(x)$ as a side-effect. This is the essence of a side-effect; a set of elementary operations, i.e. an update triggered by an elementary operation (or a set of them) due to the existence of a rule. Notice that the contents of a component set not only are updates, as we have discussed, but are updates that should be possibly enforced upon our KB in case the rule containing them is threatened. In order to pay respect a specific rule, each KB, has to imply at least one component of this rule's component set. Otherwise we are obliged to add at least one these components as an update to our belief set. Notice as updates, the components themselves could trigger another rule to be violated and subsequently more side-effects and so on.

The components of the a rule, might be many in amount, and so there would be many possibilities in order for a change, that violated this rule, to be realized. In the previous example the rule could be $c = P(x) \rightarrow Q(x) \vee R(x)$, which would alter our reaction; when adding $P(x)$ we would have to add $Q(x)$ or $R(x)$ as a side-effect. Hence, we now that side-effects are encapsulated inside the rule's themselves (actually their component set variant). We now have to set up a mechanism that exploits this property of component sets acknowledging all the different side-effects and enforcing the "best" of them according to a preference parameterization.

Cost of Updating

Actually, when a set of changes (i.e., an update U) is enforced upon a KB K, we would like to know the number and type of elementary operations that are required in order for the changes to be performed. In effect, what is the cost of the above raw application; what changes are needed. We are, firstly looking upon a formalism which would encapsulate the cost of the raw application of an update, although this cost may be increased if side-effects are taken into account. For example, the addition of $P(x)$ in K could violate an integrity constraint, thus forcing us, in turn, to add (or remove) another predicate in order to preserve the integrity of the KB. Such additional changes are called side-effects and they are not taken into account in the calculation of cost. In fact, we will later use this formalism in order to provide a rational mechanism that estimates those side-effects which unified

together with the update (as if it were parts of it all along) will produce the most desirable (profitable) cost when this reformed update is raw applied to our initial KB.

This cost we try to formulate (of raw applying U to K) will be denoted by $Cost(K, U)$ and is a set of positive and negative ground facts (i.e., an update). Notice that this cost is determined by the number of operations that really need to be performed. For example, if we are asked to add $P(x)$ in K and $P(x)$ is already in K, then there is no cost for this operation; similarly, if we are asked to add $\neg P(x)$ to K (i.e., remove $P(x)$ from K) and $P(x)$ is not in K, then this operation has no cost, or more correct: the cost is the empty set. Using the above observations, we define:

Def. 3. $Cost(K, U)$=$\{P(x) \in L^+ \mid P(x) \in U$ *and* $P(x) \notin K\} \cup \{\neg P(x) \in L^- \mid \neg P(x) \in U$ *and* $P(x) \in K\}$

Using the properties of belief sets, we can equivalently rewrite this as follows:

$$Cost(K, U) = \{P(x) \in U | K \nvdash \{P(x)\}\}$$

This cost represents a particular "edit distance" or "delta"between the belief set K and the belief set that occurs after raw applying the changes U to K. A first observation on our definitions is that $Cost(K, U)$ produces the same result with U, when applied on a KB. This is synopsized in lemma 2.

Lemma 2. *Consider K a valid belief set and U a consistent update. It holds:*

$$K + U = K + Cost(K, U)$$

Proof:

$$\begin{aligned}
K + Cost(K, U) = \\
\{P(x) \in L^+ | P(x) \in K \cup Cost(K, U) \; and \; \neg P(x) \notin Cost(K, U))\} = \\
\{P(x) \in L^+ | (P(x) \in K or (P(x) \in U and K \nvdash \{P(x)\})) \; and \\
(\neg P(x) \notin \{P(x) \in U and K \nvdash \{P(x)\}\})\} = \\
\{P(x) \in L^+ | (P(x) \in K \cup U and (P(x) \in K or \; P(x) \notin K)) \; and \\
(\neg P(x) \notin \{P(x) \in U and P(x) \notin K\})\} = \\
\{P(x) \in L^+ | P(x) \in K \cup U \; and \; \neg P(x) \notin \{P(x) \in U and P(x) \notin K\}\} = \\
\{P(x) \in L^+ | P(x) \in K \cup U \; and \; \neg P(x) \notin U = \\
= K + U \blacksquare
\end{aligned}$$

Note also, that the cost function is an operator which given an update and a KB, keeps only the meaningful operations, i.e, those that are non already implemented (implied) by the KB. Lemma 3 states that any part of the update is not contained in $Cost(K, U)$ iff it is not implied by K, and also that since $Cost(K, U)$ is already "irredundant" of void changes, then if we apply $Cost$ on itself will get back the same result. Third part of lemma 3 informs us that different updates which result in same KB, have the same (unique) cost.

Lemma 3. *Consider U a consistent update, $P(x) \in U$, and K a valid belief set. It holds:*

(a) $P(x) \notin Cost(K, U)\ iff\ K \vdash \{P(x)\}$

(b) $Cost(K, U) = Cost(K, Cost(K, U))$

(c) $Cost(K, U)$ = $Cost(K, U')$ *iff* $K + U$=$K + U'$

Proof:

(a) $P(x) \in U$, so if $P(x) \notin Cost$ then (by definition of $Cost$) $K \vdash P(x)$, as if it was not, then $P(x)$ would be a part of cost. On the other way round, if $K \vdash \{P(x)\}$ then again by definition of $Cost$ the $P(x)$ cannot be a part of it.

(b) Consider some $P(x) \in Cost(K, U)$. Then $P(x) \in U$, $K \nvdash \{P(x)\}$ and $P(x) \in Cost(K, U)$, so $P(x) \in Cost(K, Cost(K, U))$. Now consider some $P(x) \in Cost(K, Cost(K, U))$. Then $P(x) \in Cost(K, U)$ by definition. Thus: $Cost(K, U) = Cost(K, Cost(K, U))$.

(c) Suppose $Cost(K, U) = Cost(K, U')$ then $K + Cost(K, U) = K + Cost(K, U')$ and according to lemma 2(a) it holds that $K + U = K + U'$. If on the other hand $K + U = K + U'$ then :

If $P(x) \in Cost(K, U)$ then $P(x) \in U$ and $K \nvdash \{P(x)\}$; moreover $\neg P(x) \notin U$. This means that if $P(x)$ is a positive ground fact then $P(x) \in K + U$, but if $P(x)$ is a negative ground fact, then $\neg P(x) \notin K + U$; in either case: $K + U \vdash \{P(x)\}$, so (as $K + U = K + U'$) $K + U' \vdash \{P(x)\}$. For $P(x)$ to belong to U' it remains to show that $P(x) \in U'$. If $P(x)$ is a positive ground fact, then $P(x) \in K + U'$ so $P(x) \in U'$ (since $P(x) \notin K$). If, on the other hand, $P(x)$ is a negative ground fact, then $\neg P(x) \notin K + U'$; therefore, $\neg P(x) \in K$ (from that $K \nvdash \{P(x)\}$)and $\neg P(x) \notin K + U'$, so $P(x) \in U'$. Therefore $P(x) in U'$. Entirely symmetrically we show that if $P(x) \in Cost(K, U')$ then $P(x) \in Cost(K, U)$. Thus, $Cost(K, U) = Cost(K, U')$■

4.2 Selection Mechanism

Considering an update operation U, which when being enforced on K might cause some side-effects U', the set of changes that should be performed in this case is $U \cup U'$, i.e., the direct effects (U), expanded with the side-effects (U'). We can estimate the total cost for this cases that is $Cost(K, U \cup U')$.

Now suppose an update U which can have two possible (alternative) sets of side-effects U_1 and U_2. This could happen if the violation of an integrity constraint could be resolved in more than one ways. Which set of side-effects should be selected for application? The answer seems obvious: the one that is less costly. To determine the "less costly" one, we need a method to compare the two updates $U \cup U_1$ and $U \cup U_2$. Actually given such a method we can compare the cost operators $Cost(K, U \cup U_1)$ and $Cost(K, U \cup U_2)$, which are themselves also updates (in effect equivalent to the above according to lemma 3) but which contain non-void changes, thus better "measuring" the effects of $U \cup U_1$ and $U \cup U_2$.

In general, we need an ordering $\leq$ between updates; for technical reasons, this ordering should depend on K itself. This is true because the same set of operations may have different "weight" or cost in different KBs (even if they have the same side-effects). As an example in RDF, consider the removal of an IsA relation between A and B. The importance (i.e., cost), and in fact the effects and side-effects (i.e., $Cost$), of this IsA removal should depend on the importance of the concepts A, B in the RDF graph itself.

Since the ordering depends on K, we use the more adequate symbolism $\leq_K$. The set of all orderings $\{\leq_K | K : KB\}$ is called an *ordering scheme* and will be denoted by $\leq$. This ordering is defined among updates only.

An update-generating ordering scheme allows us to determine for all K and for all possible sets of effects and side-effects $U_1, U_2, ...$ the one that is the "cheapest", as the relation is transitive and all costs are comparable (totality), and even in the case that more than one "cheapest" updates may exist, these will have the same set of actual effects (Cost antisymmetry). In addition, conflict Sensitivity identifies the cost of a certain update with the actual changes required upon the KB. This way, void changes do not affect the cost of a change. Finally, monotonicity encodes the intuitively expected property that "more" changes are more expensive than "less" changes.

Def. 4. *An ordering $\leq_K$ is called update-generating iff the following conditions*

hold:

Cost Antisymmetry :

$$For\ any\ U, U' : U \leq_K U'\ and\ U' \leq_K U,$$
$$implies\ Cost(K, U) = Cost(K, U').$$

Transitivity :

$$For\ any\ U, U', U'' :\ U \leq_K U'\ and\ U' \leq_K U''$$
$$implies\ U \leq_K U''.$$

Totality :

$$For\ any\ U, U' :\ U \leq_K U'\ or\ U' \leq_K U.$$

Conflict Sensitivity :

$$For\ any\ U, U' :\ U \leq_K U'\ iff\ Cost(K, U) \leq_K$$
$$Cost(K, U').$$

Monotonicity :

$$For\ any\ U, U' :\ U \subseteq U'\ implies\ U \leq_K U'$$

Similarly, an ordering scheme $\{\leq_K | K : KB\}$ *is called update-generating iff* $\leq_K$ *is update-generating for all KBs* K.

4.2.1 Deltas

As we have observed the $Cost$ function is a kind of edit distance between the initial and the resulting belief of an update. Note here that $Cost(K, U)$, is the "distance" between K and $K + U$, not between K and U. This latter observation can be used to invent a little trick:

Suppose that:

1. $K = \{P(x)\}$
2. $K_1 = \{Q(x)\}$
3. $K_2 = \{P(x), Q(x)\}$

Viewing K_1, K_2 as beliefs, we notice that K_2 can be taken from K by the simple addition of $Q(x)$ in K; on the other hand, getting K_1 from K requires two operations, namely to remove $P(x)$ and to add $Q(x)$. So, we would expect that K_2 is closer to K than K_1. However, $Cost(K, K_1)$=$Cost(K, K_2)$=$\{Q(x)\}$, so by monotonicity and conflict sensitivity we get that $K_1 \leq_K K_2$ and $K_2 \leq_K K_1$.

In fact, the ordering defined cannot be used directly to determine the distance between two belief sets. $\leq_K$ encodes the cost of performing a certain number

of operations upon K. It determines the distance between the initial set and the resulting sets coming up from the application of the two belief sets on the original. If we had viewed K_1 and K_2 as updates upon K, they would have the same cost, as they practically enforce upon K the exact same operations. However, getting to K_1 from K is a different thing than applying K_1 upon K. Given an initial KB K and an update U $Cost(K, U)$ gives us the delta of K to $K + U$. We would find useful an operator which given two belief sets K and K', would return us their delta; a specific update U' for which $K' = K + U'$. In addition we would prefer to have not U' but U' without its void changes on K, i.e., $Cost(K, U')$

Def. 5. *We define the delta of two KBs K, K' as follows:*

$$Delta(K, K') = \{P(x) \in L^{+} | P(x) \in K' \; and \; P(x) \notin K\} \cup \{\neg P(x) \in L^{-} | P(x) \in K \; and \; P(x) \notin K'\}$$

Using the properties of beliefs we can equivalently rewrite this as follows:

$$Delta(K, K') = P(x) \in L^{0} | K' \vdash \{P(x)\} \; and \; K \nvdash \{P(x)\}$$

$Delta(K, K')$ encodes the cost of the change request that should be applied upon K in order to get to K'. Notice that $Delta(K, K')$ is an update. While we could have defined it elsewise, with respect to cost, we preferred the above definition of delta for theoretic symmetry and compatibility with a complementary to ours work, published in [89]. In this work the author's exhibit several nice properties of the same definition of delta between two KBs (and in fact they map this work on RDF/S, as we will also do later). Delta set correctly encodes the distance between two beliefs, because it contains the exact changes that should be performed upon a belief K to get a new belief K', as seen in lemma 4.

Lemma 4. *For U a consistent update and K a KB, it holds that:*

(a) $K + Delta(K, K') = K'$

(b) $Delta(K, K + U) = Cost(K, U)$

(c) If a KB $K' \vdash U$ then $Cost(K, U) \subseteq Delta(K, K')$

Proof:

(a) Take some $P(x) \in K'$. Obviously $P(x)$ is a positive ground fact and $K' \vdash \{P(x)\}$, $K' \nvdash \neg P(x)$. Thus: $\neg P(x) \nvdash Delta(K, K')$. If $P(x) \in K$, then $P(x) \in K + Delta(K, K')$. If $P(x) \notin K, then K \nvdash \{P(x)\}$, so $P(x) \in$

$K + Delta(K, K')$. Suppose now that $P(x) \notin K'$. If $P(x)$ is not a positive ground fact, then obviously $P(x) \notin K + Delta(K, K')$, so let's consider the case that $P(x)$ is a positive ground fact. Then, $K' \vdash \{\neg P(x)\}$, $K' \nvdash \{P(x)\}$. If $P(x) \in K$, then $K \nvdash \{\neg P(x)\}$, so $\neg P(x) \in Delta(K, K')$, so $P(x) \notin K + Delta(K, K')$. If $P(x) \notin K$, then, since $P(x) \notin Delta(K, K')$ we conclude that $P(x) \notin K + Delta(K, K')$. These facts imply that $K + Delta(K, K') = K'$.

(b) If $P(x) \in Cost(K, U)$ then $P(x) \in U$ and $K \nvdash \{P(x)\}$; moreover $\neg P(x) \notin U$. This means that if $P(x)$ is a positive ground fact then $P(x) \in K + U$, but if $P(x)$ is a negative ground fact, then $\neg P(x) \notin K + U$; in either case: $K + U \vdash \{P(x)\}$, so $P(x) \in Delta(K, K + U)$. If $P(x) \in Delta(K, K + U)$ then $K \nvdash \{P(x)\}$ and $K + U \vdash \{P(x)\}$. It remains to show that $P(x) \in U$. If $P(x)$ is a positive ground fact, then $P(x) \in K + U$ so $P(x) \in U$ (since $P(x) \notin K$). If, on the other hand, $P(x)$ is a negative ground fact, then $\neg P(x) \notin K + U$; therefore, $\neg P(x) \in K$ (from that $K \nvdash \{P(x)\}$)and $\neg P(x) \notin K + U$, so $P(x) \in U$.

(c) Take some $P(x) \in Cost(K, U)$. Then $K \nvdash P(x)$; moreover $P(x) \in U$, so $K' \vdash P(x)$. Thus, $P(x) \in Delta(K, K')$.■

Now, in order to determine whether K_1 is closer to K than K_2, we need to determine whether the operations required to get to K_1 from K are cheaper than the operations required to get to K_2 from K. This is equivalent to comparing $Delta(K, K_1)$ with $Delta(K, K_2)$ and whenever we want to determine whether K_1 is closer to K than K_2, we will determine whether $Delta(K, K_1) \leq_K Delta(K, K_2)$ or vice-versa, thus estimating whether $Cost(K, U_1) \leq_K Cost(K, U_2)$ for $K + U_1 = K_1$, $K + U_2 = K_2$.

4.2.2 Rational Change Operator

One of the most important properties that an update operation should satisfy is to actually perform the changes that is asked to perform. This is guaranteed even with the raw application of U upon K. Additionally, an update operation should guarantee that the result is a valid KB; this is not guaranteed by the raw application of U upon K, but can be achieved by introducing a set of side-effects to the original update effects. For some updates however, these two requirements are conflicting. It is possible that there is no adequate set of side-effects guaranteeing that at the end of the process the resulting belief set will be valid. Such changes cannot be implemented.

Def. 6. *An update U is called feasible iff there is some KB K such that $K \vdash U$.*

For an *infeasible* update, there is some rule where all its components are forced not to implied by the KB. Notice that an inconsistent update is infeasible.

We are now ready to formally define a change operator:

Def. 7. *A change operation (denoted by $\bullet$) is a function $\bullet : L^+ \times L^0 \rightarrow L^+$.*

In effect, a change operation is applied upon a belief set and an update and returns a new belief set. We will use infix notation to denote change operations (i.e., $K \bullet U$). Not all change operations are interesting, as we would like to add some nice properties to the one we will use:

Def. 8. *Consider a FOL language $< L, Cn >$, a set of integrity constraints C and some update-generating ordering scheme $\leq$. A change operation $K \bullet U$ will be called rational with respect to $\leq$ iff it satisfies the following properties for all belief sets K and updates U:*

- *Limit Cases: If K is not a KB or U is not a feasible update, then: $K \bullet U = K$.*
- *General Case: If K is a KB and U is a feasible update, then:*
  - *Primacy of New Information: $K \bullet U \vdash U$*
  - *Consistency Maintenance: $K \bullet U$ is a KB*
  - *Principle of Minimal Change: For all KBs K' such that $K' \vdash U$, it holds that $Delta(K, K \bullet U) \leq_K Delta(K, K')$*

Definition 8 dictates that applying a *rational* change operator between an update and a KB should result (in the general case) to a valid KB, which implies the update (i.e., the changes are successful). Moreover, for any other KB K' that could be an alternative result (i.e, K' is implementing the changes and is valid) the set of (non-void) side-effects leading to K' (captured by the $Delta$ function) is more "expensive" than the set of (non-void) side-effects leading to the result of the rational change operation. In effect, the rational change operator applies the "minimum", with respect to $\leq$, set of side-effects. Some results on Def. 8 follow.

Lemma 5. *Consider a FOL $< L, Cn >$, a set of integrity constraints C and an update-generating ordering scheme $\leq$, there is exactly one rational update operation because there is one minimum by the definition of $\leq_K$ (totality and Cost antisymmetry).*

Proof:
Consider an update U and a KB, there is always one rational change operation (by definition) if U is infeasible, as $K \bullet U = K$ (limit case). If U is feasible then

there is a KB K' which $K' \vdash U$. Now consider any change operation '$\star$' for which $K \star U = K'$. Obviously for $K \star U$ the first two properties of the rational change operation definition hold and if there is no other KB that implies U, the third property also holds and so '$\star$' is a rational change operator. If there is a second different than K', KB $K'' \vdash U$, then we can define another change operation $K \diamond U = K''$. Now, for $Delta(K, K \star U)$ and $Delta(K, K \diamond U)$ it must hold either that one of them is cheaper than the other in terms of $\leq_K$ (because of totality) or they are equal (by Cost antisymmetry). In any occasion if the update is feasible (so there is a change operator which satisfies the first two properties), there also is one operator '$\bullet$' which produces the minimal $Delta(K, K \bullet U)$ thus satisfying also the third property of the definition and being a rational change operator. So there exists at least one rational change operation for every language, set of integrity constraints, and update-generating ordering scheme.

Suppose that there are two different rational change operators with respect to the same update-generating ordering scheme $\leq$. We represent them '$\bullet$' and '$\star$' respectively. For a specific feasible update U let K_1, K_2 be the results of the corresponding change operations on K. In effect, $K \bullet U = K_1$ and $K \star U = K_2$. We have that $K \bullet U = K_1$ and that $K_2 \vdash U$ (primacy of new information), therefore by the principle of minimal change $Delta(K, K \bullet U) = Delta(K, K_1) \leq_K Delta(K, K_2)$. Symmetrically since $K \star U = K_2$ and $K_1 \vdash U$,by the same principle we have that $Delta(K, K \star U)$ = $Delta(K, K_2) \leq_K Delta(K, K_1)$. However, by the Cost antisymmetry property of the ordering it must be the case that $Delta(K, K_1) = Delta(K, K_2)$, which by definition of $Delta$ gives us that $K_1 = K_2$. Thus, $K \bullet U = K \star U$. Concluding there is exactly one rational update operation. ■

Lemma 6. *Given a FOL $< L, Cn >$, a set of integrity constraints C and two KBs, K, K', there is some update U such that for all $\leq$ and its associated rational update operation '$\bullet$' it holds that $K \bullet U = K'$.*

Proof:
Let $U = Delta(K, K')$. It holds that K' is a KB and by the $Delta$ definition $K' \vdash U$. So the first two properties of the rational change operation hold. In addition by lemma 4(a) it holds that $K + U = K'$. Now for a specific ordering scheme $\leq$ consider a change operation $K \bullet U = K'$. We must show that this is the rational change operation w.r.t to $\leq$, so we must prove that for any KB $K'' \vdash U$ it holds that $Delta(K, K \bullet U) \leq_K Delta(K, K'')$. We have that $K + U = K' = K \bullet U$ and lemma 4(c) tell us that $Cost(K, U) \subseteq Delta(K, K'')$.By lemma 4 $Cost(K, U) = Delta(K, K + U)$, so $Delta(K, K + U)$ = $Delta(K, K \bullet U) \subseteq Delta(K, K'')$. Lastly, by monotonicity of the ordering we have $Delta(K, K \bullet U)$

$\leq_K Delta(K, K'')$. Therefore $\bullet$ is a rational change operator.■

Lemma 6 states that, regardless of the $\leq$ defined, the user can always get the desired result (K'), starting from a KB K; an update resulting to any desired KB is guaranteed to exist. This, in turn, means that the user can always override the default behavior of the algorithm if she wants to, regardless of the order at hand and the starting KB. This is a very important property, because examples can show that no behavior is optimal (intuitively) for any type of change.

4.3 Algorithm

Consider a language $< L, Cn >$, a set of constraints C and an update-generating ordering scheme $\leq$. Our general algorithm takes as input a belief K and an update U. The first step of the algorithm is to determine whether its inputs are ok (i.e., K is a belief and U is an update). If not, return an error message and exit. Then, we must check whether K is a KB. This can be done easily by verifying that the rules are satisfied. If K is not a KB, then return K with an error message and exit. Finally, we must check that U is a feasible update.

In the following, we will assume that the first two checks have been made, so the algorithm will take for granted that K is a KB and U is an update (which may be feasible or not). We will need one special function "min" which takes as input two updates and returns the one which results in a belief closest to K (per the order $\leq_K$). We will use a special cost constant, $INFEASIBLE$, to denote infinite cost; infinite cost will be used to identify sequences of effects and side-effects that cannot possibly lead to an acceptable result (KB). The function min should return $INFEASIBLE$ iff both updates in its input are $INFEASIBLE$ (if only one is, it returns the other). If more than one updates are "preferable", one is selected arbitrarily; the actual selection makes no difference, as, due to conflict sensitivity and Cost antisymmetry, the actual cost is the same and the result of applying the two "different" updates upon K is the same as well.

Our algorithm is divided into two modules, namely the main algorithm (or general update algorithm) and the update function. The main algorithm, seen in table 4.1 takes K and U as input and returns $K' = K \bullet U$, where $\bullet$ is a rational change operation for the given language ($< L, Cn >$), constraints (C) and update-generating order ($\leq$). This algorithm is a very simple frame surrounding our major mechanism and core of our work: the update function (table 4.1). The update function is returns the $Delta(K, K + U)$ or else the $Cost(K, U \cup U')$ where U' are the "cheapest" with respect to K side-effects of U. The main algorithm simply raw applies this $Delta$ (if applicable) to the initial KB, and we will omit detailed

Table 4.1: General Update Algorithm

STEP Main1: Set $U' = \emptyset$
STEP Main2: Set $Delta = Update(U, K, U')$
STEP Main3: If $Delta =$ `INFEASIBLE`, then return K and an error message
STEP Main4: $Delta \neq$ `INFEASIBLE`, then return the belief set K+Delta

further discussion on it; note that subsequently unless otherwise stated, the term "algorithm" will refer to the $Update$ function.

$Update$ function takes as input the same inputs as the main algorithm plus two more; firstly, the set of effects and side-effects that have been already considered (ESE) this parameter is the empty set initially. It also takes the best (per $\leq_K$)result so far found (B), in order to avoid estimating more "costly" ones. B is initially set to $INFEASIBLE$, this is necessary so as for the algorithm to know that B is initially unset and therefore cannot compare a result to it; the algorithm sets the first outcome to B and subsequently comparing all the rest to B keeping the minimum each time. Function "min" represents the implementation of a specific ordering $\leq_K$. The output of $Update$ is the full set of effects and side-effects that the current thread of execution has generated.

As it can seen a specific implementation of the algorithms would be parameterizable to any update and initial KB, in fact, the framework itself is parameterizable to different orderings through different implementations of "min" function; moreover the set of rules over which our algorithms iterate corresponds to setting the framework for updating based on a different validity model. Therefore our framework is tunable in multiple ways. It supports different change strategies (through different orderings), different sets of integrity constraints as also any arbitrary set of update operations.

Update function is a recursive process and it starts by detecting whether the current process thread leads to an infeasible result (step 1); this is true iff:

(a) the remaining update operations contradict or are more expensive than the ones that have been already made, or

(b) the remaining update operations are more expensive than a solution of this specific update request already computed.

