University of Crete
Computer Science Department

Personalizing Declarative Repairing Policies for Curated KBs

Ioannis Roussakis

Master's Thesis

Heraklion, December 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΡΟΣΩΠΟΠΟΙΩΝΤΑΣ ΔΗΛΩΤΙΚΕΣ ΠΟΛΙΤΙΚΕΣ ΔΙΟΡΘΩΣΗΣ ΓΙΑ ΕΠΙΜΕΛΗΜΕΝΕΣ ΒΑΣΕΙΣ ΓΝΩΣΗΣ

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

Συγγραφέας:

Ιωάννης Ρουσσάκης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Καθηγητής, Επόπτης

Δημήτρης Πλεξουσάκης, Καθηγητής, Μέλος

Γρηγόρης Αντωνίου, Καθηγητής, Μέλος

Άγγελος Μπίλας, Αναπλ. Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Δεκέμβριος 2010

Personalizing Declarative Repairing Policies for Curated KBs

Ioannis Roussakis
Master's Thesis
Computer Science Department, University of Crete

Abstract

Curated ontologies and semantic annotations are increasingly being used in e-science to reflect the current terminology and conceptualization of various scientific domains. Such curated Knowledge Bases (KB) are usually backended by relational databases using adequate schemas. Schemas may be generic or application/domain specific and in many cases are required to satisfy a wide range of integrity constraints. As curated KBs continuously evolve, such constraints are often violated and thus KBs need to be frequently *repaired*.

Motivated by the fact that consistency is nowdays mostly enforced manually by the scientists acting as curators, we propose a *generic* and *personalized* repairing framework for assisting them in this arduous task. Modeling integrity constraints using the class of Disjunctive Embedded Dependencies (DEDs), we are capable of supporting a variety of useful integrity constraints presented in the literature. Moreover, we rely on coplex curator preferences over various interesting features of the resulting repairs that can capture diverse notions of *minimality* in repairs. As a result, other repair policies presented in the literature can be emulated within our framework.

Moreover, we propose a novel *exhaustive* repair finding algorithm which, unlike existing greedy frameworks, is not sensitive to the resolution order and syntax of violated constraints and can *correctly compute globally optimal repairs for different kinds of constraints and preferences*. Despite its exponential nature, the performance and memory requirements of the exhaustive algorithm are experimentally demonstrated to be satisfactory for real world curation cases, thanks to a series of optimizations. Finally, we propose the corresponding "greedy" algorithm wich computes *locally optimal repairs* by considering each violation individualy keeping only the preffered-per-violation repairs.

Last but not least, we propose possible extensions of our framework to describe policies where the inconsistencies are resolved during their introduction (e.g., belief revision, belief merging). This can be achieved by carefuly designing operations which modify the KB's status in order to prevent the inconsistencies from creeping into the system.

Supervisor: Vassilis Christophides
Professor

ΠΡΟΣΩΠΟΠΟΙΩΝΤΑΣ ΔΗΛΩΤΙΚΕΣ ΠΟΛΙΤΙΚΕΣ ΔΙΟΡΘΩΣΗΣ ΓΙΑ ΕΠΙΜΕΛΗΜΕΝΕΣ ΒΑΣΕΙΣ ΓΝΩΣΗΣ

Ιωάννης Ρουσσάκης
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Οι επιμελημένες οντολογίες και σημασιολογικές υποσημειώσεις χρησιμοποιούνται ολοένα και περισσότερο στην ψηφιακή επιστήμη για να αντικατοπτρίζουν την τρέχουσα ορολογία και την σύλληψη των διαφόρων επιστημονικών πεδίων. Τέτοιες επιμελημένες Βάσεις Γνώσης συνήθως υποστηρίζονται από σχεσιακές βάσεις δεδομένων χρησιμοποιώντας κατάλληλα σχήματα. Τα σχήματα μπορεί να είναι γενικού σκοπού ή συγκεκριμένα ανά εφαρμογή και σε πολλές περιπτώσεις απαιτείται να ικανοποιούν ένα ευρύ φάσμα περιορισμών ακεραιότητας. Λόγω του ότι οι επιμελημένες Βάσεις Γνώσης εξελίσσονται συνεχώς, τέτοιοι περιορισμοί συχνά παραβιάζονται άρα οι βάσεις πρέπει συχνά να επιδιορθώνονται.

Παρακινημένοι από το γεγονός ότι η ακεραιότητα σήμερα επιβάλλεται ως επί το πλείστον χειροκίνητα από τους ίδιους τους επιστήμονες οι οποίοι δρουν σαν επιμελητές, προτείνουμε ένα *γενικό* και *εξατομικευμένο* πλαίσιο διόρθωσης ώστε να τους παρέχουμε βοήθεια σε αυτό το επίπονο έργο. Μοντελοποιώντας τους περιορισμούς ακεραιότητας χρησιμοποιώντας την κλάση των Διαζευκτικών Ενσωματωμένων Εξαρτήσεων (ΔΕΕ), είμαστε σε θέση να υποστηρίξουμε μια ποικιλία από περιορισμούς ακεραιότητας που εμφανίζονται στη βιβλιογραφία. Επιπλέον, βασιζόμαστε σε σύνθετες εκφράσεις προτίμησης των επιμελητών για ενδιαφέροντα χαρακτηριστικά των διορθώσεων που προκύπτουν με σκοπό να συλλάβουμε διαφορετικές έννοιες *ελαχιστοποίησης* στις διορθώσεις. Σαν αποτέλεσμα, άλλες πολιτικές διόρθωσης που παρουσιάζονται στη βιβλιογραφία μπορούν να προσομοιωθούν στο πλαίσιο μας.

Επί πλέον, προτείνουμε ένα νέο εξαντλητικό αλγόριθμο διόρθωσης ο οποίος, εν αντιθέσει με υπάρχοντα άπληστα πλαίσια, δεν εξαρτάται από τη σειρά διόρθωσης και τη σύνταξη των παραβιασμένων περιορισμών και μπορεί να *υπολογίσει σωστά καθολικά βέλτιστες επιδιορθώσεις για διαφορετικούς τύπους κανόνων και προτιμήσεων*. Παρά την εκθετική φύση του, η απόδοση και οι απαιτήσεις σε μνήμη του αποδεικνύεται πειραματικά ότι είναι ικανοποιητικές για πραγματικές περιπτώσεις επιμέλειας χάρη σε μία σειρά από βελτιστοποιήσεις. Τέλος, προτείνουμε τον αντίστοιχο 'άπληστο' αλγόριθμο ο οποίος υπολογίζει τοπικές βέλτιστες διορθώσεις θεωρώντας κάθε παραβίαση ξεχωριστά και διατηρώντας μόνο τις προτιμώμενες-ανά-κανόνα διορθώσεις.

Τέλος, προτείνουμε πιθανές επεκτάσεις του πλαισίου μας με σκοπό να περιγράψουμε πολιτικές όπου οι ασυνέπειες επιλύονται κατά την εισαγωγή τους (π.χ. αναθεώρηση γνώσης, συγχώνευση γνώσης). Αυτό μπορεί να επιτευχθεί με το να σχεδιάσουμε προσεκτικά τις λειτουργίες που τροποποιούν την κατάσταση της Βάσης Γνώσης με σκοπό να εμποδίσουμε τις ασυνέπειες να εισχωρήσουν στο σύστημα.

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Καθηγητής

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω θερμά τον επόπτη καθηγητή μου κ. Βασίλη Χριστοφίδη για την άψογη συνεργασία μας και την πολύτιμη καθογήγηση του κατά της διάρκεια των ακαδημαϊκών μου χρόνων. Η εμπιστοσύνη που μου έδειξε ήταν καθοριστική για την ολοκλήρωση της εργασίας μου και τα εφόδια που αποκόμισα κοντά του θα αποτελέσουν τις βάσεις για τη μετέπειτα καριέρα μου. Θα ήταν μεγάλη παράλειψη μου αν δεν ευχαριστούσα το Γιώργο Φλουρή για την ουσιαστική συμβολή στην περάτωση αυτής της εργασίας. Τον ευχαριστώ για τις εύστοχες και εποικοδομητικές του παρατηρήσεις καθώς και για τον μεθοδικό τρόπο που μου έμαθε να εργάζομαι.

Δε θα μπορούσα να μην αναφέρω σε αυτό το σημείο το Ιστιντούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας τόσο για την υλική αλλά και την οικονομική στήριξη που μου προσέφερε για να καταφέρω να ολοκληρώσω την εργασία αυτή. Επίσης θα ήθελα να ευχαριστήσω όλα τα μέλη της ομάδας των Πληροφοριακών Συστημάτων του Ιστιντούτου Πληροφορικής - ΙΤΕ για την ευχάριστη συνεργασία μας όλα αυτά τα χρόνια καθώς και για το γεγονός ότι υπήρξαν όχι μόνο εξαίρετοι συνεργάτες, αλλά και πολύ καλοί φίλοι. Θα ήθελα επίσης να ευχαριστήσω τα μέλη της εισηγητικής επιτροπής κ. Γρηγόρη Αντωνίου και κ. Δημήτρη Πλεξουσάκη για τις εποικοδομοητικές τους παρατηρήσεις. Ακόμη, ευχαριστώ τους καλούς μου φίλους για τη συμπαράσταση τους και για τις όμορφες στιγμές που περάσαμε στο Ηράκλειο οι οποίες θα μείνουν για πάντα χαραγμένες στο μυαλό μου.

Το μεγαλύτερο όμως ευχαριστώ το οφείλω στους γονείς μου, Μανώλη και Άννα και στα αδέρφια μου Μιχάλη και Μαρία για τους κόπους τους, τις 8υσίες και την εμπιστοσύνη τους, χωρίς την οποία δε θα ήταν δυνατό να ολοκληρώσω την εργασία αυτή και τους στόχους μου.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Curated Knowledge Bases

An increasing number of scientific communities (e.g., Bioinformatics, Medicine and Health, Environmental and Earth Sciences, Astronomy, Cultural Informatics and Humanities[1]) rely on common terminologies and reference models related to their research field in order to facilitate annotation and inter-relation of scientific and scholarly data of interest. Such knowledge representation artifacts form a kind of curated *Knowledge Bases (KBs)* which are developed with a great deal of human effort by communities of scientists [8] and are usually backended by relational database support using adequate schemas [37]. To enforce sharing and reusability, these knowledge representation artifacts are published nowadays using Semantic Web languages such as RDF/S or OWL and essentially form a special kind of curated databases. The above observations are highlighted in Figure 1.1, where a cyclical process for maintaining a curated KB is depicted as new knowledge is accumulated and existing knowledge is modified. As we can see, existing curated data annotations and curated ontologies are used as input sources by other curated ontologies along with online resources. Moreover, once produced and published, curated ontologies may lead to modifications to, linked to them, annotations and ontologies.

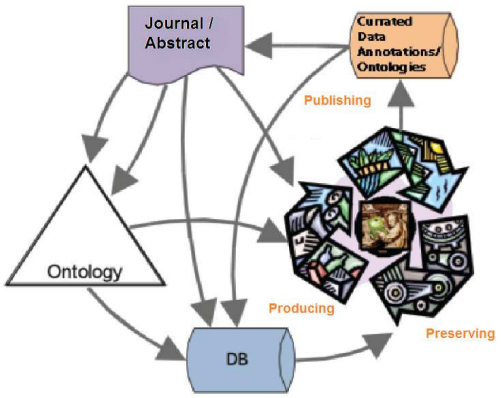

Figure 1.1: Overview of curation process

[1] www.geneontology.org, www.biopax.org, www.nlm.nih.gov/research/umls, www.co-ode.org/galen, sweet.jpl.nasa.gov/ontology, archive.astro.umd.edu/ont, cidoc.ics.forth.gr.

1.2 Motivation

Curated KBs often have to satisfy various domain or application specific constraints [36, 32, 26] which are mostly enforced nowdays manually. Inconsistencies may arise as scientists, acting as curators, have to constantly agree on the common knowledge to represent and share in their research field. Curated KBs may change as new experimental evidence and observations are acquired worldwide, due to revisions in their intended usage, or even to correct erroneous conceptualizations. Given that in most cases curated KBs are interconnected [17] and different groups of curators may adopt different constraints, inconsistencies may arise when changes get propagated from related (through copying or referencing) remote KBs, even when inconsistencies could be locally prevented. Furthermore, inconsistencies may be caused by changes in the constraints themselves. The problem of inconsistency can be addressed either by providing the ability to query inconsistent data (consistent query answering [5]) or by actually *repairing* KBs [15]. Clearly, the value of curated KBs lies in the quality of the encoded knowledge and thus they need to be frequently repaired. For example, one may want to impose acyclicity of subsumption relations between classes or properties [36], primary and foreign key constraints [10, 12], or cardinality constraints [32] of instances. In this thesis, we are interested in *declarative repairing* frameworks for assisting curators in the arduous task of repairing inconsistent KBs.

The purpose of a repairing framework is the automatic identification of a consistent KB that would preserve as much knowledge as possible from the original, inconsistent KB [15]. The latter requirement is important, because several inconsistencies may exist and each of them may be resolved in different ways, so several potential repairs may exist; the repairing process should return the one that causes *minimal* effects (updates) upon the curated KB [1, 10]. However, the notion of *minimality* in curated KBs depends on a number of underlying assumptions made by different groups of curators; e.g., under a complete knowledge assumption, curators may favor repairs performing removals [1], whereas in the opposite case, they may prefer repairs performing additions [27]. In existing algorithms (e.g., [6, 7, 13]) such preferences are embedded in the repair finding algorithms and curators cannot intervene. Furthermore, in their majority, they consider the resolution of each constraint violation in isolation from the others; as we will show in Section 1.3, this makes the result of repairing sensitive to the evaluation order of the constraints, as well as their syntactic form.

1.3 Problem Statement

To explain the intuition behind our approach and the related challenges, we consider an example of a curated ontology. Ontologies are usually backended by relational databases; an efficient representation [25, 37] of both schema and data of curated ontologies consists of tuples like $CS(B)$, denoting that B is a class, $C\_IsA(B, A)$, denoting a (direct or transitive) subsumption relationship between B, A, $PS(P)$, denoting that P is a property, and $Domain(P, B)$, $Range(P, B)$, denoting that the domain and range (respectively) of P is B. So, let us assume that the original KB is $K = \{CS(B), C\_IsA(B, A), PS(P), Range(P, B)\}$.

Various constraints related to the ontology schema and data that have been reported in the literature can be expressed using DEDs [14]. For example, the following DEDs represent schema constraints stating that all subsumption relationships should be between defined classes (c_1), that properties should have a defined domain and range (c_2), which in turn should be a defined class (c_3, c_4); for more constraints on the ontology schema and data, see Section 4.

$$c_1 : \forall u_1, u_2 C\_IsA(u_1, u_2) \rightarrow CS(u_1) \wedge CS(u_2)$$

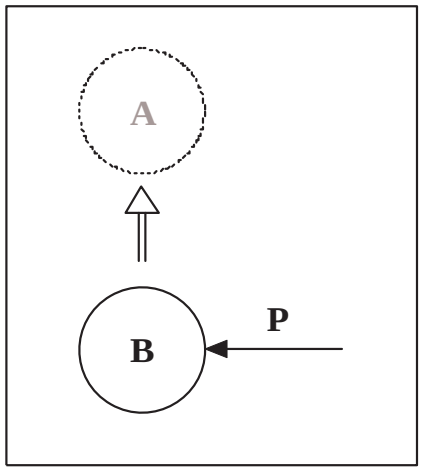

Figure 1.2: An inconsistent KB

$c_2 : \forall u PS(u) \rightarrow \exists v_1, v_2 Domain(u, v_1) \wedge Range(u, v_2)$

$c_3 : \forall u_1, u_2 Range(u_1, u_2) \rightarrow PS(u_1) \wedge CS(u_2)$

$c_4 : \forall u_1, u_2 Domain(u_1, u_2) \rightarrow PS(u_1) \wedge CS(u_2)$

Note that the KB of Figure 1.2 violates the constraint c_1 as we have the subsumption relation $C\_IsA(B, A)$ but the class A is not declared (i.e., $CS(A) \notin K$). Moreover, it violates the constraint c_2, because KB contains a property with no domain. Having an inconsistent KB which must be repaired, the widely-used repairing strategy [7, 13] consists in selecting one violated constraint, repairing it, then finding the next violated constraint etc, until no more violations exist. The resolution options for a constraint can be deduced from its syntax: for example, to resolve c_1 we must either remove $C\_IsA(B, A)$ or add $CS(A)$. Similarly, to resolve c_2, we can remove $PS(P)$, or add $Domain(P, x)$ for any x. This process can be modeled in a *resolution tree*, at each node of which one constraint violation is resolved.

Note that our repairing choices are not independent, but may have unforeseenconsequences. For example, if we choose to remove $PS(P)$ to resolve c_2, then c_3 is subsequently violated, and this violation is caused by our choice to remove $PS(P)$. Similarly, if we choose to add $Domain(P, A)$ to resolve c_2, then c_4 is subsequently violated. The latter violation is prevented if we choose to resolve c_1 by adding $CS(A)$; in this case, the addition of $CS(A)$ resolves two constraint violations at the same time.

1.3.1 Repair Strategies

In most repair finding frameworks (e.g., [7, 6, 13]), the constraint resolution tree is created by selecting the optimal resolution option(s) locally, i.e., for each violation according to the adopted policy and discarding all the non-optimal branches; this way, *all the leaves of the resolution tree correspond to preferred repairs*. We will call this strategy *violation-based (VB)*, because the selection of the preferred resolutions is done individually for each violation. Figure 1.3 shows one application of this strategy, in which the adopted policy is: "we want a minimum number of updates during repair; in case of a tie, prefer those repairs that perform most deletions". The dotted circles in Figure 1.3 indicate rejected resolution options. Under the above policy, the algorithm would first choose to remove $C\_IsA(B, A)$ (to restore c_1); this option is preferable than

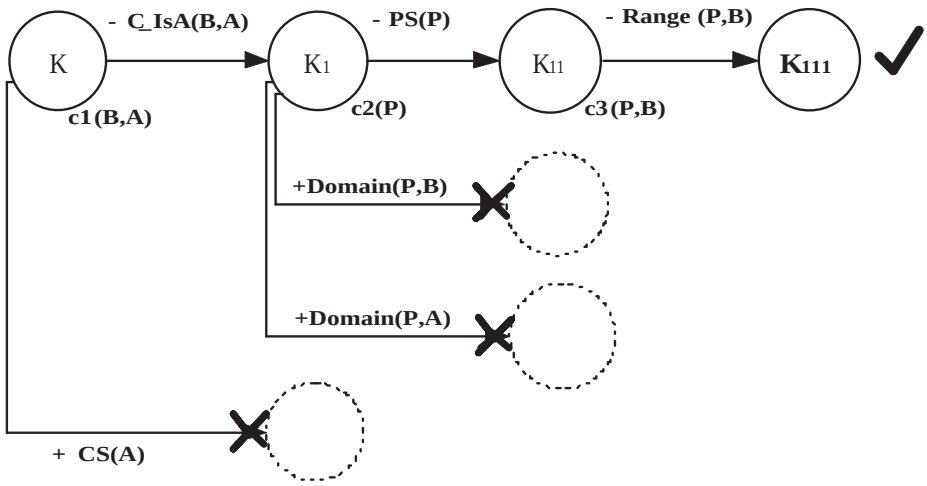

Figure 1.3: A VB Resolution Tree

the addition of $CS(A)$. The new KB (K_1) violates c_2, so we continue the process by removing $PS(P)$ (to resolve c_2) and finally remove $Range(P, B)$ (for c_3); thus the (only) returned repair would be $K_{111} = \{CS(B)\}$. The resolutions under this strategy are selected in a greedy manner, in the sense that only the current constraint is considered, and the potential consequences of a selected/rejected option are neglected. Thus, we may miss more preferred repairs, as in the repair $K' = \{PS(P), CS(B), Range(P, B), Domain(P, B)\}$, which differs from the original KB K in only two atoms, so it is more preferable than K_{111} under our policy and should have been returned instead.

An additional shortcoming of the VB strategy is that it is sensitive to the constraint evaluation order and syntactic form. To see this, consider again our running example and the repairing policy: "we prefer deletion of class-related information (CS, C_IsA) over additions; but we prefer both over the addition of property-related information ($PS, Range, Domain$), which, in turn, is preferable than deletion of property-related information". Let us consider first the evaluation order: $c_2 \rightsquigarrow c_1 \rightsquigarrow c_3 \rightsquigarrow c_4$. First, c_2 is resolved using two alternative options: add $Domain(P, A)$ or add $Domain(P, B)$. Let us concentrate on the first option, i.e., add $Domain(P, A)$. Then, the resolution of c_1 would delete $C\_IsA(B, A)$, and the resolution of c_4 would add $CS(A)$. The end result is the repair: $K_{order} = \{PS(P), CS(B), CS(A), Range(P, B), Domain(P, A)\}$ (note that this is not the only repair that will be returned). The reader can verify that, for the evaluation order $c_2 \rightsquigarrow c_3 \rightsquigarrow c_4 \rightsquigarrow c_1$, K_{order} will not be returned. The reason is that the branch that adds $Domain(P, A)$, would then resolve c_4 through the addition of $CS(A)$, which in turn resolves c_1 implicitly and $C\_IsA(B, A)$ will not be removed.

To see the effect of the constraint syntax, let us slightly change our example and consider the KB $K_{syntax} = \{ PS(P), CS(B)\})$. Also, change the repairing policy into: "we want a minimum number of updates during repair; in case of a tie, prefer those repairs that perform most additions". Then, the resolution of c_2 (the only violated rule) would only accept the branch where $PS(P)$ is removed. If we replace c_2 with the equivalent set of constraints $c_{2a} : \forall u PS(u) \rightarrow \exists v_1 Domain(u, v_1)$ $c_{2b} : \forall u PS(u) \rightarrow \exists v_2 Range\ (u, v_2)$, then the removal of $PS(P)$ is no longer a selected resolution; instead, the algorithm would add $Domain(P, x)$ and $Range(P, y)$ for some constants x, y. Thus, the returned repair depends on the constraints' syntax, rather than the constraints' semantics.

To address these shortcomings of the VB strategy, we propose an alternative resolution

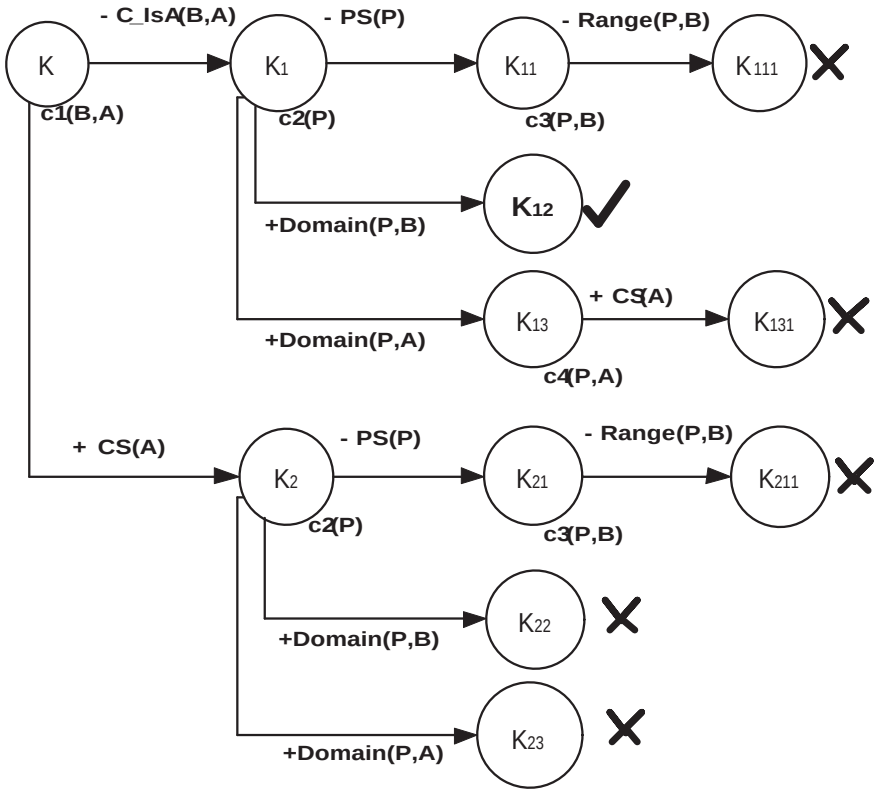

Figure 1.4: An IB Resolution Tree

strategy, which we will call *information-based (IB)*. Under IB, none of the resolution options of a violation is rejected; instead, all resolution branches are considered (see Figure 1.4). Hence, *each leaf of the resolution tree is a potential repair, but not necessarily a preferred one* according to the adopted policy. For this reason, IB needs to compare all potential repairs in order to select only the preferred ones. Figure 1.4 shows one application of this strategy: no branches are rejected, but most of the leaves (e.g., K_{111}, K_{131} etc) are rejected as non-preferred. Going back to our example, adopting the policy "we want a minimum number of updates during repair; in case of a tie, prefer those repairs that perform most deletions", the KB $K_{12} = \{PS(P), CS(B), Range(P,B), Domain(P,B)\}$ would be returned (see also Figure 1.4). Note that this is a different repair from the one returned by the VB strategy for the same KB and policy. Given that complete resolutions are considered, the returned repairs of an IB strategy are always the most preferred ones (unlike VB); in addition, as we will show later (Proposition 1), IB strategies are immune to changes in the constraints' evaluation order or syntax. However, IB strategies have to compute a larger resolution tree, so they are, on average, less efficient than VB strategies; note however that the worst-case complexity is the same in both cases (Section 3). In addition, several optimization opportunities can be considered to reduce the size of the resolution tree in the average case (e.g., by considering a more compact form of resolutions or by finding safe ways to prune the resolution tree without losing preferred solutions).