If so, the algorithm cancels this thread, as it clearly cannot lead to an acceptable (a) or profitable (b) result. Notice that a call to the update might return $INFEASIBLE$ even in the case that the update is not infeasible, but is provably more expensive than an alternative one. We use this trick and consider infinite the cost of an expensive update in order to reject it (when having a better alternative).

Table 4.2: Update Function

$STEP1$: If $U \cup ESE$ is inconsistent or $min(Cost(K, U \setminus ESE), B) == B$ return `INFEASIBLE`.

$STEP2$: If $(K \cup ESE) \vdash U$, then return $\emptyset$

$STEP3$: Take an arbitrary ground fact $P(x) \in U \setminus ESE$ such that $K \nvdash \{P(x)\}$

$STEP4$: Find a rule r, and `Select` tuple of constants $\vec{y}$ for which $\neg P(\vec{y}) \in V$ and $V \in Comp(r, \vec{y})$ and for all $V' \in Comp(r, \vec{y}), V \neq V'$ it holds that $V' \not\subseteq U \cup ESE$.

$STEP5$: If there is no such rule, then return $\{P(x)\} \cup$ `Update`$(U, K, ESE \cup \{P(x)\}, B)$.

$STEP6$: Otherwise, select (arbitrarily) one such rule, say r and for all $V' \in Comp(r, \vec{y}), V' \neq V$:

$STEP6.1$: If $B ==$ `INFEASIBLE` then $B :=$ `Update`$(U \cup V', K, ESE, B)\}$

$STEP6.2$: Else if $B \neq$ `INFEASIBLE` then $B :=$ $min(B, \{$`Update`$(U \cup V', K, ESE, B)\})$

$STEP7$: Return B

In step 2, the algorithm detects whether the current thread of execution needs to stop; this is true iff the remaining operations (parameter U) are implied by the initial KB or ESE, so no further operations need to be executed.

If both these tests fail, then more operations need to be executed. A particular effect from the "to-do list" (i.e., U) is selected (step 3) and verified against the rules (step 4). If no rule is violated by the execution of this effect (step 5), then the effect is added to the "Delta" of this particular thread of execution (to be returned to the caller), the effect is added in the "done list" ESE) and the $Update$ function is called recursively to consider new parts of the update. Notice that in step 5 it is assumed that $X \cup INFEASIBLE$ = $INFEASIBLE$ for any set X.

On the other hand, if there is some rule that is violated, then we identify each possible way to restore the rule violation, and spawn a new thread of recursive execution for each of the different alternative side-effects (step 6). This is done by adding each candidate side-effect in the "to-do list" (U) and calling the $Update$ function recursively once for each such candidate; we do this sequentially and we compare a returned threads with the minimal so-far (per $<$), that minimal one is selected and returned to the caller (step 7). Notice again that if one of the threads returns the special value $INFEASIBLE$, then this thread is by default considered to have maximal "cost", and is ignored in taking the minimum; if all threads return $INFEASIBLE$, then the same value is returned to the caller.

4.3.1 Termination and Efficiency

Notice that in certain cases, there may be an infinite number of V' that satisfy the conditions of step 6 as the existential quantifier in the DEDs might result in a $Comp$ set of infinite width. Consider for example $\forall P(x) \rightarrow \exists y_i Q(y_i)$. This turns out to the set $Comp(c,u) = \{\{\neg P(x)\}, \{Q(y_1)\}, \{Q(y_2)\}, ...\}$. Since it is the infinite number of constants symbols in our language which causes this problem, we provided a function, namely $Select$ in STEP4 of our framework, which could be used to overcome it. When adjusting our framework to a particular setting, one could choose to materialize $Select$ in such a way that would select provably less "costly" components of $Comp(c,u)$ bounding the latter in a finite set. Such an example appears in chapter 5 where we tune our framework for a specific language (the RDF/S). We thereby, based slightly on a property of our ordering, prove that for our particular rules it is pointless for the algorithm to examine (in STEP4) all the constants of our language, as we know the best solution is among a priori known finite sets of constants.

Except the above case, (where the determination of the minimum would take an infinite amount of time) our algorithm suffers from one more problem related to termination and this is also related to the infinite number of constants in our language; when bounding the variables of a $Comp$ family to $Comp(c,\vec{x})$ instantiations, any free variables in $\vec{x}$ cause infinite such instantiations. Consider for example the rule with components $Comp(c,\vec{x}) = \{\{\neg P(u)\}, \{Q(y)\}\}$. Here $\vec{x} = (u,y)$; so, when we have $\{P(z)\}$ as an update the rules violated are all the instances of the above rule family $\forall\vec{(x)}, i.e., \forall y(z,y)$; which are infinite. This fact would result to an "infinite depth" in our algorithm, i.e, an infinite number of recursive calls to $Update$ for all different rule instances. Again with the $Select$ function of STEP4, any implementation of our framework, could choose to examine those rule instances which are *actually* violated. More discussion on this issue will take place also in chapter 5, where we bound the set of selectable constants of STEP4 to a finite size for our particular setting.

Lemma 7. *Given an update U, a KB K, and $ESE = \emptyset$, under a* `finite set of constants` *in the assumed language L, the algorithm of table 4.2 terminates.*

Proof:

Consider a random run (say n^{th}) of the update function; it returns infeasible, terminates or processes to the third step. Our goal towards the proof is to point out that as recursive calls are executed the predicates allowable for selection at STEP3 will becoming lesser. At STEP3 a ground fact, say $P(\vec{x})$, is selected from

$U \setminus ESE$.

If $P(x)$ does not violate any rule, we proceed to STEP5 where the predicate is added to ESE and Update function is re-called. From this point and on the same predicate cannot be selected at STEP3 of any subsequent calls.

On the other hand, if there is a rule instance violated this means that there is a $Comp(c, \vec{y})$, for which $\neg P(\vec{x}) \in V$ and $V \in Comp(r, \vec{y})$ and for all $V' \in Comp(r, \vec{y}), V \neq V'$ it holds that $V' \nsubseteq U \cup ESE$. Now at STEP6 all these V' will be added to update, opening the respective branches of execution and comparing them upon return. Note that under a finite number of constants there is a finite number of V' that this particular $Comp(c, \vec{y})$ contains.

Let us examine one of these finite branches; the specific V' which restores the aforementioned rule instance is now a part of U.

Therefore if this branch does not return INFEASIBLE, there is a particular Update run (in one of subsequent calls), say $m^{th}, where m > n$, in which $P(\vec{x})$ will be selected in STEP3. This is certain as if not, this would mean that $P(\vec{x}) \in ESE$ in STEP3, which would in turn mean that $P(\vec{x})$ was selected in a previous call (after n^{th}), did not violate any rules and made it to ESE at STEP5. However we defined the m^{th} call to be this one.

When $P(\vec{x})$ is selected, during m^{th} call, it is checked against rule instances it might violate. Note that it cannot violate $Comp(c, \vec{y})$ for the aforementioned $\vec{y}$ again, as V' is a part of U, thus $V' \subseteq U \cup ESE$, spoiling the last condition of STEP4.

Therefore if $P(\vec{x})$ violates any rule, this would be a new rule instance, and this will be restored by adding one of its components in the update in STEP6. Based on the inclusion of a solution to a specific rule instance, we can see that the same ground fact will never break again the same rule instances in subsequent calls.

A key observation at this point is that under a finite number of constants, there is a finite number of rule instances (i.e., under a finite number of tuples of constants $\vec{a}$ there is a finite number of $Comp(c, \vec{a})$). So, as the algorithm continues (and continuing to assume that it doesn't return infeasible), there will be a call, where ground fact $P(\vec{x})$ will be selected and as not violating any rule instances and will be added to ESE to start a new branch of calls at STEP5.

From that point and on any runs of the algorithm will contain $P(\vec{x})$ in the ESE, not allowing it to be selected again.

Concluding as the algorithm runs, and if not all branches return INFEASIBLE, the allowable predicates to be selected will becoming lesser. At some point, all the predicates of U, will be unaible to be selected, by the condition of STEP3, which means the the corresponding runs will return at STEP2. Thus, under a finite number of constants, the algorithm terminates. ■

4.3.2 Complexity

Clearly as the algorithm is a brute force algorithm its complexity is exponential. Even with the pruning done by the maintenance of the best result so far (i.e, variable B of our algorithm), the algorithm in the worst case will find the best solution last. For each predicate of update it checks all instances of the rules (each rule instantiated for every possible variable combination) and when found one rule violated it opens a new branch for each part of it. The new branch re-examines all rule instances and could examine also again the same rule this branch was opened to satisfy; although for a different predicate. Therefore a detailed complexity analysis of the algorithm depends on the particular setting this is instantiated; on the language, and integrity constraints used and could also be affected by the implementation of the $Select$ and "min"(ordering) functions.

This framework is very generic and applicable to every possible FOL, set of constraints and update-generating ordering scheme. When tuning it one should take into account the peculiarities of the particular setting in which this algorithm is applied and prune a part of the recursive calls required; this would guarantee efficiency and termination.

4.3.3 Results

In all the following lemmata and propositions, we consider a particular run of the above algorithm, with inputs K, U, such that K is a KB.

We also assume that the very first call to Update (that initiates the recursion) is of the form $Update(U, K, \emptyset)$ and during that same step we verify that U is consistent. If it is not, the algorithm will return $INFEASIBLE$ immediately. Therefore our findings below take for granted that during execution and after the first step the updates are consistent. Lastly, note that we don't consider any special implementation for the $Select$ function of STEP4; we assume it returns any constant of our language arbitrarily.

Now take any particular sequence of recursive calls to $Update$. We will assume that these calls are of the form:
$Update(U_1, K, ESE_1, B_1)$,
$Update(U_2, K, ESE_2, B_2)$,
...
By our assumptions: $U_1 = U$, $ESE1 = \emptyset$, $B_1 = INFEASIBLE$. All the lemmata below are regarding to the $Update$ function.

Lemma 8. *Consider some terminating sequence of recursive calls of length n that does not return $INFEASIBLE$. Then, $P(x) \in U_i$ for some $i = 1, 2, ..., n$ implies $K + ESE_n \vdash \{P(x)\}$.*

Proof:
Take some $P(x) \in U_i$. Obviously, $U_i \subseteq U_n \subseteq$ and $K \cup ESE_n \vdash U_n \vdash U_i$, by STEP2, since the sequence terminates and does not return $INFEASIBLE$, so $K \cup ESE_n \vdash \{P(x)\}$. If $\neg P(x) \in ESE_n$ then $U_n \cup ESE_n$ inconsistent, so the sequence would return $INFEASIBLE$, by STEP1; a contradiction. So $\neg P(x) \notin ESE_n$. Thus, if $P(x) \in L^+$, then $P(x) \in K + ESE_n$ by the definition of '+' so $K + ESE_n \vdash \{P(x)\}$. If $P(x) \in L^-$, then $K \cup ESE_n \nvdash \{\neg P(x)\}$, implies that $\neg P(x) \notin K + ESE_n$, so $K + ESE_n \vdash \{P(x)\}$.■

Lemma 9. *Consider some terminating sequence of recursive calls of length n that does not return $INFEASIBLE$. Then, $K + ESE_n$ is a KB.*

Proof:
Set $K' = K + ESE_n$. Suppose, for the sake of contradiction, that there is some constraint $c \in C$ that is not satisfied by K'. Then, there is some constant (or tuple of constants) x, such that for all $V_0 \in Comp(c, x)$: $K' \nvdash V_0$. For this particular pair (c, x), since K is a KB, there is some $V \in Comp(c, x)$ such that $K \vdash V$. Therefore, for this V it holds that $K \vdash V$, $K' \nvdash V$, so there is some ground predicate, say $Q(y) \in V$ for which $K \vdash \{Q(y)\}$, $K' \nvdash \{Q(y)\}$. If $Q(y) \in L^+$ then $Q(y) \in K$, $Q(y) \nvdash K'$=$K + ESE_n$, so $\neg Q(y) \in ESE_n$. If $Q(y) \in L^-$. then $\neg Q(y) \notin K$, $\neg Q(y) \in K'$=$K + ESE_n$, so $\neg Q(y) \in ESE_n$. In either case, $\neg Q(y) \in ESE_n$, so $\neg Q(y)$ was added in STEP5 in some call, say the j^{th} call.

Thus, in the j^{th} call it holds that $\neg Q(y) \in U_j \setminus ESE_j)$ and $K \nvdash \{\neg Q(y)\}$ Furthermore, as we said $Q(y) \in V$, $V \in Comp(c, x)$, so, since STEP5 was executed, for the particular constraint c and constants x it holds that there is some $V' \in Comp(c, x)$, $V \neq V'$ for which $V' \subseteq U_j \cup ESE_j$. Take any $P(x) \in V'$. If $P(x) \notin U_j$ then $P(x) \in ESE_j$, so $P(x)$ was added in a previous call, say the i^{th} call, so in the i^{th} call the following was true: $P(x) \in U_i \setminus ESE_i$ and $K \nvdash \{P(x)\}$. Thus, $P(x) \in U_i \subseteq U_j$, a contradiction. Thus for all $P(x) \in V'$ it holds that $P(x) \in U_j$, so by the previous lemma, $K' \vdash \{P(x)\}$. Thus, $K' \vdash V'$, which contradicts our initial hypothesis.■

Lemma 10. *Consider some terminating sequence of recursive calls of length n that does not return $INFEASIBLE$. Then ESE_n=$(\cup U_i) \setminus \{P(x) | K \vdash \{P(x)\}\}$=$U_n \setminus \{P(x) | K \vdash \{P(x)\}\}$.*

Proof:
Firstly, notice that by steps 3, 4, 5, once a ground fact is added to U, it cannot later be removed from U (even though it can be re-added). Therefore $(\cup U_i)$=U_n. Consider some $P(x) \in ESE_n$. By steps 3, 5 it is obvious that once a ground fact is added to ESE, it cannot be later removed from ESE, nor re-added to ESE.

We conclude that there is some j such that $P(x)$ was added to ESE in the j^{th} call, so in that call it was the case that $P(x) \in U_j \setminus \{P(x)|K \vdash \{P(x)\}\} \subseteq (\cup U_i) \setminus \{P(x)|K \vdash \{P(x)\}\}$. Thus: $ESE_n \subseteq (\cup U_i) \setminus \{P(x)|K \vdash \{P(x)\}\}$. For the opposite inclusion, consider any $P(x) \in U_n \setminus \{P(x)|K \vdash \{P(x)\}\}$ then $P(x) \in U_n$ and $K \nvdash \{P(x)\}$. Since the sequence terminates after n calls and does not return $INFEASIBLE$, by lemma 8 it follows that $K + ESE_n \vdash \{P(x)\}$. Since K is a KB and ESE_n an update it follows. by definition 2, that $P(x) \in K \cup ESE_n$ and $\neg P(x) \notin ESE_n$. Since $K \nvdash \{P(x)\}$, $P(x) \notin K$ but $P(x) \in K \cup ESE_n$ so, $P(x) \in ESE_n$ and the proof is complete.■

Lemma 11. *Consider any KB K and any update U. Suppose that there is some KB K', such that $K' \vdash U$. Suppose also that, by running the above algorithm with inputs K, U, the algorithm terminates. Then, there is some terminating sequence of recursive calls of finite length (say n) that does not return $INFEASIBLE$, for which $Delta(K, K + ESE_n) \leq_K Delta(K, K')$.*

Proof:
By lemma 4 it holds that $Cost(K, ESE_n) = Delta(K, K+ESE_n)$ and $Cost(K, U) \subseteq Delta(K, K')$. Take any $P(x) \in Delta(K, K')$. If there is some constraint $c \in C$, some (tuple of) constants y and some $V \in Comp(c, y)$ such that $\neg P(x) \in Comp(c, y)$, then, since K' is a KB, there is some other $V' \in Comp(c, y)$, $V' \neq V$ such that $K' \vdash V'$.

We will denote by S_0 the union of all such V' for all $P(x) \in Delta(K, K')$ and all constraints and (tuples of) constants for which the above condition is true. We set $S = Delta(K, K') \cup S_0$. Obviously, $K' \vdash Delta(K, K')$ and $K' \vdash S_0$, so $K' \vdash S$. Now consider the sequence of recursive calls in which, the V' followed (chosen) in step 6 is one of the V' that comprise S, whenever possible. We will show that for this sequence, $ESE_n \subseteq S$.

By lemma 10 $ESE_n = (\cup U_i)\ \{P(x)|K \vdash \{P(x)\}\}$, so it suffices to show that for all $i = 1, 2, ..., n$ it holds that $U_i \setminus \{P(x)|K \vdash \{P(x)\}\} \subseteq S$. We will use recursion to show this. For $i = 1$, $U_1 \setminus \{P(x)|K \vdash \{P(x)\}\}$ = $U \setminus \{P(x)|K \vdash \{P(x)\}\} \subseteq Cost(K, U) \subseteq Delta(K, K') \subseteq S$ by lemma 4. Suppose that this holds for $1, 2, ..., i-1$. We will show that it holds for i too. We only need to check the ground facts in $U_i \setminus (U_{i-1} \cup K)$. By step 6, it follows that the ground facts that need to be checked belong all to some set V', which occurred in step 4. By the phrasing of step 4, the definition of S and the hypothesis of the recursion, it follows that $U_i \setminus (U_{i-1} \cup K) \subseteq S_0$, so $U_i \setminus K \subseteq S$. We conclude that $ESE_n \subseteq S$.

By the monotonicity of $\leq_K$, it follows that $Cost(K, ESE_n) \leq_K Cost(K, S)$. Now take any $P(x) \in Cost(K, S) \setminus Delta(K, K')$. Since $P(x) \in Cost(K, S)$, it follows that $P(x) \in S$ and $K \nvdash \{P(x)\}$. But $K' \vdash S$, so $K' \vdash \{P(x)\}$ and $P(x) \nvdash Delta(K, K')$, so $K \vdash \{P(x)\}$, a contradiction. We conclude that

$Cost(K,S) \setminus Delta(K,K') = \emptyset$, so $Cost(K,S) \subseteq Delta(K,K')$. By the monotonicity of $\leq_K$, it follows that $Cost(K,S) \leq_K Delta(K,K')$. Thus, using transitivity of $\leq_K$: $Delta(K, K + ESE_n) = Cost(K, ESE_n) \leq_K Cost(K,S) \leq_K Delta(K,K')$ and the proof is complete.■

Prop. 1. *Suppose that the Main algorithm of table 4.1 terminates and set, for every pair $K \subseteq L^+$, $U \subseteq L^0$: $K \bullet U = K'$, where K' is the belief returned by the algorithm. Then $\bullet$ is a rational update operation with respect to the given $\leq$.*

Proof:
If K is not a KB then the main algorithm would return K. Suppose that K is a KB, but the update is infeasible. Suppose also that the update function algorithm does not return $INFEASIBLE$. By steps 5, 6 and 7 of the $Update$ function it is obvious that the returned value $Delta = ESE_n$ for some sequence of recursive calls. By lemmata 8 and 9 it follows that $K' = K + Delta = K + ESE_n$ is a KB and $K' \vdash U$, so U is feasible (a contradiction). So the $Update$ function will return $INFEASIBLE$ and the output of the main algorithm will be K. These arguments take care of the limit cases.

Suppose now that K is a KB, and that the update is feasible. Then, there is some $K_0 \subseteq L^+$, such that K_0 is a KB, $K_0 \vdash U$. By lemma 11, there is some sequence of recursive calls (of the update function) that does not return $INFEASIBLE$, for which $Delta(K, K + ESE_n) \leq_K Delta(K, K_0)$. Suppose that the prevailing (minimal) sequence of calls that returns the result has length m. Then, by steps 5,6 and 7 it is obvious that the returned value is equal to ESE_m. By lemmata 8 and 9, no matter which sequence of recursive calls is selected from "min" for the final output of Update, the final result $K' = K + Delta = K + ESE_m$ is a KB and $K' \vdash U$. Now take any KB $K'' \vdash U$; by lemma 11, there is some sequence of recursive calls for which $Delta(K, K + ESE_k) \leq_K Delta(K, K'')$. Since the prevailing (minimal) sequence of calls returns ESE_m by step 6 and lemma 4 it obviously holds that: $Delta(K, K + ESE_m) = Cost(K, ESE_m) \leq_K Cost(K, ESE_k) = Delta(K, K + ESE_k) \leq_K Delta(K, K'')$. Therefore $Delta(K,K') \leq_K Delta(K,K'')$. We conclude that $\bullet$ is a rational update operation.■

Proposition 1 guarantees that the algorithm returns the "proper" result of the update, with the assumption that the algorithm terminates. Therefore, for any particular application of the above model (e.g., RDF) we need to do the following:

- Specify the model (i.e., FOL) that will be used; this process in fact refers to the definition of a finite number of predicates, as well as the definition of variables and constant symbols.

- Determine the integrity constraints of the model and formally describe them. (In the current setting this is done in the form of DEDs, as they are expressive enough for our case. However, other rule forms could be analogously embedded in our framework.)
- Define a particular update-generating order, that looks most "rational" intuitively.
- Based on the above constructs, prove results that allow one to prune the tree explored by the above algorithm to a finite size so that the algorithm terminates. One could just restrain the number of constants; implement $Select$ of step 4 of the $Update$ function in such a way that this function would choose a constant out of a finite set.
- Also, provide heuristics and optimization techniques that are based on the peculiarities of the model, integrity constraints and update-generating order under question to speed up the algorithm as much as possible (computational complexity).

The results presented so far show that the above process will lead to the definition of a rational change operation.

Chapter 5

Application to RDF/S

Change in all things is sweet.

Aristotle

The particular proposals described in this section for the language (predicates and rules) and the ordering are just parameters of our approach. Different parameters would result to different update operations, but the general approach (and algorithm) can be still applied, so long as the requirements set for the rules and ordering are respected. For the implementation described in the following, the parameters are fixed to be the ones that were defined here, as this particular parameterization is adequate for the purposes of the KP-Lab (e-learning) and CASPAR (digital preservation) projects, which partially supported this thesis. Mark that in this section, KBs will also be called RDF Graphs.

5.1 General Model Description

Our first task towards the development of our model is to describe RDF statements in terms of FOL predicates. This representation is shown in table 5.1. Note that the actual mapping of these statements to RDF triples is shown in table 5.2. Notice that several ground facts might be represented with the same triple. However there is a clear distinction of them based on the context. Consider, for example ground fact PI(x,y,P) and PLI(x,y,P); the same triple represents them and in order to be distinguished we should consider whether P is a common property a or a data-type one (having *Literal* as range). More discussion on the triple-view of our ontologies is postponed until chapter 6; throughout the rest of the current chapter we will be using the FOL representation.

Table 5.1: Representation of RDF facts using FOL predicates

Predicates	Intuitive meaning
$CS(C)$	The name C exists in the RDF graph and it is a class
$PS(P)$	The name P exists in the RDF graph and it is a property
$CI(x)$	The name x exists in the RDF graph and it is a class instance
$MCS(C)$	The name C exists in the RDF graph and it is a meta class
$PL(P)$	The name P exists in the RDF graph and it is a datatype property (has 'rdfs:Literal' as range)
$Domain(C, P)$	The domain of P is C (denotes the domain of a property)
$DTP\_Domain(C, P)$	The domain of P is C (denotes the domain of a datatype property)
$Range(C, P)$	The range of P is C (denotes the range of a property)
$C\_IsA(C_1, C_2)$	Class C_1 is a direct or indirect subclass of class C_2
$M\_IsA(M_1, M_2)$	Metaclass M_1 is a direct or indirect subclass of metaclass M_2
$P\_IsA(P_1, P_2)$	Property P_1 is a direct or indirect subproperty of property P_2
$DTP\_IsA(P_1, P_2)$	Datatype Property P_1 is a direct or indirect subproperty of Datatype property P_2
$C\_Inst(x, C)$	Instance x is a direct or indirect class instance of C
$M\_Inst(A, MA)$	Class A is a direct or indirect instance of metaclass MA
$PI(x, y, P)$	The pair(x,y) is a direct or indirect instantiation of property P
$DTPI(x, s, P)$	The pair(x,s) (s is a literal) is related a direct or indirect instantiation of datatype property P

Notice that mapping each statement of RDF to a FOL predicate (as seen in the Table 5.1) is not representing RDF constructs in the standard way, but is an alternative representation; this way, a class IsA between A and B, for example, is mapped to the predicate: $C\_IsA(A, B)$, while a triple denoting that the domain of a property, say P, is C, is denoted by $Domain(P, C)$. Note that the standard alternative mapping (e.g., for IsA: $\forall$x$A(x) \rightarrow B(x)$) does not allow us to map assertions of the form "C is a class", and, consequently, does not allow us to handle operations like the addition or removal of a class, property, or instance (see [21] for more details on this issue). Also note that the same representation pattern can be used for other declarative languages as well [21].

5.1.1 Constants and variables

The set of constants can be taken to be any countably infinite set. For ease of use, it will be assumed to contain all the strings of the English language up to a certain length (say N) plus a countably infinite set of auxiliary constants $(c_1, c_2, ...)$, plus the special constant $\top$, which will be used to denote the top class, as the version of RDF/S we are modeling is the one defined in [51] and it contains $\top$ which is a superclass of all the classes in an ontology. Note that only classes and not

Table 5.2: FOL Ground Facts to RDF Triples

	FOL	Triple subject	Triple Predicate	Triple object
1	CS(A)	A	rdf:type	rdfs:Class
2	PS(P)	P	rdf:type	rdfs:Property
3	CI(x)	x	rdf:type	rdfs:Resource
4	MCS(MA) (meta-class)	MA AND MA	rdfs:subClassOf rdf:type	rdfs:Class rdfs:Class
5	PL(P)	P AND P	rdf:type rdf:range	rdfs:Property rdfs:Literal
6	Domain(P,A)	P	rdfs:domain	A
7	DTP_Domain(P,A)	P	rdfs:domain	A
8	Range(P,A)	P	rdfs:range	A
9	C_IsA(A,B)	A	rdfs:subClassOf	B
10	M_IsA(A,B)	A	rdfs:subClassOf	B
11	P_IsA(P,Q)	P	rdfs:subPropertyOf	Q
12	DTP_IsA(P,Q)	P	rdfs:subPropertyOf	Q
13	C_Inst(x,A)	x	rdf:type	A
14	M_Inst(A,MA) (meta-class instantiation link)	A	rdf:type	MA
15	PI(x,y,P) (property instance)	x	P	y
16	PLI(x,y,P) (property instance of a PL)	x	P	y

metaclasses are related to the top class. Thus, the set of constants is the following set: $\{s|s : stringoflengthn, 1 \leq n \leq N\} \cup \{c_1, c_2, ..\} \cup \{\top\}$. The set of variables can be any countably infinite set, for example $u_1, u_2, ...$

Notice that our formal framework defined earlier needs a countably infinite set for our constants. Taking all the strings of the English language up to any length would result to an uncountably infinite set (isomorphic to the set of real numbers). For this reason, we constraint the length of the strings to be up to a finite length. This is not a real restriction for practical purposes, as N can be taken to be arbitrarily large. For example set $N = 10^8$; the storage of such a long name requires more bits than any computer could store anyway.

On the other hand, our framework doesn't want the set of constants to be a finite set, because, for some updates, we may need to introduce new constant names in our KB; we need to be able to do that in any circumstance and this can only be guaranteed if we have an infinite number of constant names (this is true in theory; in practice we will prove that a finite size is enough). Thus, we introduce an infinite sequence of auxiliary constants (c_i) to serve this purpose. Notice that in the previous section we stated that the infinite constants number is a problem for algorithm's termination, however tuning appropriately the $Select$ function is overcoming this

problem. We later elaborate on this issue and exhibit how the $Select$ function can always face a finite number of constants even though in principle these are infinite.

5.2 Validity Rules

Now we should determine the integrity constraints of our model. It should be emphasized that our framework is very general and can be used for many different representation languages, by carefully defining the appropriate predicates and validity rules. Here, we will use (for presentation and implementation purposes) a particular, alternative RDF semantics [51]. For example, C_IsA, P_IsA, C_Inst are assumed to be transitive (also PI is transitive) and irreflexive. In effect, we record direct and indirect IsAs, but we don"t record self-loops (e.g., $C\_IsA(x, x)$ is not recorded).

The current RDF concretization but is not in any way an inherent restriction of the framework. In particular, our algorithm is also applicable for the standard RDF semantics[1], by restricting the validity rules accordingly. In effect, any graph constructed with respect to the semantics used here is a valid RDF graph (as we are considering a sub-language of RDF). Also notice that our model considers even the implicit information in the KB (due to CWA). The aforementioned integrity constraints are shown in Table 5.3, which exposes all the rules of our language together with their Intuitive meaning. In this table, Σ denotes the set of constants in our language. Note also that we assume that every Datatype property has the standard class "rdfs:Literal" as range, and in the instance level such a property instance has any value as range. We don't record this, as from the evolution point of view we do not care about the exact values. Therefore each time we talk about datatype properties the above range information will be missing (but implied). This form of the integrity constraints gives immediately the families $Comp(c, x)$ for any constraint c and constant x. These Component sets are shown in Table 5.4.

5.3 Ordering

Our RDF KB update framework is parameterizable with respect to the ordering that chooses the minimal update result set. Any ordering could embedded neatly in our system. Before defining the update-generating order, we will need some supportive definitions.

Consider some (positive) predicate P of arity n of the FOL language and an update U. We set: $U^P = \{P(x_1, ..., x_n)$, where $P(x_1, ..., x_n) \in U$ and $x_1, ..., x_n$ are

[1] http://www.w3.org/TR/rdf-mt/

Table 5.3: Validity Rules

Rule ID/Name	Integrity Constraint	Intuitive Meaning
R1 Typing	$\forall x \in \Sigma : ((CS(x) \vee CI(x)) \wedge (CS(x) \vee PS(x)) \wedge (CI(x) \vee PS(x)) \wedge (CS(x) \vee MCS(x)) \wedge (PS(x) \vee MCS(x)) \wedge (CI(x) \vee MCS(x)) \wedge (CS(x) \vee PL(x)) \wedge (PS(x) \vee PL(x)) \wedge (CI(x) \vee PL(x)) \wedge (PL(x) \vee MCS(x)))$	The names used for all facts of our language are mutually disjoint
R2 Top Node Typing	$CS(\top)$	The Top Node is a class
R3 Domain Applicability	$\forall x, y \in \Sigma$: $Domain(x, y) \rightarrow PS(x) \wedge CS(y)$	Domain applies to properties; the domain of a property is a class
R4 Range Applicability	$\forall x, y \in \Sigma$: $Range(x, y) \rightarrow PS(x) \wedge CS(y)$	Range applies to properties; the range of a property is a class
R5 C_IsA Applicability	$\forall x, y \in \Sigma$: $C\_IsA(x, y) \rightarrow CS(x) \wedge CS(y)$	Class IsA applies between classes
R6 P_IsA Applicability	$\forall x, y \in \Sigma$: $P\_IsA(x, y) \rightarrow PS(x) \wedge PS(y)$	Property IsA applies between properties
R7 C_Inst Applicability	$\forall x, y \in \Sigma$: $C\_Inst(x, y) \rightarrow CI(x) \wedge CS(y)$	Class Instanceof applies between a class instance and a class
R8 PI Applicability	$\forall x, y, z \in \Sigma$: $PI(x, y, z) \rightarrow CI(x) \wedge CI(y) \wedge PS(z)$	Property Instanceof applies between a pair of class instances and a Property
R9 Domain is unique	$\forall x, y, z \in \Sigma$: $Domain(x, y) \rightarrow \neg Domain(x, z) \vee (x == z)$	The domain of a property is unique
R10 Range is unique	$\forall x, y, z \in \Sigma$: $Range(x, y) \rightarrow \neg Range(x, z) \vee (x == z)$	The range of a property is unique
R11 Domain and Range exists	$\forall x \in \Sigma, \exists y, z \in \Sigma$: $PS(x) \rightarrow Domain(x, z) \wedge Range(x, y)$	Each property has a domain and a range
R12 C_IsA Transitivity	$\forall x, y, z \in \Sigma$: $C\_IsA(x, y) \wedge C\_IsA(y, z) \rightarrow C\_IsA(x, z)$	Class IsA is Transitive
R13 C_IsA Irreflexivity	$\forall x, y, \in \Sigma$: $C\_IsA(x, y) \rightarrow \neg C\_IsA(y, x)$	Class IsA is Irreflexive
R14 P_IsA Transitivity	$\forall x, y, z \in \Sigma$: $P\_IsA(x, y) \wedge P\_IsA(y, z) \rightarrow P\_IsA(x, z)$	Property IsA is Transitive
R15 P_IsA Irreflexivity	$\forall x, y \in \Sigma$: $P\_IsA(x, y) \rightarrow \neg P\_IsA(y, x)$	Property IsA is Irreflexive
R16 Determining C_Inst	$\forall x, y, z \in \Sigma : C\_Inst(x, y) \wedge C\_IsA(y, z) \rightarrow C\_Inst(x, z)$	Class instance propagation
R17 Determining PI	$\forall x, y, z, w \in \Sigma : PI(x, y, z) \wedge P\_IsA(z, w) \rightarrow PI(x, y, w)$	Property instance propagation
R18 Property IsAs and Domain	$\forall x, y, z, w \in \Sigma : P\_IsA(x, y) \wedge Domain(x, z) \wedge Domain(y, w) \rightarrow C\_IsA(z, w) \vee (z = w)$	IsAs between properties reflect in their domains
R19 Property IsAs and Range	$\forall x, y, z, w \in \Sigma : P\_IsA(x, y) \wedge Range(x, z) \wedge Range(y, w) \rightarrow C\_IsA(z, w) \vee (z = w)$	IsAs between properties reflect in their ranges
R20 Property Instanceof and Domain	$\forall x, y, z, w \in \Sigma : PI(x, y, z) \wedge Domain(z, w) \rightarrow C\_Inst(x, w)$	Instanceof between properties reflect in their sources/domains
R21 Property Instanceof and Range	$\forall x, y, z, w \in \Sigma : PI(x, y, z) \wedge Range(z, w) \rightarrow C\_Inst(y, w)$	Instanceof between properties reflect in their targets/ranges

R22 Class IsAs and the Top Node	$\forall x \in \Sigma : CS(x) \rightarrow C\_IsA(x, \top)$	All classes are indirect subclasses of the top node
R23 Class Instanceof and the Top Node	$\forall x \in \Sigma : CI(x) \rightarrow C\_Inst(x, \top)$	All class instances are indirect instances of the top node
R24 Metaclass IsA Applicability	$\forall x, y \in \Sigma$: $M\_IsA(x, y) \rightarrow MCS(x) \wedge MCS(y)$	Metaclass IsA applies between metaclasses
R25 M_Inst Applicability	$\forall x, y \in \Sigma$: $M\_Inst(x, y) \rightarrow CS(x) \wedge MCS(y)$	Metaclass Instanceof applies between a class and a metaclass
R26 Metaclass IsA Transitivity	$\forall x, y, z \in \Sigma$: $M\_IsA(x, y) \wedge M\_IsA(y, z) \rightarrow M\_IsA(x, z)$	Metaclass IsA is transitive
R27 Determining M_Inst	$\forall x, y, z \in \Sigma : M\_Inst(x, y) \wedge M\_IsA(y, z) \rightarrow M\_Inst(x, z)$	Metaclass instance propagation
R28 M_IsA Irreflexivity	$\forall x, y, \in \Sigma$: $M\_IsA(x, y) \rightarrow \neg M\_IsA(y, x)$	Metaclass IsA is Irreflexive
R29 DTP_Domain Applicability	$\forall x, y \in \Sigma$: $DTP\_Domain(x, y) \rightarrow PL(x) \wedge CS(y)$	DTP_Domain applies to datatype properties; the domain of a datatype property is a class
R30 DTP_IsA Applicability	$\forall x, y \in \Sigma$: $DTP\_IsA(x, y) \rightarrow PL(x) \wedge PL(y)$	DataType Property IsA applies between datatype properties
R31 DTPI Applicability	$\forall x, y, P \in \Sigma$: $DTPI(x, y, P) \rightarrow CI(x) \wedge PL(P)$	Datatype Property Instanceof applies to a class instance and a datatype Property
R32 DTP_Domain is unique	$\forall x, y, z \in \Sigma$: $DTP\_Domain(x, y) \rightarrow \neg DTP\_Domain(x, z) \vee (y = z)$	The domain of a datatype property is unique
R33 DTP_Domain exists	$\forall x \in \Sigma \exists z \in \Sigma$: $PL(x) \rightarrow DTP\_Domain(x, z)$	Each datatype property has a domain
R34 DTP_IsA Transitivity	$\forall x, y, z \in \Sigma$: $DTP\_IsA(x, y) \wedge DTP\_IsA(y, z) \rightarrow DTP\_IsA(x, z)$	DataType Property IsA is Transitive
R35 DTP_IsA Irreflexivity	$\forall x, y \in \Sigma$: $DTP\_IsA(x, y) \rightarrow \neg DTP\_IsA(y, x)$	DataType Property IsA is Irreflexive
R36 Determining DTPI	$\forall x, y, p, q \in \Sigma : DTPI(x, y, p) \wedge DTP\_IsA(p, q) \rightarrow DTPI(x, y, q)$	Datatype Property instance propagation
R37 DataType Property IsAs and DTP_Domain	$\forall x, y, x', y' \in \Sigma : DTP\_IsA(x, y) \wedge DTP\_Domain(x, x') \wedge DTP\_Domain(y, y') \rightarrow C\_IsA(x', y') \vee (x' = y')$	IsAs between datatype properties reflect in their domains
R38 DataType Property Instance and DTP_Domain	$\forall x, y, x', y' \in \Sigma : DTPI(x, y, x') \wedge DTP\_Domain(x', y') \rightarrow C\_Inst(x, y')$	Instanceof between datatype properties reflect in their domains.

Table 5.4: Components of the Rules

Rule ID/Name	Integrity Constraint in $Comp(c, u)$ format
R1 Typing	$\forall x \in \Sigma$: $Comp(R1.0, u) = \{\{\neg CS(u)\}, \{\neg CI(u)\}\}$ $Comp(R1.1, u) = \{\{\neg CS(u)\}, \{\neg PS(u)\}\}$ $Comp(R1.2, u) = \{\{\neg CI(u)\}, \{\neg PS(u)\}\}$ $Comp(R1.3, u) = \{\{\neg CS(u)\}, \{\neg MCS(u)\}\}$ $Comp(R1.4, u) = \{\{\neg PS(u)\}, \{\neg MCS(u)\}\}$ $Comp(R1.5, u) = \{\{\neg CI(u)\}, \{\neg MCS(u)\}\}$ $Comp(R1.6, u) = \{\{\neg CS(u)\}, \{\neg PL(u)\}\}$ $Comp(R1.7, u) = \{\{\neg PS(u)\}, \{\neg PL(u)\}\}$ $Comp(R1.8, u) = \{\{\neg CI(u)\}, \{\neg PL(u)\}\}$ $Comp(R1.9, u) = \{\{\neg PL(u)\}, \{\neg MCS(u)\}\}$
R2 Top Node Typing	$Comp(R2, u) = CS(\top)$
R3 Domain Applicability	$\forall x, y \in \Sigma$: $Comp(R4, u) = \{\{\neg Domain(x, y)\}, \{PS(x), CS(y)\}\}$
R4 Range Applicability	$\forall x, y \in \Sigma$: $Comp(R3, u) = \{\{\neg Range(x, y)\}, \{PS(x), CS(y)\}\}$
R5 C_IsA Applicability	$\forall x, y \in \Sigma$: $Comp(R5, u) = \{\{\neg C\_IsA(x, y)\}, \{CS(x), CS(y)\}\}$
R6 P_IsA Applicability	$\forall x, y \in \Sigma$: $Comp(R6, u) = \{\{\neg P\_IsA(x, y)\}, \{PS(x), PS(y)\}\}$
R7 C_Inst Applicability	$\forall x, y \in \Sigma$: $Comp(R7, u) = \{\{\neg C\_Inst(x, y)\}, \{CI(x), CS(y)\}\}$
R8 PI Applicability	$\forall x, y, P \in \Sigma$: $Comp(R8, u) = \{\{\neg PI(x, y, P)\}, \{CI(x), CI(y), PS(P)\}\}$
R9 Domain is unique	$\forall x, y, z \in \Sigma$: $Comp(R9, u) = \{\{\neg Domain(x, y)\}, \{\neg Domain(x, z)\}, \{(y = z)\}\}$
R10 Range is unique	$\forall x, y, z \in \Sigma$: $Comp(R10, u) = \{\{\neg Range(x, y)\}, \{\neg Range(x, z)\}, \{(y = z)\}\}$
R11 Domain and Range exists	$\forall x \in \Sigma \exists z \in \Sigma$: $Comp(R11.1, u) = \{\{\neg PS(x)\}, \{Domain(x, z)\}\}$ $\forall x \in \Sigma \exists$y$\in \Sigma$: $Comp(R11.2, u) = \{\{\neg PS(x)\}, \{, Range(x, y)\}\}$
R12 Class IsA is Transitive	$\forall x, y, z \in \Sigma$: $Comp(R12, u) = \{\{\neg C\_IsA(x, y)\}, \{\neg C\_IsA(y, z)\}, \{C\_IsA(x, z)\}\}$
R13 Class IsA is Irreflexive	$\forall x, y \in \Sigma$: $Comp(R13, u) = \{\{\neg C\_IsA(x, y)\}, \{\neg C\_IsA(y, x)\}\}$
R14 Property IsA is Transitive	$\forall x, y, z \in \Sigma$: $Comp(R14, u) = \{\{\neg P\_IsA(x, y)\}, \{\neg P\_IsA(y, z)\}, \{P\_IsA(x, z)\}\}$
R15 Property IsA is Irreflexive	$\forall x, y \in \Sigma$: $Comp(R15, u) = \{\{\neg P\_IsA(x, y)\}, \{\neg P\_IsA(y, x)\}\}$
R16 Determining C_Inst	$\forall x, y, z \in \Sigma$: $Comp(R16, u) = \{\{\neg C\_Inst(x, y)\}, \{\neg C\_IsA(y, z)\}, \{C\_Inst(x, z)\}\}$
R17 Determining PI	$\forall x, y, p, q \in \Sigma$: $Comp(R17, u) = \{\{\neg PI(x, y, p)\}, \{\neg P\_IsA(p, q)\}, \{PI(x, y, q)\}\}$
R18 Property IsAs and Domain	$\forall x, y, x\{\prime\}, y\{\prime\} \in \Sigma$: $Comp(R18, u) = \{\{\neg P\_IsA(x, y)\}, \{\neg Domain(x, x')\}, \{\neg Domain(y, y')\},$ $\{C\_IsA(x', y')\}, \{(x' = y')\}\}$

R19 Property IsAs and Range	$\forall x, y, x', y' \in \Sigma:$ $Comp(R19, u) = \{\{\neg P\_IsA(x, y)\}, \{\neg Range(x, x')\}, \{\neg Range(y, y')\}, \{C\_IsA(x', y')\}, \{(x' = y')\}\}$
R20 Property Instance and Domain	$\forall x, y, x', y' \in \Sigma:$ $Comp(R20, u) = \{\{\neg PI(x, y, x')\}, \{\neg Domain(x', y')\}, \{C\_Inst(x, y')\}\}$
R21 Property Instance and Range	$\forall x, y, x', y' \in \Sigma:$ $Comp(R21, u) = \{\{\neg PI(x, y, x')\}, \{\neg Range(x', y')\}, \{C\_Inst(y, y')\}\}$
R22 Class IsAs and the Top Node	$\forall x \in \Sigma:$ $Comp(R22, u) = \{\neg CS(x)\}, \{C\_IsA(x, \top)\}$
R23 Class Instanceof and the Top Node	$\forall x \in \Sigma:$ $Comp(R23, u) = \{\neg CI(x)\}, \{C\_Inst(x, \top)\}$
R24 Metaclass IsA Applicability	$\forall x, y \in \Sigma:$ $Comp(R24, u) = \{\{\neg M\_IsA(x, y)\}, \{MCS(x), MCS(y)\}\}$
R25 M_Inst Applicability	$\forall x, y \in \Sigma:$ $Comp(R25, u) = \{\{\neg M\_Inst(x, y)\}, \{CS(x), MCS(y)\}\}$
R26 Meta-Class IsA is Transitive	$\forall x, y, z \in \Sigma:$ $Comp(R26, u) = \{\{\neg M\_IsA(x, y)\}, \{\neg M\_IsA(y, z)\}, \{M\_IsA(x, z)\}\}$
R27 Determining M_Inst	$\forall x, y, z \in \Sigma:$ $Comp(R27, u) = \{\{\neg M\_Inst(x, y)\}, \{\neg M\_IsA(y, z)\}, \{M\_Inst(x, z)\}\}$
R28 Metaclass IsA is Irreflexive	$\forall x, y \in \Sigma:$ $Comp(R28, u) = \{\{\neg M\_IsA(x, y)\}, \{\neg M\_IsA(y, x)\}\}$
R29 DTP_Domain Applicability	$\forall x, y \in \Sigma:$ $Comp(R29, u) = \{\{\neg DTP\_Domain(x, y)\}, \{PL(x), CS(y)\}\}$
R30 DTP_IsA Applicability	$\forall x, y \in \Sigma:$ $Comp(R30, u) = \{\{\neg DTP\_IsA(x, y)\}, \{PL(x), PL(y)\}\}$
R31 DTPI Applicability	$\forall x, y, P \in \Sigma:$ $Comp(R31, u) = \{\{\neg DTPI(x, y, P)\}, \{CI(x), PL(P)\}\}$
R32 DTP_Domain is unique	$\forall x, y, z \in \Sigma:$ $Comp(R32, u) = \{\{\neg DTP\_Domain(x, y)\}, \{\neg DTP\_Domain(x, z)\}, \{(y = z)\}\}$
R33 DTP_Domain exists	$\forall x \in \Sigma \exists z \in \Sigma:$ $Comp(R33, u) = \{\{\neg PL(x)\}, \{DTP\_Domain(x, z)\}\}$
R34 DataType Property IsA is Transitive	$\forall x, y, z \in \Sigma:$ $Comp(R34, u) = \{\{\neg DTP\_IsA(x, y)\}, \{\neg DTP\_IsA(y, z)\}, \{DTP\_IsA(x, z)\}\}$
R35 DataType Property IsA is Irreflexive	$\forall x, y \in \Sigma:$ $Comp(R35, u) = \{\{\neg DTP\_IsA(x, y)\}, \{\neg DTP\_IsA(y, x)\}\}$
R36 Determining DTPI	$\forall x, y, p, q \in \Sigma:$ $Comp(R36, u) = \{\{\neg DTPI(x, y, p)\}, \{\neg DTP\_IsA(p, q)\}, \{DTPI(x, y, q)\}\}$
R37 DataType Property IsAs and DTP_Domain	$\forall x, y, x', y' \in \Sigma:$ $Comp(R37, u) = \{\{\neg DTP\_IsA(x, y)\}, \{\neg DTP\_Domain(x, x')\}, \{\neg DTP\_Domain(y, y')\}, \{C\_IsA(x', y')\}, \{(x' = y')\}\}$
R38 DataType Property Instance and DTP_Domain	$\forall x, y, x', y' \in \Sigma:$ $Comp(R38, u) = \{\{\neg DTPI(x, y, x')\}, \{\neg DTP\_Domain(x', y')\}, \{C\_Inst(x, y')\}\}$

constants}. In effect, U^P contains all positive ground facts that use predicate P and exist in U. Similarly, we set: $U^{\neg P} = \{\neg P(x_1, ..., x_n)$, where $\neg P(x_1, ..., x_n) \in U$ and $x_1, ..., x_n$ are constants$\}$. $U^{\neg P}$ contains all negative ground facts that use predicate P (i.e., $\neg P$) and exist in U. We say that:

Def. 9. *A FOL constant x appears in K iff there is some ground fact P of arity n $P(x_1, ..., x_n) \in K$ such that $x = x_i$ for some $i = 1, 2, ..., n$.*

In the following, x, y, z refer to FOL constants. Here are some obvious definitions: We say that x is a *direct subclass* of y in K iff $C\_IsA(x, y) \in K$ and there is no z such that $C\_IsA(x, z) \in K$ and $C\_IsA(z, y) \in K$. Similarly, x is a *direct subproperty* of y in K iff $P\_IsA(x, y) \in K$ and there is no z such that $P\_IsA(x, z) \in K$ and $P\_IsA(z, y) \in K$. The same holds for datatype properties. Following the same rational we use the general term *direct sub-object*, for a ground fact, that although allowed, is not implied from two others due to transitivity (instance determining) rules of Table 5.3.

We say that x is a top object in K iff x appears in K and there is no y such that x is a direct sub-object of y. The Intuitive meaning of these definitions in RDF graphs (which satisfy the integrity constraints of the model) are obvious. The direct sub-object definition identifies the pairs of objects that are directly related through an IsA or instantiation relation. By the integrity constraints, if x is a direct sub-object of y then both x and y appear in K. Notice that all class instances are instances of the top class and all classes are subclasses of the top class; therefore none of these objects can be a top object. On the other hand the top class ($\top$) is a top object.

Properties, datatype properties, and metaclasses could be top objects. Notice that a property that has no superproperties, or a metaclass without a parent, are top objects.

Def. 10. *A path from x to the top in K is a set S of the form $S = \{x_1, x2, ..., x_n\}$ where:*

(a) $x_1 = x$

(b) x_n *is a top object in* K

(c) For all $i = 1, 2, ..., n-1$*, it holds that* x_i *is a direct sub-object of* x_{i+1} *in* K.

The size of S is n.

A path is simply a sequence of direct sub-objects from the initial object (x) to a top object ($\top$, a property or a metaclass with no super objects). Notice that, in

the general case, there may be more than one paths from x to the top. If x is a top object, then the only path from x to the top in K is $\{x\}$, whose size is 1 (special case). If x does not appear in K, then there is no path from x to the top.

Lemma 12. *If x appears in K and K is finite, there is a "shortest" (with respect to size) path from x to the top.*

Proof: There is always at least one path from x to the top. This is obviously true if x is a top object; if it is not, then we can simply trace the sequence of direct super-objects until we reach a top object. The finiteness of K and the integrity constraints (no cycles allowed) guarantee that we will reach a top object after a finite number of steps. In addition, the finiteness of K and the integrity constraints guarantee that there is a finite number of paths from x to the top. ■

Def. 11. *We define the distance of x to the top to be ∞ whenever K is infinite or x does not appear in K (limit cases). Consider a finite KB K and a FOL constant x that appears in K. The distance of x to the top in K is a natural number n such that:*

(a) There is a path from x to the top in K whose size is n.

(b) There is no path from x to the top in K whose size is smaller than n.

(c) For all $i = 1, 2, ..., n-1$, it holds that x_i is a direct sub-object of x_{i+1} in K.

We will use the notation $Dist(x)$ to denote the distance of x to the top in K (K is omitted from the symbolism, but will be obvious from the context in each case). Obviously, $Dist(x) = \infty$ in limit cases and a positive integer in any other case by lemma 12.

In other words, the distance of x (if exists and K is finite) to the top is equal to the size of the "smallest" path (in terms of path size) to the closest top object. Note that in estimating $Dist$, resources on the schema level, even in the case that they are classes, are evaluated against the closest top object, (i.e., the $\top$) and no possible metaclasses the former instantiate.