1.3.2 Repair Preferences

Given the heterogeneity of curated KBs, different groups of curators should be able to apply (and experiment with) different repair policies. Aiming at being able to to declaratively state

repairing policies, there is need for a *formal preference model* over interesting *features* of repairs. We rely on qualitative preference models proposed for relational databases supporting *atomic and composite preference expressions* over relational attributes [23, 9, 18]. For example, under a complete knowledge assumption, curators may need to obtain minimum additions, so the interesting feature is the number of additions, whereas the atomic preference is the *minimization function* (Min). The application of this preference, under an IB strategy, over the possible repairs of KB presented in Figure 1.2, would give the repair $K_{111} = \{CS(B)\}$ as preferred (cf. Figure 1.4). To express composite preferences spanning several attributes, curators may employ *constructors* between atomic preferences, such as $\otimes$ (pareto) and & (prioritized). For example, a declarative repairing policy could be to equally prefer (i.e., pareto) repairs featuring both a minimum number of updates and a minimum number of additions.

To our knowledge, the repairing frameworks proposed in the literature (e.g., [7, 6, 13]) adopt a specific policy for resolving violations, which is determined at *design-time* and may be useful for a particular curation setting but not to others (e.g., complete vs incomplete knowledge assumption). Our framework relies on qualitative preference expressions to declaratively specify a repairing policy, thereby enabling curators to adapt their policies at *run-time* (using an intuitive preference elicitation interface [31]). Last but not least, most of the policies adopted in the literature can be modeled as special cases of our repairing framework (see Propositions 2, 3 and Section 6).

1.3.3 Wildcards

One problem with the policies following the IB or VB strategy is that the number of resolution options or preferred resolution options for a given constraint may in some cases be comparable to the number of available KB constants. In our motivating example, c_2 can be resolved by removing $PS(P)$, or by adding $Domain(P, x)$ for any constant x. To avoid the need to explore a potentially large number of alternative branches, we introduce *wildcards* (denoted by ε_i), which are used to represent in a compact manner several alternative constants, and essentially constitute *existential variables* ranging over the set of constants. In our example, the resolution options for c_2 using wildcards would be just two, namely the removal of $PS(P)$ and the addition of $Domain(P, \varepsilon_1)$; the latter should be read as "$Domain(P, x)$ for any constant x". Hence, a wildcard is used to reduce the size of the resolution tree by combining several branches into one. In addition, wildcards are useful because they compact the repairs returned to the curator, allowing him to easier evaluate his options. Note that a single wildcard is not enough. For example, if two different properties violated constraint c_2 (i.e., they had no domains), then it should be possible to set the domains of the properties independently, by using different wildcards for each (say $\varepsilon_1, \varepsilon_2$).

The detection and resolution of inconsistencies when wildcards are considered is more tricky. In our previous example, suppose that we resolve c_2 by adding $Domain(P, \varepsilon_1)$. Then, we note that c_4 may, or may not be violated, depending on the value of ε_1; e.g., if we replace ε_1 with B, then c_4 is not violated, but if we replace it with A then c_4 is violated (because $CS(A) \notin K$). This problem raises the need to constrain the values that a wildcard could potentially take. Continuing our example, we would like to follow different resolution paths depending on whether c_4 needs to be resolved (i.e., is violated) or not. Thus, we introduce the *wildcard mapping*, μ, which is a set containing the values that the wildcards can take. In our example, if $\mu = \{B\}$, then c_4 is not violated.

1.4 This Thesis

The main contribution of this work is a *generic* and *personalized* framework for assisting curators in repairing inconsistent KBs, which supports a *variety of useful integrity constraints* using Disjunctive Embedded Dependencies (DEDs) [14], as well as complex curators' *preferences* over interesting features of the resulting repairs (e.g., the number and type of repairing updates). To formalize the desired repairing policies we rely on qualitative preference models that have been proposed for relational databases [9, 18, 23]. Then, we demonstrate that the proposed framework is able to express very diverse repairing preferences, as well as different notions of minimality that have been proposed so far in the literature (e.g., [1, 15]).

We propose a novel, algorithm, which we call *information-based (IB)*, which successfully addresses the aforementioned issues by considering all possible resolutions in order to identify globally optimal repairs (per the minimality preference). In contrast, existing algorithms (e.g., [6, 13, 7]), which we call *violation-based (VB)*, are greedy and consider minimality only locally during the resolution of each inconsistency (by ignoring non-optimal resolution options). We prove that the two algorithms do not give, in general, the same resulting KB repairs, because the IB algorithm computes repairs which are globally optimal w.r.t. to curators' preferences, whereas VB computes locally optimal repairs. Finally, we provide the complexity bounds for both IB and VB algorithms along with some experimental results for typical synthetic curated KBs.

The structure of this thesis is as follows: In Chapter 2 we describe our repairing framework. In Section 2.1 we present some preliminary notions such as the considered representation language and integrity constraints and in Section 2.2 we present its declarative nature.

In Chapter 3 we present the two main repair finding algorithms we implemented; an IB and a VB algorithm. Next, in Section 3.1, we show some proposed optimizations which can be applied over IB and VB algorithms and the chapter ends with the complexity bounds for both IB and VB algorithms for various constraint categories which are found in the literature (Section 3.2).

The experimental findings of our repairing algorithms are presented in Chapter 4. Firstly, in Section 4.1 we describe the experimental setting which was used and next we show the results from the evaluation of IB, VB algorithms. Finally, in Section 4.4 we show the quality of the repairs produced by VB algorithm comparing them with the respective produced by IB algorithm.

Apart from repairing, in Chapter 5, we introduce an alternative way to deal with invalidities by preventing them from creeping into the KB. Next in Chapter 6 we show existing repair approaches which are found in the literature and how they can be integrated within our framework. Finally, in Chapter 7 we present our main conclusions and some ideas for future work.

Chapter 2

Framework

2.1 Preliminaries

2.1.1 Knowledge Bases

In this work, we consider the standard relational semantics, whereas we do not commit ourselves to any particular schema. Our only restriction is that we will always assume some arbitrary, but predetermined finite set of relations $\mathcal{R}$ and infinite set of constants ($\mathcal{U}$). Thus, A *KB* K is any set of relational atoms of the form $R(\vec{A})$. Obviously, for any relational atom $R(\vec{A})$ or tuple and KB K it holds that $K \vDash R(\vec{A})$ iff $R(\vec{A}) \in K$, and $K \vDash \neg R(\vec{A})$ iff $K \nvDash R(\vec{A})$ (i.e., iff $R(\vec{A}) \notin K$). We consider that changes performed during repairs are represented from a special tuple which forms a *delta*. A delta $\delta = \langle \delta_a, \delta_d \rangle$ essentially contains the sets of relational atoms to be added (δ_a) or removed (δ_d) from the KB. We denote by $\mathcal{K}$ the set of all KBs and by Δ the set of all deltas. Next we define the application of a delta $\delta = \langle \delta_a, \delta_d \rangle$ upon a KB K.

Definition 1 *Consider a KB K and a delta $\delta = \langle \delta_a, \delta_d \rangle$. The* application *of a δ upon K is an operation $\bullet$ s.t. $K \bullet \delta = (K \cup \delta_a) \setminus \delta_d$.*

Deltas can be composed as well to produce a delta with a cumulative effect.

Definition 2 *Consider two deltas $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle$, $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle$. The* composition *of deltas is an operation $\uplus$ s.t. $\delta_1 \uplus \delta_2 = \langle \delta_{a1} \cup \delta_{a2}, \delta_{d1} \cup \delta_{d2} \rangle$.*

We equip our framework with *wildcards* ($\varepsilon_1, \varepsilon_2, \ldots$), taken from an infinite set $\mathcal{E}$, which is disjoint from $\mathcal{U}$ to avoid confusion. A wildcard provides a compact representation of several constants and essentially introduces an *existential variable*. A *wildcard mapping* μ is a set that determines the constants that each wildcard can be mapped to. For example, if $\mu_1 = \{A, B\}$, then ε can be replaced by either A or B, but not, e.g., by C. Thus, $R(\varepsilon)$ denotes a set of different relational atoms, which depend on the mapping; in the above example, $R(\varepsilon)$ represents $\{R(A), R(B)\}$. The latter set is denoted by $[[R(\varepsilon)]]^{\mu_1}$ which in fact contains all the assigned values from μ_1 to ε. Obviously, if $R(\vec{A})$ does not contain wildcards, then $[[R(\vec{A})]]^{\mu} = \{R(\vec{A})\}$ for all μ. Note that if we had more than one wildcard to deal with, the mapping would contain tuples of constants (each element of the tuple representing one wildcard). For example consider the tuple $R(\varepsilon_1, \varepsilon_2)$ and the mapping $\mu_1 = \{\langle A, C \rangle, \langle A, B \rangle, \langle B, C \rangle, \langle B, B \rangle\}$. The corresponding $[[R(\varepsilon_1, \varepsilon_2)]]^{\mu_1}$ is equal to $\{R(A, C), R(A, B), R(B, C), R(B, B)\}$. We assume that a wildcard can range only over the constants that appear in the KB K, i.e., wildcards cannot be used to introduce new constants in the KB.

As explained in Section 1.3, KBs and deltas may feature wildcards. A KB K or a delta δ with wildcards corresponds to a set of KBs or deltas as determined by some mapping μ. These sets will be denoted by $[[K]]^{\mu}$, $[[\delta]]^{\mu}$, or simply $[[K]]$, $[[\delta]]$ when the corresponding mapping can be easily deduced from the context. Note that a mapping μ in $[[\delta]]^{\mu}$ is used to restrict the values that wildcards can take when they appear in KBs/deltas. Therefore, the application operation for KBs/deltas with wildcards is defined as follows:

Definition 3 *Consider a set of KBs $[[K]]^{\mu_K}$ and a set of deltas $[[\delta]]^{\mu_\delta}$ determined by the corresponding mappings μ_K, map_δ respectively. The application of such deltas over KBs when they feature wildcards is defined as $[[K]]^{\mu_K} \bullet [[\delta]]^{\mu_\delta} = [[(K \cup \delta_a) \setminus \delta_d]]^{\mu_K \cap \mu_\delta}$.*

The only subtle issue that needs to be clarified is related to the wildcards' (in)dependence. In particular, given a KB K and a delta δ, featuring the same wildcard, the application $K \bullet \delta$ will force this wildcard to be mapped to the same constant(s) in all its appearances in $K \bullet \delta$. In some cases, this may not be the desired behavior. A similar case appears when different wildcards, which should have been the same, appear in K, δ. However, this is a non-issue in our framework, because wildcards are introduced during the repair finding algorithm (see Subsection 2.1.2), so we can control which wildcards are used, and the application semantics given previously is sufficient. Similarly, the composition operator $\uplus$ for two deltas δ_1, δ_2 with wildcards is defined as follows:

Definition 4 *Given two deltas which may feature wildcards, $[[\delta_1]]^{\mu_1}$, $[[\delta_2]]^{\mu_2}$, their composition is defined as $[[\delta_1]]^{\mu_1} \uplus [[\delta_2]]^{\mu_2} = [[\delta]]^{\mu \cap \mu_2}$ where $\delta = \langle \delta_{a1} \cup \delta_{a2}, \delta_{d1} \cup \delta_{d2} \rangle$.*

2.1.2 Integrity Constraints

We assume that, in general, not every Knowledge Base K is a valid representation of our knowledge. To discriminate between "valid" and "non valid" representations, we allow the introduction of *Integrity Constraints* which are FOL formulas over relational atoms without wildcards. For technical reasons, we assume that all rules can be encoded in the form of DEDs (disjunctive embedded dependencies), which have the following general form:

$$\forall \vec{A} \bigwedge_{i=1,\ldots,n} P_i(\vec{A}) \rightarrow \vee_{j=1,\ldots,m} \exists \vec{B}_j Q_j(\vec{A}, \vec{B}_j)$$

where:

- $\vec{A}$,$\vec{B}_i$ are tuples of variables.
- Q is a (maybe empty) conjunction of relational atoms of the form $R(w_1, \ldots, w_n)$, where $w_1, \ldots, w_n$, are variables or constants.

We say that a KB K *satisfies* a constraint c when $K \vDash c$. Obviously, for any given set $\mathcal{C}$ of integrity constraints, we will call a KB K to be *consistent* iff it satisfies $\mathcal{C}$ ($K \vDash C$). Note that $K \vDash \mathcal{C}$ iff K satisfies c, for all $c \in C$. To guarantee the existence of at least one valid KB, the set of integrity constraints must be consistent in the standard logical sense. For a given DED c, we denote by $c(\vec{A})$ the *constraint instance* that occurs by replacing $\vec{u}$ in c with a tuple of constants $\vec{A}$. For example consider the constraint $c : \forall x, y P(x, y) \rightarrow Q(y)$. A constraint instance of c is $c(a, b) = P(a, b) \rightarrow Q(b)$. As before, we assume that some arbitrary, but predetermined set of constraints C is given.

The DED form of constraints allows both the easy detection of an invalidity, as well as the determination of all possible options for repairing such an inconsistency, using just syntactical

manipulations over the violated constraint. This form is general and expressive enough, such that it can be used to express consistency in several contexts [14]. For instance, in the field of relational databases, DEDs be used to express the traditional functional and inclusion dependencies [12] along with more complex constraints as tuple generating dependencies(i.e., tgds), LAV tgds and full tgds [1]. Inclusion and functional dependencies can be extended with conditions to increase the quality of data on which they are applied [15] and DEDs are capable of capturing this extention as well. However, DEDs can be used to capture constraints over ontological data as well. Among others, we can capture subsumption transitivity, acyclicity and symmetric relations [25].

The form of DED constraints allows both the easy detection of a violation as well as the determination of all possible repairing options for this violation. The underlying idea is very simple: a DED constraint can be written as a universally quantified disjunction(DNF) of (possibly negated) ground facts, existentially quantified conjunctions of positive ground facts and equality axioms. For example, the DNF form of $c : \forall x, y P(x, y) \rightarrow Q(y)$ is writen as $\neg P(x, y) \vee Q(y)$. For KBs without wildcards, this is based on the fact that $K \vDash Q(\vec{A})$ iff $Q(\vec{A}) \in K$. Thus, for a DED constraint c, all we have to do to find its violated instances is to search for those tuples $\vec{A}$ for which:

1. For all $i \in \{1, \ldots, n\}$, $P_i(\vec{A}) \in K$
2. For all $i \in \{1, \ldots, m\}$, there is no tuple of constants $\vec{B}$ s.t. $Q_{ij}(\vec{A}, \vec{B}) \in K$ for all $j \in \{1, \ldots, k_i\}$

A KB K satisfies a constraint $c(\vec{x})$ iff at least one such conjunction is implied by K; by our definitions, the latter check is reduced to checking whether all the positive and none of the negative ground facts of the conjunction appear in K. Similarly, given a KB K and a violated constraint instance $c(\vec{x})$, we can change K to satisfy $c(\vec{x})$ by simply adding all the positive, and removing all the negative, ground facts of one of the conjunctions $c(\vec{x})$ to/from K. Considering the motivating example of Section 1.3, the DNF form of constraint $c_3(P, B)$ is $\neg Range(P, B) \vee (PS(u_1) \wedge CS(u_2))$ and we can see that K satisfies $c_3(P,\ B)$ because $PS(P), CS(B) \in K$; it also satisfies $c_2(P')$ because $PS(P') \notin K$, but violates $c_2(P)$ because both the above clauses are false. The resolution options for a violation can be similarly determined: if none of the above conditions are true, we simply need to add or remove some fact from the KB in order to make one of them true. For example, the violation of $c_2(P)$ by K can be resolved either by removing $PS(P)$ or by adding $Domain(P, x)$ for some x (note that adding $Range(P, y)$ for some y is unnecessary here, as $Range(P, B) \in K$). Thus, the resolution of this violation can be made with one of the following deltas: $\delta_0 = \langle \emptyset, \{PS(P)\} \rangle$, $\delta_x = \langle \{Domain(P, x), \emptyset\} \rangle$ for all x. As explained in Section 1.3, wildcards can be used to compact the above set of deltas $\{\delta_x |$ for all $x\}$, by using $[[\delta]]^\mu = [[\langle \{Domain(P, \varepsilon)\}, \emptyset \rangle]]^\mu$ for an unused wildcard $\varepsilon$and $\mu\varepsilon = \mathcal{E}$. To formalize these ideas, for a constraint instance $c(\vec{x})$, we define the *resolution set* of $c(\vec{x})$ with respect to a KB K:

Definition 5 *Consider a constraint c and an instance of c, say $c(\vec{x}) = P(\vec{x}) \rightarrow \vee_{i=1,\ldots,n} \exists \vec{y_i} Q_i(\vec{x}, \vec{y_i})$, s.t. $P(\vec{x}) = P_1(\vec{x}) \wedge P_2(\vec{x}) \wedge \ldots \wedge P_k(\vec{x})$ for some $k \geq 1$ and $Q_i(\vec{x}, \vec{y_i}) = Q_{i_1}(\vec{x}, \vec{y_i}) \wedge Q_{i_2}(\vec{x}, \vec{y_i}) \wedge \ldots \wedge H_{i_m}(\vec{x}\vec{y_i})$ for some $m \geq 0$ depending on i. Without loss of generality, we assume that for each i there is some k_i ($0 \leq k_i \leq m_i$) such that $Q_{ij}(\vec{x}, \vec{v_i})$ is a relational atom for $0 < j \leq k_i$ and an equality atom for $k_i < j \leq m_i$. Then the* resolution set of c with respect to $\vec{x}$*, denoted by $Res(c(\vec{x}), K)$, is a set of deltas defined as:*

$$Res(c(\vec{x}), K) = \{\delta \mid \delta = \langle \emptyset, \{P_j(\vec{x})\} \rangle,\ 1 \leq j \leq k\} \cup \{\delta \mid \delta = \langle \{Q_{i1}(\vec{x}, \vec{y}),\ Q_{i2}(\vec{x}, \vec{y}), \ldots,$$

$Q_{ik_i}(\vec{x}, \vec{yi})\}, \emptyset\rangle$, *where* $\vec{y}$ *is a tuple of constants and for* $\vec{x}, \vec{z}$ *the equality atoms* $Q_{i,j}(\vec{x}, \vec{z})$ *for* $k_i < j \leq m_i$ *hold where* $1 \leq i \leq n\}$. *When* $\vec{x}$ *is obvious from the context, we will say* $Res(c, K)$.

Back in the motivating example, the corresponding resolution set for c_2 is defined as $Res(c_2(P), K) = \{\langle\emptyset, \{PS(P)\}\rangle, [[\langle\{Domain(P, \varepsilon)\}, \emptyset\rangle]]^{\mu}\}$.

When a KB features wildcards, the detection and resolution of violations becomes challenging. Formally, we define $[[K]] \vDash c$ iff $K' \vDash c$ for all $K' \in [[K]]$; using this, the definition of satisfaction and violation of constraints can be easily extended for the case with wildcards. However, this definition is not useful from a practical point of view since $[[K]]$ could be very large. So, our first challenge is to determine whether $[[K]] \vDash c$ without actually computing $[[K]]$. Then, once a violation has been detected we need to find a way to resolve it. The second challenge is to restrict resolution only to some of the KBs that the original KB with wildcards is mapped to, namely those that actually violate the constraint. Obviously, it does not worth to resolve the violation for those KBs of $[[K]]$ for which there is no constraint violation in the first place. For this reason, before launching the resolution algorithm we need to discriminate between *violating* and *non-violating* elements of $[[K]]$ (equivalently: violating and non-violating mappings). Then, only for the violating mappings, we need to add or remove KB facts for resolving the violated constraints, as in the case of KBs without wildcards.

All three problems (i.e., *detection of violations*, *identification of violating and non-violating mappings* and *violation resolution*) can be solved at the same time. In particular, we just need to determine whether there is any allowed mapping for the wildcards in K, for which the conditions 1,2 above hold. This gives both the constraint instances that are being violated, and the mapping that violates such instances. It also implicitly gives the KB facts that must be added or removed for resolving the violation.

However, to check the above conditions #1, #2, a simple membership test is not anymore sufficient, since we also need to consider the wildcards. So, suppose that, per condition #1, we seek a tuple $P(a_1, \ldots, a_n)$ in $[[K]]^{\mu}$. We need to find some $P(a'_1, \ldots, a'_n)$ in K such that for all i $a'_i = a_i$ *or* $a'_i \in \mathcal{E}$ and $a_i \in \mu(a'_i)$, where $\mu(a'_i)$ are the allowed mappings for a'_i (similarly if we look for a tuple that should not be in $[[K]]^{\mu}$, per condition #2). If this test succeeds, the corresponding constraint (say c) is violated, for the constraint instance $c(a_1, \ldots, a_n)$. The mapping that violates $c(a_1, \ldots, a_n)$ is the one that was used in the *or* clause of the above condition causing the searches to succeed/fail as necessary. Thus, the mapping restrictions will be of the form $a \in \mu(\varepsilon)$, or $a \notin \mu(\varepsilon)$; these restrictions can be used to determine the violating (and non-violating) mappings. Violated KBs in $[[K]]$ can be resolved by removing $P_i(\vec{A})$ or adding $Q_{ij}(\vec{A}, \vec{B})$ as in the simple case.

An example will clarify the above process. Consider the constraint c_4 and a KB $K_0 = K \cup \{Domain(P, \varepsilon)\}$ where where K is as defined in the motivating example. It holds that $K_0 = \{CS(B), C\_IsA(B, A), PS(P), Range(P, B), Domain(P, \varepsilon)\}$ for $\mu = \{A, B\}$, We are looking for x, y such that $Domain(x, y) \in K_0$ and $PS(x) \notin K_0$ or $CS(y) \notin K_0$. We note that the only tuple of the form $Domain(x, y)$ that appears in K_0 is $Domain(P, \varepsilon)$ and that $\mu = \{A, B\}$; thus, our first test (for $Domain(x, y)$) succeeds only for the pairs (P, A), (P, B). Our second test ($PS(x) \notin K_0$ or $CS(y) \notin K_0$) succeeds only for the pair (P, A), because $PS(P) \in K_0$, $CS(A) \notin K_0$ and $CS(B) \in K_0$. Thus, the only violated instance of c_4 is $c_4(P, A)$. Moreover, $c_4(P, A)$ is violated when $Domain(P, \varepsilon)$ is mapped to $Domain(P, A)$, i.e., for the map $\mu_V = \{A\}$ (stemming from the restriction $A \in \mu_V$); the non-violating mapping is the complement of μ_V, i.e., $\mu_{NV} = \mu \setminus \mu_V = \{B\}$. For the violating elements of $[[K_0]]$, the resolution options are, either the removal of $Domain(P, \varepsilon)$, or the addition of $CS(\varepsilon)$ (under the mapping μ_V in both cases).

Therefore, given a KB with wildcards and some violated constraint instance $c(\vec{A})$, the resolution can be made either by restricting ourselves to the non-violating mappings, or to take the

violating ones and resolve the violation in the standard manner (adding / removing atoms). Per the definition of application, the first option corresponds to applying the delta $[[\delta_{NV}]]^{\mu_{NV}}$, where $\delta_{NV} = \langle \emptyset, \emptyset \rangle$ and μ_{NV} is the non-violating mapping; the second option corresponds to applying the deltas $[[\delta_{Vi}]]^{\mu_V}$, $i = 1, 2, \ldots$ where δ_{Vi} are the deltas that would resolve the constraint, calculated as in the case of violations without wildcards, and μ_V is the violating mapping. In our running example, the resolution options are $[[\delta_{NV}]]^{\mu_{NV}}$, $[[\delta_{V1}]]^{\mu_V}$, $[[\delta_{V2}]]^{\mu_V}$ where: $\delta_{NV} = \langle \emptyset, \emptyset \rangle$, $\mu_{NV} = \{B\}$, $\delta_{V1} = \langle \emptyset, \{Domain(P, \varepsilon)\} \rangle$, $\delta_{V2} = \langle \{CS(\varepsilon)\}, \emptyset \rangle$, $\mu_V = \{A\}$. These deltas can be used to resolve the violation of the instance $c(\vec{A})$ w.r.t. KB K and belong in the resolution set $Res(c(\vec{A}), K)$. We will use the same symbol for KBs with wildcards: $Res(c(\vec{A}), [[K]])$. The set $Res(c(\vec{A}), K)$ (or $Res(c(\vec{A}), [[K]])$) can be computed as above. For the given example, $Res(c_4(P, A), [[K]]) = \{ [[\delta_{NV}]]^{\mu_{NV}}, [[\delta_{V1}]]^{\mu_V}, [[\delta_{V2}]]^{\mu_V} \}$.

2.2 Declarative Repairing Policies

We have seen in Section 1.3 that repairing policies essentially reflect the notion of minimality [1], a curator wants to impose to the constraint resolution choices. However, our objective is to define a highly customizable method for restoring invalidities and the above modeling is inadequate for this purpose. To achieve our goal, the curator should be able to provide a set of "specifications" (e.g.,"I want repairs to differ as little as possible from the original KB; in case of ties, I prefer to remove information") to determine his repair policy. To formalize the idea of specifications and declaratively specify such a policy, we consider a finite set of interesting *features* of a repair (e.g., number of updates) along with atomic and composite *preferences* upon these features. Recall that depending on the employed strategy (VB/IB), curators' preferences can be applied after each individual resolution (in each node of the resolution tree) or at the end of the repairing process (in each leaf of the resolution tree).

So, let's take some K and a potential repair K_r, and set $\delta_r = \langle \delta_a, \delta_d \rangle$; it follows that $K \bullet \delta_r = K_r$. To capture the idea of features, we will associate each repair with a certain value (usually a number), which represents the value of the "significant property"; thus, a feature can be encoded as a function. Features could be defined over the repairs themselves (like K_r) or over the repairing deltas (like δ_r); given the 1-1 association between K_r and δ_r, both approaches are equivalent from a formal point of view. This association results from the relation of K_r with δ_r ($K_r = K \bullet \delta_r$). Here, we chose the latter, because, intuitively, we tend to define specifications over repairing deltas rather than repairs. More formally, a feature is a functional attribute of the delta under question, whose value (usually a number) represents some interesting property of the delta. As a result it can be modeled as a function over a set S such that $f_A : \Delta \mapsto S$. For example, to specify the aforementioned repairing policy we need to define the "number of updates" in a delta $\delta = \langle \delta_a, \delta_d \rangle$ for a KB K as a feature: $f_{size}(\delta) = f_{additions}(\delta) + f_{deletions}(\delta)$ where $f_{additions}(\delta) = |\delta_a|$, $f_{deletions}(\delta) = |\delta_d|$. We refer the reader in Appendix B where we have defined a number of "interesting" features to us. Obviously, anyone can define his own features over repairing deltas with respect to the above definit.

To formalize the notion of declarative repairing policies, we rely on qualitative preference models proposed for relational databases [18, 9, 23]. An atomic *preference* $(f_A, >_P)$ over a delta feature f_A states essentially a binary relation among its possible values. Preference relations usually satisfy some intuitive properties like reflexivity and transitivity, i.e., they are preorders [18]. Intuitively, $x >_P y$ means "x is more preferred than y". For numerical values, for which a natural order is defined, such preference relations can be stated in a compact way using aggregate functions over feature values. For example, the expression $Min(f_A)$ states that "the lowest available values for f_A are more preferred than others" and thus corresponds to the

preference relation: $x >_P y$ iff $f_A(x)$ is minimal, but $f_A(y)$ is not; for more details, see [18]. As usual, for categorical values, the preference relation has to be explicitly enumerated by the curators.

Atomic preferences can be further composed using operators such as & (prioritized) and $\otimes$ (pareto). For example, the composite preference $P_1 \& P_2$ states that P_1 is more important than P_2, so P_2 should be considered only for values which are equally preferred w.r.t. P_1. Similarly, $P_1 \otimes P_2$ states that P_1 and P_2 are of equal importance. We refer the reader to [18] for an extended formal description of the semantics of these preference expressions. Note also that we maintain the distinction between equally preferred and incomparable values [18]. Going back to our example repairing policy, we need to define the atomic preferences $P_1 = Min(f_{size})$ and $P_2 = Max(f_{deletions})$ and compose them using lexicographic composition ($P = P_1 \& P_2$), giving priority to P_1, as the total delta size is more important. Given an atomic or composite preference expression, we can trivially induce an order $(\Delta, >)$ over deltas.

Features and preferences can be easily generalized to support deltas with wildcards. Specifically, for a feature f_A, we set $f_A([[\delta]]) = \{f_A(\delta_0) | \delta_0 \in [[\delta]]\}$ and $[[\delta_1]] >_P [[\delta_2]]$ iff $\delta_{10} >_P \delta_{20}$ for all $\delta_{10} \in [[\delta_1]]$, $\delta_{20} \in [[\delta_2]]$.

Besides the order $>_P$, a repairing policy must also specify the strategy (IB/VB) to follow for the filtering of the non-preferred repairing options. In the following, we will use the symbol $>_P^I$ ($>_P^V$) to denote the IB (VB) policy that this order defines. A KB K' is called a *preferred repair* for a KB K iff it is consistent and optimal (per $>_P$) w.r.t. K. Note that the preferred repairs are different depending on the employed strategy (IB/VB).

For the VB strategy, a KB K' is a preferred repair for K iff it is consistent and there is some sequence of deltas, whose sequential application upon K leads to K'. Since the order of violation resolutions in VB creates different repair sequences, we assume the existence of some arbitrary, but fixed evaluation order for constraints, encoded as a *violation selection function* ($NextV$); given an inconsistent KB, this function determines the next constraint violation to consider. Note that VB requires each delta in the sequence of resolutions to be *locally* optimal, i.e., w.r.t. the other resolution options in the resolution set for the considered violation. Formally:

Definition 6 *Consider a VB repairing policy $>_P^V$, a set of integrity constraints $\mathcal{C}$ and some KB K, and a violation selection function $NextV$. A sequence of KBs $SEQ = \langle K_1, K_2, \ldots \rangle$ is called a* preferred repairing sequence *of K for $>^V$ iff:*

1. *$K_1 = K$.*

2. *If $K_i \vDash \mathcal{C}$ then $K_{i+1} = K_i$, else $K_{i+1} = K_i \bullet \delta$, where $\delta \in Res(NextV(K), K)$ and there is no $\delta' \in Res(NextV(K), K)$ such that $\delta' > \delta$.*

We say that SEQ terminates after n steps *iff $K_n = K_{n+1}$ and either $n = 1$ or $K_{n-1} \neq K_n$. A KB K' is called a* preferred repair *of K for $>_P^V$ iff there is some preferred repairing sequence SEQ of K for $>_P^V$ which terminates after n steps and $K' = K_n$.*

Let us now consider the preference expression described earlier $P = Min(f_{size}) \& Max(f_{deletions})$, the KB K of our motivating example (see Section 1.3), and the corresponding order that defines the repairing policy $>_P^V$. If we choose to repair $c_1(B, A)$ first (i.e., $NextV(K) = c_1(B, A)$), we would take $K_1 = K$, $K_2 = K_1 \bullet \langle \emptyset, \{C\_IsA(B, A)\} \rangle$ as this is the only delta in $Res(c_1(B, A), K)$ with size 1 and 1 deletion (cf. Figure 1.3 and Table 2.2). Similarly, $K_3 = K_2 \bullet \langle \emptyset, \{PS(P)\} \rangle$ (to resolve $c_2(P)$, where we assumed that $NextV(K_2) = c_2(P)$), and $K_4 = K_3 \bullet \langle \emptyset, \{Range(P, B)\} \rangle$

Table 2.1: Selecting the Preferred Resolution Option for $c_1(B, A)$

Resolution option (delta) for $c_1(B, A)$	size	deletions
$\langle \emptyset, \{C\_IsA(B, A)\} \rangle$	1	1
$\langle \{CS(A)\}, \emptyset \rangle$	1	0

(to resolve $c_3(P, B)$). Then, K_4 is consistent so $K_4 = K_5 = \ldots$. This creates a preferred repairing sequence terminating after 4 steps (which in this example is the only possible one), so $K_4 = \{CS(B)\} = K_{111}$ is a unique preferred repair.

Definition 6 can be generalized to support KBs and deltas with wildcards in the standard way, i.e., by replacing K (δ) with $[[K]]$ ($[[\delta]]$). Note also that $NextV$ must be overloaded to be applicable on sets of KBs (i.e., $NextV([[K_i]])$).

For the IB strategy, a KB K' is a preferred repair for K iff it is consistent and there is some delta which is *globally* optimal per $>_P^I$, and whose application upon K leads to K'. Unlike VB strategies, the individual resolution options are not considered to determine the preferred repairing deltas; instead, the cumulative effect of individual resolutions is considered and compared. When considering policies adopting the IB strategy, there is a subtle issue deserving further clarification. If we restrict ourselves only to optimality defined by the preference expression (per $>_P^I$) we may not always obtain *useful repairs*. For instance, consider the resolution tree of Figure 1.4 and the preference $Min(f_{deletions})$. Under this preference, the optimal repairs occur from the branches K_{22}, K_{23}, each of which causes no deletions. Consider now the following KB $K_I = K_{22} \cup \{CS(C)\}$. Compared to the original K, K_I also contains no deletions, and is consistent, so it should be equally preferred to K_{22}, K_{23}. However, K_I contains one extra fact ($CS(C)$), which did not exist in the original KB and is totally irrelevant to the resolution process (i.e., it was arbitrarily added, without being dictated as a resolution of some constraint violation). Therefore, it is not useful to return K_I as a preferred repair along with K_{22}, K_{23}. To avoid such cases, we put an additional requirement, namely that the delta leading to the preferred repair should be minimal, w.r.t. the *subset relation*, i.e., there should be no other delta δ' leading to a consistent KB ($K \bullet \delta'$: consistent) that contains a subset of the updates in δ. Such deltas will be called *useful*. This requirement is in accordance to the minimality notion established in [1, 10]. This issue does not arise in VB strategies, because the sequence of deltas that leads to a preferred repair contains (by definition) only updates that are actually dictated by the violation resolution.

Definition 7 *Consider an IB repairing policy $>_P^I$, a set of integrity constraints $\mathcal{C}$ and some KB K. A delta $\delta = \langle \delta_a, \delta_d \rangle$ is a* preferred repairing delta *of K for $>^I$ iff:*

1. *$K \bullet \delta \models C$.*
2. *The δ is useful, i.e., there is no $\delta' = \langle \delta'_a, \delta'_d \rangle$ such that $K \bullet \delta' \models C$, $\delta'_a \subseteq \delta_a$, $\delta'_d \subseteq \delta_d$ and $\delta \neq \delta'$.*
3. *There is no δ' satisfying the above two requirements for which $\delta' >^I \delta$.*

A KB K' is called a preferred repair *of K for $>_P^I$ iff there is some preferred repairing delta δ of K for $>_P^I$ such that $K' = K \bullet \delta$.*

Note. The second bullet states that we require repairing deltas to be minimal w.r.t. $\subseteq$; this is imposed in order to guarantee that we discard repairing deltas which violate the Principle of Minimal Change [2].

Table 2.2: Selecting the Preferred Repairing Delta of K

Potentially Preferred Repairing Delta	size	deletions
$\delta_{111} = \langle\emptyset, \{$C_IsA(B,A), PS(P),Range(P,B)$\}\rangle$	3	3
$\delta_{12} = \langle\{$Domain(P,B)$\}$, $\{$C_IsA(B,A)$\}\rangle$	2	1
$\delta_{131} = \langle\{$Domain(P,A), CS(A)$\}$, $\{$C_IsA(B,A)$\}\rangle$	3	1
$\delta_{211} = \langle\{$CS(A)$\}$, $\{$PS(P), Range(P,B)$\}\rangle$	3	2
$\delta_{22} = \langle\{$CS(A),Domain(P,B)$\}$, $\emptyset\rangle$	2	0
$\delta_{23} = \langle\{$CS(A),Domain(P,A)$\}$, $\emptyset\rangle$	2	0

Consider again the KB K of Section 1.3, the preference $P = Min(f_{size})$ & $Max(f_{deletions})$ and the corresponding order that defines the repairing policy $>^I_P$. Then, δ_{12} is a preferred repairing delta (cf. Figure 1.4 and Table 2.2). Note that δ_{131} also repairs the KB (i.e., $K \bullet \delta_{131} \vDash C$), but is not useful (cf. δ_{23}). Similarly, δ_{111} satisfies the first two requirements of Definition 7, but not the third, because its size is 3, so for δ_{12} (whose size is 2) it holds that $\delta_{12} >_P \delta_{111}$. (cf. Table 2.2).

Again, Definition 7 can be easily generalized for KBs/deltas with wildcards by replacing K with $[[K]]$ (same for deltas). Note that requirement #2 in Definition 7, should hold for all $\delta_s \in [[\delta]]$, i.e., there should be no δ' satisfying these relations for any $\delta_s \in [[\delta]]$.

2.2.1 Formal Properties

In Section 1.3 it was shown that when the VB strategy is employed, the syntax of the constraints affects the preferred repairs. This is not true when the IB strategy is employed:

Proposition 1 *Consider two sets of integrity constraints C, C' such that $C \equiv C'$ and a repairing policy $>^I$. Then K' is a preferred repair of K per $>^I$ for the constraints C iff it is a preferred repair of K per $>^I$ for the constraints C'.*

To compare existing repair approaches with our framework, we will model a *repair finding algorithm* as a function R taking as input an inconsistent KB and returning a non-empty set of consistent KBs: $R : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset$, such that for all K and all $K' \in R(K)$ it holds that $K' \vDash C$. Our objective is to characterize exactly the properties that a repair finding algorithm must satisfy in order to be *expressible* by some policy $>^I$ (or $>^V$); in other words, we are looking to characterize exactly the repair finding algorithms that can be captured by our framework using a policy under the IB (or VB) strategy. Formally, a repair finding algorithm R will be called *IB-expressible* (*VB-expressible*) iff there is some repairing policy $>^I$ ($>^V$) such that for all KBs $\mathcal{K}$ it holds that $K' \in R(K)$ iff K' is a preferred repair for K per $>^I$ ($>^V$). We formally define the corresponding sets of preffered deltas and repairs of a KB K with respect to a repair policy:

Definition 8 *Consider a KB K, a set of integrity contraints $\mathcal{C}$ and a repairing policy $>^P$ where $P \in \{I, V\}$. We define the set $PR^P(K) = \{K' |$ K' is a preffered repair of K per the repairing policy $>^P\}$.*

Definition 9 *Consider a KB K, a set of integrity contraints $\mathcal{C}$ and an information based repairing policy $>^I$. We define the set $PRD^I(K) = \{\delta | \delta$ is a preferred repairing delta of K for $>^I\}$.*

Note. Based on the above two definitions, for a given KB K, it holds $PR^I(K) = \{K \bullet \delta | \delta \in PRD^I(K)\}$.

The following propositions describe the aforementioned characterization and prove the generality of our framework:

Proposition 2 *A repair finding algorithm R is IB-expressible iff for all KBs $\mathcal{K}, K_r, K'_r$ for which $K_r \in R(K)$, $K'_r \vDash \mathcal{C}$, $K'_r \setminus K \subseteq K_r \setminus K$ and $K \setminus K'_r \subseteq K \setminus K_r$, it holds that $K'_r = K_r$.*

Proposition 3 *A repair finding algorithm R is VB-expressible iff $R(K) = \{K\}$ when $K \vDash \mathcal{C}$ and there is a family of repair finding algorithms $\{R_0^{c(\vec{A})} | c(\vec{A})$: constraint instance$\}$ such that $R_0^{c(\vec{A})}$ is an IB-expressible repair finding algorithm which considers only one integrity constraint, namely $\{c(\vec{A})\}$, and $R(K) = \bigcup_{K_0 \in R_0^{c(\vec{A})}(K)} R(K_0)$ where $c(\vec{A}) = NextV(K)$, when $K \nvDash \mathcal{C}$.*

The condition of Proposition 2 is quite general and implies that a repair finding algorithm is IB-expressible iff it returns useful repairs. Similarly, the condition of Proposition 3 captures the recursive and "memory-less" character of VB strategy: we select a violated rule ($c(\vec{A}) = NextV(K)$), repair it in an optimal manner ($R_0^{c(\vec{A})}(K)$), then start over. The discrimination between consistent and inconsistent KBs in Proposition 3 is necessary, because for a consistent KB K, $NextV(K)$ is not defined. Note that, in both cases, the requirements stem from the intuition behind IB/VB strategies, not by the use of preferences. Therefore, preferences form an extremely powerful tool for modeling repair approaches; in Section 6, we will review existing repairing frameworks that are reducible to our framework.

2.3 The Issue of Modifications

Apart from additions and deletions, one could also imagine that *modification* of tuples could also be useful to resolve certain violations. For instance, consider the DED $c_{v1} = \forall x, y, z, x', y', z'$ $P(x, y, z) \wedge P(x', y', z') \rightarrow (x = x')$ and the KB $K_v = \{P(a, b, c),\ P(a', b, c'),\ R(a, b, c)\}$. Obviously, K_v violates c_{v1}; the available resolutions in our framework are the removal of $P(a, b, c)$ and the removal of $P(a', b, c')$ leading to the repairs $K_{v1} = \{P(a', b, c'), R(a, b, c)\}$, $K_{v2} = \{P(a, b, c), R(a, b, c)\}$ respectively. However, the equality in the rule's head provides two more options if we consider tuple modifications: namely, we could replace a with a' (or vice-versa) and thus modify the tuples $P(a, b, c)$ (or $P(a', b, c')$) into $P(a', b, c)$ ($P(a, b, c')$). These options would give the repairs $K_{v3} = \{P(a', b, c),\ P(a', b, c'), R(a, b, c)\}$, $K_{v4} = \{P(a, b, c), P(a, b, c'), R(a, b, c)\}$ respectively.

However, special care should be taken in this respect for equality axioms, as it is not clear whether the corresponding replacements should be made "locally", i.e., affecting only the tuples that participate in the violated rule, or "globally", i.e., affecting all the tuples; depending on the application, and the semantics associated with the constants, different options may be more adequate. Note also that in other applications, we may want to overrule the replacements altogether. To allow the application to determine the desired semantics, we consider three different types of equality relations. All three have the same semantics, as far as diagnosis (constraint checking) is concerned, but they differ on the resolution policy that they imply:

1. $x = y$, which means that the unique name assumption should be imposed for x, y, i.e., if x, y are the same constant, then the equality is true, but if they are not the same constant,

then the equality is false. Note that expressions of the form $x = y$ cannot be resolved, i.e., the expression $x = y$ in a DED means that the knowledge engineer does not want to allow equating x, y as part of the repair process. This semantics for equality are useful in certain contexts [1], [7], [10] e.g., for primary key constraints $\forall x, y, z P(x, y) \wedge P(x, z) \rightarrow y = z$, where one wants to either delete $P(x, y)$ or $P(x, z)$, but not equate x with y.

2. $x \equiv y$, which essentially allows the two constants to be equated as part of the repair process. Again, the equality is true if and only if the two constants (x, y) are the same. The difference from $=$ is that the unique name assumption is not made. Therefore, one could resolve the corresponding violated DED rule by replacing x with y (i.e., $y \rightarrowtail_{c(\vec{a})} x$) or vice-versa (i.e., $y \rightarrowtail_{c(\vec{a})} x$); note that $\equiv$ enforces a "local" replacement, i.e., only the occurences of x (respectively y) that are related to the given constraint instance $c(\vec{a})$ (will be defined next) will be replaced by y (respectively x). Note that $\equiv$ is particularly useful in the database context, where x,y are usually tuple values, which need to be locally replaced only [6, 39]; it is also useful in the ontological context, whenever x, y represent literals, rather than URIs.

3. $x \equiv_u y$, which is similar to $\equiv$, but enforces a universal replacement. In particular, $\equiv_u$, like $\equiv$, allows the two constants to be equated as part of the repair process; again, the equality is true if and only if the two constants (x, y) are the same. As with $\equiv$, the difference from $=$ is that the unique name assumption is not made, so one could resolve the corresponding violated DED rule by replacing x with y (i.e., $y \rightarrowtail_u x$) or vice-versa (i.e., $x \rightarrowtail_u y$). The difference from $\equiv$ is hat $\equiv_u$ enforces a universal replacement, i.e., all occurences of x (respectively y) will be replaced by y (respectively x). Note that $\equiv_u$ is particularly useful in cases where x, y represent schema constants (e.g., URIs in ontologies); in such cases, replacement should be done universally, essentially eliminating one of the costants in favor of the other.

Discussion. The type of "data" on which we examine consistency, affects the expected semantics of equality atoms which appear in the head of the integrity constraints. We argue that only the curator of the KB knows the semantics of the equality atoms he wants to enforce with respect to the context of his KB. For example, if the constraint $\forall x, y, z P(x, y) \wedge P(x, z) \rightarrow (z = y)$ refers to data (e.g., in context of relational databases), it could be restated as: $\forall x, y, z P(x, y) \wedge P(x, z) \rightarrow z \equiv y$, causing "local" replacements at the level of single constraint instances during repair. On the other hand, if it refers to schema information, where we need global replacements, the constraint could be written as $\forall x, y, z P(x, y) \wedge P(x, z) \rightarrow z \equiv_u y$. Note that if the curator wants to try different resolution policies regarding equality, he could use more than one "equalities" in his DEDs using disjunction and the rule could, e.g., be written as $\forall x, y, z P(x, y) \wedge P(x, z) \rightarrow (z \equiv y) \vee (z \equiv_u y)$. We should mention here that this kind of transformation, does not change the expressiveness of DED rules for capturing various types of constraints as all these equality relations have the same semantics as far as diagnosis is concerned.

2.3.1 Modifications In Our Framework

The option of tuple modifications has been already proposed in the literature, e.g., in [13] and the field of relational databases. It could be easily incorporated as additional options in the resolution set of constraints for both the IB and the VB strategy, without affecting the rest of our formal model.

First of all, we should extend the form of the DED rules in order to express which type of the above equality semantincs we would like to impose. For example, the above rule $c_{v1} = \forall x, y, z, x', y', z'\ P(x, y, z) \wedge P(x', y', z') \rightarrow (x = x')$ can be rewriten as next w.r.t. the previously defined categories:

c_{v11}: $\forall x, y, z, x', y', z'\ P(x, y, z) \wedge P(x', y', z') \rightarrow (x = x')$.

c_{v12}: $\forall x, y, z, x', y', z'\ P(x, y, z) \wedge P(x', y', z') \rightarrow (x \equiv x')$.

c_{v12}: $\forall x, y, z, x', y', z'\ P(x, y, z) \wedge P(x', y', z') \rightarrow (x \equiv_u x')$.