According to the above remarks, the distance of x to the top exists whenever x appears in K and K is finite; so $Dist(x) \in \mathbb{N}$ and $Dist(x) \geq 1$. Considering the limit cases also, $Dist(x) \in N \cup \{\infty\}$. Notice that it might be the case that $Dist(x) = Dist(y)$ but $x \neq y$, even if $Dist(x)$, $Dist(y)$ are finite (i.e., not equal to ∞). We define a relation $\leq$ between positive and negative predicates, denoted by $\leq_P$. For two positive or negative predicates P, Q we say that P is *cheaper* than Q iff $P \leq_P Q$. This order is formally defined as the transitive closure of the one shown in Table 5.5.

Table 5.5: Ordering of predicates

$PI \leq_P DTPI \leq_P C\_Inst \leq_P M\_Inst \leq_P P\_IsA \leq_P DTP\_IsA \leq_P C\_IsA \leq_P M\_IsA \leq_P \neg PI \leq_P \neg DTPI \leq_P \neg C\_Inst \leq_P \neg M\_Inst \leq_P \neg P\_IsA \leq_P \neg DTP\_IsA \leq_P \neg C\_IsA \leq_P \neg M\_IsA \leq_P \neg Domain \leq_P \neg DTP\_Domain \leq_P \neg Range \leq_P \neg CI \leq_P \neg PS \leq_P \neg PL \leq_P \neg CS \leq_P \neg MCS \leq_P Domain \leq_P DTP\_Domain \leq_P Range \leq_P CI \leq_P PS \leq_P PL \leq_P CS \leq_P MCS$

Obviously, the relation $\leq_P$ is reflexive, antisymmetric, transitive and total. We also define a ordering $\leq_{lex}$ based on the lexicographic ordering on the constants of the FOL language; notice that a lexicographic ordering makes sense only for strings, so we extend the standard lexicographic ordering as follows:

Def. 12. *We define $\leq_{lex}$ relation, which for brevity, we will call "lexicographic", even though, technically, it is not entirely lexicographic:*

- *If x and y are auxiliary constants (say $x = c_i$, $y = c_j$ respectively), then $x \leq_{lex} y$ iff $i \leq j$.*
- *If x is an auxiliary constant c_i and y is a string, then $x \leq_{lex} y$.*
- *If x and y are strings, then $x \leq_{lex} y$ iff x is "before" y in the standard lexicographic ordering of the strings x, y.*
- *If x is equal to the constant $\top$ (i.e., x=$\top$), then $y \leq_{lex} x$.*

In simpler words, def. 12 states that the "first" (minimal) element of this ordering is c_1, followed by c_2, c_3,... All strings are "after" the auxiliary constants in the ordering and are ordered lexicographically. Finally, the constant $\top$ is "last" (maximal) in the order. Obviously, the relation $\leq_{lex}$ is reflexive, antisymmetric, transitive and total. In addition, our lexicographic ordering has the following very useful property:

Lemma 13. *For any finite or infinite set S of constants, there is some constant x such that $x \leq_{lex} y$ for all $y \in S$, i.e., S has a minimum.*

Proof: If there is any auxiliary constant in S, take $x = c_i$ where i is the "first" index such that $c_i \in S$. If there is no auxiliary constant in S, then S is finite (because all strings have finite size, so there is a finite number of them, so the union $\{s|s$: string of length n, $1 \leq n \leq N\} \cup \{\top\}$ is finite and $S \subseteq \{s|s$: string of length n, $1 \leq n \leq N\} \cup \{\top\}$ is also finite).■

Finally, we define the relation $\leq_G$ between ground facts as summarized in Table 5.6. Consider two (positive or negative) ground facts $P(x_1, ..., x_n), Q(y_1, ..., y_m)$. Then, if $P \neq Q$ we set $P(x_1, ..., x_n) \leq_G Q(y_1, ..., y_m)$ iff $P \leq_P Q$. If $P = Q$ we Compare the arguments of the ground facts as described in the table. We therefore say that $P(x_1, ..., x_n)$ is cheaper than $Q(y_1, ..., y_m)$, iff $P(x_1, ..., x_n) \leq_G Q(y_1, ..., y_m)$. The first column of the table represents the order $\leq_P$, as well as the order $\leq_G$ whenever $P \neq Q$, i.e., when the Compared predicates are different. In this column, the most "expensive" type of predicate is first in the list (in the first raw). The predicate order does not depend on K. The second column represents the order $\leq_G$ whenever $P = Q$, i.e., when the Compared predicates are the same. The second column informally describes how to determine the cheapest fact when $P = Q$. The ground fact order depends on K (because it depends on Dist). So, in simpler words, when having to Compare two ground facts, one would find cheaper the ground fact which uses a predicate lying at a lower raw in the table. If the ground facts are in the same raw in the table (i.e., we are dealing with the same predicate), the Comparison is determined as described in the right cell of this raw.

This relation depends on $\leq_{lex}$, $\leq_P$ and the standard $\leq$ relation of natural numbers. The nice properties of these relations (reflexivity, antisymmetry, transitivity and totality), allows us to show that the relation $\leq_G$ is reflexive, antisymmetric, transitive and total. The full proof requires considering each case separately (omitted). In addition, this ordering has the same very useful property as the lexicographic one:

Lemma 14. *For any KB K and any non-empty, finite or infinite set U of ground facts, there is some ground fact $P(x_1, ..., x_n)$ such that $P(x_1, ..., x_n) \leq_G Q(y_1, ..., y_m)$ for all $Q(y_1, ..., y_m) \in U$, i.e., U has a minimum.*

Proof: Suppose that K is finite; then all $Dist(.)$ are finite natural numbers and there is a finite number of them, so there is a minimum and a maximum $Dist(.)$ for the constants that appear in K. Now take any constant x; if the constant appears in K, then $Dist(x)$ is a value from a finite set; if the constant does not appear in K, then $Dist(x) = \infty$. On the other hand, if K is infinite, then for all constants x, $Dist(x) = \infty$. We conclude that for any K there is some finite set S, such that for any constant x, $Dist(x) \in S$. In addition, there is a finite number of predicates, so the number of different predicates that appear in U is finite. Notice that $\leq_G$ takes into account the predicate involved ($\leq_P$), the $Dist$ of the constants in the predicates (under the standard numerical ordering $\leq$) and the lexicographic ordering ($\leq_{lex}$). All three orderings have minimums (see above); exploiting this fact, we can prove the result. Details are omitted.■

Notice that the inclusion of K in the proposition is necessary, because $\leq_G$ depends on K.

The ordering $\leq_G$ provides an order of "cheapness" for each ground fact that appears in an update U. We need to expand this definition so as to be able to Compare sets of ground facts (i.e., updates); this expansion should result to an update-generating order.

Def. 13. *Consider any two updates U_1, U_2 and a KB K. Set $U_1' = Cost(K, U_1)$, $U_2' = Cost(K, U_2)$. Then $U_1 \leq_K U_2$ iff any of the following are true:*

(a) There is some predicate P such that $|U_1'^P| < |U_2'^P|$ or $U_1'^P \subset U_2'^P$ and for all predicates Q such that $P \leq_P Q$, $Q \neq P$ it holds that $|U_1'^Q|$=$|U_2'^Q|$, $U_2'^P \subsetneq U_1'^P$.

(b) For all predicates P it holds that $|U_1'^P| = |U_2'^P|$, $U_1'^P \subsetneq U_2'^P$ and $U_2'^P \subsetneq U_1'^P$ and there is some (positive or negative) ground fact of the form $Q(x_1, ..., x_n) \in U_1' \setminus U_2'$, such that for all (positive or negative) ground facts of the form $Q'(y_1, ..., y_m) \in U_2' \setminus U_1'$ it holds that $Q(x_1, ..., x_n) \leq_G Q'(y_1, ..., y_m)$.

(c) $U_1' = U_2'$.

This defines an ordering $\leq_K$ (which depends on K, because $\leq_G$ depends on K). Suppose that we are given two updates $U_1, U_2 \subseteq L^0$. Instead of considering the facts in U_1, U_2 directly, we consider the facts in $U_1' = Cost(K, U_1)$, $U_2' = Cost(K, U_2)$ (this guarantee conflict sensitivity). To determine whether $U_1 \leq_K U_2$, we start with the most "expensive" predicate (per $\leq_P$) and determine whether the number of facts that use this statement in U_1' is smaller than the number of facts that use this statement in U_2'. If this is so, we return $U_1 \leq_K U_2$. Notice that in the (limiting and uninteresting for practical purposes) case where the number of statements in these sets for the particular predicate is infinite, it could be the case that $U_1'^P \subset U_2'^P$ but $|U_1'^P|$=$|U_2'^P|$; in this case, as $U_1 \leq_K U_2$ monotonicity still holds, because of the "or" clause in the first bullet of the definition.

If the opposite is true, i.e., the number of facts that use this statement in U_2' is smaller than the number of facts that use this statement in U_1' (or it is its proper subset), it holds $U_2 \leq_K U_1$. Note that a characteristic of this property is that a single ground fact may be more expensive than a great number (in effect infinite) of other ground facts. For example, adding a single CS(A) fact to our RDF graph is more expensive than adding an infinite number of C_IsA statements. The Intuition behind this choice is that due to transitivity it should be "cheap" to add cascading subsumption or instance relations triggered by a single such addition. On the other

Table 5.6: Our Ordering

Predicate Order	Ground Fact Order
PI	Use the distance (Dist function) of the first parameter from the top; if comparing finite distances the larger the distance, the cheaper the ground fact, if one of the distances is infinite then the smaller the distance, the cheaper the ground fact. If this distance is identical, compare, using exactly the same algorithm, the second parameters. If again that distance is identical use the distance of the third parameter (the property) from the top; in this case the smaller the distance, the cheaper the groundfact at any occasion. If again this distance is the same use lexicographic ordering ($\leq_{lex}$) on the first parameter, and subsequently on the second and the third parameter until you find a winner; when comparing lexicographically the winner is (by convention) the "smallest".
$DTPI$	Same as PI
C_Inst	Same as C_IsA
M_Inst	Same as C_IsA
P_IsA	Same as C_IsA
DTP_IsA	Same as C_IsA
C_IsA	Use the distance (Dist function) of the first parameter from the top; if comparing finite distances the larger the distance, the cheaper the ground fact, if one of the distances is infinite then the smaller the distance, the cheaper the ground fact. If this distance is identical, use the distance (Dist function) of the second parameter from the top; the smaller the distance, the cheaper the ground fact. If both distances are identical, use lexicographic ordering ($\leq_{lex}$) on the first parameter. If both distances are identical and the first parameter is also identical, use lexicographic ordering ($\leq_{lex}$) on the second parameter.
M_IsA	Same as C_IsA
$\neg PI$	Same as PI
$\neg DTPI$	Same as PI
$\neg C\_Inst$	Same as C_IsA
$\neg M\_Inst$	Same as C_IsA
$\neg P\_IsA$	Same as C_IsA
$\neg DTP\_IsA$	Same as C_IsA
$\neg C\_IsA$	Same as C_IsA
$\neg M\_IsA$	Same as C_IsA
$\neg Domain$	Same as $Domain$
$\neg DTP\_Domain$	Same as $Domain$
$\neg Range$	Same as $Domain$
$\neg CI$	Same as CI
$\neg PS$	Same as CI
$\neg PL$	Same as CI
$\neg CS$	Same as CI
$\neg MCS$	Same as CI
$Domain$	Use the distance (Dist function) of the first parameter from the top; if comparing finite distances the larger the distance, the cheaper the ground fact, if one of the distances is infinite then the smaller the distance, the cheaper the ground fact. If this distance is identical, compare, using exactly the same algorithm, the second parameters.If both distances are identical, use lexicographic ordering ($\leq_{lex}$) on the first parameter. If both distances are identical and the first parameter is also identical, use lexicographic ordering ($\leq_{lex}$) on the second parameter.
DTP_Domain	Same as $Domain$
$Range$	Same as $Domain$
CI	Use the distance (Dist function) of the only parameter from the top; if comparing finite distances the larger the distance, the cheaper the ground fact, if one of the distances is infinite then the smaller the distance, the cheaper the ground fact. If this distance is identical, use lexicographic ordering ($\leq_{lex}$) on the second parameter.
PS	Same as CI
PL	Same as CI
CS	Same as CI
MCS	Same as CI

hand addition of a new concept to our graph should be avoided if possible as it would seem rather awkward.

If none of the above is true, i.e., the number of ground facts that use the most expensive predicate is equal in each of U_1', U_2' and none is a proper subset of the other, then we can't determine the result just yet, so we repeat the process with the second most "expensive" predicate (per $\leq_P$). We continue until the result is determined, or we run out of predicates. If we run out of predicates, then all predicates appear an equal number of times in both updates and they are not related with the (proper) subclass relation (taken per predicate); however, this does not necessarily mean that U_1' and U_2' are equal, because the parameters of the predicates might be different. To determine whether $U_1 \leq_K U_2$ (or vice-versa) in this case we need to consider the ground facts that appear in U_1' and U_2', i.e., the arguments (constants) of the predicates.

In this case, the "cheapest" update is the one that contains the "cheapest" ground fact (per $\leq_G$) out of all the ground facts that appear in only one of the two updates (U_1', U_2'). That is, we seek the cheapest (per $\leq_G$) ground fact in the set $(U_1' \setminus U_2') \cup (U_2' \setminus U_1')$; if this appears in $(U_1' \setminus U_2')$ then $U_1 \leq_K U_2$; if this appears in $(U_2' \setminus U_1')$ then $U_2 \leq_K U_1$. Notice that we can always find a minimum in $(U_1' \setminus U_2') \cup (U_2' \setminus U_1')$, unless $(U_1' \setminus U_2') \cup (U_2' \setminus U_1') = \emptyset$ in which case $U_1' = U_2'$; in this case, both $U_1 \leq_K U_2$ and $U_2 \leq_K U_1$ (this satisfies conflict sensitivity and reflexivity).

Below we verify that the above ordering is update-generating:

Prop. 2. *The ordering $\leq_K$ as defined is an update-generating order for all KBs K.*

Proof: Consider any KB K and any updates U_1, U_2, U_3. Set $U_i' = Cost(K, U_i)$, for $i = 1, 2, 3$. Conflict sensitivity is an immediate consequence of lemma 3: since $Cost(K, U_i) = Cost(K, Cost(K, U_i))$ for all $i = 1, 2, 3$, we conclude that $U_i' = Cost(K, U_i')$ for all $i = 1, 2, 3$. The rest is obvious by the fact that, in order to determine whether $U_1 \leq_K U_2$, we consider the sets U_1', U_2'. For Cost antisymmetry suppose that $U_1 \leq_K U_2$ and $U_2 \leq_K U_1$ and that $U_1' \neq U_2'$. Since $U_1 \leq_K U_2$, either the first or the second bullet of the definition is true. If the first bullet is true, then it cannot be the case that $U_2 \leq_K U_1$, a contradiction. So the second bullet is true for U_1, U_2. Thus, there is some ground fact $G_1 \in U_1' \setminus U_2'$ with the required properties, so $U_1' \setminus U_2' \neq \emptyset$.

Using similar arguments we can show that the second bullet is true for U_2, U_1, there is some ground fact $G_2 \in U_2' \setminus U_1'$ with the required properties and $U_2' \setminus U_1' \neq \emptyset$. Thus, $(U_1' \setminus U_2') \cup (U_2' \setminus U_1') \neq \emptyset$, so it has a minimum (see above); set G the minimum of the set $(U_1' \setminus U_2') \cup (U_2' \setminus U_1')$. Since $U_1 \leq_K U_2$ and

$U_2 \leq_K U_1$, using the above remarks we are forced to conclude that $G \in U_1' \setminus U_2'$ and $G \in U_2' \setminus U_1'$, which is obviously a contradiction.

For Monotonicity, suppose that $U_1 \subseteq U_2$; then by the definition of Cost, $U_1' \subseteq U_2'$. Set $U = U_2' \setminus U_1'$. If $U = \emptyset$ then $U_1' = U_2'$ so $U_1 \leq_K U_2$. If $U \neq \emptyset$ then there is some minimum for U (per $\leq_G$), say $P(x_1, ..., x_n)$. Then, obviously, $U_1'^P \subset U_2'^P$ and for all predicates Q such that $P \leq_P Q$, $Q \neq P$ it holds that $U^Q = \emptyset$, so $U_1'^Q = U_2'^Q$; thus $|U_1'^Q| = |U_2'^Q|$, $U_2'^P \subsetneq U_1'^P$. We conclude that $U_1 \leq_K U_2$.

For Totality: Set S=$\{P|P$: predicate for which $|U_1'^P| \neq |U_2'^P|$ or $U_1'^P \subset U_2'^P$ or $U_2'^P \subset U_1'^P\}$. If $S \neq \emptyset$, then S is obviously finite (since there is a finite number of predicates) so we can find the "most expensive" (per $\leq_P$) predicate that appears in S (say $Q \in S$). If, for Q, it holds that $|U_1'^Q| < |U_2'^Q|$ or $U_1'^P \subset U_2'^P$ then $U_1 \leq_K U_2$; if it holds that$|U_2'^P| < |U_1'^P|$ or $U_2'^P \subset U_1'^P$ then $U_2 \leq_K U_1$.

So, let us suppose that $S = \emptyset$. Set $S' = (U_1' \setminus U_2') \cup (U_2' \setminus U_1')$. If $U_1' \setminus U_2' = \emptyset$ then $U_1' \subseteq U_2'$; by monotonicity: $U_1' \leq_K U_2'$; by conflict sensitivity: $U_1 \leq_K U_2$. Similarly, if $U_2' \setminus U_1' = \emptyset$ then $U_2 \leq_K U_1$. So, let us suppose that $U_1' \setminus U_2' \neq \emptyset$ and $U_2' \setminus U_1' \neq \emptyset$. Take the minimum of $U_1' \setminus U_2'$, say G_1 and the minimum of $U_2' \setminus U_1'$, say G_2. If $G_1 \leq_G G_2$ then the bullet and our assumptions imply that $U_1 \leq_K U_2$. If $G_2 \leq_G G_1$ then the second bullet and our assumptions imply that $U_2 \leq_K U_1$. Since $\leq_G$ is total, there is no other possibility and we are done.

For Transitivity: If $U_1' = U_2'$ or $U_2' = U_3'$ then the proof follows from conflict sensitivity. If the first bullet is true for either U_1, U_2 or U_2, U_3, then the proof is obvious. If the second bullet is true for both U_1, U_2 and U_2, U_3, then, similar to the totality proof we can show that the sets $(U_1' \setminus U_2')$, $(U_2' \setminus U_1')$, $(U_2' \setminus U_3')$, $(U_3' \setminus U_2')$ are nonempty. Suppose that $U_1' \setminus U_3' = \emptyset$; then $U_1' \subseteq U_3'$, so combining monotonicity and conflict sensitivity we get that $U_1 \leq_K U_3$ and we are done.

Suppose that $U_3' \setminus U_1' = \emptyset$; then $U_3' \subseteq U_1'$, so combining monotonicity and conflict sensitivity we get that $U_3 \leq_K U_1$; using the proof of monotonicity, we conclude that the relation $U_3 \leq_K U_1$ can be shown using the first bullet of the definition. But transitivity has been shown to hold when one of the two inequalities are based on the first bullet, so $U_3 \leq_K U_1$ and $U_1 \leq_K U_2$ imply $U_3 \leq_K U_2$. In addition, $U_2 \leq_K U_3$; using Cost antisymmetry, we conclude that $U_2' = U_3'$, which leads us to the case we studied in the first line of this proof.

So, let us suppose that $U_1' \setminus U_3' \neq \emptyset$, $U_3' \setminus U_1' \neq \emptyset$. By our results, each of these sets has a minimum per $\leq_G$. Set $G_{ij} = min(U_i' \setminus U_j')$ for $i, j = 1, 2, 3$, our assumptions: $G_{12} \leq_G G_{21}$ and $G_{23} \leq_G G_{32}$. It remains to show that $G_{13} \leq_G G_{31}$. We will use a tree-like structure to cover all the different cases for G_{ij} (see fig 5.3. First, consider some triple $i, j, k \in 1, 2, 3$ such that i, j, k are mutually disjoint. Then: If $G_{ij} \in U_k'$, it follows that $G_{ij} \in U_k' \setminus U_j'$, so $G_{ki} \leq_G G_{ij}$. If $G_{ij} \notin U_k'$, it follows that $G_{ij} \in U_i' \setminus U_k'$, so $G_{ik} \leq_G G_{ij}$. In addition, it cannot be the case that $G_{ij} = G_{ji}$ because $G_{ij} \in U_i'$, $G_{ji} \notin U_i'$ by definition.

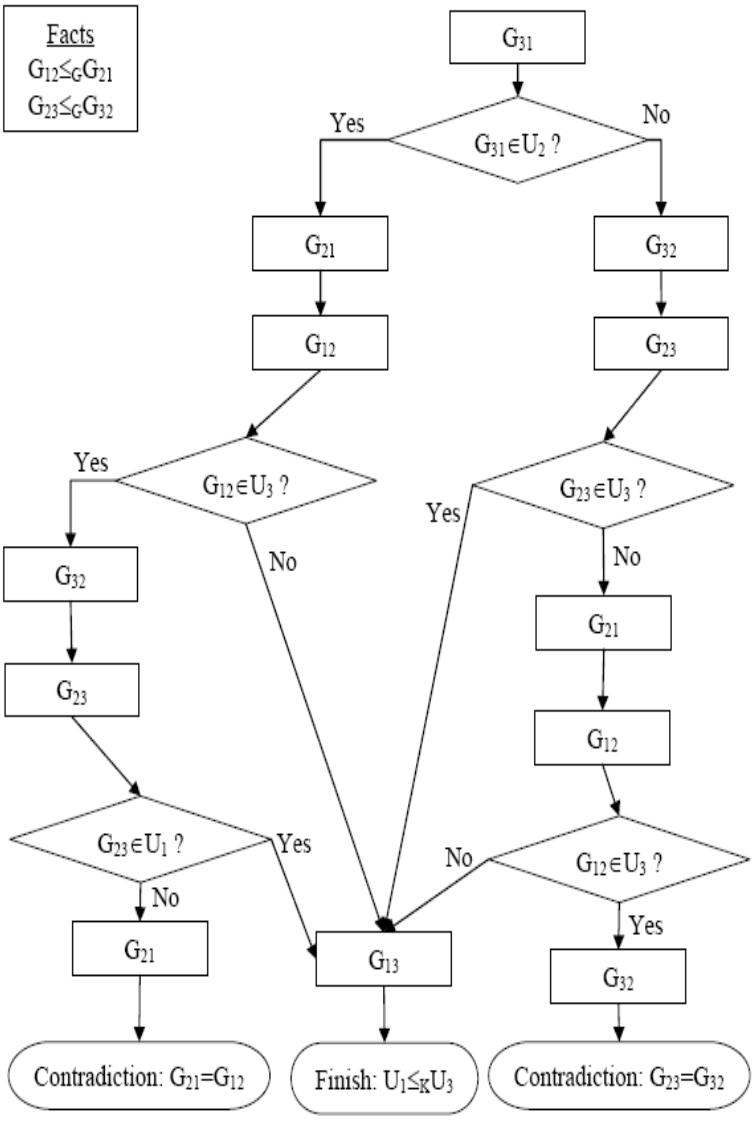

Figure 5.1: Ordering is update generating

Using these remarks, we can create the tree of possibilities shown in the Fig. 5.3; this tree is described as a flowchart. The facts with which we start are shown in a box in the upper left corner. As we move down the tree, we encounter boxes which contain ground facts (G_{ij}) which are successively "smaller" (per $\leq_G$). We start with G_{31}; the process stops when we reach G_{13} (in which case, by the transitivity of $\leq_G$, $G_{13} \leq_G G_{31}$ so $U_1 \leq_K U_3$) or if we reach a G_{ij} twice, with an intermediate ground fact G_{ji} (in which case $G_{ij} \leq_G G_{ji}$ and $G_{ji} \leq_G G_{ij}$, so by the antisymmetry of $\leq_G$, implies $G_{ij} = G_{ji}$, i.e., a contradiction).

At each step, we add a new ground fact implied either by the facts in the upper left corner, or by a particular assumption made (described by rhombus in the diagram) and the observations above. As shown in the diagram, for all possible sets of assumptions, we either reach a contradiction (i.e., the particular set of assumptions is not possible) or we reach a position where $G_{13} \leq_G G_{31}$ (i.e., $U_1 \leq_K U_3$) can be shown. This Completes the proof. Thus $\leq_K$ is update-generating.■

5.4 Termination

In the 4^{th} step of our algorithm, we try to find a rule c and a special set of constants u, which violate the specific instance of this rule. Most of the rules are a set of a finite number of sets, except those with an existential quantifier. All rule families $Comp(c, \vec{u})$ except $R11$ and $R33$ although having finite size, they have an infinite number of instances, one for each of the possible set of constants $\vec{u}$. In this section we emphasize that during the 4^{th} step of the $Update$ function we can constrain the available constants to a finite size, in effect we can implement $Select$ to consider only the existing constants in $K \cup U$ at that particular run, plus the "cheapest" auxiliary non-existing constant.

As also explained in the previous chapter, when selecting a rule instance that contains a specific ground fact, we bound some of this rule instance's variables (with the constants existing in the ground fact) leaving a number of free variables. With infinite constants in our language these free variables might lead to an infinite number of rule instantiations, or to rule instantiations with an infinite size.

Below we solve the infinite instances/sizes problem by examining our specific set of rules and showing that the free variables in the rules suffering from this problem, can be bounded to a finite set. Bounding the free variables to non-existing constants (in the ontology or update) is shown to be fruitless and can be avoided. For those rules that contain an existential quantifier we also consider one, the "cheapest" per our $\leq_{lex}$ to be considered.