It is clear that again, the syntactic form of the constraint denotes the way it will be resolved. As a result, taking into account the above possible semantics of the equality, the corresponding repairing deltas would be:

- $\delta_{11} = \langle \emptyset, \{P(a, b, c)\}\rangle$, $\delta_{12} = \langle \emptyset, \{P(a', b, c')\}\rangle$.
- $\delta_{21} = \langle \emptyset, \{P(a, b, c)\}\rangle$, $\delta_{22} = \langle \emptyset, \{P(a', b, c')\}\rangle$, $\delta_{23} = \langle \{P(a', b, c)\}, \{P(a, b, c)\}\rangle$ (i.e., replace a with a' locally), $\delta_{24} = \langle \{P(a, b, c')\}, \{P(a', b, c')\}\rangle$ (i.e., replace a' with a locally).
- $\delta_{31} = \langle \emptyset, \{P(a, b, c)\}\rangle$, $\delta_{32} = \langle \emptyset, \{P(a', b, c')\}\rangle$ $\delta_{33} = \langle \{P(a', b, c), Q(a', b)\}, \{P(a, b, c), Q(a, b)\}\rangle$ (i.e., replace a with a' universally), $\delta_{34} = \langle \{P(a, b, c')\}, \{P(a', b, c')\}\rangle$ (i.e., replace a' with a universally).

Nevertheless, tuple modifications have not been incorporated in our framework because they cause the IB strategy to lose some of its nice properties(see Appendix A, corollary 3); in particular, unlike the case where no tuple modifications are considered, the leaves of the IB resolution tree may be different depending on the constraint evaluation order when tuple modifications are involved.

To illustrate this, consider the KB K_v and c_{v1} as above, and the constraint $c_{v2} = \forall x, y, z P(x, y, z) \rightarrow Q(x, y, z)$. Let us suppose that c_{v2} is considered first, and let us consider a very specific branch, namely the one that resolves c_{v2} by subsequently adding $Q(a, b, c)$ and $Q(a', b, c')$. Then, c_{v1} must be also resolved; let us consider the branch that replaces $P(a, b, c)$ with $P(a', b, c)$ (i.e., we decided to replace a with a' locally). Then c_{v2} is violated again, due to the addition of $P(a', b, c)$; suppose that this is resolved by adding $Q(a', b, c)$. This will lead to one of the leaves of the resolution tree, which corresponds to the potentially preferred KB $K_a = \{P(a', b, c),\ P(a', b, c'),\ Q(a, b, c),\ Q(a', b, c),\ Q(a', b, c')\}$. If we had chosen the opposite evaluation order, then c_{v1} would be considered first. Regardless of the resolution option chosen, one of $P(a, b, c)$ or $P(a', b, c')$ would be removed (possibly to be replaced with another tuple). Thus, when c_{v2} is subsequently violated, at least one of $P(a, b, c), P(a', b, c')$ will not exist, so the corresponding constraint instance will not be violated; as a result, one of $Q(a, b, c), Q(a', b, c')$ will not appear in any of the potentially preferred repairing KBs, so K_a will not be returned as one of the potentially preferred KBs. Depending on the preference, K_a could have been the preferred repairing KB. Thus, the inclusion of tuple modifications in our framework jeopardizes the correctness of our creation process for the IB resolution tree.

Chapter 3

Repair Finding Algorithms

In this chapter we provide the two algorithms implementing IB and VB strategies using the formal framework presented previously. As usual, when wildcards are considered, the output of the algorithms is repairs with wildcards as well and the employed symbols (K, δ, $\models$, $Res(c(\vec{A}), K_c)$, $\uplus$, $\bullet >$, etc) should be read under their extended semantics. Next we suggest some two general purpose optimizations for IB, VB algorithms (i.e., IB_{opt}, VB_{opt}) and some specific optimizations for IB algorithm (i.e., IB_{prun}) and finally we give the complexity bound for these algorithms. We use a simple *Repair* function (Algorithm 1), which takes as input the preferred repairing policy $>^P$(i.e., $>^V$ or $>^I$) a KB K, and, depending on the policy, calls either IB or VB (Algorithms 2, 3), which calculates the preferred repairs using either an information based or a violation based strategy respectively. Note that although K does not initialy contain ground facts with wildcards, no one can guarantee that they will not be inserted during the repair process (i.e., when constraints with existential quantifiers are violated). For generality purposes, in our algorithms we consider that we deal with KBs and deltas which contain wildcards.

Algorithm 1: $repair(>^P, [[K]])$

```
1: if P = I then
2:     RD = ∅
3:     IB([[K]], ⟨∅, ∅⟩)
4:     PR = {[[K]] • [[δ]] | [[δ]]: preferred (per >^I) deltas in RD}
5: else
6:     PR = ∅
7:     VB([[K]])
8: end if
9: return PR
```

Let's consider the information-based algorithm first (Algorithm 2), IB. The input to IB is the current KB K_c and the delta that has been computed so far δ_{tot} (initially $\delta_{tot} = \langle \emptyset, \emptyset \rangle$), which is necessary to perform the final filtering using $>$. If K_c is invalid, then we find the next violated constraint (if any), say $c(\vec{x})$, and compute $Res(c(\vec{x}), K_c)$. In this approach, it is not necessary to use a violation selection function (like $NextV$), because the order we chose to resolve invalidities does not affect the result. Note that in line 5, we use the composition operator ($\uplus$) to guarantee that δ_{tot} always corresponds to a delta which, if applied to the original KB K, would give K_c.

We should mention here that the composition of deltas may create deltas which are "problematic". I.e., consider a $\delta_s = \langle \delta_{as}, \delta_{ds} \rangle$.t. $\delta_{as} \cap \delta_{ds} \neq \emptyset$. This means that delta contains *conflicting*

Algorithm 2: $IB([[K_c]], [[\delta_{tot}]])$

1: **if** $[[K_c]] \not\models \mathcal{C}$ **then**
2: Find $Res(c(\vec{A}), [[K_c]])$ for a violated constraint instance $c(\vec{x})$
3: **for all** $[[\delta]] \in Res(c(\vec{A}), [[K_c]])$ **do**
4: $[[\delta_{tmp}]] = [[\delta_{tot}]]$
5: $[[\delta_{tot}]] = [[\delta_{tot}]] \uplus [[\delta]]$
6: $clean([[\delta_{tot}]])$
7: **if** $isConflicting([[\delta_{tot}]]) = false$ **then**
8: $IB([[K_c]] \bullet [[\delta]], [[\delta_{tot}]])$
9: **else**
10: $[[\delta_{tot}]] = [[\delta_{tmp}]]$
11: **end if**
12: **end for**
13: **else**
14: $RD = RD \cup \{[[\delta_{tot}]]\}$
15: $RD = RD \setminus \{[[\delta]] \mid [[\delta]]$ is a non useful repairing delta$\}$
16: **end if**

changes and such changes in fact introduce violations over constraint instances which was repaired in previous states in the same repairing sequence (i.e., branch in the resolution tree). In IB algorithm, such branches are pruned in the repair process this is succeeded by the use of two assisting algorithms, namely *clean* and $isConflicting$. Algorithm $isConflicting$(Algorithm 4) takes as input a repairing delta, possibly with wildcards, and checks if its sets δ_a, δ_d have any common element. The same idea is followed in algorithm *clean*((Algorithm 5)), but this time we are interested on the mapping which is related with the δ given as input. More specificaly, each delta which can be constructed by replacing the wildcards of the initial with the appropriate values denoted by μ, is passed into $isConflicting$ function. If it contains conflicting changes (line 3 of *clean* function), we remove the corresponding delta from $[[\delta_{tot}]]^{\mu}$. The corresponding algorithms for the functions *clean* and $isConflicitng$ can be found in Appendix C. If we will not prune such branches which bring the KB into a previous state we have two main problems: a) there will exist branches which will not lead to repairs (i.e., non terminating) b) for the rest branches which will terminate, they will lead to non useful deltas. Let's illustrate this with a simple example taken from propositional logic. Consider the constraints $c_1 : C \rightarrow A$, $c_2 : C \rightarrow B$, $c_2 : A \rightarrow E$ and a KB $K = \{C\}$ which violates constraints c_1, c_2. Considering also the resolution order $c_1 \rightsquigarrow c_2 \rightsquigarrow c_3$ the user can verify that we have the resolution tree depicted in Figure 3.1. Let us consider the repairing option which inserts A into K to repair c_1. KB K_2 violates c_2 and the repairing options are either the removal of C or the addition of B. Let us consider the removal of C which leads to KB K_{21} which in turn violates c_3. At this point we clearly see the two problems we discussed before. If we choose the repairing option which enforces the removal of A we have the delta of the form $\langle\{A\}, \{C, A\}\rangle$ which does not lead to a repairing delta (it violates constraint c_1) and we have a delta of the form $\langle\{A, E\}, \{C\}\rangle$ which is not useful due to delta $\langle\emptyset, \{C\}\rangle$. Back in state K_2, the repairing option to which enforces the insertion of B leads into the KB K_{22} which in turn violates c_3. Next, the choosing to remove A creates the delta of the form $\langle\{A, B\}, \{A\}\rangle$ which is not a repairing delta as it violates c_2 and choosing the insertion of E creates the delta of the form $\langle\{A, B, E\}, \emptyset\rangle$ which is a repairing delta.

Back in IB algorithm, if the composed delta $[[\delta_{tot}]]$ does not contain conflicting changes (line 7), then the possible repairing delta ($\delta \in Res(c(\vec{x}), K_c)$) is applied upon K_c and the result is

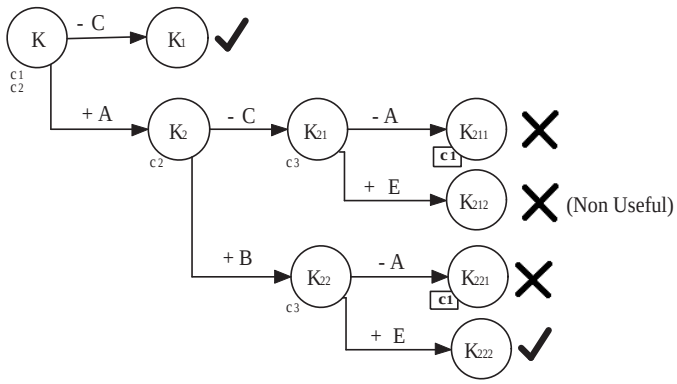

Figure 3.1: Dealing with Conflicting Changes

used to spawn a new recursive branch ($IB(K_c \bullet \delta, \delta_{tot} \uplus \delta)$). Otherwise, we bring the delta to its previous state $[[\delta_{tot}]]$ (line 9) and we examine the next repairing delta from $Res(c(\vec{x}), K_c)$, if any. If K_c is valid, then one potential solution has been found. A potential solution is a repair of K but as the preference based repair ordering is applied upon repairing deltas we must firstly compute and store the respective repairing deltas (line 13 in IB algorithm) in a global set of repairing deltas RD (initialy empty). Next we remove the non useful repairing deltas from set RD because as mentioned in def. 7, the preferred repairing deltas must be also useful. Note that not all remaining deltas preferred deltas; they are just potential solutions. Upon return, the *Repair* function, compares the elements of RD, keeping only the "optimal" (preferred) ones per the $>^I$. Also, in *Repair* function those preferred deltas are applied to K, and the result of each application is put in PR and returned.

The violation-based strategy, VB (Algorithm 3), is very similar. There are three main differences: first, we don't arbitrarily select the next rule to consider (line 2), but function $NextV$ is used instead for this purpose; second, not all deltas in $Res(r(\vec{x}), K)$ are used to spawn a new recursive branch, only the preferred ones (per $>$); thirdly, when reaching a leaf node (i.e., when $K_c \vDash \mathcal{C}$), the current KB, K_c, is definitely a preferred repair, so it is simply added to a global variable, PR (initially empty). In this case, the *Repair* function need not make any additional steps; the contents of PR are returned. As described in def. 6 we can model the process described in VB function as a repairing sequence which terminates when a preferred repair is found. To avoid the case where a repairing delta, creates a KB instance K_i which is the same with an instance K_j, $j < i$ (i.e., a circular sequence) we use a stack KB (initialy empty) to store the intermediate KB instances (line 7). Next, in line 6, we check whether we revisit a KB's instance and rejecting this sequence(i.e., apperance of a circular sequence) or not.

As implied by Definition 6, the preferred repairs returned by the VB Algorithm 3 are sensitive to the constraint resolution order ($NextV$). For IB Algorithm 2, no matter which evaluation order is considered, the computed preferred repairs will be the correct ones, which, per Definition 7 are irrelevant to the evaluation order of constraints. The described repair algorithm (*repair*) can be shown to return the correct result per the selected repairing policy $>^I$ or $>^V$ (see also the discussion on complexities below). Formally:

Proposition 4 *Consider a KB K, a constraint selection function $NextV$ and an information based repairing policy $>^I$. Then, function repair returns exactly the preferred repairs of K with respect to $>^I$.*

Algorithm 3: $VB([[K_c]])$

```
1: if [[K_c]] ⊭ C then
2:    Find Res(c(A⃗), [[K_c]]), where c(A⃗) = NextV([[K_c]])
3:    RD = {preferred (per >) deltas in Res(c(A⃗), [[K_c]])}
4:    for all [[δ]] in RD do
5:       [[K_c]] = [[K_c]] • [[δ]]
6:       if K_c ∉ KB then
7:          push(KB, [[K_c]])
8:          VB([[K_c]])
9:       end if
10:   end for
11: else
12:   PR = PR ∪ {[[K_c]]}
13:   pop(KB)
14: end if
```

Proposition 5 *Consider a KB K, a constraint selection function $NextV$ and a violation based repairing policy $>^V$. Then, function repair returns exactly the preferred repairs of K with respect to $>^V$.*

3.1 Optimizations

In this section we illustrate a series of optimizations which aim at reducing the execution time and the memory requirements of our repair finding algorithms. These optimizations can be separated into two categories; a) those wich can be applied in both IB and VB algorithm and b) those which can be applied in IB algorithm exclusively.

3.1.1 Optimizations for IB Algorithm

Despite the exhaustive nature of the IB algorithm, there are cases where we know, a priori, that a certain branch will not lead to a preferred repair, so it can be pruned. As an example, consider that the preference is $Min(f_{size})$; if we have already found a potential repairing delta with a certain size (e.g., n), then we can prune all branches whose delta δ_{tot} has size larger than n, because none of the deltas eventually produced by this branch can be a preferred one. Unfortunately, such optimizations are preference-dependent and cannot be applied in the general case.

On the other hand, we can exploit requirement for useful repairs in Definition 7 (which does not depend on the employed preference) in order to prune branches that cannot lead to a preferred delta. To grasp the intuition, consider the motivating example of Section 1.3 and the corresponding Figure 1.4. Consider the branch that led to K_{131}: in the last step, we resolve $c_4(P, A)$, by adding $CS(A)$. The same operation (adding $CS(A)$) would have resolved the first violated constraint in the resolution tree, $c_1(B, A)$, but the branch under consideration chose an alternative resolution, namely the removal of $C\_IsA(B, A)$. Since this branch (K_{131}) eventually forced us to add $CS(A)$, we conclude that if we had chosen the addition of $CS(A)$ to resolve $c_1(B, A)$ in the first place, we would have got a "smaller" delta, so the current delta (δ_{131}) is not useful (δ_{23} in figure). Thus, K_{131} could not have been a preferred repair, regardless of preference. This observation can be generalized:

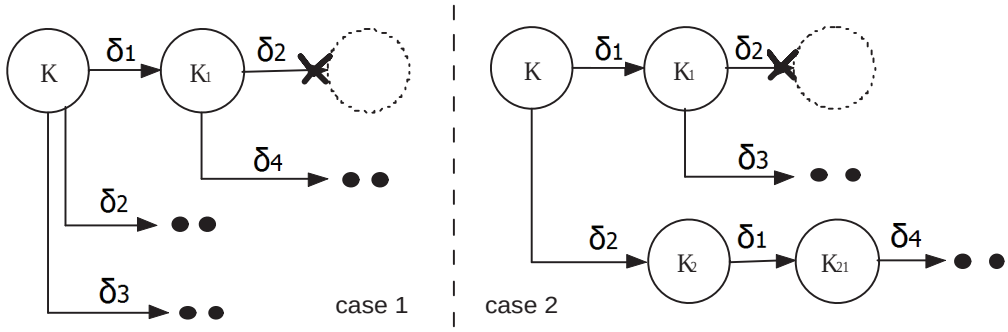

Figure 3.2: Examples of Tree Pruning

Proposition 6 *Consider a KB K and a violation selection function $NextV$. Suppose that there are two nodes, D, L, and a delta δ_{rej} for which:*

- *L is a leaf node, i.e., $desc(L) = \emptyset$.*
- *D is an ancestor of L, i.e., $D \in anc(L)$.*
- *δ_{rej} is a possible resolution for c^D, i.e., $\delta_{rej} \in Res(c^D, K)$.*
- *δ_{rej} was not selected in the branch that led to L, i.e., for the only node $D' \in anc(L) \cap desc_0(D)$ it holds that $\delta^{D'} \nsubseteq \delta_{rej}$.*
- *$K \bullet \delta^{L*} \models \delta_{rej}$*

Then, there is a second leaf node, say $L' \neq L$, for which $\delta^{L'*} \subseteq \delta^{L*}$.

Section A.3 in Appendix A explains the terminology used in the above proposition which can be used in several ways to optimize Algorithm 2. Firstly, assume that the four conditions of Proposition 6 hold and consider a delta δ^{L*} such that $K \bullet \delta^{L*} = L$ (equivalently $K \bullet \delta^{L'*} = L'$). Then, based on the proposition, there is some node L' and if $\delta^{L*} \neq \delta_{L*'}$, it follows that δ_L is not useful and must be rejected per Definition 7; if $\delta^{L*} = \delta_{L*'}$, then there is no need to keep both of $\delta^{L*}, \delta_{L*'}$. Thus, in either case, δ^{L*} need not be considered. The catch here is that, in the latter case (i.e., when $\delta^{L*} = \delta_{L*'}$), we should be careful enough to *not* ignore $\delta_{L*'}$ as well.

A more drastic optimization occurs if we incorporate the above checks inthe internal nodes. If, for some node, it holds that the current delta (δ_{tot}) contains two deltas (δ_1, δ_2) that would resolve a constraint that was previously resolved, then we have the option to prune entire branches of the resolution tree. Formally:

Proposition 7 *Consider a KB K and a violation selection function $NextV$. Suppose that there are two nodes, D, L, and a delta δ_{rej} for which:*

- *L is a leaf node, i.e., $desc(L) = \emptyset$.*
- *D is an ancestor of L, i.e., $D \in anc(L)$.*
- *δ_{rej} is a possible resolution for c^D, i.e., $\delta_{rej} \in Res(c^D, D)$.*
- *δ_{rej} was not selected in the branch that led to L, i.e., for the only node $D' \in anc(L) \cap desc_0(D)$ it holds that $\delta^{D'} \nsubseteq \delta_{rej}$.*

- $\delta^{L*} \supseteq \delta_{rej}$

Then, there is a second leaf node, say $L' \neq L$, for which $\delta^{L'*} \subseteq \delta^{L*}$.

As depicted in Figure 3.2, for the case 1, we follow the resolution option δ_1 in the first step of the resolution process; then, in the second step, we follow δ_2. Both δ_1, δ_2 are now included in δ_{tot}, and both are resolution options for a previously considered constraint. Thus, this branch must be rejected. In case 2, the same situation is shown, but then, in another branch, the same pair (δ_1, δ_2) appears in a different order. As shown in Figure 3.2, the second branch is processed normally, because it may correspond to the case where $\delta_L = \delta_{L'}$ that we explained in the previous paragraph.

A last optimization stems from the fact that, if we incorporate the above checks, we can reduce significantly the checks for useful deltas (per requirement #2 of Definition 7) when considering potentially preferred deltas (leafs). Namely, if the fifth bullet of Proposition 6 does not hold, then the delta is certainly useful, so we can avoid this check.

The above optimizations can be easily incorporated in the initial IB algorithm leading to the optimized IB_{prun} algorithm(Algorithm 13) which is presented in Appendix C. Again, for generality purposes, in our algorithms we consider that we deal with KBs and deltas which contain wildcards.

3.1.2 Optimization for IB and VB algorithms

As we described in IB, IB_{prun} and VB algorithms, in every state of the repair process, we have to check if the KB is valid with respect to the considered set of constraints (line 1) by implementing a full diagnosis. Since this repeated process over individual repairs incurs a significant overhead especially in the case of constraints with interdependencies. This cost can be avoided in both IB and VB strategies when constraints do not feature interdependencies. In this case we can consider a bulk constraint satisfaction check (i.e., diagnosis) over a set of nodes (vs. individual nodes) of the resolution tree. In other terms, when we have to perform diagnosis over a KB, we can restrict the search space only to constraint instances which may be violated by the changes a repairing delta has enforced. After diagnosing the constraints which are violated, their corresponding instances are stored into a global tuple, which is essentially an ordered set, called Errors. As a result, the *repair* function (Algorithm 1) is replaced with function $repair_{opt}$ (Algorithm 9) and this step is applied in line 1 of function $repair_{opt}$. The detailed description of the algorithms mentioned in this Subsection, can be found in Appendix C under Section C.2.

Let's see the above idea with the ontology example highlighted in Figure 1.2. Consider the IB resolution tree for this KB depicted in Figure 1.4. Based on Algorithm 2, in each node of the resolution tree, we have to find a constraint which is not satisfied (line 1) before continuing to the steps presented in the algorithm. Thus, for this resolution tree we have to apply 12 diagnosis checks (i.e., the number of nodes). In each diagnosis process, we have to examine the verification of all the constraints $c_1 - c_4$.

On the other hand, if we follow the idea of the aforementioned optimization we firstly diagnose the KB, which in turn returns the two violated constraints initializing tuple $Errors = \langle c_1(B, A), c_2(P) \rangle$. Next, we start repairing these constraints until we have no more constraint to repair. The latter depends on the returned value from function $fetchNextError$ (see Algorithm 10 in Appendix C). Next, we apply diagnosis (line 15) taking into account the followed branch. Repeat that we examine only the constraint instances which may be violated by the enforced changes in said branch. For example, following the branch $K \rightsquigarrow K_1 \rightsquigarrow K_{11}$, we notice that we removed $PS(P)$. We also notice that there is a constraint (i.e., c_3) which may be violated by this

change as it requires a property in its head. Additionally, the removal of a C_IsA tuple cannot create any violations as a side effect because it is not related with any other constraint. Based on the above, the diagnosis which has to be applied is restricted to verifying the satisfaction of constraint c_3 only for property P instead of verifying all the constraints $c_1 - c_4$ and for all the tuples in the KB. Obviously, re-diagnosing the KB can be much faster with this optimization. Continuing our example, the applied diagnosis will return the constraint instance $c_3(P, B)$, which in turn will update the tuple with the violated constraint instances (i.e., $Errors$). Last when $c_3(P, B)$ is repaired we notice that this change will not create any inconsistency, thus the branch terminates and we have a potential repairing delta. This process is repeated in each branch of the resolution tree and we notice that the number of diagnosis check is reduced to 6 and for the nodes K_{11}, K_{12}, K_{13}, K_{21}, K_{22}, K_{23}. Notice that for the nodes K_{111} K_{131}, K_{211} we don't have to apply diagnosis because the corresponding changes which led to those nodes (i.e., -$Range(P, B)$, $+CS(A)$, -$Range(P, B)$) cannot introduce any inconsistencies as side effects. This outcome derives from the syntactic form of the considered integrity constraints as these specific changes cannot violate them.

Concluding, this optimization in most cases will reduce the execution time significantly as it a) reduces the number of diagnosis applications and b)reduces the search space where diagnosis is applied. The first is succeeded as we do not apply diagnosis in every node of the resolution tree, but only in the biginning and on every leaf node. The second is succeeded by considering the corresponding branch (i.e., delta) when reaching a leaf node and the constraint instances which may be violated by its enforced changes. Obviously this optimization can be applied "as is" in the VB algorithm taking into account preferred repairing deltas. Moreover, it can be coupled with the proposed optimizations of Subsection 3.1.1 for the case of IB algorithm.

3.2 Complexity Analysis

The complexity of *Repair* algorithm is basically determined by the size of the corresponding resolution trees (say NOD) that need to be constructed. and the cost of each individual call to IB or VB. In our subsequent analysis we assume a fixed number of constraints in $\mathcal{C}$, the largest of which has size N_r; the number of universally (resp. existentially) quantified variables in a rule are at most N_x (resp. N_y); the original KB has size N_K and contains c constants. In this work, we also examined other constraint categories found in [1] which are special cases of DED constraints and are shown in table 3.1. Analyzing the constraint categories, in the case of Denial Constraints, ϕ is a (possibly) empty conjunction of comparison atoms (e.g., $u_i = u_j$) using variables from $\vec{u}$. Atom $B(\vec{u})$ denotes a non-empty conjunction of predicates except the case of LAV tgds where $B_0(\vec{u})$ is a single predicate. Also, predicate $H(\vec{u})$ (or $H(\vec{u}, \vec{v})$ denotes a non-empty conjunction of predicates as well. Readers are referred to [1, 14] for a detailed description of these rules and a more extended classification of DEDs expressivity. It was interesting to notice that the syntactic form of the rule, does not affect the cost to determine whether $K_c \vDash \mathcal{C}$ (step 1 in algorithms 2, 3) or not, which is equal to $O(N_K^{N_r})$. However, it affects the cost of the rest repair process as we will see next.

Table 3.2 illustrates the complexity bounds for the height (H) and width (W) of the resolution tree for various forms of DEDs in C (see Figure 6.1). The results on W are computed by finding the maximum size of the resolution set for each type of constraint; for constraints with existential quantifiers, the incorporation of wildcards significantly reduces the width, as expected. H is calculated by finding the maximum number of violated constraint instances in a branch, and the analysis includes violations that could be caused by the resolution of another violated instance. We noticed that the form of the rules' set (cyclic or not) combined with the

Table 3.1: Integrity Constraints' Types

Rule	**Form**
Denial Constraints	$\forall \vec{A} P(\vec{A}) \rightarrow \phi$
Full tgds	$\forall \vec{A} P(\vec{A}) \rightarrow Q(\vec{A})$
LAV tgds	$\forall \vec{A} P_0(\vec{A}) \rightarrow \exists \vec{B} Q(\vec{A}, \vec{B})$
Tgds	$\forall \vec{A} P(\vec{A}) \rightarrow \exists \vec{B} Q(\vec{A}, \vec{B})$
Full DEDs	$\forall \vec{A} P(\vec{A}) \rightarrow \vee_{i=1,\ldots,n} Q_i(\vec{A})$
DEDs	$\forall \vec{A} \bigwedge_{i=1,\ldots,n} P_i(\vec{A}) \rightarrow \vee_{j=1,\ldots,m} \exists \vec{B_j} Q_j(\vec{A}, \vec{B_j})$

set of constants (finite or not) determines the number of violations. For example, if we have a cyclic set of rules, a rule correction, may cause a violation of a rule which has been already corrected and if we have rules with existential quantifiers (LAV tgds, tgds DEDs), the termination of this process depends on the number of contants. On the other hand, having an acyclic set of rules, we are sure that a rule correction will not violate a rule which has been already corrected. Also, as we have a consistent finite set of rules, the height of the tree will be finite as well. Our analysis for the height relies on the existence of a loop detection procedure. For the IB algorithm, this is embedded in the check for useful deltas (Propositions 6, 7), because deltas produced by loops are always non-useful (see also algorithm 5); for the VB algorithm, loop detection need to identify that the same K_c is produced twice in the tree and thus reject the looping branch (recall that VB is recursive, so if the same K_c appears twice, the branch will not terminate).

In this context the size of the resolution tree NOD in the worst case will be $O(W^H)$ (see Table 3.3). Obviously, since the total number of tree nodes is finite, the algorithm will always terminates. It should be stressed that the same results hold, in the worst-case, for both IB and VB strategies; however, for certain combinations of constraints and preference, there is only one preferred delta in each $Res(c(\vec{A}), K)$ (i.e., $W = 1$), therefore the tree (in VB) is reduced to a chain, possibly with a polynomial number of nodes. More specifically, the above holds in the next cases:

- incorporating a set of Full tgds and the preference $Min(Deletions)$ (resp. $Highest(Additions)$)
- incorporating a set of LAV tgds and the preference $Min(Deletions)$ (resp. $Highest(Additions)$) or $Lowest(Additions)$ (resp. $Highest(Deletions)$)

Now let's compute the cost of comparing δ_i, $i = 1, \ldots, n$. If $>$ is defined by a composite preference, we need to compute the feature of each delta (costing, say, T_f), so the total cost is $O(n \cdot T_f)$. We assume that the atomic preferences induce a total order (e.g., Min, Max etc), as these are sufficient for capturing widely used repair policies. For such preferences, a simple scan over the feature values, costing $O(n)$, is enough, so the total cost of computing an atomic preference is $O(n \cdot T_f)$. Now let us combine k such atomic preferences (P_i, $i = 1, \ldots, k$) via the same constructor ($\otimes$ or $\&$). Table 3.3 contains the complexity results for $T_\&$ and the result found in [19] for $T_\otimes$. Note that, in all cases, the cost of computing k features for n deltas ($O(k \cdot n \cdot T_f)$) has been included. Since $T_\otimes$ is larger than $T_\&$, we will use $T_\otimes$ as the final comparison cost.

In both IB and VB algorithms, the maximum size of K_c is $O(N_K + H)$, and appears in the leafs when each node in the branch adds facts in K_c. In both algorithms, we check whether K_c is consistent, costing $O((N_K + H)^{N_r})$. The selection of the next constraint instance to consider can be done along with the consistency check, so it doesn't have any extra cost. For the VB

Table 3.2: Maximum Height and Width of the Resolution Tree

Constraint	**Height (H)**		**Width (W)**	
	Acyclic Constraints	**Cyclic Constraints**	**Without W/cards**	**With W/cards**
FD/CFD	O(N_K)	–	O(1)	O(1)
EGD/Denial	O(N_K)	–	O(N_r)	O(N_r)
Full TGD / Full DED	O($N_K{}^{N_r}$)	O(c^{N_x})	O(N_r)	O(N_r)
IND/CIND	O(N_K)	O(c^{N_x})	O(c^{N_y})	O(N_r)
LAV TGD	O(N_K)	O(c^{N_x})	O(c^{N_y})	O(N_r)
TGD/DED	O($N_K{}^{N_r}$)	O(c^{N_x})	O(c^{N_y})	O(N_r)

Table 3.3: Complexities

$T_{\otimes}(n,k)$	$O(k \cdot n \cdot T_f) + O(min\{n \cdot log^{(k-2)} n, k \cdot n^2\})$
$T_{\&}(n,k)$	$O(k \cdot n \cdot T_f)$
T_{IB}	$O((N_K + H)^{N_r}) + O(W \cdot (N_K + H))$
T_{VB}	$O((N_K + H)^{N_r}) + O(W \cdot (N_K + H)) + T_{\otimes}(W,k)$
NOD	$O(W^H)$
T_{Repair}	$max\{NOD \cdot T_{IB} + T_{\otimes}(W^H,k), NOD \cdot T_{VB}\}$

function, we need to compare W deltas, which costs $T_{\otimes}(W,k)$. The FOR loop in both functions will be executed W times at most, with a total cost of $O(W \cdot (N_K + H))$. The accumulated costs for the two functions are shown in Table 3.3 (T_{IB}, T_{VB}). The total cost (T_{Repair}) is computed by multiplying NOD with T_{IB} or T_{VB} costs. Note that for the IB case we have an additional cost $T_{\otimes}(W^H,k)$ to compare all the possible repairing deltas (which are $O(W^H)$ in total).

Last but not least the above worst-case complexity bounds are identical with or without the use of wildcards. This holds in resolution trees where, for a given node, the violating branches are equal to the number of constant values a given wildcard can be mapped to. For example consider the constraints $c : \forall x\ P(x) \rightarrow \exists y\ Q(x,y)$, $c' : \forall x\ Q(x) \rightarrow R(x)$. Consider also a KB $K = \{P(a)\}$ and a finite set of constants $\mathcal{C} = \{a,b\}$. The user can verify that the produced repairing repairing deltas without the use of wildcards are: $\delta_1 = \langle\emptyset, \{P(a)\}\langle$, $\delta_2 = \langle\{Q(a,a), R(a)\}, \emptyset\langle$, $\delta_3 = \langle\{Q(a,b), R(b)\}, \emptyset\langle$. On the other hand, using wildcards, the violated instance of constraint c is resolved by deltas $\delta'_1 = \langle\emptyset, \{P(a)\}\langle$, $\delta'_2 = \langle\{Q(a,\varepsilon)\}, \emptyset\langle$ where $[[\delta'_2]]^{\mu} = \{\langle\{Q(a,a)\}, \emptyset\langle, \langle\{Q(a,b)\}, \emptyset\langle\}$. However all the mapping of ε violate constraint c', thus we have to replace wildcard *any* with its corresponding values and then continue the repair process by adding the appropriate tuples to resolve contstraint c'. As a result the resolution turns out to be the same as if we did not use wildcards from the first place. Eventually, starting from $[[\delta'_2]]^{\mu}$, we will create the repairing deltas: $\delta'_{21} = \langle\{Q(a,a), R(a)\}, \emptyset\langle$, $\delta'_{31} = \langle\{Q(a,b), R(b)\}, \emptyset\langle$.

Wildcards will essentially reduce the average size of the resolution tree (NOD) when constraints with existential quantifiers exist, as expected by Table 3.2. Note also that the optimizations related to Propositions 6, 7 similarly do not affect the reported worst-case complexity, but reduce the average tree size, by pruning it in several cases. This holds because the main for loop contained in functions $nodeCheck$, $leafCheck$ is executed at most H times. In each iteration constant operations are executed, thus the main complexity of both functions is $O(H)$.

This complexity is already included in T_{IB} depicted in Table 3.3, thus it can be ommited.

The optimizations presented in Subsection 3.1.2 essentially reduce the execution time for both IB and VB algorithms as the number of Diagnosis applications are reduced. However, in a worst-case scenario, the above complexity bounds are identical with or without this optimization. For example consider a repairing sequence (using either IB or VB), where each repairing option introduces a single inconsistency. This means that after each application, we have to diagnose the KB again to capture the introduced inconsistency, leading to the first version of the algorithm.

Chapter 4

Experimental Evaluation

To experimentally evaluate the performance of our IB and VB repair finding algorithms, we relied on synthetically generated RDF/S KBs with certain features such as the number of classes and properties. As a next step, we created an errors generation algorithm to synthetically insert errros into the created KBs. Next, we studied the impact of critical parameters over our algorithms, such as the number and type of violated constraints the used preference expressions, on the execution time and memory requirements. Last but not least, we compared the obtained preferred repairs of the IB and VB algorithms and examined thir quality.

The chapter is organized as follows; in Section 4.1 we describe the experimental setting such as the implementation language, the KBs which were used for repairing (Subsection 4.1.1, the integrity constraints we used(Subsection 4.1.2) and the errors insertion algorithm we used to violate these constraints(Subsection 4.1.3). Last, in Sections 4.2, 4.3, 4.4 we present our experiment findings for the evaluation of out algorithms along with the qualitative and quantitative comparison of them.

4.1 Experimental Setting

We implemented both IB and VB algorithms as Java-based applications. We offer an API of method calls which can be used for the production of the preferred repairing deltas for a given (invalid) KB and a specific preference expression. All the experiments were contacted on a quad core CPU machine at 2.40GHz with 3GB of memory running Ubuntu 10.04. In order to load RDF/S KBs into main memory and manipulate them we used the Semantic Web Knowledge Middleware (SWKM) [1] that provides both an Object-based Model and a Triple-based Model for Main Memory RDF manipulation. In the triple based view a retrieval method for all triples of the model, as well as one for all triples with some subject, predicate and/or object are offered. In the object based view one can retrieve all the necessary information starting at any of the objects without having to retrieve and process any triples in between. This provides, for example, easy access to subsumption relationship information like subClassOf, superClassOf, subPropertyOf and superPropertyOf (both direct and indirect descendants/ancestors).

4.1.1 Synthetic Data Sets

For our experiments, we created synthetic synthetic RDF/S KBs featuring different structural characteristics using PowerGen [38] [2]. In particular, we created one KB privileging classes which

[1] http://139.91.183.8:3026/hudson/job/swkmmodel2/
[2] http://139.91.183.30:9090/RDF/PoweRGen

will be called Class Centric (i.e., CC) and one KB privileging properties which will be called Property Centric (i.e., PC). KB CC, features 80 classes and 40 properties, while KB PC features 40 classes and 120 properties. Moreover, we randomly inserted subsumption relationships between classes creating subsumption hieratchies with max depth equal to 5. The number of leaf nodes (i.e., classes with no descendants) is about 75% of the total number of classes in each KB. Last but not least, none of the KBs contains subsumption relationships between properties as PowerGen does not incorporate this function.

Additionaly, using PowerGen again, we enriched KBs CC and PCD with instances in order to examine instance-based constraints. In particular, we inserted 15 instances per class and we normally distributed 1-11 instances per property for both KBs PC, CC. As a result, we created the corresponding KBs with instances namely, PCD, CCD.

Table 4.1: Table of Predicates

Predicate	**Meaning**	**Triple**
$isCl(C)$	Name C exists in RDF KB and it is a Class	-
$isPr(C)$	Name C exists in RDF KB and it is a Property	-
$isMC(C)$	Name C exists in RDF KB and it is a MetaClass	-
$isMP(C)$	Name C exists in RDF KB and it is a MetaProperty	-
$isIn(c)$	Name c exists in RDF KB and it is an Individual	-
$Domain(P, C)$	The domain of P is C	$\langle P\ rdfs : domain\ C\rangle$
$Range(P, C)$	The range of P is C	$\langle P\ rdfs : range\ C\rangle$
$Cl\_IsA(C_1, C_2)$	Object C_2 is a direct subclass of object C_1	$\langle C_1\ rdfs : subclassof\ C_2\rangle$
$Cl\_IsA_{inf}(C_1, C_2)$	Object C_2 is a direct or indirect subclass of object C_1	-
$Type(x, C)$	Object x is of type (instance of) C	$\langle x\ rdf : type\ C\rangle$
$Type_{inf}(x, C)$	Object x has the inferred type of C	-
$P\_Inst(x, y, P)$	The pair (x,y) is an instantiation of property P	$\langle x\ P\ y\rangle$

4.1.2 Integrity Constraints

As we mentioned in Chapter 2 and Section 2.1.2, integrity constraints are encoded as FOL statements. Because of the fact that we used RDF/S KBs we firstly had to describe RDF statements in terms of FOL predicates in order to use them in the constraints. Table 4.1.1 presents all the predicates along with their meaning which appear in the integrity constraints.

At this point, we should mention that predicates which denote a specific object type (e.g., $isCl(A)$, $isPr(A)$) do not require the presence of specific triples into the KB to characterize their

corresponding object type. This holds because SWKM provides a typing mechanism which holds the type of each object internally thus, it can be used as a predicate value. In SWKM, objects are typed taking into account their correlatios with other objects (e.g., subclassof, domain relation). In Appendix D under Section D.1, we present the constraints which denote the type of the aforementioned objects. Any violation of these rules creates typing errors. However, in our work we do not consider the violations over typing rules, because the repairing of such rules would lead the change of object types which is not the case. However, we could enrich the set of integrity constraints with typing constraints and repair them as well considering that the objects may appear with different type into the repaired KB (e.g., a class was transformed into a metaclass). In constrast, in our experiments we consider violations over the integrity constraints which are defined in Table 4.1.2. Table 4.1.2 also shows how we create the resolution set per violated constraint. Notice that constraints c_3, c_4 introduce the use of wildcards and how they significantly reduce the size of their resolution set.

We should mention here that for efficiency purposes, in our implementation the created resolutions sets had some differences compared to the corresponding sets shawn in Table 4.1.2. In Appendix 4.1.2 under Section D.2 we show all the assumptions we made during our implementation and the respective resolution sets.

4.1.3 Synthetic Error Insertion

All the synthetic KBs we created using Powergen are consistent with respect to the constraints we previously defined (see Table 4.1.2). In order to evaluate our repair finding algorithms, we developed an *errors insertion algorithm (EIA)* to make the KBs inconsistent. One of the required parameters is the number of errors we would like to introduce. This can be an absolute number (e.g., 3 errors) or a percentage with respect to the number of triples of the KB (e.g., 2% errors). The other parameter is a set of DED constraints which will be violated.

Given a set of constraints to be violated, we randomly violate as many instances as the number of errors denotes. A violation of a constraint is done by randomly adding/removing some tuple that would invalidate the selected constraint. Let's illustrate this procedure with an example. Consider that we want to insert 3 errors and the set of constraints consists of c_1 to c_4(see Table 4.1.2). Then we have to randomly pick 3 constraints of this set. Consider that we pick c_1 (equiv. c_2). This constraint can be violated by adding one extra domain (resp. range) for the same property. The extra domain can be any of the classes, metaclasses, metaproperties contained in the KB. Now consider that we pick c_3 (equiv. c_4). This constraint can be violated by removing the domain (resp. range) of the property. Note that this rule could also be violated by adding a new property. However, based on assumption As. 5 (Appendix D, Section D.2) we do not want to add new URIs(i.e., properties) during repairs. If we allow the violation of a rule by adding a new URI, then one of the repaired KBs will contain this URI, thus it is forbidden due to our assumptions. Back in our example we similarly randomly select and violate as many rules as the number of violations we want to insert.

As we highlighted in Chapter 3 and Section 3.2, the repair process is affected by the width(W) of the resolution tree. Obviously different constraints, have different resolution sets, thus a different fan-out upon repairing. Focusing at this fact, that different constraint violations affect the whole repair process, we considered violations over schema constraints (c_1-c_5) when we applied EIA over PC and CC KBs and violations over instance constraints (c_6, c_7) when we applied EIA over PCD and CCD KBs. Note that the resolution of c_6,c_7 could potentially invalidate c_1-c_7, so our analysis below takes into account the *actual* errors produced and repaired, not the initial errors introduced by the EIA. The actual number of errors cannot be predicted

Table 4.2: Integrity Constraints

Constraint	**Intuition**	**Resolution Set**
$c_1 : \forall x, y, z\ Domain(x, y) \wedge Domain(x, z) \rightarrow \perp$	Each property should have exactly one domain	$Res(c_1(\langle x, y, z\rangle), K) = \{\langle \emptyset, \{Domain(x, y)\}\rangle, \langle \emptyset, \{Domain(x, z)\}\rangle\}$
$c_2 : \forall x, y, z\ Range(x, y) \wedge Range(x, z) \rightarrow \perp$	Each property should have exactly one range	$Res(c_2(\langle x, y, z\rangle), K) = \{\langle \emptyset, \{Range(x, y)\}\rangle, \langle \emptyset, \{Range(x, z)\}\rangle\}$
$c_3 : \forall x\ isPr(x) \rightarrow \exists y Domain(x, y)$	Each property should have an explicitly defined domain	$Res(c_3(\langle x\rangle), [[K]]) = \{\langle \emptyset, \{isPr(P)\}\rangle, [[\delta]]^{\mu}\}$
$c_4 : \forall x\ isPr(x) \rightarrow \exists y Range(x, y)$	Each property should have an explicitly defined range	$Res(c_4(\langle x\rangle), [[K]]) = \{\langle \emptyset, \{isPr(P)\}\rangle, [[\delta]]^{\mu}\}$
$c_5 : \forall x, y\ Cl\_IsA_{inf}(x, y) \wedge Cl\_IsA_{inf}(y, x) \rightarrow \perp$	The subsumption relationship must be acyclic	$Res(c_5(\langle x, y\rangle), K) = \{\langle \emptyset, \{Cl\_IsA_{inf}(x, y)\}\rangle, \langle \emptyset, \{Cl\_IsA_{inf}(y, x)\}\rangle\}$
$c_6 : \forall x, y_1, z_1, y\ P\_Inst(y_1, z_1, x) \wedge Domain(x, y) \rightarrow Type_{inf}(y_1, y)$	If a property instance is defined between two elements, then the subject of the corresponding triple should be correctly classified (directly, or through inference) to the domain of the property. Note that multiple classification is also possible.	$Res(c_6(\langle x, y_1, z_1, y\rangle), K) = \{\langle \emptyset, \{P\_Inst(y_1, z_1, x)\}\rangle, \langle \emptyset, \{Domain(x, y)\}\rangle, \langle \{Type_{inf}(y_1, y)\}, \emptyset\rangle\}$
$c_7 : \forall x, y_1, z_1, y\ P\_Inst(y_1, z_1, x) \wedge Range(x, y) \rightarrow Type_{inf}(y_1, y)$	If a property instance is defined between two elements, then the object of the corresponding triple should be correctly classified (directly, or through inference) to the range of the property. Note that multiple classification is also possible.	$Res(c_7(\langle x, y_1, z_1, y\rangle), K) = \{\langle \emptyset, \{P\_Inst(y_1, z_1, x)\}\rangle, \langle \emptyset, \{Domain(x, y)\}\rangle, \langle \{Type_{inf}(y_1, y)\}, \emptyset\rangle\}$

from the beginning and becomes known only at the end of the repair process.

In this chapter, we scaled the number of actual violations from 1 to 20, which represents a maximum of approximately 8% of the total amount of facts for the CC KB, 6% for PC and 4%, 3% for CCD, PCD respectively. These error percentages are motivated by the fact that only a small fraction of ontologies actually change from one version to another eg., [34] reports that 97.9% of the RDF/S data in each version remains unchanged. In addition, small error percentages are also considered when repairing databases (with of course a more important total size than in the case of ontologies). For instance, [6, 13] reports in their experiments from 1% to 5% errors (over databases of size 45000 tuples) although the error generation procedure is different. In addition, [6, 13] is interested in repairing only a fraction (e.g., up to 50%) of the introduced inconsistencies.

4.2 IB Algorithm Evaluation

Per the complexity analysis of Table 3.3, the performance of the IB algorithm is expected to scale linearly w.r.t. the size of the resolution tree. Thus, we first investigate experimentally how the type and number of the introduced violations in each of the four KBs of our testbed affect the size and form of the resolution tree under the IB strategy. Then, we study the time and memory requirements of the IB algorithm and experimentally verify the linear correlation between the tree size and the execution time. To stick on the essentials of our algorithm, the reported performance figures do not consider any repairing preference, because the computational time for determining the preferred deltas is negligible using efficient algorithms such as those proposed in [18].

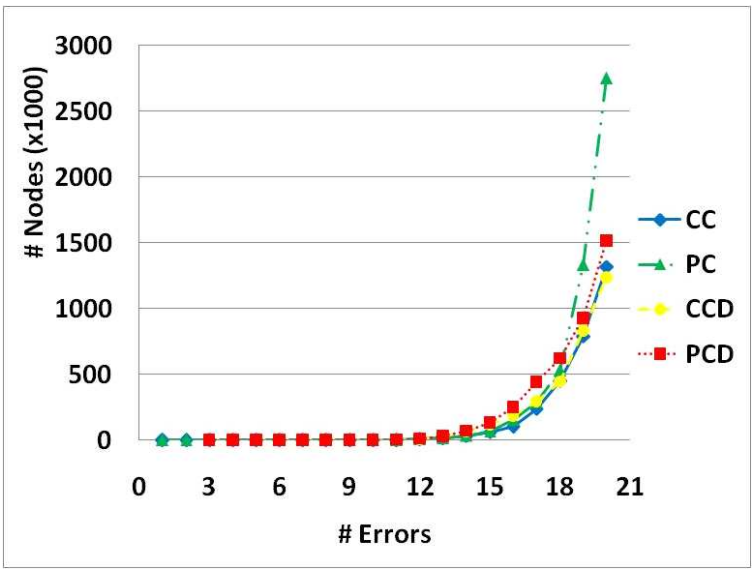

Figure 4.1: Size of the Resolution Tree for IB

As we can see in Figure 4.1 the size of the resolution tree grows *exponentially* as the number of introduced errors increase. This is due to the fact that the height of the tree increases with the number of constraint violations. On the other hand, the tree's maximum fan-out is determined by the type of the violated constraints, i.e., the size of their resolution set (note that the fan-out is increased by 1 when wildcards are included in the KB). This explains why the four KBs

of our testbed give resolution trees of different size for the same number of errors. For KBs CC and PC schema constraints $c_1 - c_5$ are violated and the resolution sets for all the violated constraints in these KBs has the same number of elements and in particular is equal to 2. One could expect the corresponding curves to coincide which is not the case. This holds because in PC KB, the EIA violates more constraints which refer on properties and have as a potential resolution the removal of a specific property (i.e., $c_3 - c_4$). Based on As. 4 in Section D.2 of Appendix D, the side effect of this choice, is the removal of the triples which use this property leading to the increment of the nodes in the resolution tree.