Prop. 3. *Suppose a run of the $Update$ function with initial inputs $U, K, \emptyset$. For the set of rules presented in Table 5.3, constraining $Select$ function of the algorithm to*

search over the finite space of all existing constants in U or K, plus the cheapest per $\leq_{lex}$, "auxiliary" constant, will detect all possible rule instances violations.

Proof:

The rules $R1$, $R13$, $R15$, $R22$, $R23$, $R28$, $R35$ have no free variables. When adding or removing a ground fact $P(\vec{x})$, there is being created only one instance $Comp(c, \vec{y})$ of those rules, the one having $\vec{u} = \vec{x}$. Therefore there is no point in examining in $Select$ function any constants other than $\vec{x}$.

$R2$ is a special rule. There are no free variables here and the rule does not break. Trying to remove the top class, the only update which could violate this rule, is an infeasible update.

The rest of our rule's instances, except $R11$ and $R33$, are $Comp(c, \vec{u})$ of the form $\{\{\neg A_1(\vec{u_1})\}, \{\neg A_2(\vec{u_2})\}$,..., $\{\neg A_n(\vec{u_n})\}$, $\{B_{00}(\vec{u}_{B_{00}})$,$B_{01}(\vec{u}_{B_{01}})$,...,$B_{0l}(\vec{u}_{B_{0l}})$ $\}$, $\{B_{10}(\vec{u}_{B_{10}})$,$B_{11}(\vec{u}_{B_{11}})$,..., $B_{1l}(\vec{u}_{B_{0l}})$ $\}$, $\{B_{m0}(\vec{u}_{B_{m0}})$,$B_{m1}(\vec{u}_{B_{m1}})$,..., $B_{ml}(\vec{u}_{B_{0l}})$ $\}\}$ for some $m \geq 0$, l depending on m, $n \geq 1$, c a specific Integrity Constraint, A_i and B_{ij} positive ground facts and $\vec{u} = \bigcup_{i=1}^{i=n} \vec{u_i} \cup \bigcup_{i=B_{00}}^{i=B_{ml}} \vec{u_i}$

An important notice is that in our case and for all integrity constraints of ours that can be written in the above form it holds that $\bigcup_{i=B_{00}}^{i=B_{ml}} \vec{u_i} \subseteq \bigcup_{i=1}^{i=n} \vec{u_i}$. Therefore $\vec{u} = \bigcup_{i=1}^{i=n} \vec{u_i}$.

Now consider that we bound a part of the set of tuples $\vec{u}$ of the above rule, when adding one or more $A_i(\vec{u}_i)$ and/or removing one or more $B_{ij}(\vec{u}_{B_{ij}})$. Rule instances of c are violated if either some $A_i(\vec{u}_i)$ is present in the update or the ontology, or a $B_{ij}(\vec{u}_{B_{ij}})$ is absent. When trying to see whether a $\neg A_i(\vec{u}_i)$ holds, there is no point examining any tuple of constants $\vec{v}$ which does not exist in the initial ontology or the update as for this tuple, $\neg A_i(\vec{v})$ holds for sure. Therefore if there are free variables in any set $\{A_i(\vec{u}_i)\}$ of the above rule, there's no point examining non existing constants, as all the rule instances these constants construct are satisfied.

Suppose on the other hand, that there are not any free variables in $A_i(\vec{u}_i)$, so the c's instances that are violated are those who have at least one $B_{ij}(\vec{u}_{B_{ij}})$, not satisfied, i.e., not present in the ontology. However, as we stated, when all the variables appearing in A_is are all the variables of the rule: $\vec{u} = \bigcup_{i=1}^{i=n} \vec{u_i}$. So any $\vec{u}_{B_{ij}}$ is immediately bounded.

Therefore there are not any instances of rules $R3$-$R10$, $R12$, $R14$, $R16$-$R21$, 24-$R27$, 29-$R32$, $R34$, $R36$-$R38$, violated for non-existing constants in the update or the ontology.

Actually, the above arguments are subject to one exception for the rules that have an existential quantifier, i.e., $R11$ and $R33$. In these cases we don't face an "infinite" rule instance problem but a single's rule's instance "infinite width" problem.. When one tries to bound the variables of these rules to constants, each of $R11$, $R33$ generates only one instance of the rule with infinite number of sets, one

for each of the infinite number of pairs of $(\vec{y}, z)$. The instance that these rules have, takes the form $Comp(c, \vec{u})$ = {{$A(x)$}, {$B(x, y_1)$,$C(x, z_1)$}, {$B(x, y_2)$,$C(x, z_2)$}, ..., {$B(x, y_n)$,$C(x, z_n)$}} , where, x, y_i, z_i are constants and $n \rightarrow \infty$. Because of the constraint "$\exists y, z$" in these rules, only x can be bounded to a constant.

However notice that for {$B(x, y_i)$,$C(x, z_i)$}, choosing among auxiliary non existing constants c_1, c_2 to restore this rule by bounding y_i, z_i both to c_1 or to c_2 will lead to two almost identical solutions (Deltas). The only difference will be that one result will contain {$B(x, c_1)$,$C(x, c_1)$} where the other will contain {$B(x, c_2)$,$C(x, c_2)$}. We now based on our ordering that the "cheapest" result will be the one containing the "cheapest" (per $\leq_{lex}$) of c_1,c_2

Concluding when searching rule violations of the above rules including the first "cheapest", non-existing in K or U, auxiliary tuple of constants in our search space, should be enough. This "cheapest" is depending on the ordering used, so any ordering following the properties we defined here should have one.

Therefore termination of our algorithm is independent of the ordering used and depends only on the rules of the language. In our case termination is guaranteed, by defining $Select$ function to be searching over the finite set of all existing in U or K, constants plus the cheapest per $\leq_{lex}$ auxiliary constant.■

5.5 Example and Optimizations

Let K be our initial KB and suppose $U = \{\neg CS(A)\}$ is an update to be applied upon K. Next we elaborate on the algorithm's behavior. In STEP2, we check if the requested update, is already implemented in K. If $K \cup | U \nvdash \{\neg CS(A)\}$, (i.e., class A is in K), we continue to STEP4 (STEP3 is trivial),where we try to identify the integrity constraints that might be violated. These are the (instances of the)rules that contain $\neg(\neg CS(A))$, i.e., $CS(A)$, in at least one element of their component set.

For example, consider rule R3 with $Comp(R3,(x, A))$ = {{$\neg\ Domain(x, A)$}, {$PS(x)$, $CS(A)$}}. If we instantiate this rule for a constant P it states that when information $Domain(P, C)$ exists in our KB, then property P and class C, should also exist. As STEP4 demands, $\neg(\neg CS(A))$ = $CS(A) \in V$={$PS(x)$,$CS(A)$}, while $V \in Comp(R3, (x, A))$. R3 will be violated if there is no other component of $Comp(R3, (x, a))$ implied by K. Suppose for start that K doesn't contain any property, which has A as its domain. Therefore the rule is not violated. Thus, the update $\neg CS(A)$ can be returned as an output for raw application on K.

Notice here that, if not restricting the $Select$ function as discussed in the previous section, instead of returning immediately the result, the algorithm makes "another round of calls", in case our update contained more than one ground facts;

$\forall x \in L$ (which are infinite) the algorithm would do the following: It proceeds to STEP6, where it re-calls Update function with input $\{\neg CS(A), \neg Domain(x, A)\}$ (we avoid explaining B for the moment). It reaches again STEP3 (where it cannot select $\neg Domain(x, A)$, as this is implied by K (there's no $Domain(x, A)$ in K) and so it reselects $\neg CS(A)$. In STEP4 it re-examines R3, but now there's a $V' \neq V \in Comp(R3, (x, A))$, for which it holds $V' \subseteq U \cup ESE$ (U contains $\neg Domain(x, A)$). Hence it goes in STEP5 and returns $\{\neg CS(A)\} \cup O$, where O is the result of a last call to the update function (after adding $\neg CS(A)$ to ESE) that will evaluated to $\emptyset$ in STEP2. Therefore, for the infinite number of x that are absent from our KB and Update, the algorithm would for sure, check and find no violations, returning its input ($\{\neg CS(A)\}$) unharmed.

On the other hand, if the class A, we are removing, is a domain of some property say P,and R3 is violated, then in STEP6 we unify each side-effect with the update set and recall the update function for each one of them. We keep always (in B) the minimum result, comparing them with the use of *min* (which implements our $\leq$), and return the minimum solution. In our example, $Comp(R3, (P, A))$ has only two elements (V'), and therefore there is left only one of them to be applied as a side-effect.This is the $\neg Domain(P, A)$. Thus STEP6 returns the result of Update($\{\neg\ CS(A)\ ,\ \neg\ Domain(P, A)\}$, K , $\emptyset$). In subsequent calls for this property P, R3 will not be violated again due to the existence of $\{\neg Domain(P, A)\}$ in the update, but it could be violated again if class A was also a domain of some other property, R. Now, during the algorithm's execution $\{\neg Domain(P, A)\}$ will also be examined for causing side-effects.

For presentation purposes, we don't set out in detail, the procedure that the addition of $\{\neg Domain(P, A)\}$ will cause. We just state that when removing $Domain(P, A)$, R11 suggests that either a new Domain should be inserted for P, or the property P should be completely deleted. Our ordering favors the second solution, despite its side-effects (remove P and the information of its range,remove all property subsumption relations that contain P, and remove all instantiation links of P) as we have placed $Domain(x, y)$ high in our ordering, instating it more "expensive" than all these actions.

An entirely symmetrical, to the above, procedure is materialized in case the under deletion class, A, has the role of some property's range, in K (rule R4). Integrity Constraint R5 is also affected by a class removal if there is an IsA to or from this class. Thereupon, R5 yields the removal of any such IsA. In addition, R7 dictates the removal of any instantiation link of this class ($C\_Inst(x, C)$). Lastly, if the class we try to remove is the Top Class, i.e., $A = \top$, R2 is being violated having not any possibility to be restored and leading this way to INFEASIBLE.

The analysis of this section leads to a simpler form of the algorithm following in table 5.7,that is specific for the discussed operation. Similarly, following the

Table 5.7: Remove Class

```
If CS(A) isn't already absent from K:
    Remove all class IsA relationships deriving from A.
    Remove all class IsA relationships arriving in A.
    Remove all instantiation links between a resource and A.
    For every property P whose range/domain is A:
        Remove all property IsA relationships deriving from P.
        Remove all property IsA relationships arriving in P.
        Remove all instantiation links of P.
        Remove P and the information on its range/domain.
    Remove A.
```

execution of the algorithm we can develop simpler, special-purpose algorithms, for the particular application that we are interested at (e.g., RDF in our case). These "instantiations" are much faster than the general algorithm, but can still be proven equivalent to it, i.e., formally sustained.

Thus, we can guarantee that they exhibit the expected/desired behavior, by verifying them against the general-purpose algorithm above. Notice that they are similar to ad-hoc methodologies applied by other systems but without resorting to the tedious and error-prone case-based reasoning usually employed for this purpose. Moreover, the general algorithm could still be used to implement any possible operation, beyond these specific solutions. Below we pose a number of common operations that could be implemented this way. Yet, the general algorithm could always be used to implement any possible operation, these specific operations haven't covered. We have developed such special case algorithms for 32 singular operations, in effect for all the singular sets (sets that have only one ground fact) we can form with all our 32 positive and negative ground facts. These algorithms are given in Appendix A.

5.6 Necessity of our algorithm

Before moving on to implementation details we would like to elaborate on the necessity of our general algorithm. As we have already pointed out, one interesting advantage of the formal foundation upon which we establish our methodology is the ability to support arbitrary in size, and in form, composite updates without resting this effort on the existence of elementary ones. In fact, and while remaining in the space of standard RDF semantics, we can argue that there is not a finite number of pre-determined operations upon which any composite update can be

decomposed.

Let's consider as an update the set $\{\neg C\_IsA(A, C)\}$ applied upon an ontology $O = C\_IsA(A, B), C\_IsA(B, C), C\_IsA(A, C)$. Rule $R11$ (which also belongs to the standard RDF semantics), captures the intuition that in order to remove an implicit IsA one needs to remove an explicit that "stands in the way". Therefore, for the sake of this example (and without loss of generality), suppose that $C\_IsA(A, B)$ is preferable to be removed together with $C\_IsA(A, C)$, in order for the update to be "rational".

Let an update $U = \{\neg C\_IsA(A, C), C\_IsA(A, B)\}$. In this case, the existence of the explicit requirement that $C\_IsA(\ A, B)$ should be "added", forces us to select $\neg\ C\_IsA(B, C)$ as a side-effect of $\neg C\_IsA(A, C)$ (which is not what we, and also other implemented systems, would normally chose, e.g. $\neg C\_IsA(A, B)$).

Now suppose that we decomposed this update into its two elementary constituents and executed them in some order. It is easy to see that any such decomposition would give us a different result, as the effects of the first action would disappear by the time the second was executed. For example, removing $C\_IsA(A, C)$ ($\neg C\_IsA(A, C)$) first, as an isolated operation, would remove $C\_IsA(A, B)$. Then adding the second part of the update ($C\_IsA(A, B)$) would revert the effects of the previous change and in fact re-construct the implicit IsA $C\_IsA(A, C)$ (due to $R11$). On the other hand, if we first added $C\_IsA(A, B)$, this would be a void change as this IsA already exists, leaving the ontology O unaltered. Thereinafter, removal of $C\_IsA(A, C)$, would remove $C\_IsA(A, B)$, leading to a wrong result.

Hence, there are operations which cannot be decomposed to their elementary constituents so as to be executed in a serialized way. A sceptical counterpoint to the above testimony would dictate to replace U with the removal of $C\_IsA(B, C)$ followed by the removal of $C\_IsA(A, C)$, thus giving the same results as U. However, we cannot determine this decomposition automatically. Furthermore, if we generalize this example so that the number of successive IsAs that form the implicit $C\_IsA(A, C)$ are n, we can see that while one can invent a number of heuristics capturing these circumstances, it is pointless to try to do so while n grows bigger. This means, that unless a sophisticated algorithm like ours handles the problem, storing operations into a library will never provide full support of composite updates. This is an intrinsic peculiarity of ad-hoc approaches.

Chapter 6

Implementation

Change does not roll in on the wheels of inevitability
but comes through continuous struggle.

Martin Luther King, Jr

6.1 General Setting: The SWKM

The Semantic Web Knowledge Middleware (a.k.a Manager) platform is a semantic management server-side stack, implemented in *Java* and *C++*. SWKM's purpose is to provide its users scalable middleware services for managing voluminous representations of Semantic Web data (schemata and data expressed in RDF/S). SWKM[1] is developed by ICS-FORTH institute, and partially supported by EU projects KP-Lab[2] and CASPAR[3]. The results of the current research have been implemented in the Evolution Service of the SWKM. Note that Evolution Service is also sometimes referred to as the Change Impact service in order to emphasize on the fact that in reality it doesn't apply the requested changes but just computes their side-effects (impacts) handing them to another service for implementation. The architecture of the SWKM is shown in Fig. 6.1.

For the purpose of giving the platform the maximum interoperability, it was chosen to offer all services as an array of SOAP-enabled[4] Web Services. All services depend directly or indirectly to the internal components shown in Fig. 6.1,

[1] http://athena.ics.forth.gr:9090/SWKM
[2] http://www.kp-lab.org//
[3] http://www.casparpreserves.eu/
[4] http://www.w3.org/TR/soap/

but a description of the SWKM framework goes out of the scope of this work. For a thorough discussion on SWKM as well as comparison to other Semantic Web middlewares and main memory RDF/S management systems refer to [4]. However since our service is relating upon the main memory model of SKWM we should mention a few things.

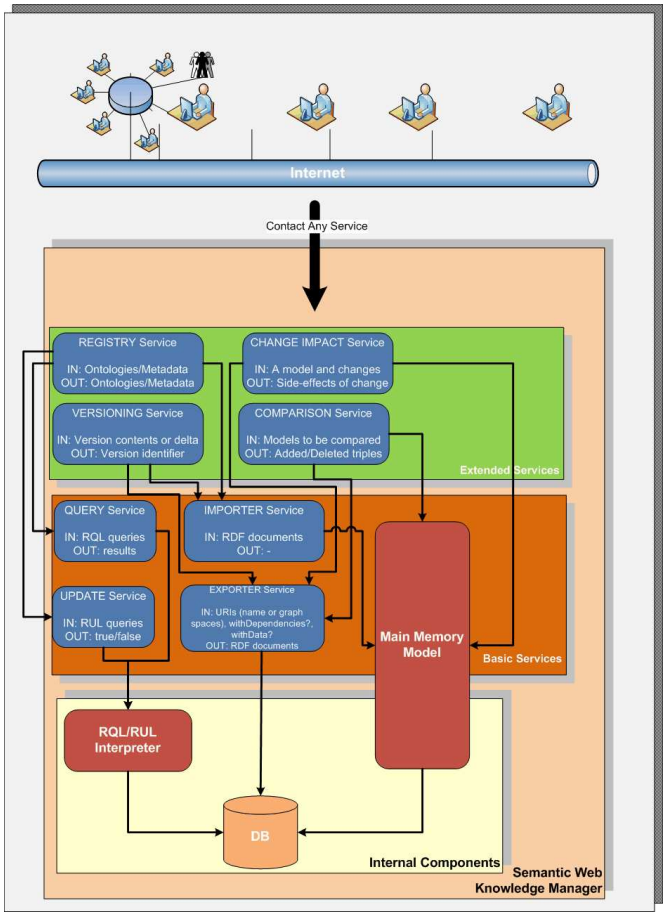

Figure 6.1: An architectural overview of the SWKM services

SWKM's model is the only main memory RDF/S management system that supports a fully-fledged object based view of RDF/S. This view allows typing information to be carried along with the objects themselves while provides object methods for storing and accessing RDF/S schemas and instances. Additionally, it offers higher abstractions to Semantic Web developers such as *Namespaces* (viewed as a container of classes and properties defined in a schema) and *Graphspaces* [12]

(viewed as a container of edges relating objects through various kinds of properties). In a nutshell, the *subject*, *predicate* or *object* of a triple in SWKM are Java objects whose state and type information is determined by the triples of an RDF/S model. Thus, SWKM supports seamlessly both triple and object views allowing to construct the latter from the former in a transparent to the user way.

6.2 The Evolution Service

The Evolution Service of the SWKM (shown in Fig. 6.1 under the name ChangeImpact service) is published as a webservice exposing a single service which is the one that applies the update request upon an RDF/S ontology. In effect, as mentioned the evolution service does not implement the changes, this is left to another service (which, by the time this document was written was the Exporter Service, although this choice is subject to change). The result of the evolution service is the estimation and report of the side-effects a given change produces on a specific ontology; not the materialization of these side-effects.

The ChangeImpact Service is internally written in *Java*. This module is responsible for determining the changes that should occur on a set of name or graph spaces in response to a change request. The change request comes in the form of a Delta, which basically encapsulates a pair of TRIG formatted strings, containing RDF/S triples. One file corresponds to the to-be-added triples and one to the to-be-deleted. A diagram showing the connection with the internal implementation of the ChangeImpact webservice is the one shown in Fig. 6.2.

The RDF/S ontology or KB is specified using any, arbitrarily large, collection of name and/or graph spaces, and this is the parameter $nameGraphSpaceURI$ of the service. The update request (delta) can affect any of the triples in this collection. However, the side-effects could potentially affect triples in other, depended or depending name or graph spaces. Therefore, the RDF/S knowledge base in this case is the union of all the triples that appear in all the name or graph spaces that are directly or indirectly depending (or are depended) on the given ones. Letting the side-effects of the update address the whole name /graph space "dependency" closure as was explained above corresponds to setting the mode parameter of ChangeImpact to $NamespaceClosure.ON$. In case $NamespaceClosure.OFF$ is chosen, ChangeImpact ignores side-effects concerning triples/resources with name or graph spaces out of the given collection.

The output of our service comes in the form of a pair of strings (or files); the first string represents the collection (set) of RDF/S triples that should be added to the ontology, whereas the second represents the collection (set) of triples that should be removed from the ontology, in order for the requested change to be

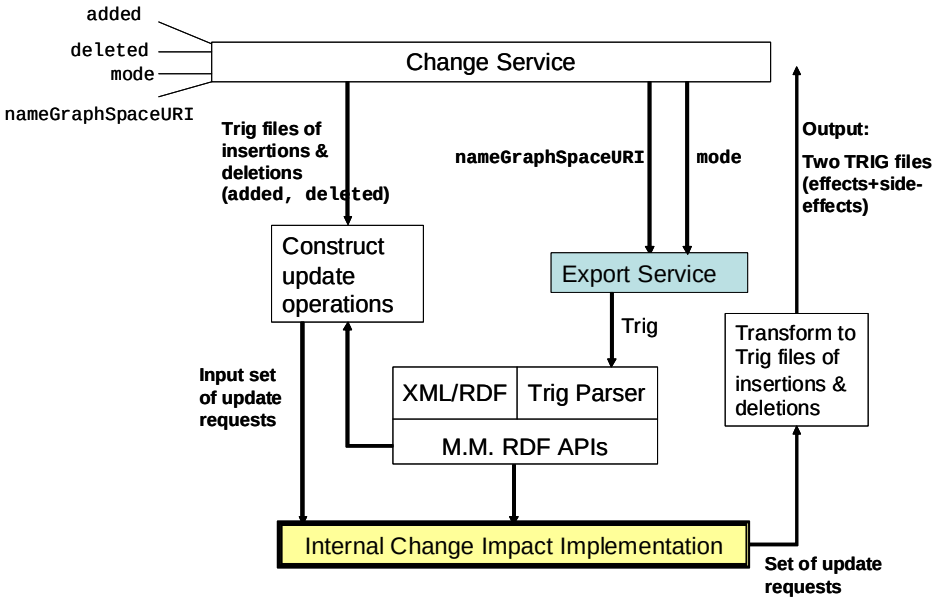

Figure 6.2: The Evolution or ChangeImpact service

applied (according to our algorithms and ordering). Both sets of triples are encoded into two strings which follow the TRIG syntax, so as to follow a standard, globally recognizable and re-usable format.

After the successful execution of the service, the output triples (in the form of another Delta object) include both the effects that were directly dictated by the original update request and the ones computed by our algorithm as so dictated by validity considerations, (i.e., the side-effects to be used in order to avoid introducing invalidities in the original ontology) due to the update request. All these triples are in fact primitive update operations (i.e., another update request) that captures all the effects and side-effects of the original change request upon the target KB. Void additions and removals have been filtered from the output.

Preconditions of the Service:

- The Collection of $nameOrGraphspaceURIs$ at input, contains all the different name or graph spaces that come in the triples contained in the input delta.
- Format is of the files in Delta is “TRIG”. Triples are fully-qualified.
- In cases that the update contains one of the following triples:

1. $P\ rdfs:subPropertyOf\ R$
2. $A\ rdf:type\ B$
3. $A\ rdfs:subClassOf\ B$

(with A,B,P,R fully qualified Uris)

Then at least one resource out of subject or object must have its URI contained in the update or the initial KB (unless the resources are part of RDF Schema namespace). This condition holds in order for the changeImpact mechanism to decide of the actual operation that is attempted. For example triple (a) could be attributed to adding an isa to a normal property or adding an isa to a data type property (that is a property with range = $rdfs:Literal$). These two operations have different semantics and therefore have to be distinguished. Similarly in case of triple (b) for example, the service needs to know if it is about an instantiation link between a class and a meta-class, or an instantiation link between a class instance and a class.

- The same is true for the triple::

  1. $P\ rdfs:domain\ A$

  (with P,A fully qualified Uris)

but, in this case, only the subject (P) might be ambiguous, an so is obligatory to have been specified.

Effects of the Service:

After the successful execution of the operation, the output will be a delta ready for set-theoretical addition/removal upon the initial sets of triples (if mode is OFF) or upon the "dependency" closure of these initial sets of triples (if mode is ON).

However in the case that successful execution of an update upon an initial RDF KB does not guarantee a consistent resulting KB, an $InfeasibleUpdateException$ is thrown to inform that the change requested cannot be applied to the name or graph spaces provided.

Changes demanding the addition of triples that already exist (or the deleting of triples that are already absent) are missing from the output delta, as they are void changes. Moreover, when an addition of a specific resource is confronted with the existence of this URI as a different resource, the user is notified by an exception, modeled with the $NameExistsAsDifferentObjectException$ object.

In addition, if the precondition that delta's URIs are included in the parameter $nameOrGraphspaceURIs$ is violated, the user is notified by the exception

named $UpdatingIllegalNameSpacesException$ is thrown. Also another exception named $NotEnoughInfoException$ is thrown in the violation of the precondition that some triples need to be related to other existing triples so as to disambiguate their meaning.

Moreover, one should note that the triples $\{A\ type\ rdfs:Class\}$, $\{P\ type\ rdf:Property\}$, when presented in the update in isolation (i.e., there is neither A nor P anywhere else in the update), then they are considered to be a normal Class (and not a meta-class) and a normal Property (with range a class) correspondingly.

In general, normal Classes and Properties are considered different and distinct from Meta-Classes (subclasses of rdfs:Class) and data type properties (properties with rdfs:Literal as their range), they have different semantics, treated separately and should not be confused. For example, an addition of the sole triple $\{P\ type\ rdf:Property\}$ to an KB is interpreted as the addition of a Property which can have only classes as range, so if the KB contains the triple $\{P\ rdfs:range\ rdfs:Literal\}$ then a $NameExistsAsDifferentObjectException$ is thrown.