On the other hand, in KBs CCD and PCD, the fan-out ranges from 2 to 3 (due to the constraints c_6, c_7 and their resolution set). Given that PCD contains more properties and property instances, it causes more violations of c_6, c_7 (which have a fan-out of 3) thereby causing a larger resolution tree than CCD. Someone would expect KBs CCD and PCD to create larger resolution trees than the corresponding CC and PC which is not the case. This holds because as the number of inconsistencies increases, the optimizations based on Propositions 6, 7 are appliead more and more intensively in PCD, CCD KBs(unlike PC, CC) by pruning their resolution tree.

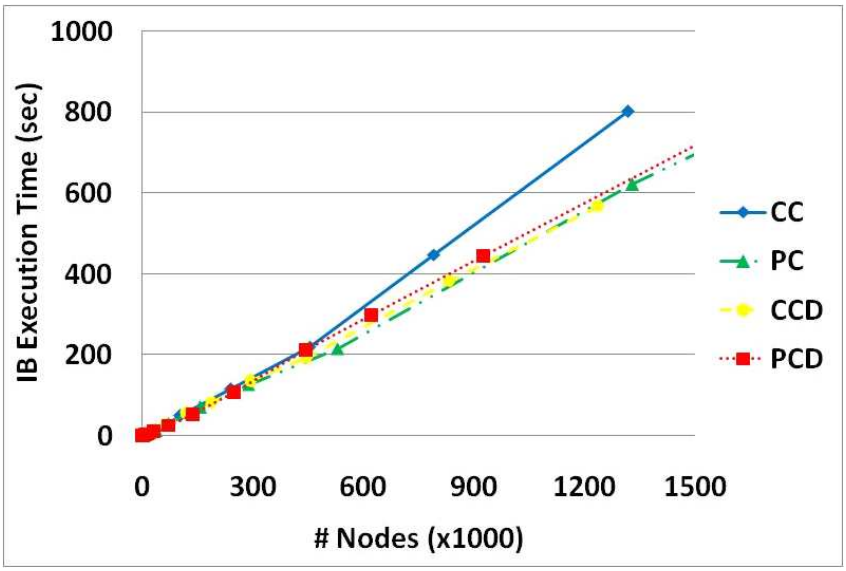

Figure 4.2: IB Execution Time

As we can see in Figure 4.2, the execution time scales linearly with the size of the corresponding resolution trees in all cases, confirming our complexity results of Table 3.3. We notice that the four lines have similar inclinations indicating a similar execution time per node of the tree in the four KBs. Notice however that the CC line has a slight higher execution time per node. This holds because in PC and CC KBs we have constraint violations which involve wildcards. Thus, the number of objects which can be assigned over wildcards(i.e., classes) affect the execution time.

The memory requirements of our IB algorithm are mainly affected by the number of potentially preferred deltas stored in main memory before performing the filtering (per the given policy). The introduction of wildcards has significantly reduced both the number of deltas to return/store, but also the size of the resolution tree. For example, consider a single violation of c_3 in CC KB, i.e., a property with no domain. Without wildcards, we would have 81 resolution options, namely one which removes the property and 80 which apply as a domain one of the classes contained in CC KB. Using wildcards, only two resolution options must be considered,

reducing the number of branches (and deltas) by about 98%. Furthermore, given that many facts are often replicated in different deltas, we rely on a bitsets-based implementation[3] to efficiently store and retrieve deltas. This representation could save up to an order of magnitude of storage space. Thanks to the above optimizations none of our IB experiments required more than 200MB of memory.

4.3 VB Algorithm Evaluation

For the evaluation of the VB algorithm we used the preference expression $Min(f_{additions})$. Our focus is again on the size of the resolution tree and the parameters it affects/is affected by, as well as on the comparison of the VB results with the ones presented for IB in the previous subsection.

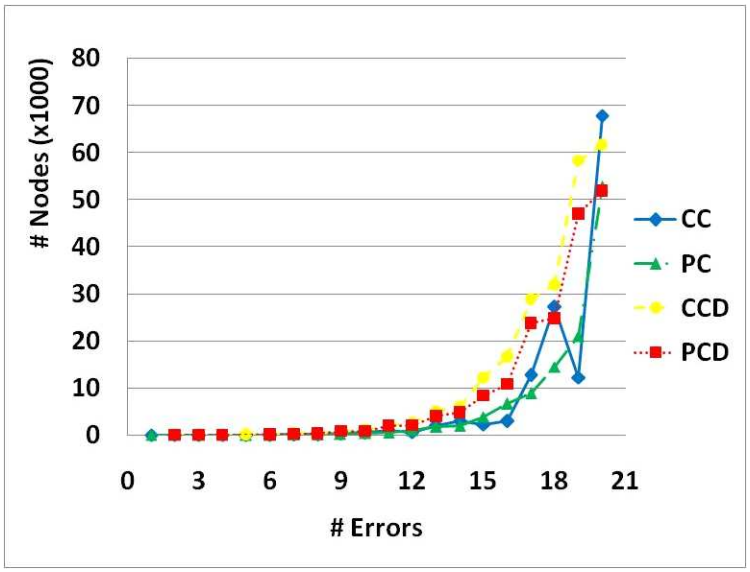

Figure 4.3: Size of the Resolution Tree for VB

As we see in Figure 4.3, the size of the VB resolution tree grows exponentially with the number violations despite VB algorithm's greedy nature. However, the actual number of nodes is significantly smaller than in the IB case, as many non-preferred branches are pruned. For the considered preference, the algorithm would ignore all additions of facts, essentially reducing the fan-out of several constraints (c_3, c_4, c_6, c_7) by 1. As a result, the constructed resolution trees of the IB algorithm is about 2 orders of magnitude larger than the corresponding ones created by the VB (depending on the case). Comparing Figures 4.1, 4.3, we note that the relative order of the curves for the different KBs is different than the IB case. This holds because the pruning w.r.t. the given preference depends on the actual constraints violated, which is different per KB. For example, the pruning over constraints violated in CC, PC KBs is more intense than in KBs CCD, PCD leading to smaller resolution tree. Note that in the case of CC KB and for 19 errors the number of nodes is smaller than when we inserted 18 errors. This can be explained by the random errors insertion process. When we inserted 19 errors, we inserted more errors over constraints c_3, c_4 (than in the case of 18 errors) where the used preference reduces the fan-out from 2 to 1 for these constraints leading to an overall smaller resolution tree.

[3] http://en.wikipedia.org/wiki/Bit_array

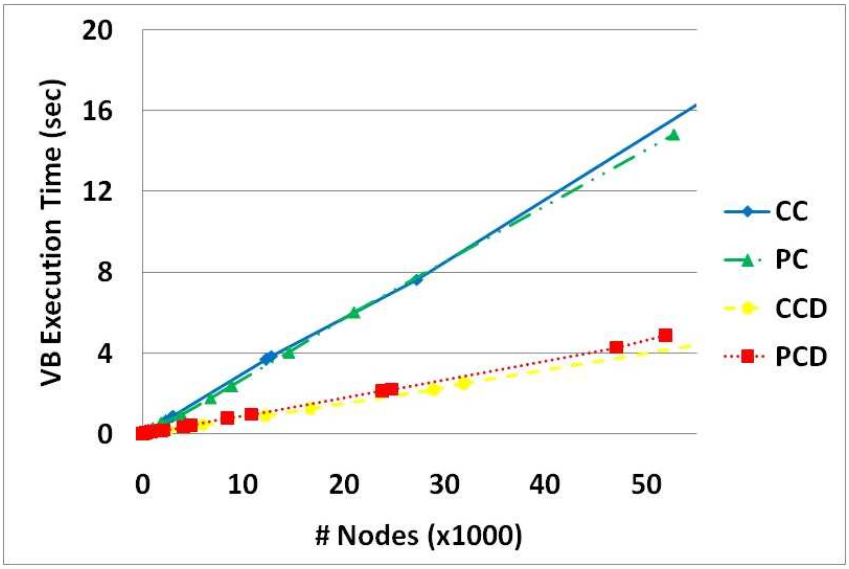

Figure 4.4: VB Execution Time

As with the IB case, Figure 4.4 shows that the execution time scales linearly with the tree size. The execution time for the VB algorithm, for the same KB and number of errors, is about 2 orders of magnitude smaller (depending on the case) than the time required for the IB algorithm. This is partly explained by the smaller size of the resolution tree, but is also due to the fact that the execution time per node is about 3-4 times smaller for the VB case. This is because the IB algorithm includes the optimization checks presented in Chapter 3 under Section 3.1.1 in order to prune a-priori the branches which lead to non useful repairing deltas (see also Propositions 6, 7). Note however that this extra cost in the IB algorithm is more than compensated by the reduction in the number of nodes caused by these optimizations. Note also that the curves in the VB case have different inclinations indicating a different execution time per node of the tree in the four KBs. This is caused by the different (average) cost for determining whether the KB is consistent (line 11 in Algorithm 12. In our testbed, we concluded that this cost is higher in CC, PC KBs than in the corresponding CCD, PCD. This was an interesting result as we concluded that in the case of IB algorithm, the type of constraint violations does not affect the average execution time per node something which does not hold for the VB algorithm.

The VB algorithm required a maximum of 20MB in our experiments. This is quite large, compared with the IB requirements, given the fact that the VB algorithm needs to store much less deltas than the IB. This is explained by our use of the bitsets optimization: the fewer deltas in the VB cause the bitsets to be sparse, thereby reducing the impact of this optimization. Despite that, bitsets are still reducing the VB memory requirements compared to the simple representation of deltas as sets of tuples.

4.4 Quality of VB Repairs

We consider that the preferred repairs returned by IB algorithm are optimal as the curator's preference is applied over all the possible repairing deltas. In this section, we judge the quality of the repairs returned by the VB algorithm. To do so, we measure the number of globally optimal repairs (returned by IB) which are not returned by the VB which will be called *false*

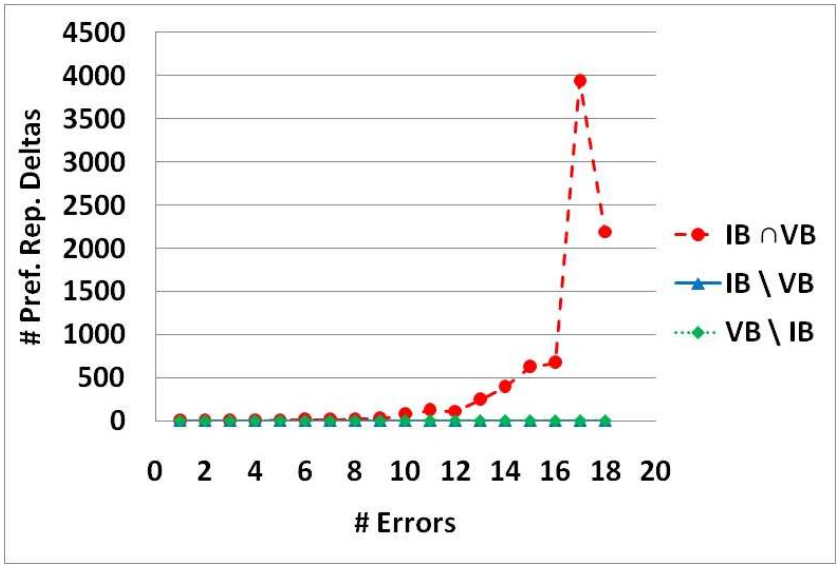

Figure 4.5: VB vs IB Repairs in CC for $Min(f_{additions})$

negatives, the number of preferred repairs returned by VB which are not globally optimal which will be called *false positives* and the number of common preferred repairs returned by VB and IB. Note that we count repairs with wildcards, i.e., a repair $[[K]]$ is counted as a single repair. As expected, the quality of results depends both on the types of the violated constraints and the employed preferences.

Common Repairs are denoted by $VB \cap IB$ in said figures. Using the preferences $Min(f_{additions})$, $Min(f_{deletions})$, $Min(f_{size})$ over the KBs CC or PC, algorithms IB and VB return exactly the same preferred repairs (see also Appendix B). The same preferred repairs are returned also using preference $Min(f_{deletions})$ of the KBs CCD or PCD (see also figure 4.5).

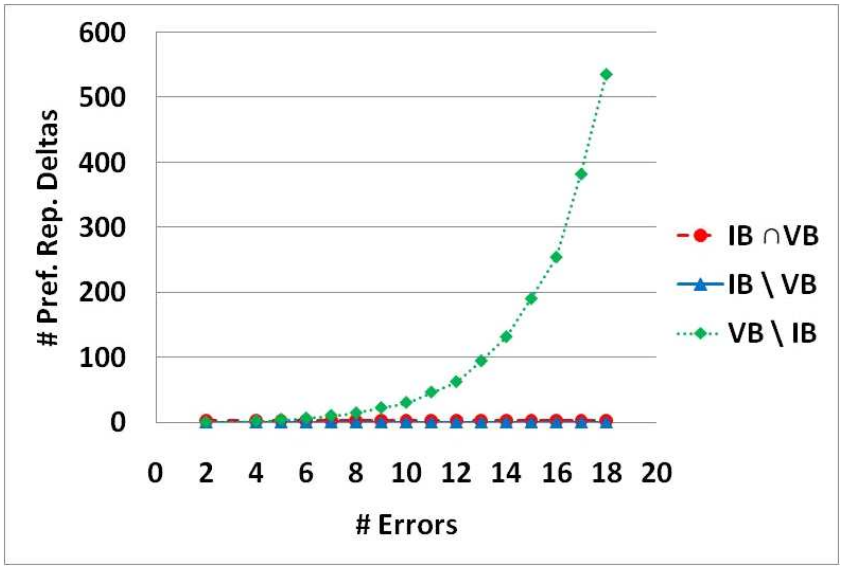

Figure 4.6: VB vs IB Repairs in CCD for $Min(f_{additions})$

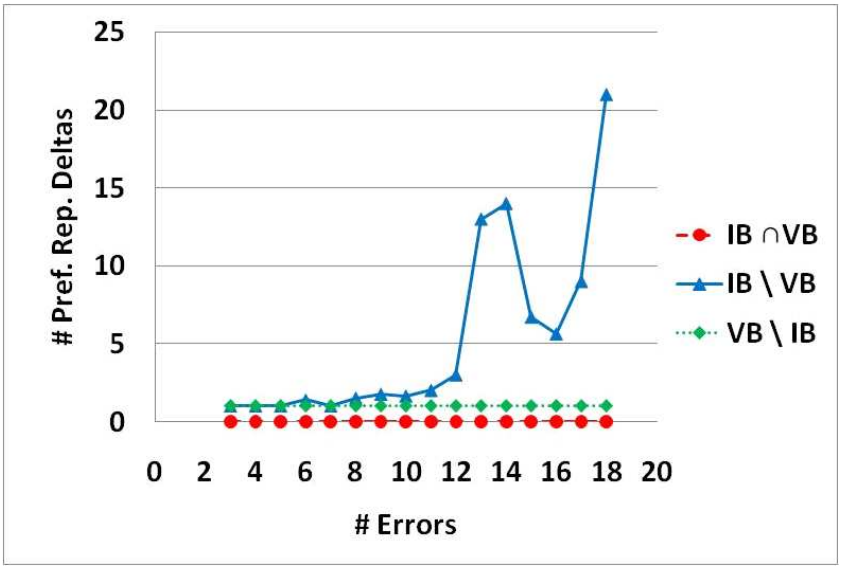

Figure 4.7: VB vs IB Repairs in CCD for $Max(f_{additions})$

However, when CCD and PCD are considered and using the preference $Min(f_{additions})$, VB contains several false positive results which are denoted by $VB \setminus IB$ in said figures. For the CCD in particular (see Figure 4.6), we observe that IB always returns two repairs regardless the number of errors we introduce. On the other hand, VB returns these repairs as well (i.e., no false negatives), plus a number of locally optimal repairs (false positives), which grows exponentially with the number of inserted errors. The reason is the following: all inserted violations are related to c_8, c_9, and can be optimally resolved by deleting a property instance in each case; however, VB will also consider the option to delete a domain or range in each case, which generates several non-optimal repairs as side effects (i.e., false positives). We will get similar results if we use the preference $Min(f_{size})$ over the KBs CCD or PCD. Again, IB algorithm will always return two repairs, while VB algorithm will return these repairs plus with a number of false positive repairs.

If we change the preference constructor into Max favoring additions (i.e., $Max(f_{additions})$) we found that repairing KBs CCD or PCD with this preference, a completely disjoint set of deltas is returned (i.e., no common preferred deltas are returned). As depicted in Figure 4.7, we see that VB always returns a single repair regardless of the number of introduced errors, which is not returned by IB (i.e., one false positive). On the other hand, IB (unlike VB) returns several more repairs which are not returned by VB (i.e., false negatives) whose number is not correlated by the number of violations. I.e., the number of preferred repairing deltas is not ascending w.r.t. the number of errors. The reason is similar as above: VB will only consider the addition of class instantiations to resolve all violations of c_8, c_9, whereas some of the deletions could also lead to globally optimal repairs.

Chapter 5

Declarative Prevention Policies

In previous chapters, we presented a declarative framework which can be used for the repairing of a KB. In that setting, the inconsistency is already there, and we have to repair it (e.g., ontology debugging [35, 20] database repairs [1, 7]). However, an alternative strategy to deal with inconsistencies is to prevent them from creeping into the KB in the first place, by carefully designing the operations that modify the KB. This repair strategy is followed in various settings such as belief revision [16, 25], belief merging [24], ontology evolution [25]). The latter approach can be easily seen as an extended case of the standard repair. To see this, consider the case of updating a KB K with an update U. Figure 5.1 shows that the update process can be split in two steps: (a) apply the update upon K in a straightforward way (using $\bullet$); (b) use a repair finding algorithm R on the result $(K \bullet U)$ to get a consistent KB (K').

Under this viewpoint, there are two main differences between repair finding and prevention algorithms. The first is that, in the former (repair), we have no clue on how the inconsistency was originally introduced and have to repair an already inconsistent KB, whereas in the latter (prevention), we are repairing the inconcistency the moment it is created, so we can use the information on how it was created to perform better repairs (e.g., in belief revision, the update often takes priority). The second difference is related to the input: in repair finding algorithms the input is a KB, whereas in prevention algorithms the input can be any piece of data that describes a change, depending on the context (e.g., a set of KBs to merge, a KB and an update etc). In addition, prevention algorithms depend on the initial operation (e.g., union, for the case of merging) that is performed upon the data. Another important observation here is that repair finding algorithms can be viewed as special cases of prevention algorithms, where the initial data is the KB itself and the "operation" performed is the identity function.

Extending the theory on repairs to include prevention algorithms is a simple exercise. First, we need to assume that the original input for prevention strategies is an element of some domain, say D, and there is some operation, say $\mathcal{O} : D \mapsto \mathcal{K}$, that returns a KB based on the input (e.g., for updates, the input is a KB and an update, which is a delta, so $D = \mathcal{K} \times \Delta$, and $\mathcal{O}(\langle K, U\rangle) = K \bullet U$). We overload the symbols PR^I, PR^V to describe the sets of preferred repairs w.r.t. a prevention policy $>^I$, $>^V$ respectively. Notice that theses symbols are now defined over a domain D and not over the KB K as in repairing policies. Based on the above, a *prevention algorithm* can be formalized as a function P, whose input belongs to a domain D (e.g., a KB K and an update U) and output is a *non empty* set of conistent KBs: $PD \mapsto 2^{\mathcal{K}} \setminus \emptyset$ s.t. for all $w \in D$ and all $K' \in P(w)$ it holds that $K' \vDash \mathcal{C}$. The definition of prevention policies is identical to repair policies, except that prevention policies are not coupled with a KB K as repairing policies but with a domain D which is affected by the operation $\mathcal{O}$. Similarly, the definitions of preferred repairs and preferred repairing deltas are the same. Note that, under this

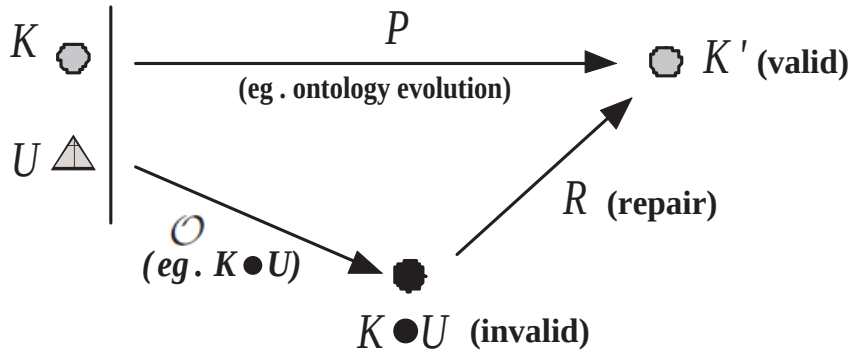

Figure 5.1: Prevention Strategies

formalization, repair finding algorithms (and repairing policies) are special cases of prevention algorithms (and prevention policies), for $D = \mathcal{K}$ and $\mathcal{O}(K) = K$, for $K \in \mathcal{K}$.

Let us illustrate the above with the next example, setting $K = \{C\_IsA(B, A), CS(A), CS(B)\}$ and considering the update $U = \langle\{PS(P), Range(P, B)\}, \{CS(A)\}\rangle$. Update U requests the addition of $PS(P), Range(P, B)$ and the removal of $CS(A)$. The updating operation is shown in Figure 5.2 and the result of this operation $(K \bullet U)$ is the KB of the motivating example shown in Figure 1.2.

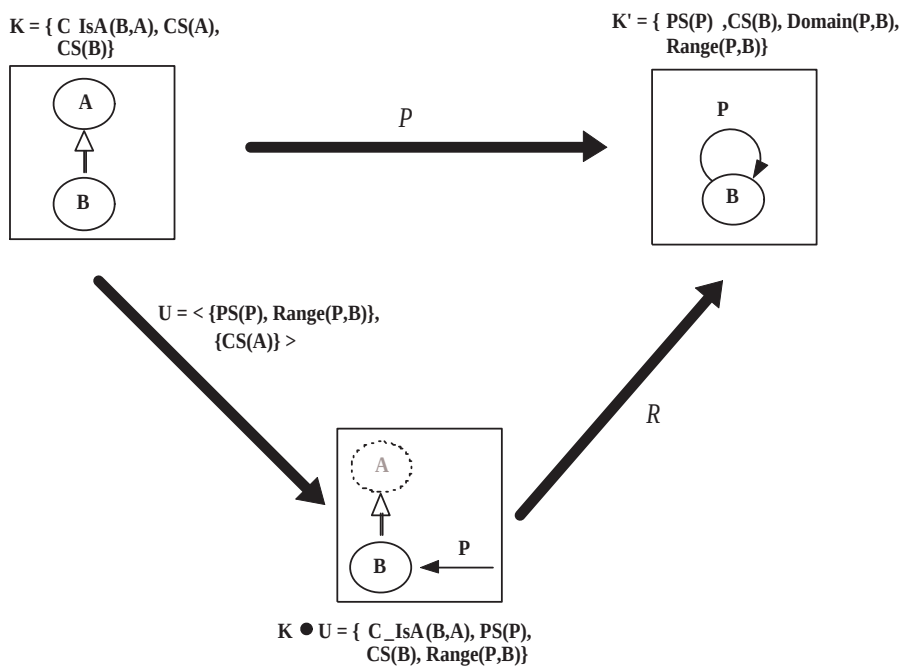

Figure 5.2: An Updating Operation

Then, $D = \mathcal{K} \times \Delta$ and $\mathcal{O}(\langle K, U\rangle) = K \bullet U$. Suppose that the curator's preferences are: "I want the update to be respected; secondarily, I prefer solutions with a minimal number of additions". We notice that the current definition of features is insufficient and cannot be used in the context of prevention algorithms as is. Thus, we restate the definition of features into a function over a set S such that $f_A : \Delta \times D \mapsto S$. We remind that prevention algorithms relies on

a domain D which is not fixed but it is affected by the operation $\mathcal{O}$ which led to the inconsistent result. Thus, in order to have features which refer to this domain D(as in the running example) we extended the domain of features with D. Back in the curator's preference, we will use two features which will be defined using this "extended" form. The first will count the number of changes in δ that do not respect the update: $Upd(\langle\delta_a, \delta_d, \rangle, \langle U_a, U_d\rangle) = |\delta_a \cap U_d| + |\delta_d \cap U_a|$. The second counts the number of additions: $Additions(\langle\delta_a, \delta_d\rangle, \langle U_a, U_d\rangle) = |\delta_a|$. The final, composite preference is $Min(Upd)\&Min(Additions)$, which defines an ordering, say $>$. If we use the IB approach, table 2.2, shows the possible repairing deltas. We then calculate the preferred repairs of $K_0 = \mathcal{O}(\langle K, U\rangle) = K \bullet U$, as in Definitions 7, and end up with only one preferred repairing delta ($\delta = \langle\{Domain(P, B)\}, \{C\_IsA(B, A)\}\rangle$), which respects the update and adds only one fact. This delta is applied upon K_0 (not upon K), giving the preferred KB $K' = K_0 \bullet \delta = \{CS(B), PS(P), Range(P, B), Domain(P, B)\}$ (see also Figure 5.2) which is the only KB in $P(\langle K, U\rangle)$.