Lastly, an $InvalidTriplesException$ is thrown in case a parsed triple falls in the following cases:

1. {X Y rdfs:Literal}, if Y exists in update and is not comment,label, or property
2. {X rdf:type rdfs:Class}, if X exists in update and is not a class
3. {X rdf:type rdf:Property}, if X exists in update and is not property
4. {X rdf:type rdfs:Resource}, if X exists in update and is not class instance
5. {X rdf:type Y}, if Y exists in update and is a class and X exists in update and is not a class instance
6. {X rdf:type Y}, if Y exists in update and is a metaclass and X exists in update and is not a class
7. {X rdf:type Y}, if X,Y exist in the update and are not resources (e.g. are literals)
8. {X rdf:type Y}, if Y exists in update and is a class instance
9. {X rdf:type Y}, if X exists in update and is a meta class
10. {X rdfs:domain A} if X is not property (normall or datatype) or A is not class (this should never break as X,A are immediately transformed to the necessary types by the model when we parse the trig files)

11. {X rdfs:range A} if X is not property or A is not a class or rdfs:Literal
12. {X rdfs:subclassOf rdfs:Class} if X exists in update and is not a class
13. {X rdfs:subclassOf X}
14. {P rdfs:subPropertyOf R} if P,R are not properties (this should never break as P,R are immediately transformed to the necessary types by the model when we parse the trig files)
15. {P rdfs:subPropertyOf P}
16. For any unsupported type of triple (for example rdf:member, list, reification;rdf:subject,rdf:object)

Signature of the Service:

The discussion above rounds up to the following signature:

```
Delta changeImpact(Delta delta, Collection <String>
nameGraphSpaceURI, NamespaceClosure mode) throws
InfeasibleUpdateException,
UpdatingIllegalNameSpacesException,
NotEnoughInfoException, InvalidTripleException,
NameExistsAsDifferentObjectException;
```

Interconnection of webservice with inner API

When the webservice parses a Delta object in order to apply it as an update (actually to estimate its side-effects), an $AddDelTriples$ object is constructed for internal use. The $ChangeImpact$ internal implementation receives in the input and returns to the output $AddDelTriples$ objects representing the sets of changes. In summary an $AddDelTiples$ object has two HashSets of $TripleWrapper$ objects. A $TripleWrapper$ is our representation of an RDF/S triple. Actually the main memory model already has a representation for RDF/S triples which is the the class RDF_Triple instantiating the interface $IRDF\_Triple$. However, in our case the semantics considered for RDF triples were a little bit more sophisticated than these of the model.

For example, when two main memory model $IRDF\_Triple$ objects are compared for equality this is done by reference. On the other hand we considered a value equality context. Towards this we constructed class $TripleWrapper$ which

contains (wraps) an $IRDF\_Triple$ object and is equipped with the following equality condition: Two $TripleWrapper$ objects are equal if the $IRDF\_Triple$ they contain is of the same type and their subject/object are equal using the semantics of $ResourceLiteralWrapper$. The first notice here is that a $TripleWrapper$ contains a reference to an $IRDF\_Triple$ and by having the latter's type we have the predicate of the RDF triple that these structures represent. The second notice is that the subject and object of the RDF triple (which could be accessible through the reference to the $IRDF\_Triple$) are also contained in the wrapper as two $ResourceLiteralWrapper$ objects.

Exactly symmetrically to the above rational we substituted the use of an $IRDF\_Resource$ object or $IRDF\_Literal$ object (which an $IRDF\_Triple$ could have has as subject and object) with the use of a $ResourceLiteralWrapper$. This was done for the same reasons, as while $IRDF\_Resource$ and $IRDF\_Literal$ carry by reference equality, our wrapper for both of these objects (which is represented by the $ResourceLiteralWrapper$) carries equality by value.

Notice that equality by value is somewhat obvious: in the case the former objects represent literals two different $ResourceLiteralWrapper$ objects are considered equal if their contained mm representation of a literal, i.e.,$IRDF\_Literal$ objects, have the same string content; in the case they represent other resources two any such $ResourceLiteralWrapper$ classes are equal if their underlying $IRDF\_Resources$ (a mm representation of a resource) have the same URI.

An $AddDelTriples$ object contains one set (of $TripleWrapper$ classes) for the to-be-added triples and one for the to-be-deleted. We excessively say the $AddDelTriples$ objects keep references to $IRDF\_Triples$ although in fact, it is the sets the former retain that do that. Based on the equality context within which we placed our objects, our object of representing a RDF triple, i.e., $TripleWrapper$ can check whether this triple belongs to a $IRDF\_Model$ (the main memory representation of SWKM for an RDF/S ontology).

This is useful as we want our internal representation of a change, (i.e., an $AddDelTriples$ object) to keep references to existing $IRDF\_Triple$ classes in the initial model (which is an $IRDF\_Model$); when a user requests the deletion of a triple and there is a main memory representation of this triple, we would like to keep a reference there, rather than constructing a second main memory representation of a triple which is "equals" to the first.

However sometimes it might be the case that a triple belonging to the update does not belong to the initial model. Therefore when we parse the Delta object given to the webservice we are faced with the problem of how to create an $AddDelTriples$ instance. The problem actually is risen because of the following conditions: the $AddDelTriples$ contains collections of $TripleWrapper$ classes, as already mentioned, and as also already told a $TripleWrapper$ contains

an $IRDF\_Triple$, and the fact is that is object is coupled with the existence of an $IRDF\_Model$. No main memory object representation of a triple can exist without being associated to a model.

At this point one could ask: why not keeping triads of strings in the memory, as that is what a RDF/S triple basically is? this is a rational question given that a lot of main memory RDF management systems keep triples of strings for representing ontologies. However, the benefits of an "object-view" representation of an RDF ontology have been already acknowledged, including among other memory speedups in answering queries and in reasoning. Yet, arguing in favor of such choices is out of the scope of this document; for a comparison of main memory RDF management systems as well as the benefits of an "object-view" representation refer to [4].

Coming back to our problem, we need to generate an $IRDF\_Model$ (which, considering that models an ontology, is a rather "heavy" construct) in order to have some unforseen triples referring to it. Instead of building a new $IRDF\_Model$ on demand (i.e., when we want construct a triple) we decided to initialize and keep throughout the entire life of a $ChangeImpact$ object, two instances of $IRDF\_Model$, namely the $added\_model$ and the $deleted\_model$. Obviously, $added\_model$ contains the to-be-added triples that exist in update but not in the initial model, and $deleted\_model$ the to-be-deleted triples for which the same holds.

Therefore, $ChangeImpact$ is initialized with three $IRDF\_Model$ objects, the initial upon which the update will be enforced, the $added\_model$ and the $deleted\_model$. We have already explained how the initial model is constructed through the arguments of the web service (the $nameGraphSpaceURI$ collection and the $NamespaceClosure$ mode); as far as the other models, when the $ChangeImpact$ service transforms the input $Delta$ object to $AddDelTriples$ it searches for the triples in the initial model, and when not found it adds them to the corresponding models. Fig. 6.3 depicts the objects introduced so far are shown. Fig. 6.4 depicts the objects so-far analyzed. It also depicts an object called $URIMap$ this is basically a map between resources that we should consider as equal. Note that current version is not taking advantages of this map, but it exists for future support.

A design choice that should be emphasized here, is the choice of maintenance of two different auxiliary models, instead of one; due to nature of our update such a choice is obligatory. As we have seen, sometimes, the existence of two triples together in the update is interpreted as something different than the existence of only one of them. With this in mind, suppose we had only one auxiliary main memory model instead of $added\_model$ and $deleted\_model$ and consider the example of a client of the service who might want to add class "A" but delete metaclass "A".

This is interpreted as an update which contains the triple $\{[Ardf : typerdfs :$

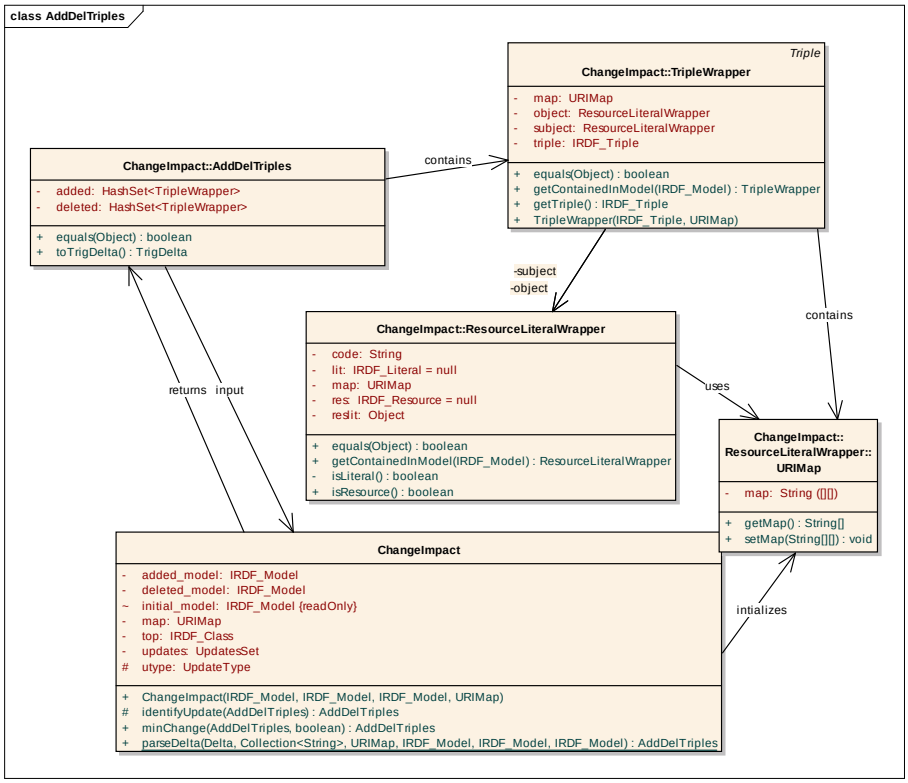

Figure 6.3: ChangeImpact: Building Blocks

$Class]\}$ in the added set and (at least) the triple $\{[Ardfs : subClassOfrdfs : Class]\}$ in the deleted set. Now if neither of these triples exists in the initial model and both these triples are added to the same auxiliary main memory model, the latter is going to end up having an object $IRDF\_MetaClass$ to represent concept "A", which shouldn't be the case. Thus, we cannot avoid having at least two auxiliary main memory models, one for the added and one for the deleted sets of triples. In other words, the two (added and deleted) sets of the update if unified onto a model might cause misinterpretations of each other's elements.

The main memory model has a certain consistency[5] context for an $IRDF\_Model$ to be properly functional. For example a constructed consistent model with a property P (for example in a triple $\{[Prdfs : domainrdfs : Literal]\}$ rdf)should always contain the triple $\{[Prdfs : subClassOfrdfs : Class]\}$. Implemen-

[5]The reader shouldn't confuse the term consistent here with the consistency nor the validity of KBs and RDF Graphs as presented in previous chapters

tation details of the RDF_Model, which is the only and basic implementation of the $IRDF\_Model$ interface, restrict the functionality of its methods to have as a precondition the "consistency" (in the terms explained), of the set of triples it contains. So, as we might use some method of the object $IRDF\_Property$ (corresponding to the above mentioned property P), we have to have the triple $\{[Prdfs : subClassOfrdfs : Class]\}$ in the model containing this method, in order to have guarantees that this method will work.

This additional triple might be included in the update all along or might be added importing-triples procedure of the model as it would be the case for example with property P above. In effect when we parse the triple $\{[Prdfs : subClassOfrdfs : Class]\}$ to the $deleted\_model$, the parsing procedure will generate and add also to the same model the $\{[Prdfs : subClassOfrdfs : Class]\}$. However, the AddDelTriple object should not contain both triples; only one is part of the update the other is there for main mm model's own consistency reasons. Fortunately, when adding a statement to a model this is marked with a flag $isInferred = false$; every $IRDF\_Triple$ not explicitly added (i.e., generated automatically) is marked with the $isInferred$ flag to be true. This way we differentiate the explicit and the inferred triples; the $AddDelTriples$ object contains only the explicitly added triples, which are not part of the initial model.

Thankfully, we can ourselves set the $inferred$ flag of a triple also; it might be the case that we want to add extra information to the update to clarify it. For example, consider that the initial model contains two meta-classes and the update contains the removal of a non-existing subsumption relation between them. As this triple doesn't exist we will build it and add it to the $deleted\_model$. However, the model would demand the declaration of the subject and object of this triple as metaclasses through a series (two in this case) of RDF statements; these statements will declare both subject and object as $rdfs : subClassOf$ of $rdfs : Class$. Note that here we should ourselves manually add this kind of information to the $deleted\_model$; a subsumption triple inserted to the model could be of several types, like a class or a property IsA. The algorithms dealing with each such case are, as we saw, different and so we add extra triples to make the model aware of of the type of the triple. Yet, as we stated although we add these triples explicitly we will manually set the $inferred$ flag to be true.

There are other cases, where the inferred triple, that is automatically inferred, is part of the update, or added by our module, is contained also in the initial model. Although this triple is contained in the initial model for consistency reasons it should belong also to the $added\_model$ or $deleted\_model$, which (although inferred) invalidates the reason we introduced the two auxiliary models all along, which was to hold non-existing triples.

At this point a detail, intentionally hidden until now for presentation purposes,

should be revealed: The two auxiliary models used to hold the references to the non-existing (in the initial model) triples, are in fact holding all the to be added or removed triples (even those that do exist in the initial model). Of course an $AddDelTriples$ object keeps references to the existing triples (i.e., the initial model) whenever possible, and references to the other models in case of non-existing triples. The rational on this choice is that as the auxiliary model is condemned to contain also existing-triples as described in the previous paragraph, it would obtain a nice property if we added the complete set of the update triples to it (actually the complete to-be-added set to the $added\_model$ and the complete to-be-deleted set to the $deleted\_model$).

This nice property regards a method which any $IRDF\_Model$ is equipped with, and which can serialize the content triples of a model to a string (formatted as TRIG or RDF-XML). Therefore, as the output of the webservice is a $Delta$ object (more specifically a $TrigDelta$ which is an implementation of a $Delta$) containing two Trig formatted strings, it is easy to print the two models directly to the two strings needed to construct the output. This means these models need to have all the triples of the update (as also as all the triples generated as side-effects which they would have anyway). The catch here, is that these models might contain inferred triples that we wouldn't want to be printed and returned. The solution came with a special version of the corresponding serializer of the model, which serializes to a string only the non-inferred triples.

Notice that through the above functionalities we could entirely bypass the $AddDelTriples$ object. However this object was kept for interoperability reasons as other components or services of the SWKM, which need to somehow model a "change" or a "delta", make use of this object, and compatibility with these components is desirable. In addition $AddDelTriples$ is a helpful structure containing the equality semantics and other secondary functions we require making very useful for the rest of the module.

6.3 Implementation Level Details

We will now turn our focus towards the internal implementation of the service (which is visually represented by the box at the bottom of Fig. 6.2), providing the architecture of this module, its intra and interconnections and also implementation details. This section is addressing to anyone interested in lightening the different implementation spots as also to developers who potentially want to "touch" the source code.

Dispatching the update

Essentially, the $ChangeImpact$ service is equipped with two major components which can be conceptually categorized as:

(a) The set of tools dealing with singular updates.

(b) The set of tools dealing with bulk updates.

The update operations that we need are singlefffmethod objects. They all implement the RDF_Update interface which has a method a method called $execute$. For example, one singular update operation is the $AddClass$ which looks somewhat like this:

```
Class AddClass implements RDF_Update{

AddDelTriples execute(AddDelTriples update_input,

RDF_Model initial_model) {.. }
..

}
```

In fact the $execute$ method takes also one more argument which we are not mentioning yet for simplicity reasons. In every RDF_Update object, the two seeable arguments of $execute$ correspond to the set with the demanding by the user changes on an initial model and the initial model itself. The output of $execute$ is also a set of changes, superset (in the most cases) of the initial.

By now, we can imagine that for an update operation, one could create a new RDF_Update object. For example $AddClass\ A\ =\ new\ AddClass()$; and then call the $execute$. A first remark is that actually several RDF_Update objects call execute methods of other objects and as a result we will be burdened with the generation of many new objects for each time a certain method is needed. This can be solved by the introduction of the Class $UpdatesSet$ and the dispatching technique.

In general the aim of employing dispatching techniques, is to retain a data structure or an object, containing references (or pointers) to certain pre-stored static methods. The data structure is usually a map with keys an identifer of the methods and values the methods themselves (or as in our case, objects which contain the methods). Plenty of further information on these techniques is available in the software engineering literature and on the Web.

So, the $UpdatesSet$ class basically holds a map of the RDF_Update objects. Thus, all the RDF_Update objects are generated once and "dropped" on a hashmap with key an identifier, (for example "ADD_CLASS") and value the object itself. Now, whenever we want to call $execute$ function of the $AddClass$ RDF_Update object, we simply ask the map (i.e., the $UpdateSet$) to give us the object that lies in the entry with key "ADD_CLASS" and hence we don't need to regenerate it. The keys of this hashmap are a enumeration named $UpdateType$.

Thus, an $UpdateType$ value is the "ADD_CLASS", while another one is for example the "$BULK$", which dispatches the execution of the program to the algorithms dealing with a bulk (or composite) update. The benefit of the dispatching methodology that we employ, is that we decouple the process of recognizing what kind of an update we are dealing with with the process of executing it upon the initial model. During the translation of the a $Delta$ to maim memory objects, like IRD_Triple and $AddDelTriple$ we identify a RDF_Update object that the delta represents but we don't want to call it yet and thus we can simply set the particular $UpdateType$. Later we just automatically call whichever $execute$ method is stored in the corresponding (to the saved $UpdateType$) entry in the $UpdatesSet$.

Therefore, a third argument on $execute$ is the set of updates with which we work, that is an $UpdatesSet$ with all our RDF_Update objects. So, when $execute$ of any RDF_Update wants to call, for example, the $AddClass$ execute, it will call this function on the object with $UpdateType =$ "ADD_CLASS", taken from the instance of $UpdatesSet$ that was parsed as an argument to the former. Thus, the signature of the method is:

```
AddDelTriples execute(UpdatesSetup set,
AddDelTriples update_input, RDF_Model initial_model)
```

With the employed technique, when necessary we could change a set of operations. For example, if there was a need to use a different functionality (or semantics) for adding a class, we would use another object (e.g., $AddClass1$) instead of $AddClass$, having a different implementation of $execute$. To do that we would substitute the value (of the $UpdatesSet$ instance we use) of the entry with key the $UpdateType$"ADD_CLASS", placing our new class $AddClass1$ where $AddClass$ was lying.

Consequently, we can always call $execute$ on a RDF_Update with a particular value of enumeration, thus knowing the "type" of the update, and without caring about the object (i.e., the implementation). Note that this way we could substitute only the objects holding the implementation of the semantics; plug them into the same framework, and they will work. Another usage would come up if we wanted to create a new operation; we would create a new object and put it in the

$UpdateSet$ that we are using, with key a new type of enumeration. We would need only to add code to appropriately setting the type of our update when recognized. The dispatching to the $execute$ would stay unaltered.

An auxiliary class to construct the $UpdatesSet$, is the class $UpdatesBuilder$. The main functionality of this object is the method:

```
UpdatesBuilder set(UpdateType ut,RDF_Update up);
```

This method "fills" the $UpdateSet$ with new types and objects. It returns the constructed set through another of its method, namely the method $build$. Example usage:

```
UpdatesBuilder upbuild = newUpdatesBuilder();
upbuild.set(UpdateType.ADD_CLASS, new AddClass());

UpdatesSet updates = upbuild.build();
```

Note that, for coding facility, the method set returns the $UpdatesBuilder$ object itself, so that we can perform all the "sets" in the same call, as:

```
UpdatesBuilder upbuild = new UpdatesBuilder();
upbuild.set(UpdateType.ADD_CLASS, new
AddClass()).set(UpdateType.ADD_RANGE,new AddRange());
```

After the aforementioned analysis, the comprehension of our central class object, $ChangeImpact$, is pretty straightforward: when a new $ChangeImpact$ is created, an $UpdatesBuilder$ loads all our update operations on the $UpdatesSet$ which will be used by all the update operations. The class $ChangeImpact$ provides methods for parsing a Delta object to construct the $AddDeltriples$ which represents the update and also functionalities for recognizing the type of the update. However, its main functionality is the method $minChange$ which executes the update on the initial model. The $minChange$ method is the interface of update operations with the outer world and calls the $execute$ of the proper RDF_Update object in order to implement the update operation that the client needs. Different update operations would simply mean "feeding" these operations to $UpdatesBuilder$ when initializing $ChangeImpact$. Fig. 6.4 depicts the objects so-far analyzed showing two of the singular updates and the bulk update (the others are similarly participating in the architecture).

Note that the $UpdatesSet$ is is dispatched to every update object, in case these objects want to use each other (in a lot of occasions as we saw in section 5.5 they do). In addition, in Fig. 6.4 you can notice a field of $UpdatesSet$

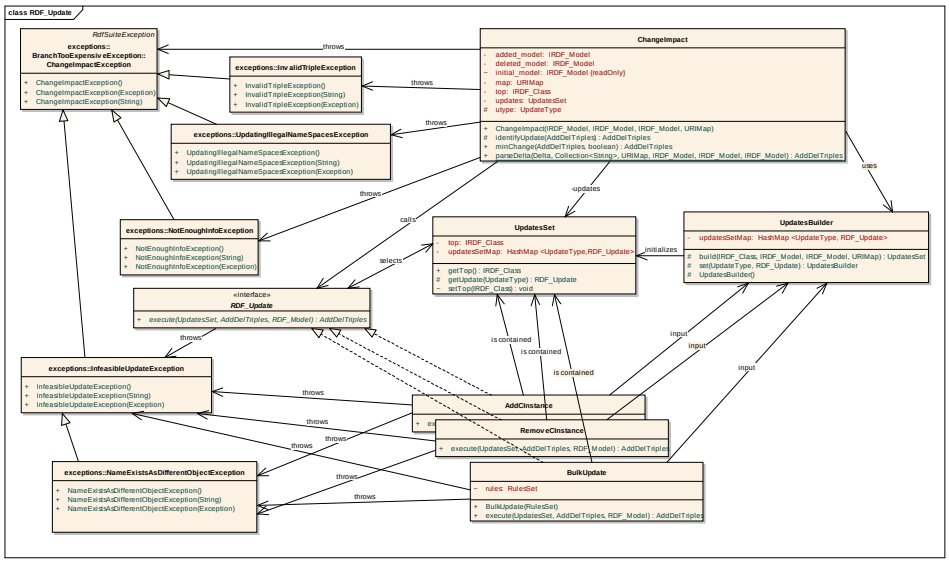

Figure 6.4: Dispatching the updates

named Top, that is an IRD_Class set by $ChangeImpact$ to represent the Top class of our algorithms, and propagate them to the update objects with the use of $UpdatesSet$. $ChangeImpcat$ passes this class to $UpdatesBuilder$ and in the case no class is set, the default top class is set to be the "rdfs:Resource". Fig. 6.4 also depicts the emergence of the several exceptions; these are the ones thrown either by the RDF_Update objects, in reaction to the operation attempted or by the $ChangeImpact$ object while parsing and identifying the update. The exceptions of our module, those which we have already explained and also some that are used only internally, are depicted in Fig. 6.5. The internal exception $BranchTooExpensiveException$ is used only by the implementation of our algorithm during the bulk update, in the process of pruning expensive branches. The other unmentioned exception $ComparingDifferentTriplesException$ shown in the figure, also used internally by the groundfact comparators for which we talk below.

6.3.1 Bulk Update

When the bulk update is executed (i.e., the $execute$ method of the RDF_Update object $BULK$), it takes use of an object called $GeneralUpdateFunction$ which models the logic of our main algorithm presented in section 4.3. Its main function is the recursive $update$. In the current version and the update function we

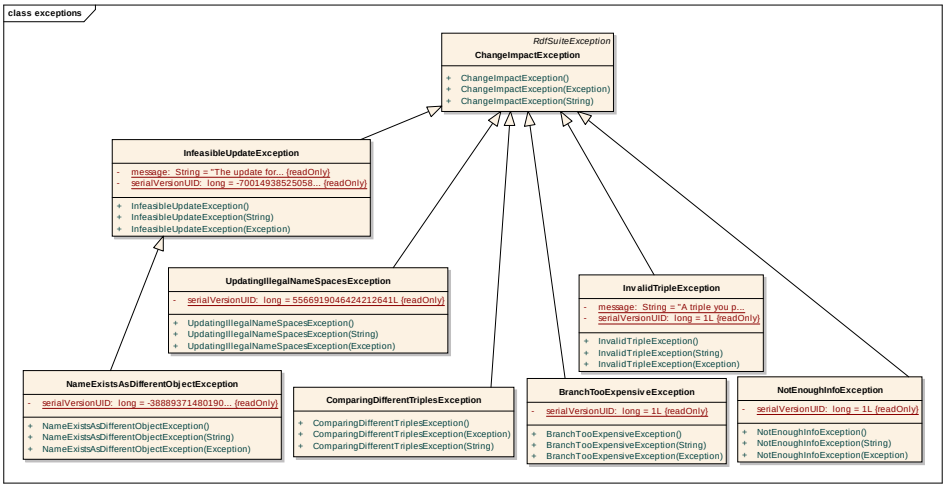

Figure 6.5: Exceptions

chose to use $Delta$ objects, instead of $AddDelTriples$; as the recursion goes into a respectable depth (as we have already discussed), and in order to lighten up the execution we chose to avoid the generation of a lot of "heavy" objects (such as $TripleWrapper$) in the large amount of branches that are going to end in infeasible. The price we pay for such a convention is that we are translating the $AddDelTriples$ input of the update function back to $Delta$ (which was the original form in the input of the web service). After the execution is returned we also transform the $Delta$ result of $update$ to $AddDelTriples$, as this what every RDF_Update must return, and therefore, also $Bulk$ update. Actually the next enhancement we are planning is to bypass entirely the translation of the input form $Delta$ to $AddDelTriples$ when it is about a bulk update. Note however that we can not get entirely rid of $AddDelTriples$ as our module is not only designed to serve the web service but possible also other inner modules of SWKM, which "speak" in $AddDelTriples$ terms. $GeneralUpdateFunction$ uses a number of auxiliary interfaces and classes, upon which we will elaborate in the following.

The variable B of the algorithm is being modeled with the $BestResultSoFar$ object. This object is set to keep the minimum (w.r.t to our ordering) delta (output) currently produced by the recursion. Bulk update initializes an instance of this object giving at as an argument to the constructor of $GeneralUpdateFunction$. This constructor takes also a $URISet$ instance, which is a multi-set of different sets of URIs. The latter hold all the URIs of the initial and auxiliary models, divided into several conceptual sets; $URIs$ maintains six hash sets of strings, namely *classes*,*properties*,*datatypeproperties*, *metaclasses*,*classinstances*,*literals*. As we

have discussed in step 4 our algorithm would not necessarily iterate over all the constants of our language (i.e., all the URIs) but only the existing ones. In addition when we look for instance for an existing class we shouldn't iterate over the whole set of URIs of our ontology but only the classes' URIs. Thus, $URISet$ help us optimizing this search (and step of the algorithm) keeping the existing URIs of each category.

Selection of the predicates

Recall that during the second step our algorithm arbitrarily selects a predicate out of the update. Practically, we chose to implement a somewhat parameterizable mechanism for choosing the predicate. We provided the interface $PredicateSelector$ which offers methods that sort that update set, put and take predicates from it. While the implementation of the $PredicateSelector$ we made for the particular version of our framework, namely the $RDFSuite\_PredicateSelector$ (the name is a leftover of the predecessor of SWKM suite), materializes "dummy" implementations of the methods of $PredicateSelector$, one is able to provide any alternative implementation. The naive implementation we provided simply picks up the next predicate from the update (it does not sort it nor uses any particular preference when inserting predicates to it). How to initialize our algorithms with a specific $PredicateSelector$ implementation will be discussed subsequently. Figure 6.6 shows the interactions of $Bulk$ update within our framework.

Looking closer to the bulk update in the diagram of Fig. 6.4 one could notice that the $Bulk$ update instance actually has a constructor, which is initiated with a $RulesSet$ object. Similar to $UpdatesBuilder$ in our module, there exists a class by the name $RulesBuilder$. This class helps $ChangeImpact$ build a map holding an implementation of the $Rule$ interface in entries where the keys are again values of the $UpdateType$ enumeration. The map constructed is the $RulesSet$.

Each $Rule$ object (i.e., implementation of the $Rule$ interface) corresponds to the set of integrity constraints, that should be counter examined for invalidation in reaction to the addition of removal of a single predicate. They are single-method objects implementing method $checkInvalidities$ containing the reasoning that needs to be done, in reaction to a predicate's addition or removal in step 4 of our main algorithm.

The dispatching technique is similarly used. When selecting a predicate in step 4 of $update$ function in $GeneralUpdateFunction$, we identify it to be of an $UpdateType$, e.g., ADD_CIsA; then the method $checkInvalidities$ is called upon any object is stored in the $RulesSet$ with key ADD_CIsA. This methods implements all the checks that are needed in step 4 when $C_IsA(x, y)$ is the object selected in step 3. Given the predicate and the $U \cup ESE$, $checkInvalidities$ of

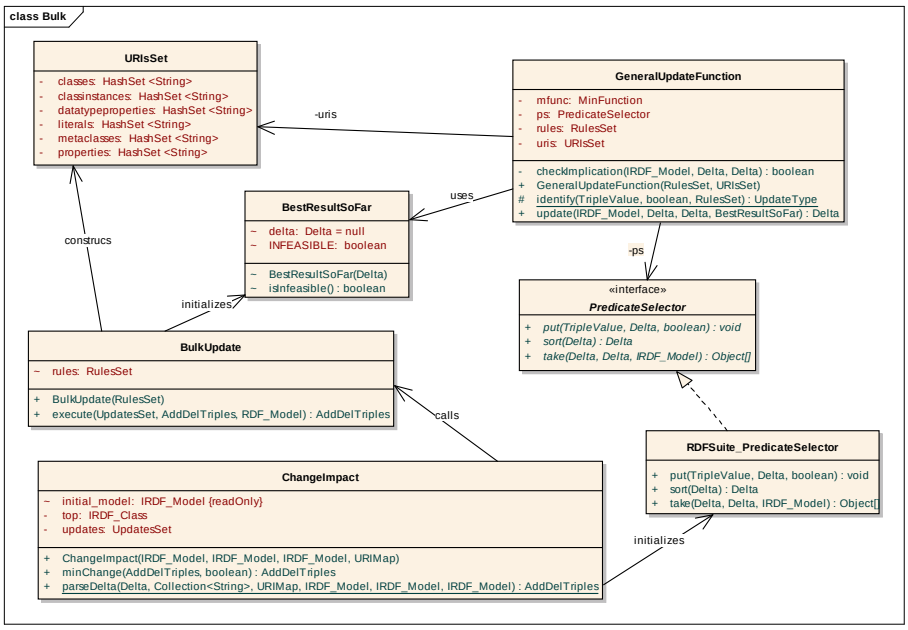

Figure 6.6: Bulk Update

the appropriate $Rule$ returns a set of side-effects for the specified action. If the side-effects are empty we proceed to step 5, else we go to step 6.

$RulesBuilder$ is called by $ChangeImpact$ to construct a $RulesSet$, which will be subsequently "feeded" to the constructor of the $Bulk$ instance. Therefore that point and $RulesSet$ are a suitable opportunity to encapsulate additional information from the $ChangeImpact$ that $Rule$ instances may need. For example $ChangeImpact$ initializes a particular $PredicateSelector$, namely the $RDFSuite\_PredicateSelector$, and hands it over to $RulesBuilder$ in order to be maintained in the $RulesSet$. The Top class, the $URIMap$, and the two auxiliary models are also kept in $RulesSet$. Having all this information the $RulesSet$ is parsed as an argument to the $checkInvalidities$ method of each $Rule$ object making these information available to the implementations of the rules.

In addition to the above, $RulesSet$ keeps a reference to the implementation of "min" function that we use. Interface $MinFunction$ provides the generic under which one can implement his own "min". Fig. 6.7 shows a diagram of the objects involved in the implementation of the $GeneralUpdateFunction$ (in effect, our algorithm's) steps. The dispatching technique that was used here also, offers the same advances that have already been discussed. One could easily implement a different reasoning (rule violation) for all (or any) predicate addition or removal.

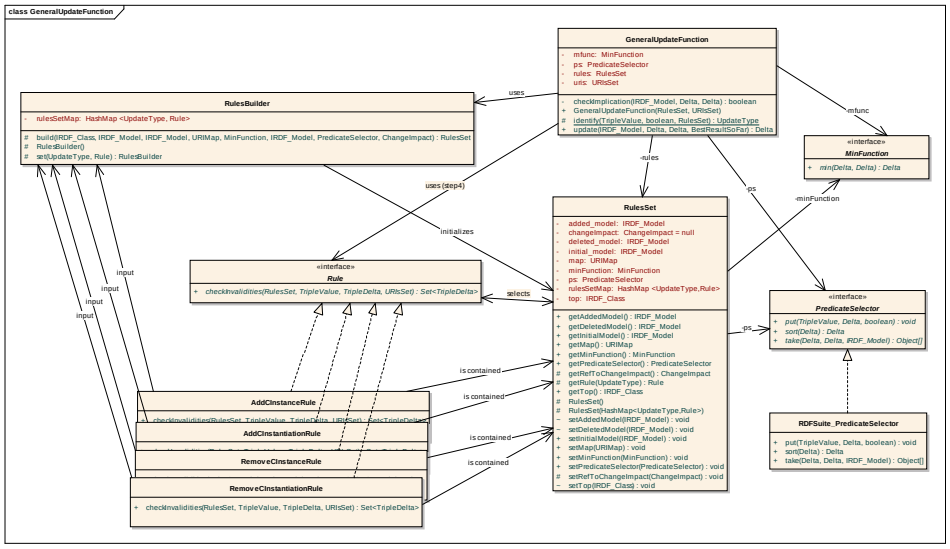

Figure 6.7: Update function

Function "min"

As we mentioned, the comparison functionality of our algorihtm's "min" is provided through implementations of the $MinFunction$ interface. Our implementation of $MinFunction$, namely $OurMinFunction$ is relying upon an ordering parameterization. At this point we should note that whatever implementation of the $MinFunction$ would fit in with rest of the framework without problems. However, using $OurMinFunction$ is not so restricting either as we have designed a very flexible mechanism. A designer has the ability to use $OurMinFunction$ with an ordering of his own just implementing the $Ordering$ interface.

The $Ordering$ interface was designed to be used with $OurMinFunction$, representing the $\leq_P$ and $\leq_G$ ordering. Nevertheless our implementation of $Ordering$, that is $OurOrdering$, encapsulates $\leq_P$ ordering and it remains still parameterizable to the $\leq_G$, providing even more flexibility and granularity in order for one to keep as many mechanisms as she likes and changing only a few things in order to tune the system otherwise. In addition to the above, the mechanism we constructed provides a very useful building-dispatching mechanism for somebody to load the $\leq_G$ ordering onto $OurOrdering$. Actually this same mechanism can be used also autonomously in order to compare two predicates in any context in the application.

The mechanism implementing $\leq_G$ ordering is the class $GroundFactOrdering$; this basically holds a map of $GroundFactComparator$ implementations. $GroundFactComparator$ is an interface allowing single-method implementa-

tions that compare two groundfacts. Fig. 6.8 shows the implementations we made for our $\leq_G$ of table 5.6. All our groundfact comparisons are supported with the five implementations shown in the figure. A $GroundFactOrderingBuilder$ is used by $OurOrdering$ is order to build the $GroundFactComparator$ instances into a map ($GroundFactOrdering$). This builder follows the same conventions as the other builders presented earlier and actually possesses the same advantages; based on the $UpdateType$ which is used again as key in $GroundFactOrdering$, the latter provides dispatched access to $GroundFactComparator$ objects where we automatically call $compareGroundFacts()$. Any implementation of $Ordering$ could use the $GroundFactOrderingBuilder$, to associate $UpdateType$ values with comparators.

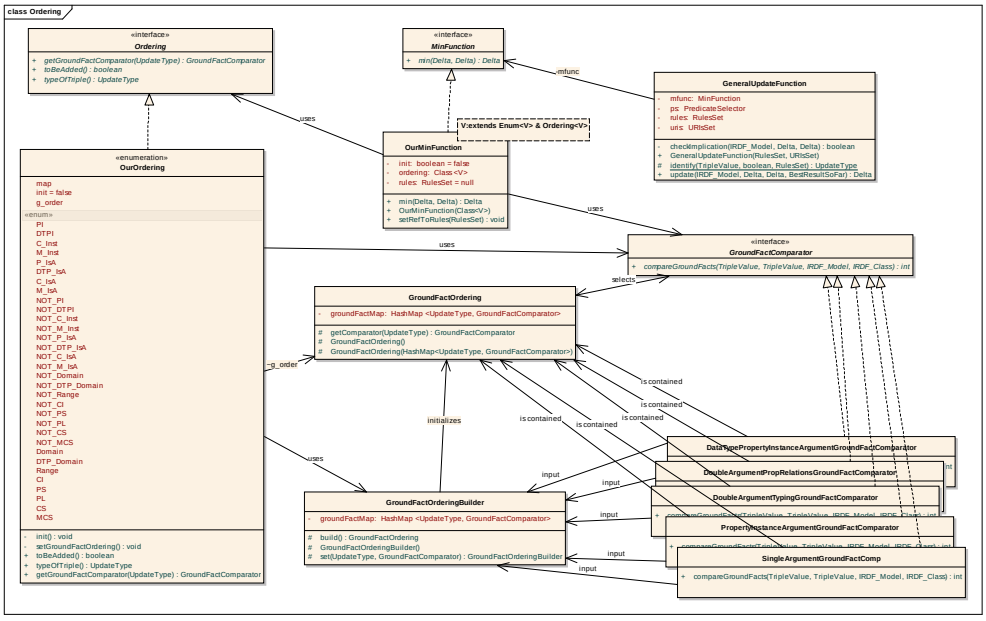

Figure 6.8: Implementing the “min” through Ordering

To sum up until here, $OurMinFunction$ computes the comparison between two deltas by advising the $\leq_P$ of $Ordering$, and for ties uses the appropriate comparator for $\leq_G$ comparisons that $Ordering$ automatically provides (in $OurOrdering$ implementation this happens through the $GroundFactOrdering$ the latter has builded). To automate the mechanism even more a “trick” was invented in order for any implementation of $Ordering$ to be able both tho provide a $\leq_P$ comparison and to represent an ordering in a “proper” way. The declaration of interface ordering is as following:

```
public interface Ordering<E extends Enum<E>&Ordering<E>>
```

With the generic use of Enum in the way showing, we can actually enforce any implementation of the $Ordering$ to be an enumeration (i.e., a Java "enum"). Thus, $OurOrdering$ has the following declaration:

```
public enum OurOrdering implements Ordering<OurOrdering>
```

Straight after the beginning of the class we can normally write down, the enumeration values we want for our ordering. This enumeration is also the $\leq_P$, and it can be used in a automatic way; we describe this in the following. $OurMinFunction$ has declaration:

```
public class OurMinFunction <V extends Enum<V> &
Ordering<V>> implements MinFunction{
```

Also $OurMinFunction$ has a field

```
Class <V> ordering
```

which is initialized through the constructor of $OurMinFunction$. Therefore $OurMinFunction$ can take any implementation of $Ordering$ and work with it. Such implementations will be basically simple enumerations with some obligatory implementations of some methods. One such method that any enumeration that implements $Ordering$ should provide is the method:

```
public GroundFactComparator getGroundFactComparator
(UpdateType type);
```

Hereafter the things are straightforward. $OurMinFunction$ takes the enum values of the implementation of the ordering provided to its constructor. This way it immediately has the $\leq_P$ ordering. Iterating over:

```
for(V o:ordering.getEnumConstants())
```

It iterates over the predicates of the language in a descending, w.r.t. our $\leq_P$, order. The enumeration provided as an implementation of $Ordering$ could be any arbitrarily, therefore this implementation, the enum should also implement another method of the $Ordering$ interface:

```
public UpdateType typeOfTriple();
```

This should be done in order to know which predicate corresponds to every enumeration value. For instance while iterating over all the enum constants in the above "for" we have the ability to:

```
UpdateType type = o.typeOfTriple();
```

thus having the type of the predicate in the current order of the iterator. Now, as every enum has also implemented the third method of $Ordering$, that is:

```
public GroundFactComparator getGroundFactComparator
(UpdateType type);
```

(with or without the help of $GroundFactOrderingBuilder$), we have immediate access to the corresponding comparator, calling automatically its method $compareGroundFacts$:

```
GroundFactComparator comp;
comp = o.getGroundFactComparator(type);
comp.compareGroundFacts(...);
```

6.3.2 Conclusion

Applicator

The output set of changes is ready to be raw "unified" with the initial model in order to give a result. The exact raw application of the result is out of the specifications of our part and we don't step into it. The only reference we are going to make here regarding the materialization of the changes is concerning the implementation of a special component which we constructed and which can serve as a third-party service between the ChangeImpact service and the one that is going to materialize the changes.

It a common phenomenon for the ontologies to be held in an "irredundant" form. We define such an *irredundant* RDF Graph as a belief set, as discussed in section 4.1.1, valid to all the rules of Table 5.3 except zero or more of: $R12$, $R14$, $R16$, $R17$, $R26$, $R27$, $R34$, $R36$ which deal with transitive relations and instance propagation. All these rules are common to most RDF Fragments used, and so often ignored but implied. Notice that all these rules are straightforwardly restored. For any combination of ground facts that violates one such rule, there is exactly one ground fact that could be added to restore it.

Although an alternative deletion of a ground fact might restore these rules, it is commonly accepted, as also encoded by our ordering, that the addition should be chosen. Consider for example two sequential IsAs; they imply a transitive one unless you delete one of them. Both common practise and our ordering imply tha addition of the transitive IsA as a restoration. This third IsA might be absent in an irredundant graph. Notice that we call a graph irredundant even if it does not violate all (or any) of these rules. In effect, with this definition any valid KB is an irredundant graph. Notice that as restoration of these rules is deterministic, we can eliminate some "information" knowing exactly what it is and how to produce it back.

We define the *closure* of an irredundant graph as "minimal" in terms of size (i.e., amount of predicates) KB which contains the graph (all its ground facts) and the positive ground facts needed for satisfying all the rules the graph didn't. This "transitive" closure is of course a unique valid KB which contains the graph; to produce it one should examine the rules and add only the necessary positive ground facts (not choose any alternative removals) in order to satisfy all rules. Notice that such a valid KB could have also other predicates, but we keep only the necessary in order for our graphs (the irredundant and the closure), to be "'semantically" or "transitive" equivalent. We define also an operator Cn which gives us the closure ginen an irredundant graph.

This special component this section is devoted to is the $UpdateApplicator$ which basically is a function $F()$ doing the following:

Suppose an update U and an initial ontology (RDF Graph or valid KB) K. Then for our algorithm, in particular $Update$ function, holds: $Update(K, U) = Update(K', U) \cup F(K', U)$, where K' is any irredundant graph, for which $Cn(K')$ = K. In simpler words, method $minChange$ of $ChangeImpact$ (which is the interface of the $Update$ function our algorithm) might be called to produce a delta given an update which is not closed with respect the above definition of closure; in that case in order to be flexible we don't reject the update but use $F(K', U)$ which computes the extra information that we would otherwise loose since (K', U) is irredundant. For example let $K' = \{C\_IsA(A, B), C\_IsA(B, C)\}$ and $U = \{\neg C\_IsA(A, B)\}$; as $ChangeImpact$ assumes that its initial ontology is a valid RDF Graph it will just report as delta the set $\{\neg C\_IsA(A, B)\}$. However after the raw application of this update on K' the transitive IsA will be gone for ever, and none could know it existed.

The output of F, in the above case is $\{C\_IsA(A, C)\}$, i.e., its work is to explicitly state the facts that are implied in an irredundant graph but are in danger to disappear due to an update. $Update$ would have no reason to include this IsA in its output; as this isa is not affected by any rules, so its deletion is not reported, it is assumed to continuing being in the graph.

The only object we haven't mentioned so far is the $GeneralAlgorithm$ object, which in fact is an auxiliary class to the whole project. It maintains several methods for use by different classes of the model, for example it keeps the $dist()$ functions or methods common to more than one RDF_Update objects. Concluding, we have build a fully flexible parameterizable and re-programmable framework for our algorithms. Part of the future work of this thesis, that is to evaluate our ordering, is now easier with the provided tool suite of $ChangeImpact$.

Chapter 7

Conclusions

Be the change you want to see in the world.

Mahatma Gandhi

In this work, we studied the problem of updating an ontology (or corpus of knowledge in general) in the face of new information. We criticized the currently used paradigm which consists of selecting a number of supported operations and determining the proper effects of each operation on a per-case basis; such an approach is tedious, ad-hoc, error-prone, inherently restrictive and unable to formally guarantee the "faithfulness" and "consistency" of the returned results or the exhaustiveness of the cases considered.

As an answer to this problem, we proposed a formal framework to describe updates and their effects, as well as a general-purpose algorithm to perform those updates, which is inspired by the general belief revision principles of Validity, Success and Minimal Change [34]. Our approach was based on a mapping of facts of a language (as RDF/S) into FOL ground facts (and KBs) interpreted under CWA. Two conditions were introduced for a "correct" update: success and validity; these conditions were formally defined using the standard CWA FOL inference and a set of FOL rules respectively. Success and validity may be satisfied by many different update results, so we devised a parameterizable way to determine the result that is "as close as possible" to the original RDF/S KB, following the principle of minimal change. Our methodology was based on determining the side-effects of an update operation onto a KB and then ordering the various possible update results on the basis of an ordering upon facts (and, consequently, upon sets of facts).

The end result is an algorithm that is highly parameterizable, both in terms of the language used and in terms of the implementation of the Principle of Minimal Change (through the ordering relation). We decomposed the process of coping

with ontology evolution into 5 discrete steps. This way, devising an ontology evolution algorithm is reduced to the process of instantiating each step in a modular way. To this end, the formal framework we presented offers a basis, with the aid of which an evolution algorithm can be materialized can be as a set of adequate parameterizations which are the following:

1. The model used and its mapping to FOL using the pattern described in [21].
2. The definition of the allowed operations in the model. Notice that this is not necessary, as the framework is general enough to support any update operation, but we may, for some reason, disallow certain "dangerous" or "unwanted" (application specific) operations, if any.
3. The consistency rules that encode the consistency model and allows us to detect inconsistencies, as well as to determine how the inconsistencies can be resolved.
4. The ordering that encodes the selection mechanism.

Once these parameters are set, the general algorithm presented in Table 4.1 can be directly used standing as an evolution algorithm. The main advantages of our method is that it exhibits a "faithful" behavior with respect to the various choices involved, regardless of the particular ontology or update at hand. It lies on a formal foundation, issuing a solid, consistent, transparent and customizable method to handle any type of operation upon any ontology, including unconsidered at design time operations. In addition, it avoids having to resort to the error-prone case-based reasoning of other systems, as all the alternatives regarding an update's side-effects can be derived from the language's rules themselves, in an exhaustive and provably correct manner. Our theory is modular in the sense that it could work with any language, rule and/or ordering given, (even though the implementation described in this study is applicable only for the particular setting.)

Exploiting our general algorithm's applicability to a variety of languages, we specified a set of parameters, namely the RDF/S language and an ordering of RDF/S facts, for which we evaluated and presented our framework. This case study set the proper parameter values for the RDF/S model of [51]; these parameters led to a number of special-purpose algorithms which provably exhibit behavior identical to the general-purpose one (so they are rational change operators), but also enjoy much better computational properties. In general, for efficiency reasons, it may be useful to generate simpler and more efficient case-special algorithms that "emulate" the behavior of the general algorithm. However, this can be done only for specific instantiations of the above parameters, as in the case study of RDF/S updating presented here.

We recently implemented our methodology for the purposes of the EU KP-Lab and CASPAR projects; our preliminary experimentation results are promising. Future work includes the incorporation of techniques and heuristics that could further speed up our algorithms as well as the verification of the effectiveness of our proposed ordering, by experiments with real users; we also plan to evaluate the feasibility of applying the same methodology in richer languages, like DLs or OWL.

Bibliography

[1] C. Alchourron, P. Gardenfors, and D. Makinson. On the Logic of Theory Change: Partial Meet Contraction and Revision Functions. *The Journal of Symbolic Logic*, 50(2):510–530, 1985.

[2] C. Alchourrón and D. Makinson. On the logic of theory change: Safe contraction. *Studia Logica*, 44(4):405–422, 1985.

[3] C. Alchourrón and D. Makinson. Maps between some different kinds of contraction function: The finite case. *Studia Logica*, 45(2):187–198, 1986.

[4] D. Andreou. Semantic Web Middlewares and Versioning Services. Master's thesis, University of Crete, 2007.

[5] G. Antoniou. *Nonmonotonic Reasoning*. MIT Press, 1997.

[6] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: A Reason-able Ontology Editor for the Semantic Web. *Ki 2001: Advances in Artificial Intelligence: Joint German/Austrian Conference on AI, Vienna, Austria, September 19-21, 2001: Proceedings*, 2001.

[7] S. Benferhat, S. Kaci, D. Le Berre, and M. Williams. Weakening conflicting information for iterated revision and knowledge integration. *Artificial Intelligence*, 153(1-2):339–371, 2004.

[8] D. Billington, G. Antoniou, G. Governatori, and M. Maher. Revising Nonmonotonic Theories: The Case of Defeasible Logic. *KI-99: Advances in Artificial Intelligence: 23rd Annual German Conference on Artificial Intelligence, Bonn, Germany, September 13-15, 1999: Proceedings*, 1999.

[9] A. Bochman. Entrenchment versus Dependence: Coherence and Foundations in Belief Change. *Journal of Logic, Language and Information*, 11(1):3–27, 2002.

[10] A. Bochman. Two Representations for Iterative Non-prioritized Change. *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning(NMR-02)*, 2002.

[11] S. Burris. *Logic for mathematics and computer science*. Prentice Hall Upper Saddle River, NJ, 1998.

[12] J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. *Proceedings of the 14th international conference on World Wide Web*, pages 613–622, 2005.

[13] M. Dalal. Investigations into a theory of knowledge base revision: Preliminary report. *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, 2:475–479, 1988.

[14] M. Dalal. Updates in propositional databases. Technical Report DCS-TR-222, Department of Computer Science, Rutgers University, 1988.

[15] A. Darwiche and J. Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 89(1-2):1–29, 1997.

[16] J. Delgrande and T. Schaub. A consistency-based approach for belief change. *Artificial Intelligence*, 151(1-2):1–41, 2003.

[17] J. Doyle. Rational belief revision (preliminary report). *Proceedings of the Second Conference on Principles of Knowledge Representation and Reasoning*, pages 163–174, 1991.

[18] H. Enderton. *A mathematical introduction to logic*. Academic Press New York, 1972.

[19] R. Fagin, J. Ullman, and M. Vardi. On the semantics of updates in databases. *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 352–365, 1983.