Next we restate Proposition 2 for information based prevention policies with the obvious changes. Proposition 8 below is the "prevention counterpart" of Proposition 2. Unfortunately though, we cannot do the same with Proposition 3, because the recursion (inside the big union in the theorem) would require prevention strategies to be defined over $\mathcal{K}$, which is not the case. Furthermore, VB prevention policies are not totally memory-less, as they always depend on the original $w \in D$.

Proposition 8 *A prevention algorithm P is IB-expressible iff for all $w \in D$, $K = \mathcal{O}(w) \in \mathcal{K}$, $K_r \in P(w)$, $\delta_r = \langle\delta_{ar}, \delta_{ar}\rangle$ s.t. $K_r = K \bullet \delta_r$, $K'_r \in \mathcal{K}$, $\delta'_r = \langle\delta'_{ar}, \delta'_{ar}, R'_r\rangle$ s.t. $K'_r = K \bullet \delta'_r$ for which $K'_r \vDash \mathcal{C}$, $\delta'_{ar} \subseteq \delta_{ar}$, $\delta'_{dr} \subseteq \delta_{dr}$ and $R'_r \subseteq R_r$, it holds that $K'_r = K_r$.*

Prevention algorithms can be described using a repair finding algorithm as an intermediate step (see Figure 5.1). Given that the result of a prevention algorithm depends on its input ($w \in D$), rather than the input to the repair strategy ($\mathcal{O}(w) \in \mathcal{K}$), we could say that, in fact, a prevention algorithm is described by a family of repair finding algorithms (one for each $w \in D$), which also characterizes it. Formally:

Definition 10 *Consider a prevention strategy P over some domain D. Consider also an operation, say $\mathcal{O} : D \mapsto \mathcal{K}$, that returns a KB.The family of repair finding algorithms $\{R_w | w \in D\}$* implements *P iff for all $w \in D$ it holds that $P(w) = R_w(\mathcal{O}(w))$.*

Proposition 9 *Consider an operation $\mathcal{O} : D \mapsto \mathcal{K}$ and a prevention algorithm $P : D \mapsto 2^{\mathcal{K}} \setminus \emptyset$. Then P is an IB prevention algorithm iff there exists a family $\{R_w | w \in D\}$ of IB repair finding algorithms that implements P.*

We should mention here that under this formalization, VB prevention algorithms cannot be modeled using an adequate family of VB repair finding algorithms. This holds based on Proposition 3, VB repair finding algorithms are memoryless and in each state, they depend on the current KB instance. However, as VB prevention policies depend on the original $w \in D$, we should "carry" w in each step. The last is not supported by the current formalization of VB repair finding algorithms.

As with repair finding algorithms, prevention algorithms from the literature can be mapped to our framework. For example, the ontology evolution approach of [25] corresponds to a family of IB prevention strategies, if we restrict ourselves to consistent updates. The formal proof can be found in Appendix A and under Section A.5. Moreover, given that Rondo encodes the necessary information for the repair inside the merged (inconsistent) KB ($K_0 = \mathcal{O}(w)$), we

proved that we can can model it using a VB repair policy. We can also model its repair strategy using a VB prevention algorithm by setting $\mathcal{O}$ to be the merging operator and w the sources K_1, K_2 which will be merged.

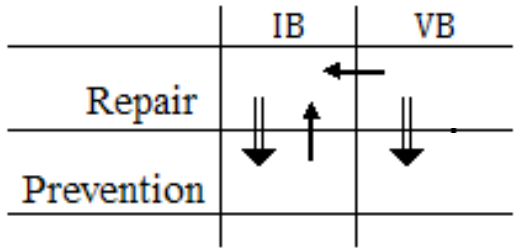

Figure 5.3: Correlating Strategies depending on the used Repair Approach

Discussion. Figure 5.3 shows how repair finding algorithms are related with prevention algorithms and similarly IB with VB strategies. As we mentioned, setting the performed "operation" in prevention algorithms to be the identity function, we can view repair finding algorithms as prevention algorithms (arrow $\Downarrow$). Moreover, taking into account Proposition 9, we conclude that any prevention algorithm is an IB if and only if there exists a family of IB that implements it (arrow $\uparrow$). Finally, we conclude from Proposition 3 that any VB strategy can be described as a family of IB strategies when we have repair strategies.

Chapter 6

Related Work

6.1 Introduction

As already mentioned, curated KBs are developed with a great deal of human effort by communities of scientists [8] and co- evolve along with the experimental evidence produced. In such a dynamic setting, the introduction of various forms of inconsistencies is a common situation with adverse effects on the operation of knowledge-intensive e-science applications. There are various types of integrity constraints in the literature (see also Figure 6.1) used in order to express consistency over the data. However as we noticed, their form varies with respect to the representation language of the KB. However, all the constraint forms we examined, can be considered as disjunctive embedded dependencies [14] and are separated in the depicted subcategories of Figure 6.1.

For instance, in ontological environments the curator may want to impose acyclicity of subsumption relations between classes or properties [36] which can be encoded using functional dendencies(FDs) (e.g., $P(\vec{x}) \wedge P(\vec{y}) \rightarrow x_i = y_j$). in the same context, cardinality constraints [32] of instances could be expressed using inclusion dependencies (INDs) (e.g., $P(\vec{x}) \rightarrow \exists \vec{y} Q(\vec{x}, \vec{y})$). Functional dependencies can be also used to express primary key constraints in relations whereas inclusion dependencies can be used to express foreign key constraints [10, 12]. In [15], they managed to extend the expressiveness of the previous constraints introducing the respective conditional functional(CFDs) and inclusion dependencies (CINDs) wich refer to specific fragments of each corresponding relation. Using more expressive constraints(which are still special cases of DEDs [14]) the quality of data increases [15], although the tasks of diagnosis and repair become more challenging.

The problem of dealing with inconsistencies has motivated many works in various contexts. However, they encounter them following different approaches and can be listed into:

1. Invalidity Tolerant strategies which do not repair the data, but allow working over invalid data and try to retrieve meaningful information over it(e.g., consistent query answering [5]).

2. Prevention Strategies which try to prevent the introduction of invalidities (e.g., belief revision [2])

3. Repair strategies which repair the inserted inconsistencies with a straightforward way(e.g., e.g., ontology debugging [35], database repairs [1])

We should mention here that in strategies of the last two categories, the resolution of inconsistencies should cause *minimal changes* upon the curated KB (per the Principle of Minimal

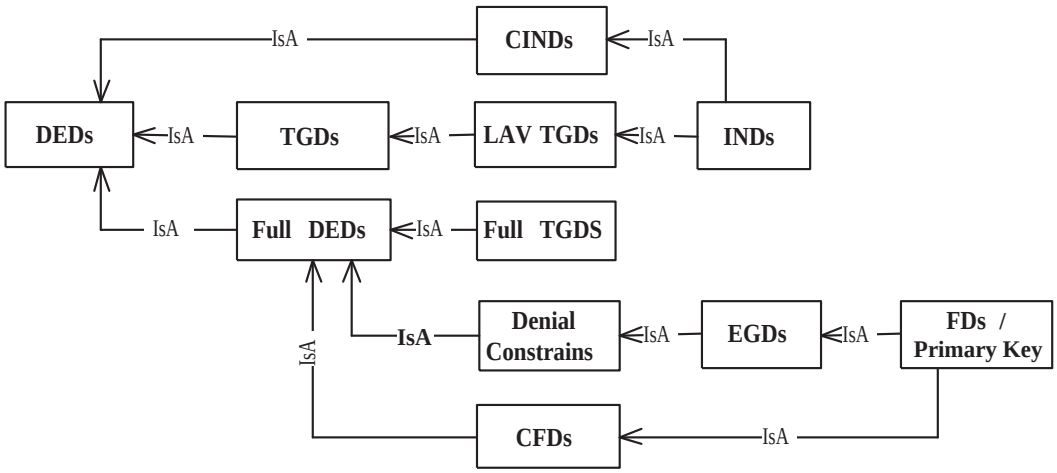

Figure 6.1: Types of DED Constraints

Change [2]). However, the notion of *minimality* in curated KBs is context dependent, as it may depend on the underlying assumptions over the data. In the following sections we will present such strategies along with the notions which determine minimality in their context.

6.2 Invality Tolerant Strategies

This approach is followed in case the materialization of repair is not an option i.e. when the repair is costly, non-deterministic, may lead to loss of potentially useful data or add potentially unuseful data. Despite the fact that we have an inconsistent KB, we must be able to derive meaningful information (via queries) about its contents; otherwise it is useless to keep such an inconsistent KB. In this case, instead of repairing the KB (which in most cases produces more than one possible repairs), we try to derive meaningful information via queries. In fact, the considered query answers are those which are true in all repairs of the initial inconsistent KB, i.e., consistent query answers [5, 10, 20, 3, 11, 4]. This can be achieved by rewriting the initial queries into new which ideally fetch the same(or as close as possible) result we could get if we applied the initial query to every possible repaired version of the KB. It is important to note that this must be done hopefully avoiding as much as possible the explicit computation of the repairs and checking candidate answers in them.

This strategy could be useful in virtual data integration [27] where there is no centralized consistency maintenance mechanism that makes the data in the mediated system satisfy certain global integrity constraints(ICs). Another case where consistency cannot be preserved at all, thus such strategies would be useful, is when schema-level and instance-level of ontologies are evolved separately without synchronizing the changes continuously. In such cases, inconsistencies cannot be repaired, but still one may wants to derive meaningful answers when reasoning [22]. Last but not least, consider a knowledge base which is updated and changes frequently and the queries applied upon the KB are known in advance. In that case, it is very efficient to transform the queries once at the beginning and use the transformed queries afterwards to obtain consistent answers on the changing KB [39].

However, we argue that there are cases and scenarios where the materialization of the repair will be more efficient and the provided query answers will be of higher quality. The efficiency can be shown in a scenario where we have KBs which do not change constantly and many ad hoc

queries are applied over the KB [39]. In such a case, the KB may have to be repaired only in cases where it changes its state (i.e., seldom) and then consistent answers are produced to any query without the overhead of the latter's transformation. We are convinced that in most cases the approach followed in consistent query answering returns low quality results. This holds because this approach considers results which are implied by all possible valid repairs associated with the (inconsistent) KB, so we cannot set any preferred repairs, or prevent certain repairs from being considered. We should mention here that the considered repairs in all the aforementioned works are minimal with respect to the subset relation (called useful repairs in our framework). Let's illustrate this with an example. Consider a scenario where the curator would prefer to add, rather than delete in order to repair inconsistencies(i.e., data integration). Consider also a KB $K = \{P_1(a), P_2(a)\}$ and a single integrity constraint $c : \forall x P_1(x) \wedge P_2(x) \rightarrow Q(x)$ which is violated by K. The possible repairs of K are $K_1 = \{P_1(a)\}$, $K_2 = \{P_2(a)\}$, $K_3 = \{P_1(a), P_2(a), Q(a)\}$. As we mentioned, under consistent query answer, we fetch the common tuples which are contained in every possible repair which is minimal under set inclusion. However in this example there are no common tuples, thus any query over K would return *empty* results. On the other hand, taking into account the curator's preference and choosing to repair K, we would obtain repair K_3 as preferred. At this point, it is clear that queries over K_3 will return results of much higher quality.

In this work, we do not focus in this category which contains strategies that do not perform repairs. In our setting of curated databases, curators have complete control of the curated KB, thus they can repair the created ivalidities.

6.3 Repair Strategies

6.3.1 Repair Checking

In the database setting, there has been a substantial amount of work related to the computational complexity of the problem of determining whether a given KB is a preferred repair for an inconsistent KB (*repair checking*), under various notions of minimality and kinds of DED constraints [14] (see also Figure 6.1). Note that determining whether a repair is preferred is different than actually computing a preferred repair, which is the problem we deal with in this thesis. Four variations of minimality have been proposed in [1], all of which can be captured in our framework, and fit under the IB strategy, because they consider the final repairing delta and not the individual resolutions performed at each intermediate step.

The first type is *subset repairs*, which require that the repaired KB should be a maximal sub-instances of the initial KB. This means that it should contain the least possible (per the $\subseteq$ relation) updates (i.e., deletions). This minimality notion can be captured using the preference $Min(f_{additions})$ over the feature $f_{additions}$ defined in Appendix B.

The second is *symmetric difference repairs*, which require that the repaired KB should contain the least possible (per the $\subseteq$ relation) updates (additions and deletions). Note that symmetric difference repairs are actually the useful repairs in our terminology, so this minimality notion is captured by default if no preference is provided.

Cardinality Repairs require that the repairing KB should contain the least possible (in terms of cardinality) updates (additions and deletions). This minimality notion can be captured using the preference $Min(f_{size})$ over the feature f_{size} defined in Section 2.2.

Finally, *component cardinality repairs*, require that the repairing KB should contain the least possible (in terms of cardinality) updates (additions and deletions) per relation. This minimality notion can be captured by using the feature $f^i_{pr\_appearance}$ defined in Appendix B, each one

counting the number of appearances of a certain relational atom R_i in the delta (i.e., $\mathcal{P}_i = \{R_i\}$). Each of those features defines a preference of the form $P_i = Min(f^i_{pr\_appearance})$, which should be combined using pareto to get the final composite preference (i.e., $Min(f^1_{pr\_appearance}) \otimes \ldots \otimes Min(f^n_{pr\_appearance})$).

In Appendix A and under Section A.4, we provide the formal proofs of how the above variations can be emulated within our framework.

6.3.2 Repair Finding

Automatic Techniques

Works of this category repair inconsistencies automaticaly without any user intervention. They are based on intuition and heuristics leading them to be often application-specific and their techniques cannot be generalized in other contexts. Next we present several repair finding techniques found in the literature and how they can be emulated within our framework. Formal proofs of these emulations can be found in Appendix A and under Section A.4.

In [6], they deal with the actual problem of automatically finding the repairs. The framework of [6] only considers Functional and Inclusion Dependencies (FD/IND); in [13], this work has been extended to support Conditional Functional Dependencies (CFD) as well (see Figure 6.1). Each violated constraint is resolved in a predetermined manner, depending on its kind. FD and CFD constraints (e.g., $\forall x, y, z P(z, x) \wedge P(z, y) \wedge \phi \rightarrow x = y$) are resolved through tuple modifications (in effect replacing x with y in the corresponding tuples or vice-versa). Note that ϕ is a (maybe empty) conjunction of formulas of the form $a = b$, $a \neq b$, $a\ b$ etc. and denotes the partition of table's rows which the CFD should be satisfied. IND constraints (e.g., $\forall x P(x) \rightarrow \exists y Q(x, y)$) are resolved by adding a tuple of the form $Q(x, null)$. This repairing option is generalized by replacing all the existential attributes with the special value *null*. This option reduces the maybe infinite number of possible repairing options significantly and avoids considering all possible assignments for the existential values (value y in our example). As resolution is made independently for each constraint, this framework implements a VB strategy. In addition, the involved repairing policy is embedded in the algorithm and curators cannot intervene. Since tuple modifications are not supported in our framework (for the reasons explained in Section 2.3), we cannot capture the repair policy used in [6, 13] for FDs and CFDs. We can only capture the repair policy for INDs, by stating a preference that would prefer the additions that use *null* for the existential quantifiers, over the rest of the additions and deletetions as well. To do so we use the feature $f_{const\_additions}(\delta)$ defined in Appendix B which count the occurences of some specific values of a set $\mathcal{CV}$ (here $\mathcal{CV} = \{null\}$ into the set δ_a of a possible repairing delta $\delta = \langle \delta_a, \delta_d \rangle$ for a given violation. The corresponding prefence is $Max(f_{const\_additions})$, where $\mathcal{CV} = \{null\}$.

The use of *null* in order to avoid producing a large number of repairs has also been employed in the framework of [7] which is capable of resolving violations for all kinds of DED constraints. The resolution is made by taking all possible resolution options, with one exception: when existential quantifiers are involved (e.g., in INDs), only the *null* value is considered for the existentially quantified variables, instead of all possible values. Since violations are resolved independently, this framework also implements a VB strategy. We can express the employed repair policy in our framework using the above preference with two exceptions: (a) we now don't want to filter out deletions, and (b) the rules considered are not just INDs, so a slightly more complicated preference expression is necessary to consider other DED types as well. To capture this repair policy we require the following preferences:

- $Max(f_{deletions})$ prefers deltas which apply deletions.
- $Min(f_{deletions})$ prefers deltas which apply additions.
- $Max(f_{const\_additions})$, where $\mathcal{CV} = \{null\}$ to prefer deltas which add ground facts with the maximum number of *null* values.
- A composite preference of the form: $P = Min(f^1_{pr\_additions}) \otimes \ldots \otimes Min(f^n_{pr\_additions})$, where each feature $f^i_{pr\_additions}$(see Appendix B), counts the number of appearances of a certain relational atom R_i in the δ_a set of a delta. (i.e., $\mathcal{P}_\rangle = \{R_i\}$).

The final composite preference is $Max(f_{deletions}) \otimes (Min(f_{deletions})\ \&\ (P \otimes Max(f_{const\_additions})))$.

Another repair finding algorithm appears in Rondo [29], which is actually a generic framework for relational and XML model management. Violation resolution takes place during schema merging since Rondo systematically checks whether the resulting merged schema satisfies certain FD constraints. Before merging the input schemata, Rondo requires a user-defined mapping to be provided, which determines their identical elements. Using this mapping, each tuple of the merged schema will be assigned a label, which determines its importance. There are 9 types of labels, namely (in decreasing level of importance): 00,0+,0−,+0,−0,++,+−, −+, −−. Rondo's repair policy considers the tuples' labels: in particular, when an FD constraint of the form $P_1(\vec{A}) \wedge P_2(\vec{A}) \rightarrow x = y$ is violated, we remove the least important of $P_1(\vec{A})$, $P_2(\vec{A})$ as determined by their labels (we remove either if they have the same label). To capture Rondo's repair policy in our framework, we define 9 preferences of the form $Max(f^l_{const\_deletions})$(or simply P_l), where $f^l_{const\_deletions}$ is a feature which counts the number of appearance of a specific label l, from the above in the tuple in the set δ_d pf a delta. We combine these preferences using & to form the final preference asfollows: $P_{--}\&P_{-+}\ \&P_{+-}\&P_{++}\ \&P_{-0}\&P_{+0}\ \&P_{0-}\&P_{0+}\ \&P_{00}$.

An interesting example of a repair strategy which cannot be modeled as a preference based repair policy $>^P$ within our framework appears in [30]. In that approach, they consider the integration of KBs described in description logics. Each integrated KB is modeled as a stratum which is integrated itself within the existing stratified knowledge. The arising inconsistencies from the KBs integration are resolved by removing knowledge from the outermost stratum until we reach a valid KB. However if the consistency is not reached, we move on to the next stratum and so on. Therefore, if the invalidity is caused by an inner stratum, all strata that have less priority will be removed completely, and unnecessarily. This causes the conditions of Propositions 2, 3 to fail.

User Interactive Techniques

Works of this category consider that a fully automatic repair process without human intervention is unrealistic (i.e., curated KBs) [35]. They are based on the underlying idea that only experts or KB engineers (i.e, curators) have all the required knowledge to perform the most optimal repair.

Several works have been also proposed for repairing ontologies (such as PROMPT [33] and Chimaera [28]). These works only address violations of constraints which can be expressed in the underlying knowledge representation formalism (such as OWL [21]) and essentially correspond to a subset of the DEDs considered in our framework (e.g., functional and inclusion dependencies). Furthermore, they provide automatic support only for detecting constraint violations while curators have to manually find the correct resolution and perform the repair. The main difference between PROMPT and Chimaera is that in case an inconsistency is found, PROMPT not only present it to the user, but also *suggests* possible repairing solutions if possible, based on the

syntactic form of the inconsistency found. This is exactly the ontology curation task that our framework aims to facilitate.

6.4 Prevention Strategies

Prevention strategies use mechanisms which act as monitors over KBs. They are activated when changes(eg., knowledge additions or removals) over the contained knowledge could potentialy create invalidities and try to prevent their introduction. As a result, KB remains valid after such modifications.

In the field of belief revision [2] and ontology evolution [25] we suppose that "new" knowledge (i.e., update) is more important and reliable compared to the existing knowledge in the knowledge base. Such assumptions could prevent the creation of invalidities by applying the changes which do not contradict to those imposed by updating changes, or if the previous is not possible, the minimal changes which are in concflict with those imposed by the update. (eg., we can not prevent an invalidity created by the "new" addition of a ground fact by removing it). As we proved in Proposition 17, this notion of minimality can be captured within our framework using a prevention algorithm and a violation based strategy

In the field of belief merging [24], one of the sources is often considered to be more reliable. As a result, invalidities are prevented by respecting the preferred source, and changing the non-preferred one. Also, we saw that Rondo [29] supports the merging operator as well. Given that Rondo encodes the necessary information for the repair inside the merged (invalid) KB ($K_0 = \mathcal{O}(w)$), in the previous section, we proved that we can can model it using a VB repair policy. We can also model its repair strategy using a VB prevention by setting $\mathcal{O}$ to be the merging operator and w the sources K_1, K_2 which will be merged.

Some works in these fields automatically determine the result (based on certain assumptions) [25], whereas others do not commit to any particular approach, but introduce *postulates* [2, 24] that determine which approaches are "rational", thereby allowing the exact approach to be set by the application at hand. Our approach is closer to the second (general) category, but puts more emphasis on allowing the curator to specify the exact policy in an intuitive and declarative manner (i.e., using formal preferences).

Chapter 7

Conclusions and Future Work

Scientific communities (e.g., in Bioinformatics) more and more rely on *Curated Knowledge Bases (KBs)* like ontologies or annotations in their everyday scholarly and scientific work. These KBs are developed with a great deal of human effort by scientists acting as curators, and co-evolvealong with the experimental evidence produced. In such a dynamicsetting, the introduction of various forms of invalidities is acommon situation with adverse effects on the operation ofknowledge-intensive e-science applications.

In this work we proposed a declarative and intuitive framework for assisting curators in one of the most frequently performed tasks, i.e., the *repairing* of various forms of inconsistencies arising in curated KBs. Our framework is generic enough as it captures a wide class of integrity constraints using disjunctive embedded dependencies (DEDS). Moreover, our framework is highly customizable as is able to capture various curation settings and scenarios using qualitative preference expressions upon interesting features of the repairs. Such preferences are used to filter out non-optimal repairs, either during the resolution of each individual inconsistency or at the end, after having resolved all invalidities and acquired all potential repairs. The above characterize the very flexible nature of our framework since it supports both greedy (VB) and exhaustive (IB) repair finding strategies.

Our research over existing repair finding algorithms in the literature showed that they can be integrated within our framework. We proved that using adequate repair policies, such algorithms can be emulated and these facts make our framework very expressive. We believe that one of the major contributions of our work is the distinction between locally optimal(VB) and globally optimal(IB) repairs with respect to a preference, which opened the way to devise a repair finding algorithm (IB) which is immune to changes in the constraints' syntax and evaluation order.