[20] G. Flouris. *On belief change and ontology evolution*. PhD thesis, University of Crete,Greece, 2006.

[21] G. Flouris. On the Evolution of Ontological Signatures. *Proceedings of the Workshop on Ontology Evolution*, 2007.

[22] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. Ontology change: Classification and survey. *Knowledge Engineering Review (KER) , to appear*.

[23] G. Flouris and D. Plexousakis. Belief Revision Using Table Transformation. Technical report, Technical Report FORTH-ICS, TR-290, July 2001.

[24] G. Flouris and D. Plexousakis. Belief Revision in Propositional Knowledge Bases. *Proceedings of The 8th Panhellenic Conference on Informatics, Nicosia, Cyprus*, 2001.

[25] G. Flouris and D. Plexousakis. On the Use of Matrices for Belief Revision. *Advances in Informatics: Post-proceedings of the 8th Panhellenic Conference in Informatics*, 2003.

[26] G. Flouris and D. Plexousakis. Handling Ontology Change: Survey and Proposal for a Future Research Direction. Technical report, Technical Report, ICS-FORTH, TR-362, 2005.

[27] G. Flouris and D. Plexousakis. Bridging Ontology Evolution and Belief Change. *Proceedings of the 4th Hellenic Conference on Artificial Intelligence (SETN-06)*, 2006.

[28] G. Flouris, D. Plexousakis, and G. Antoniou. AGM Postulates in Arbitrary Logics: Initial Results and Applications. Technical report, Technical Report FORTH-ICS/TR-336, April 2004.

[29] G. Flouris, D. Plexousakis, and G. Antoniou. Generalizing the AGM Postulates: Preliminary Results and Applications. *Proceedings of the 10th International Workshop on Non-Monotonic Reasoning*, pages 171–179, 2004.

[30] N. Foo. Ontology Revision. *Conceptual Structures: Applications, Implementation, and Theory: Third International Conference on Conceptual Structures, ICCS'95, Santa Cruz, CA, USA, August 14-18, 1995: Proceedings*, 1995.

[31] A. Fuhrmann. Theory contraction through base contraction. *Journal of Philosophical Logic*, 20(2):175–203, 1991.

[32] A. Fuhrmann and S. Hansson. A survey of multiple contractions. *Journal of Logic, Language and Information*, 3(1):39–75, 1994.

[33] T. Gabel, Y. Sure, and J. Voelker. KAON–ontology management infrastructure. *SEKT informal deliverable*, 3(1).

[34] P. Gärdenfors. Belief Revision: An Introduction. *Belief Revision*, 29:1–28, 1992.

[35] P. Gärdenfors. The dynamics of belief systems: Foundations versus coherence theories. *Knowledge, Belief, and Strategic Interaction*, 1992.

[36] P. Gärdenfors and D. Makinson. Revisions of knowledge systems using epistemic entrenchment. *Proceedings of the 2nd conference on Theoretical aspects of reasoning about knowledge*, pages 83–95, 1988.

[37] A. Grove. Two modellings for theory change. *Journal of Philosophical Logic*, 17(2):157–170, 1988.

[38] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[39] P. Haase and L. Stojanovic. Consistent Evolution of OWL Ontologies. *The Semantic Web: Research and Applications: Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29-June 1, 2005: Proceedings*, 2005.

[40] P. Haase and Y. Sure. D3.1.1.b state of the art on ontology evolution. Technical report, 2004.

[41] C. Halaschek-Wiener and Y. Katz. Belief base revision for expressive description logics. In *Proceedings of OWL: Experiences and Directions 2006 (OWLED-06)*, 2006.

[42] S. Hansson. In defense of base contraction. *Synthese*, 91(3):239–245, 1992.

[43] S. Hansson. Theory Contraction and Base Contraction Unified. *The Journal of Symbolic Logic*, 58(2):602–625, 1993.

[44] S. Hansson. Knowledge-level analysis of belief base operations. *Artificial Intelligence*, 82(1-2):215–235, 1996.

[45] S. Hansson. Revision of Belief Sets and Belief Bases. *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, pages 17–25, 1998.

[46] S. Hansson. A Survey of non-Prioritized Belief Revision. *Erkenntnis*, 50(2):413–427, 1999.

[47] J. Heflin, J. Hendler, and S. Luke. Coping with Changing Ontologies in a Distributed Environment. *AAAI Conference Ontology Management Workshop*, pages 74–79, 1999.

[48] Z. Huang and H. Stuckenschmidt. Reasoning with multi-version ontologies: A temporal logic approach. *The Semantic Web-ISWC 2005: 4th Int'l Semantic Web Conf.*

[49] A. Hunter and J. Delgrande. Iterated Belief Change: A Transition System Approach. *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.

[50] S. Kang and S. Lau. Ontology Revision Using the Concept of Belief Revision. *Proceedings of the 8 thInternational Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES-04), part*, 3:8–15, 2004.

[51] G. Karvounarakis, A. Magkanaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. RQL: A Functional Query Language for RDF. *The Functional Approach to Data Management*, 2004.

[52] H. Katsuno and A. Mendelzon. Propositional knowledge base revision and minimal change. *Artificial Intelligence*, 52(3):263–294, 1991.

[53] H. Katsuno, A. Mendelzon, J. Allen, R. Fikes, and E. Sandewall. On the Difference Between Updating a Knowledge Base and Revising It. *KR'91: Principles of Knowledge Representation and Reasoning*, pages 387–394, 1991.

[54] K. Kelly. The learning power of belief revision. *Proceedings of the 7th conference on Theoretical aspects of rationality and knowledge*, pages 111–124, 1998.

[55] M. Klein and D. Fensel. Ontology Versioning on the Semantic Web. *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 75–91, 2001.

[56] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, pages 197–212, 2002.

[57] M. Klein and N. Noy. A component-based framework for ontology evolution. *Workshop on Ontologies and Distributed Systems at IJCAI*, 2003.

[58] S. Konieczny. On the difference between merging knowledge bases and combining them. *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, pages 135–144, 2000.

[59] S. Konieczny and R. Pino Pérez. Propositional belief base merging or how to merge beliefs/goals coming from several sources and some links with social choice theory. *European Journal of Operational Research*, 160(3):785–802, 2005.

[60] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. On rdf/s ontology evolution. In *To appear In LNCS Post-Proceedings of the Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases*, 2007.

[61] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. Ontology evolution: A framework and its application to rdf. In *Proceedings of the Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases (SWDB-ODBIS-07)*, 2007.

[62] K. Lee and T. Meyer. A Classification of Ontology Modification. *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence (AI-04)*, pages 248–258, 2004.

[63] P. Liberatore. The Complexity of Iterated Belief Revision. *Database Theory–ICDT'97: 6th International Conference, Delphi, Greece, January 8-10, 1997: Proceedings*, 1997.

[64] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. An infrastructure for searching, reusing and evolving distributed ontologies. *Proceedings of the 12th international conference on World Wide Web*, pages 439–448, 2003.

[65] D. Makinson. How to give it up: A survey of some formal aspects of the logic of theory change. *Synthese*, 62(3):347–363, 1985.

[66] E. Mendelson. Introduction to Mathematical Logic. Wadsworth & Brooks. *Cole, Monterey*, 1987.

[67] T. Meyer, K. Lee, and R. Booth. Knowledge Integration for Description Logics. *Proceedings of the 7th International Symposium on Logical Formalizations of Commonsense Reasoning*, 2005.

[68] S. Munoz, J. Perez, and C. Gutierrez. Minimal deductive systems for rdf. In *Proceedings of the 4^{th} European Semantic Web Conference*, 2007.

[69] A. Nayak. Iterated belief change based on epistemic entrenchment. *Erkenntnis*, 41(3):353–390, 1994.

[70] B. Nebel. A knowledge level analysis of belief revision. *Proceedings of the first international conference on Principles of knowledge representation and reasoning table of contents*, pages 301–311, 1989.

[71] N. Noy, R. Fergerson, and M. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. *Lecture Notes in Artificial Intelligence (LNAI)*, 1937:17–32.

[72] N. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440, 2004.

[73] N. Noy and M. Musen. An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support. *Proceedings of the Workshop on Ontology Management at the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 1999–0799, 1999.

[74] J. Pan and I. Horrocks. Metamodeling Architecture of Web Ontology Languages. *Proceedings of the Semantic Web Working Symposium*, 149, 2001.

[75] H. Pinto, A. Gomez-Perez, and J. Martins. Some issues on ontology integration. *IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5)*, 1999.

[76] P. Plessers and O. De Troyer. Ontology change detection using a version log. *Lecture notes in computer science*, pages 578–592.

[77] G. Qi, W. Liu, and D. Bell. Knowledge base revision in description logics. In *Proceedings of the 10^{th} European Conference on Logics in Artificial Intelligence (JELIA-06)*, 2006.

[78] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *Proceedings of the 4^{th} International Semantic Web Conference (ISWC-05)*, 2005.

[79] E. Sosa. The raft and the Pyramid: Coherence Versus Foundations in Theory of Knowledge. *Midwest Studies in Philosophy*, 1980.

[80] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW*, 2002.

[81] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. *Proceedings of the 2nd international conference on Knowledge capture*, pages 162–171, 2003.

[82] L. Stojanovic and B. Motik. Ontology Evolution within Ontology Editors. *Proceedings of OntoWeb-SIG3 Workshop*, pages 53–62, 2002.

[83] H. Stuckenschmidt and M. Klein. Integrity and change in modular ontologies. *Proceedings of the International Joint Conference on Artificial Intelligence-IJCAI"03*, pages 900–905.

[84] G. Stumme and A. Maedche. Ontology Merging for Federated Ontologies on the Semantic Web. *Proceedings of the International Workshop for Foundations of Models for Information Integration (FMII-2001)*, pages 413–418, 2001.

[85] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative ontology development for the semantic web. In *Proceedings of the first International Semantic Web Conference 2002 (ISWC 2002), June 9-12 2002, Sardinia, Italia.* Springer, LNCS 2342, 2002.

[86] R. Wassermann. Revising concepts. *Proceedings of the Fifth Workshop on Logic, Language, Information and Comunication (WoL-LIC), Sao Paulo*, 1998.

[87] A. Weber. Updating propositional formulas. *Proceedings First Conference on Expert Database Systems*, pages 487–500, 1986.

[88] M. Williams. Applications of Belief Revision. *Transactions and Change in Logic Databases, Lecture Notes in Artificial Intelligence (LNAI)*, 1472.

[89] D. Zeginis, Y. Tzitzikas, and V. Christophides. On the Foundations of Computing Deltas Between RDF Models. *Proceedings of the sixth International Semantic Web Conference 2007 (ISWC 2007), November 11-15 2007, Busan, Korea.*

APPENDIX A:

ALGORITHMS FOR SINGULAR UPDATES

F1: Add a Class: U={CS(A)}

If name A already exists not as a class:

Return INFEASIBLE

If Class A doesn't exist

Make it a subclass of ⊤

F2: Add a Property: U={PS(P)}

If name P already exists not as a property:

Return INFEASIBLE

If property P doesn't exist

Insert a new Property P, with domain and range ⊤.

F3: Add a New Class Instance: U={CI(x)}

If name x exists as some other object

Return INFEASIBLE

If CI(x) doesn't exist:

Insert it as an instance of ⊤.

F4: Insert (Change) Domain U={Domain(P,A)}

If the relationship doesn't already exist:

Do what the F2 function does for PS(P)

- (DO NOT CALL F2 – DON'T ARRANGE 'T' AS DOMAIN)

Call the F1 function for CS(A)

- If the F1 function returns INFEASIBLE, return INFEASIBLE

Remove the old Domain WITHOUT calling the corresponding function (you don't want to erase the property instances for example)

Add the new Domain.

If P is a subproperty/superproperty of another property (say Q), verify that the new Domain is a subclass a subclass/superclass (respectively) of the Domain of Q.

- If not, attempt to add an IsA between the respective Domains
  - may not always be possible using only a single addition of an IsA because maybe it would cause cyclic IsAs. If this is the case don't call the F6 but remove the subsumption relation between P and Q instead, (NOT by calling the F23 which might destroy an implicit IsA in way different than it might already being destroyed, by execution of this very bullet several times).

        - Else add the IsA through F6 (will not exploit all its functionality as we know we don't have a cycle)

If P has an instance of a property instance (say Q), verify that the new Domain has as its instance the source of Q.

- If not, add an instance relationship to A by calling F8

F5: Insert(Change) Range U={Range(P,A)}

If the relationship doesn't exist:

If PS(P) doesn't exist:

- Add it calling the F2 function.
- If the F2 function returns INFEASIBLE, return INFEASIBLE

If CS(A) doesn't exist:

- Add it calling the F1 function.
- If the F1 function returns INFEASIBLE, return INFEASIBLE

Remove the old Range WITHOUT calling the corresponding function (you don't want to erase the property instances for example)

Add the new range.

If P is a subproperty/superproperty of another property (say Q), verify that the new range is a subclass a subclass/superclass (respectively) of the range of Q.

- If not, attempt to add an IsA between the respective ranges
    - may not always be possible using only a single addition of an IsA because maybe it would cause cyclic IsAs. If this is the case don't call the F6 but remove the subsumption relation between P and Q instead, not by calling the F23
    - Else add the Isa through F6.

If P has an instance a property instance (say Q), verify that the new range has as its instance the target of Q.

- If not, add an instance relationship to A by calling F8. (will not use all functionality of F8).

F6: Add Class Subsumption U={C_IsA(A,B)}

If there is not an IsA between A and B:

If CS(A) doesn't exist:

- Add it calling the F1 function.
- If the F1 function returns INFEASIBLE, return INFEASIBLE

If CS(B) doesn't exist:

- Add it calling the F1 function.
- If the F1 function returns INFEASIBLE, return INFEASIBLE

Remove C_IsA (B, A) through F22
Add the class subsumption relation between A and B.
Make every subclass of A, a subclass of B
Make every superclass of B, a superclass of A
Make every instance of A, an instance of B

F7: Add Property Subsumption (i.e., U={P_IsA(P,Q)})
If there is not an IsA between P and Q:
If PS(P) doesn't exist:
- Add it calling the F2 function.
- If the F2 function returns INFEASIBLE, return INFEASIBLE

If PS(Q) doesn't exist:
- Add it calling the F2 function.
- If the F2 function returns INFEASIBLE, return INFEASIBLE

Remove P_IsA (Q, P) through F23:
Add the property subsumption relation between P and Q.
Add a class IsA relationship between the domain of P and the domain of Q through F6 (return infeasible if F6 returns infeasible)
Add a class IsA relationship between the range of P and the range of Q through F6 (return infeasible if F6 returns infeasible)
Make every subproperty of P, a subproperty of Q
Make every superproperty of Q, a superproperty P
Make every property instance of B, a property instance of Q.

F8: Add Class Instantiation: U={C_Inst(x,A)}
If the relationship doesn't exist:
If CI(x) doesn't exist:
- Add it calling the F3 function.
- If the F3 function returns INFEASIBLE, return INFEASIBLE

If CS(A) doesn't exist:
- Add it calling the F1 function.
- If the F1 function returns INFEASIBLE, return INFEASIBLE

Add the class instantiation relation between x and A
Add any instantiations that follow from this due to transitivity
(∀x,y,z for which it holds C_Inst(x,y) AND C_IsA(y,z) add C_Inst(x,z))

F9: Add Property Instantiation U={PI(x,y,P)}
If there is not a P_Inst relation between p and P:
If CI(x) doesn't exist:
- Add it calling the F3 function.
- If the F3 function returns INFEASIBLE, return INFEASIBLE

If CI(y) doesn't exist:
- Add it calling the F3 function.
- If the F3 function returns INFEASIBLE, return INFEASIBLE

If PS(P) doesn't exist:

- Add it calling the F2 function.
- If the F2 function returns INFEASIBLE, return INFEASIBLE

Add the property Instance relation PI(x,y,P).

Add a Class Instance relationship between the source, x, and the domain of P through F8 (return infeasible if F8 returns infeasible)

Add a Class Instance relationship between the target, y, and the range of P through F8 (return infeasible if F8 returns infeasible)

Add a property Instance to every one of P's superpoperties

F10: Add a DatatypeProperty: U={PL(P)} (this means a property with range Literal)

If name P already exists not as a datatype property:

Return INFEASIBLE

If property P doesn't exist

Insert a new Property P, with domain $\top$, and range the 'rdfs:Literal'

F11: Insert (Change) Domain of a DataType Property U ={DataTypeDomain(P,A)}

If the relationship doesn't already exist:

Do what the F10 function does for PL(P)

Call the F1 function for CS(A)

- If the F1 function returns INFEASIBLE, return INFEASIBLE

Remove the old DTDomain WITHOUT calling the corresponding function (you don't want to erase the property instances for example)

Add the new DTDomain.

If P is a subproperty/superproperty of another data type property (say Q), verify that the new DTDomain is a subclass a subclass/superclass (respectively) of the DTDomain of Q.

- If not, attempt to add an IsA between the respective DTDomains
  - may not always be possible using only a single addition of an IsA because maybe it would cause cyclic IsAs. If this is the case don't call the F6 but remove the subsumption relation between P and Q instead, (NOT by calling the F23 which might destroy an implicit IsA in way different than it might already being destroyed, by execution of this very bullet several times).
  - Else add the IsA through F6 (will not exploit all its functionality as we know we don't have a cycle)

If P has a datatypeproperty instance (say Q), verify that the new DTDomain has as its instance the source of Q.

- If not, add an instance relationship to A by calling F8

F12: Add Datatype Property Subsumption (i.e., U={DataP_IsA(P,Q)})

If there is not an IsA between P and Q:

If PL(P) doesn't exist:

- Add it calling the F10 function.
- If the F10 function returns INFEASIBLE, return INFEASIBLE

If PL(Q) doesn't exist:

- Add it calling the F10 function.
- If the F10 function returns INFEASIBLE, return INFEASIBLE

Remove DataP_IsA (Q, P) through F23:

Add the dtproperty subsumption relation between P and Q.

Add a class IsA relationship between the domain of P and the domain of Q through F6 (return infeasible if F6 returns infeasible)

Make every subproperty of P, a subproperty of Q

Make every superproperty of Q, a superproperty P

Make every property instance of P, a property instance of Q.

F13: Add a DatatypePropertyInstance: U={ DataP_Inst (x,y,P)}

If there is not a DataP_Inst relation between p and P:

If CI(x) doesn't exist:

- Add it calling the F3 function.
- If the F3 function returns INFEASIBLE, return INFEASIBLE

If PL(P) doesn't exist:

- Add it calling the F10 function.
- If the F10 function returns INFEASIBLE, return INFEASIBLE

Add the property Instance relation PI(x,y,P).

Add a Class Instance relationship between the source, x, and the domain of P through F8 (return infeasible if F8 returns infeasible)

Add a datatype property Instance to every one of P's superpoperties

F14: Add a MetaClass: U={MCS(A)}

If name A already exists not as a class:

Return INFEASIBLE

If Class A doesn't exist

Add it

F15: Add MetaClass Subsumption U={M_IsA(A,B)}

If there is not an IsA between A and B:

If MCS(A) doesn't exist:

- Add it calling the F14 function.
- If the F14 function returns INFEASIBLE, return INFEASIBLE

If MCS(B) doesn't exist:

- Add it calling the F14 function.
- If the F14 function returns INFEASIBLE, return INFEASIBLE

Remove M_IsA (B, A) through F31

Add the class subsumption relation between A and B.

Make every subclass of A, a subclass of B

Make every superclass of B, a superclass of A

Make every instance of A which are classes, an instance of B

F16: Add MetaClass Instantiation: U={M_Inst(x,A)}

If the relationship doesn't exist:

If CS(x) doesn't exist:

- Add it calling the F1 function.
- If the F1 function returns INFEASIBLE, return INFEASIBLE

If MCS(A) doesn't exist:

- Add it calling the F14 function.
- If the F14 function returns INFEASIBLE, return INFEASIBLE

Add the meta class instantiation relation between x and A

Add any instantiations that follow from this due to transitivity

(∀x,y,z for which it holds M_Inst(x,y) AND M_IsA(y,z) add M_Inst(x,z))

F17: Remove Class U={¬CS(A)}

If there exists CS(A)

Remove all IsAs starting from A.

Remove all IsAs ending in A.

Remove all instantiation links ending in A.

Remove all properties (including datatype) whose range/domain is A by calling F18 or F26.

Remove A.

F18: Remove Property U={¬PS(P)}

If there exists PS(P)

Remove all IsAs starting from P.

Remove all IsAs ending in P.

Remove all instantiation links ending in P.

Remove P and the information on its range/domain.

F19: Remove Class Instance x U={¬CI(x)}

If there exists CI(x)

Remove all classification links starting from x.

Remove all property instances (including datatype) whose target/source is x.

Remove x.

F20: Delete Domain U={¬Domain(P,A)}

Equivalent to removing the relevant property

F21: Delete Range U={¬Range(P,A)}

Equivalent to removing the relevant property

F22: Remove Class Subsumption (i.e., U={¬C_IsA(A,B)})

If IsA from A to B exists:

If B=⊤, call F17 to remove class A.

Remove the direct class subsumption relation (IsA) between A and B:

Find all sequences of IsA that start from A and end in B. Select one IsA from each sequence for removal. The selection should be made so as to select the fewest IsAs possible; in the case of ties, we choose the solution which includes the IsAs that are furthest from the top. If again we have ties, favour the solution that has the "cheapest" IsA among all.

To determine the cheapest we use the distance (Dist function) of the first parameter from the top; the larger the distance, the cheaper the ground fact. If this distance is identical, use the distance (Dist function) of the second parameter from the top; the smaller the distance, thecheaper the ground fact. If both distances are identical, use lexicographic ordering ($\leq_{lex}$) on the first parameter. If both distances are identical and the first parameter is also identical, use lexicographic ordering ($\leq_{lex}$) on the second parameter.

In addition, for each of the IsAs selected for removal in the above steps (for each direct and for each of the indirect IsAs that were removed due to the above process and for the initial IsA from A to B that we were asked to remove) do the following:

- Determine whether there is any property (including datatype properties) (say P) whose range/domain (only domain for datatype properties) is the source of the IsA and P is subsumed by a property (say Q) whose range/domain (respectively) is the target of the IsA;
- For each such property, remove the subsumption link between P and Q.

F23: Remove Property Subsumption (i.e., U={¬P_IsA(P,Q)})

If IsA from P to Q exists:

Remove the direct subsumption relation (IsA) between P and Q:

Find all sequences of IsA that start from P and end in Q. Select one IsA from each sequence for removal. The selection should be made so as to select the fewest IsAs possible; in the case of ties, we choose the solution which includes the IsAs that are furthest from the top class. . If again we have ties, favour the solution that has the "cheapest" IsA among all, as explained in F22.

F24: Remove Class Instantiation (i.e., U={¬C_Inst(x,A)})

If there is an Instantiation link between x and A

If A=⊤, remove of the class instance x, calling F19

Remove the instantiation link between x and A:

- Remove every C_Inst from x to a subclass of A.

Check if there is any property instance ((including datatype property instances) containing P such that:

- The source/target (respectively) is x
- The range/domain (respectively) of P is A or another subclass of A, which until the previous step was instantiated by x.

If there are such properties, remove the property instantiation containing them.

F25: Remove Property Instantiation U={¬PI(x,y,P)}

If there is an Instantiation link containing P

Remove the property instantiation

Remove every PI containing a subproperty of P (and x,y)

F26: Remove DataTypeProperty U={¬PL(P)}

If there exists PL(P)

Remove all IsAs starting from P.

Remove all IsAs ending in P.

Remove all instantiation links ending in P.

Remove P and the information on its domain.

F27: Remove DTDomain U={¬DataTypeDomain(P,A)}

Equivalent to removing the relevant datatype property

F28: Remove DataTypeProperty Subsumption (i.e., U={¬DataP_IsA(P,Q)})

If IsA from P to Q exists:

Remove the direct subsumption relation (IsA) between P and Q:

Find all sequences of IsA that start from P and end in Q. Select one IsA from each sequence for removal. The selection should be made so as to select the fewest IsAs possible; in the case of ties, we choose the solution which includes the IsAs that are furthest

from the top class. . If again we have ties, favour the solution that has the "cheapest" IsA among all, as explained in F22.

F29: Remove DatatypeProperty Instantiation U={¬DataP_Inst(x,y,P)}

If there is an Instantiation link containing P

Remove the property instantiation

Remove every DataP_Inst containing a subproperty of P (and x,y)

F30: Remove MetaClass U={¬MCS(A)}

If there exists MCS(A)

Remove all IsAs starting from A.

Remove all IsAs ending in A.

Remove all instantiation links ending in A.

Remove A.

F31: Remove MetaClass Subsumption (i.e., U={¬M_IsA(A,B)})

If IsA from A to B exists:

Remove the direct meta class subsumption relation (IsA) between A and B:

Find all sequences of IsA that start from A and end in B. Select one IsA from each sequence for removal. The selection should be made so as to select the fewest IsAs possible; in the case of ties, we choose the solution which includes the IsAs that are furthest from the top. If again we have ties, favour the solution that has the "cheapest" IsA among all.

To determine the cheapest we use the distance (Dist function) of the first parameter from the top; the larger the distance, the cheaper the ground fact. If this distance is identical, use the distance (Dist function) of the second parameter from the top; the smaller the distance, thecheaper the ground fact. If both distances are identical, use lexicographic ordering ($\leq_{lex}$) on the first parameter. If both distances are identical and the first parameter is also identical, use lexicographic ordering ($\leq_{lex}$) on the second parameter.

F32: Remove MetaClass Instantiation (i.e., U={¬M_Inst(x,A)})

If there is an Instantiation link between x and A

Remove the instantiation link between x and A:

Remove every M_Inst from x to a subclass of A.