We implemented the corresponding algorithms for the respective strategies which are supported by our framework. We proved that the two algorithms do not give, in general, the same preferred KB repairs, because the IB algorithm computes repairs which are globally optimal w.r.t. to curators' preferences, whereas VB computes locally optimal repairs. Finally, we computed the complexity bounds for both IB and VB algorithms for various types of constraints along with some experimental results for typical synthetic curated KBs.

Apart from the scenario where we are faced with an inconsistent KB with no hints on the causes of the inconsistency, we investigated cases where inconsistencies are caused by an operation upon the KB, so the information on the operation (e.g., an update, merging) can be used to guide the resolution process. We showed that the problem of preventing invalidities from arising during changes is in fact a different facet of the repair problem and we presented a possible extention of our framework in order to describe prevention policies for curated KBs in e- science.

As a future work, will apply our repair finding algorithms over real KBs along with the corresponding integrity constraints which must be satisfied. We plan to conduct experiments to evaluate the quality of the contained data and highlight the constraints which are most likely violated in practise. An additional topic of future work could be to devise an anytime repair algorithm, i.e., an algorithm that would exhibit the performance that a curator requires, possibly at a cost of lower-quality repairs. Low-quality repairs can be considered those which are not fully consistent w.r.t. said integrity constraints. For example, one could consider stopping the evaluation at a certain tree depth (assuming breadth-first search) thereby producing optimal, but incomplete repairs, or at a certain tree size (assuming depth-first search) thereby producing non-optimal, but complete repairs. Moreover, we will formally integrate the suggested extention, to capture prevention policies, into our framework. Last but not least, we will implement and experimentaly evaluate typical operations from the literature over KBs which may introduce inconsistencies such as merging, updates etc. and use this knowledge to perform repairs.

Bibliography

[1] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT-09*, 2009.

[2] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. Symb. Log.*, 50(2):510–530, 1985.

[3] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-99)*, pages 68–79, 1999.

[4] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP*, 3(4-5):393–424, 2003.

[5] L. E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 2006.

[6] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD Conference*, pages 143–154, 2005.

[7] Loreto Bravo and Leopoldo Bertossi. Semantically correct query answers in the presence of null values. In *EDBT-06*. 2006.

[8] Peter Buneman. Curated databases. Keynote, PODS-08, 2008.

[9] J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems (TODS)*, 28(4), 2003.

[10] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. *Inconsistency Tolerance*, 2005.

[11] Jan Chomicki. Consistent query answering: Five easy pieces. In *ICDT*, pages 1–17, 2007.

[12] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1/2):90–121, 2005.

[13] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB-07*, 2007.

[14] A. Deutsch. Fol modeling of integrity constraints (dependencies). In *Encyclopedia of Database Systems*. 2009.

[15] Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the 28th ACM Symposium on Principles of Database Systems (PODS-08), invited talk*, 2008.

[16] Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. Evolving ontology evolution. In *SOFSEM*, pages 14–29, 2006.

[17] M. Galperin and G. Cochrane. The 2010 nucleic acids research database issue and online database collection: A community of data resources. *Nucleic Acids Res*, 38, 2010.

[18] P. Georgiadis, I. Kapantaidakis, V. Christophides, E. Mamadou Nguer, and N. Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE-08*, 2008.

[19] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB Journal*, 2007.

[20] P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure. A framework for handling inconsistency in changing ontologies. In *ISWC*, 2005.

[21] Ian Horrocks, Peter Patel-Schneider, and Frank van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[22] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *BNAIC*, 2005.

[23] W. Kießling. Foundations of preferences in database systems. In *VLDB-02*, 2002.

[24] Sébastien Konieczny. On the difference between merging knowledge bases and combining them. In *KR*, pages 135–144, 2000.

[25] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A formal approach for rdf/s ontology evolution. In *ECAI*, 2008.

[26] G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for rdf. In *EDBT-08*, 2008.

[27] M. Lenzerini. Data integration: A theoretical perspective. In *PODS-02*, 2002.

[28] D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *KR*, 2000.

[29] S. Melnik. *Generic Model Management: Concepts and Algorithms*. Springer, 2004.

[30] T. A. Meyer, K. Lee, and R. Booth. Knowledge integration for description logics. In *AAAI-05*, 2005.

[31] D. Mindolin and J. Chomicki. Preference elicitation in prioritized skyline queries. Submitted for publication; also preprint arXiv:1008.5357v1, 2010.

[32] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *WWW-07*, 2007.

[33] N. F. Noy and M. A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI*, 2000.

[34] Natalya Fridman Noy and Mark A. Musen. Ontology versioning in an ontology management framework. *IEEE Intelligent Systems*, 19(4):6–13, 2004.

[35] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *IJCAI-03*, 2003.

[36] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *ISWC-05*, 2005.

[37] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *ISWC-05*, 2005.

[38] Yannis Theoharis, Yannis Tzitzikas, Dimitris Kotzinos, and Vassilis Christophides. On graph features of semantic web schemas. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):692–702, 2008.

[39] Jef Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, pages 375–390, 2003.

Appendix A

Proofs

A.1 Proofs of Theorems for Chapter 2

Lemma 1 *Consider a delta* $\delta = \langle \delta_a, \delta_d \rangle = \langle K_2 \setminus K_1, K_1 \setminus K_2 \rangle$*; then* $K_1 \bullet \delta = K_2$*. Moreover, for any* $\delta' = \langle \delta'_a, \delta'_d \rangle \in \Delta$ *such that* $K_1 \bullet \delta' = K_2$*, it holds that* $\delta_a \subseteq \delta'_a$*,* $\delta_d \subseteq \delta'_d$*.*

Proof. Obvious by the definitions. □

Lemma 2 *Consider a* $>^V$ *repairing policy for some* $K \in \mathcal{K}$*. Then for* $c(\vec{x}) = NextV(K)$*, it holds that there is some* $\delta \in Res(c(\vec{x}), K)$ *such that there is no* $\delta' \in Res(c(\vec{x}), K)$ *for which* $\delta' > \delta$*.*

Proof. Consider a KB K' to be a preferred repair for K per $>^V$. Then, there is some preferred repairing sequence $SEQ = \langle K_1, K_2, \ldots \rangle$, terminating after n_{SEQ} steps, for which $K_{n_{SEQ}} = K'$ and $K_1 = K$. Set $\delta = \langle K_2 \setminus K_1, K_1 \setminus K_2 \rangle$. By lemma 1, and the definition of a preferred repairing sequence, it follows that δ has the properties required. □

Lemma 3 *Consider a KB* K*, a constraint instance* $c(\vec{x})$ *and some* $\delta \in Res(c(\vec{x}), K)$*. Set* $K_0 = K \bullet \delta$*. Then* $K_0 \vDash c(\vec{x})$*. Moreover, if* $K \nvDash r(\vec{x})$*, then for all* $K' \vDash r(\vec{x})$ *it holds that there is some* $\delta_0 \in Res(c(\vec{x}), K)$ *such that for* $K_0 = K \bullet \delta_0$ *it holds that* $K_0 \setminus K \subseteq K' \setminus K$ *and* $K \setminus K_0 \subseteq K \setminus K'$*.*

Proof. Obvious by the definitions. □

Proposition 2 A repair finding algorithm R is IB-expressible iff for all KBs $\mathcal{K}, K_r, K'_r$ for which $K_r \in R(K)$, $K'_r \vDash C$, $K'_r \setminus K \subseteq K_r \setminus K$ and $K \setminus K'_r \subseteq K \setminus K_r$, it holds that $K'_r = K_r$.

Proof. ($\Rightarrow$) Suppose that R is an IB-epressible repair finding algorithm. Then, there is some repairing policy $>^I$ such that for all KBs $\mathcal{K}$ it holds that $K' \in R(K)$ iff K' is a preferred repair for K per $>^I$.
Take some $K_r \in R(K)$, $K'_r \in \mathcal{K}$ such that $K'_r \vDash \mathcal{C}$, $K'_r \setminus K \subseteq K_r \setminus K$ and $K \setminus K'_r \subseteq K \setminus K_r$. Set $\delta_r = \langle K_r \setminus K, K \setminus K_r \rangle$, $\delta'_r = \langle K'_r \setminus K, K \setminus K'_r \rangle$. Then $K \bullet \delta_r = K_r$ and $K \bullet \delta'_r = K'_r$. By the third condition of definition 7, and our hypotheses, it follows that $\delta_r = \delta'_r$, so $K_r = K'_r$.
($\Leftarrow$) Consider the set $S = \{0, 1\}$ and a feature f such that $f(\delta) = 1$ iff $K \bullet \delta \in R(K)$, $f(\delta) = 0$ iff $K \bullet \delta \notin R(K)$. We define preference $P = (S, Max(S))$, and denote by $>$ the order defined by P. Then, $\delta > \delta'$ iff $f(\delta) = 1$ and $f(\delta') = 0$, i.e., $\delta > \delta'$ iff $K \bullet \delta \in R(K)$ and $K \bullet \delta' \notin R(K)$. We will show that for any preferred repair K' for K, per $>^I$ it holds that $K' \in R(K)$ for all $K \in \mathcal{K}$.

Thus, we have to show that $PR^I(K) = R(K)$. So, take some $K \in \mathcal{K}$.
Consider some $K_1 \in R(K)$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$, so $K \bullet \delta_1 = K_1$. Then, the following hold for δ_1:

1. $K \bullet \delta_1 = K_1 \in R(K)$, so $K \bullet \delta_1 \vDash \mathcal{C}$.

2. Suppose that there is some $\delta' = \langle \delta'_a, \delta'_d \rangle$ such that $K \bullet \delta' \vDash \mathcal{C}$, $\delta'_a \subseteq \delta_{a1}$, $\delta'_d \subseteq \delta_{d1}$. Set $K_2 = K \bullet \delta'$ and $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle = \langle K_2 \setminus K, K \setminus K_2 \rangle$. Then, by Lemma 1 and our hypotheses, $K \bullet \delta_2 = K_2$ and $\delta_{a2} \subseteq \delta'_a \subseteq \delta_{a1}$, $\delta_{d2} \subseteq \delta'_d \subseteq \delta_{d1}$. Therefore, $K_2 \setminus K \subseteq K_1 \setminus K$ and $K \setminus K_2 \subseteq K \setminus K_1$, so by the hypotheses of the proposition, $K_1 = K_2$, i.e., $\delta_2 = \delta_1$, thus, $\delta' = \delta_1$.

3. Given that $K_1 = K \bullet \delta_1$ and $K_1 \in R(K)$, it follows that $f(\delta_1) = 1$. So, there is no δ' such that $\delta' > \delta_1$.

The above conditions show that δ_1 is a preferred repairing delta for K per $>^I$, so $K_1 \in PR^I(K)$, thus $R(K) \subseteq PR^I(K)$.
Consider now some $K_1 \in PR^I(K)$. There is some preferred repairing delta $\delta = \langle \delta_a, \delta_d \rangle$ for K per $>^I$ such that $K \bullet \delta = K_1$. By definition 7, $K_1 \vDash \mathcal{C}$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$. By lemma 1, $K \bullet \delta_1 = K_1$, and $\delta_{a1} \subseteq \delta_a$, $\delta_{d1} \subseteq \delta_d$, so, by definition 7, $\delta_1 = \delta$.
Let's assume that $K_1 \notin R(K)$. This means that $f(\delta) = 0$.
As we mentioned, a repair finding algorithm returns a *non-empty* set of consistent KBs i.e., $R : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset$ so there is some $K' \in R(K)$. Setting $\delta' = \langle K' \setminus K, K \setminus K' \rangle$, it follows that $K \bullet \delta' = K'$ and $f(\delta') = 1$.
Given the above, it follows that $\delta' > \delta$ and $K \bullet \delta' = K' \vDash \mathcal{C}$, because $K' \in R(K)$. By definition 7, it follows that $\delta_a \subseteq \delta'_a$, $\delta_d \subseteq \delta'_d$. Given that $\delta_1 = \delta$, we get the following: $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$.
But then we have that $K' \in R(K)$, $K_1 \in \mathcal{K}$, $K_1 \vDash \mathcal{C}$ and $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$, so by our hypotheses, $K' = K_1$, i.e., $K_1 \in R(K)$, a contradiction.
This implies that $PR^I(K) \subseteq R(K)$, which completes the proof. $\square$

Proposition 3 A repair finding algorithm R is VB-expressible iff $R(K) = \{K\}$ when $K \vDash \mathcal{C}$ and there is a family of repair finding algorithms $\{R_0^{c(\vec{A})} | c(\vec{A}) : \text{constraint instance}\}$ such that $R_0^{c(\vec{A})}$ is an IB-expressible repair finding algorithm which considers only one integrity constraint, namely $\{c(\vec{A})\}$, and $R(K) = \bigcup_{K_0 \in R_0^{c(\vec{A})}(K)} R(K_0)$ where $c(\vec{A}) = NextV(K)$, when $K \nvDash \mathcal{C}$.

Proof. ($\Rightarrow$) Take any violation selection function $NextV$. Given that R is a VB-expressible repair repair finding algorithm, there is some repairing policy $>^V$ such that for all KBs $\mathcal{K}$ it holds that $K' \in R(K)$ iff K' is a preferred repair for K per $>^V$.
For any given constraint instance $c(\vec{x})$, define the function $RSI^{c(\vec{x})} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset$ as follows:

- $RSI^{c(\vec{x})}(K) = \{K\}$ iff $K \vDash c(\vec{x})$

- $RSI^{c(\vec{x})}(K) = \{K \bullet \delta | \delta \in Res(c(\vec{x}), K)\}$ iff $K \nvDash c(\vec{x})$ and $NextV(K) \neq c(\vec{x})$.

- $RSI^{c(\vec{x})}(K) = \{K \bullet \delta | \delta \in Res(c(\vec{x}), K)$ and there is no $\delta' \in Res(c(\vec{x}), K) : \delta' > \delta\}$ iff $K \nvDash c(\vec{x})$ and $NextV(K) = c(\vec{x})$.

We must first show that for all $c(\vec{x})$, $RSI^{c(\vec{x})}$ is an IB-expressible repair finding algorithm. To see this note that:

- For all $c(\vec{x})$, $Res(c(\vec{x}), K) \neq \emptyset$; this result, combined with lemma 2 shows that $RSI^{c(\vec{x})}(K) \neq \emptyset$ for all $c(\vec{x})$ and $K \in \mathcal{K}$.

- For all $K \in \mathcal{K}$, $K' \in RSI^{c(\vec{x})}(K)$, $K' \vDash rc\vec{x})$ (see also lemma 3). This bullet and the previous one show that $RSI^{c(\vec{x})}$ is a repair finding algorithm.

- Consider some $K \in \mathcal{K}$, $K_r \in RSI^{c(\vec{x})}(K)$, $K'_r \in \mathcal{K}$ such that $K'_r \vDash \mathcal{C}$, $K'_r \setminus K \subseteq K_r \setminus K$ and $K \setminus K'_r \subseteq K \setminus K_r$. By lemma 3, it follows that $K_r \setminus K \subseteq K'_r \setminus K$ and $K \setminus K_r \subseteq K \setminus K'_r$. Thus, $K_r = K'_r$, so by proposition 2, we get that $RSI^{c(\vec{x})}$ is an IB-expressible repair finding algorithm

Now take some $K \in \mathcal{K}$. Let us initially suppose that $K \vDash \mathcal{C}$. Then, the only preferred repairing sequence is $\langle K, K, \ldots \rangle$, which terminates after 1 step, so $PR^V(K) = \{K\}$. Thus, $R(K) = \{K\}$. Let us now suppose that $K \nvDash \mathcal{C}$. Consider some $K_r \in R(K) = PR^V(K)$. There is a preferred repairing sequence SEQ terminating, say, after n_{SEQ} steps, for which $K = K_1$, $K_r = K_{n_{SEQ}}$. Now set $c(\vec{x}) = NextV(K) = NextV(K_1)$. By definition 6 and the definition of $RSI^{c(\vec{x})}$, it follows that $K_2 \in RSI^{c(\vec{x})}(K_1) = RSI^{c(\vec{x})}(K)$. Moreover, the sequence $SEQ' = \langle K_2, K_3, \ldots \rangle$ is a preferred repairing sequence of K_2 w.r.t. $>^V$ and $NextV$, which terminates after $n_{SEQ} - 1$ steps. The element in position $n_{SEQ} - 1$ is $K_{n_{SEQ}} = K_r$, so $K_r \in R(K_2)$ for some $K_2 \in RSI^{c(\vec{x})}(K)$. Thus, $R(K) \subseteq \bigcup_{K_0 \in RSI^{NextV(K)}(K)} R(K_0)$.
Now consider some $K_r \in \bigcup_{K_0 \in RSI^{NextV(K)}(K)} RS^{\mathcal{S}}(K_0)$. Then, there is some $K_0 \in RSI^{NextV(K)}(K)$ such that $K_r \in R(K_0)$. Given that $R(K_0) = PR^V(K_0)$, we can find a preferred repairing sequence $SEQ = \langle K_1, K_2, \ldots$ which terminates after n_{SEQ} steps, for which $K_1 = K_0$ and $K_{n_{SEQ}} = K_r$. By the definition of $RSI^{c(\vec{x})}$, it follows that the sequence $SEQ = \langle K'_1, K'_2, \ldots \rangle$ for which $K'_1 = K$, $K'_i = K_{i-1}$ for $i > 1$, is a preferred repairing sequence of K for $>^V$ and $NextV$, which terminates after $n_{SEQ} + 1$ steps. Therefore, $K_r = K_{n_{SEQ}} = K'_{n_{SEQ}+1}$, so $K_r \in R(K)$, thus $R(K) = \bigcup_{K_0 \in RSI^{NextV(K)}(K)} R(K_0)$.
($\Leftarrow$) Consider the set $S = \{0, 1\}$ and and a feature f such that $f(\delta) = 1$ iff $K \bullet \delta \in RSI^{NextV(K)}(K)$, $f(\delta) = 0$ iff $K \bullet \delta \notin RSI^{NextV(K)}(K)$. We define preference $P = (S, Max(S))$, and denote by $>$ the order defined by P. Then, $\delta > \delta'$ iff $f(\delta) = 1$ and $f(\delta') = 0$, i.e., $\delta > \delta'$ iff $K \bullet \delta \in RSI^{NextV(K)}(K)$ and $K \bullet \delta' \notin RSI^{NextV(K)}(K)$. We will show that $PR^V(K) = R(K)$ for all $K \in \mathcal{K}$. So, take some $K \in \mathcal{K}$.
Assume initially that $K \vDash \mathcal{C}$. Then, the sequence $SEQ = \langle K, K, \ldots \rangle$ is the only preferred repairing sequence, and terminates after 1 step, so $PR^V(K) = \{K\} = R(K)$.
Let us now assume that $K \nvDash \mathcal{C}$. We will first show that:
$PRV(K) = \bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0)$.
So, take some $K_r \in PR^V(K)$.
Suppose that $NextV(K) = c(\vec{x})$. Given that $RSI^{NextV(K)}(K) \neq \emptyset$, there is at least one $K' \in RSI^{NextV(K)}(K)$. By lemma 3, it holds that there is some $\delta_0 \in Res(c(\vec{x}), K)$ such that for $K_0 = K \bullet \delta_0$ it holds that $K_0 \setminus K \subseteq K' \setminus K$ and $K \setminus K_0 \subseteq K \setminus K'$. By proposition 2, this implies that $K_0 = K'$, so $K \bullet \delta_0 = K' \in RSI^{NextV(K)}(K)$. Therefore, by definition, $f(\delta_0) = 1$, i.e., there is at least one delta in $Res(c(\vec{x}), K)$, namely δ_0, for which $f(\delta_0) = 1$.
Given that $K_r \in PR^V(K)$, it follows that there is a preferred repairing sequence $SEQ = \langle K_1, K_2, \ldots \rangle$, terminating after n_{SEQ} steps, for which $K_1 = K$, $K_{n_{SEQ}} = K_r$. For SEQ, it holds that $K_2 = K_1 \bullet \delta = K \bullet \delta$ for some $\delta \in Res(c(\vec{x}), K)$. If we assume that $f(\delta) = 0$, then $\delta_0 > \delta$ for the above $\delta_0 \in Res(c(\vec{x}), K)$, a contradiction by definition 6. So, $f(\delta) = 1$, i.e., $K_2 = K \bullet \delta \in RSI^{NextV(K)}(K)$.
Moreover, the sequence $SEQ' = \langle K'_1, K'_2, \ldots \rangle$ is a preferred repairing sequence, terminating at

step $n_{SEQ}-1$, for which $K'_i = K_{i+1}$, so $K'_1 = K_2 \in RSI^{NextV(K)}(K)$ and $K'_{n_{SEQ}-1} = K_{n_{SEQ}} = K_r$, so $K_r \in PR^V(K_2)$ for some $K_2 \in RSI^{NextV(K)}(K)$, i.e., $K_r \in \bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0)$. Thus, $PR^V(K) \subseteq \bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0)$.
Now take some $K_r \in \bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0)$. Then, $K_r \in PR^V(K_0)$ for some $K_0 \in RSI^{NextV(K)}(K)$, so there is some preferred repairing sequence $SEQ = \langle K_1, K_2, \ldots \rangle$, terminating after n_{SEQ} steps for which $K_1 = K_0$, $K_{n_{SEQ}} = K_r$.
By the fact that $RSI^{NextV(K)}$ is an IB repair strategy and lemma 3, it follows that $K_0 = K \bullet \delta$ for some $\delta \in Res(c(\vec{x}), K)$, for which there is no $\delta' \in Res(c(\vec{x}), K)$ such that $\delta' > \delta$, where $c(\vec{x}) = NextV(K)$. Thus, the sequence $SEQ' = \langle K, K_1, K_2, \ldots \rangle$ is a preferred repairing sequence for K, which terminates after $n_{SEQ}+1$ steps, so $K_{n_{SEQ}} = K_r \in PR^V(K)$, i.e., $\bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0) \subseteq PR^V(K)$.
We conclude that, whenever $K \nvDash \mathcal{C}$, $PR^V(K) = \bigcup_{K_0 \in RSI^{NextV(K)}(K)} PR^V(K_0)$. By our hypotheses, it also holds that $R(K) = \bigcup_{K_0 \in RSI^{NextV(K)}(K)} R(K_0)$, whenever $K \nvDash \mathcal{C}$. Given this, and the fact that $PR^V(K) = R(K)$ when $K \vdash \mathcal{C}$, it follows that $PR^V(K) = R(K)$ for all $K \in \mathcal{K}$ and the proof is complete. $\square$

A.2 Proofs of Theorems for Chapter 5

Proposition 8 A prevention algorithm P is IB-expressible iff for all $w \in D$, $K = \mathcal{O}(w) \in \mathcal{K}$, $K_r \in P(w)$, $\delta_r = \langle \delta_{ar}, \delta_{ar} \rangle$ s.t. $K_r = K \bullet \delta_r$, $K'_r \in \mathcal{K}$, $\delta'_r = \langle \delta'_{ar}, \delta'_{ar}, R'_r \rangle$ s.t. $K'_r = K \bullet \delta'_r$ for which $K'_r \vDash \mathcal{C}$, $\delta'_{ar} \subseteq \delta_{ar}$, $\delta'_{dr} \subseteq \delta_{dr}$ and $R'_r \subseteq R_r$, it holds that $K'_r = K_r$.

Proof. ($\Rightarrow$) Suppose that P is an IB prevention strategy. Then, there is some IB prevention policy $>^I$ such that for all $w \in D$ it holds that $P(w) = PR^I(w)$.
Set $K = \mathcal{O}(w)$ and take some $K_r \in P(w)$, $K'_r \in \mathcal{K}$ such that $K'_r \vDash \mathcal{C}$, $K'_r \setminus K \subseteq K_r \setminus K$ and $K \setminus K'_r \subseteq K \setminus K_r$. Set $\delta_r = \langle K_r \setminus K, K \setminus K_r \rangle$, $\delta'_r = \langle K'_r \setminus K, K \setminus K'_r \rangle$. Then $K \bullet \delta_r = K_r$ and $K \bullet \delta'_r = K'_r$. By the second condition of definition 7, and our hypotheses, it follows that $\delta_r = \delta'_r$, so $K_r = K'_r$.
($\Leftarrow$) Consider the set $S = \{0, 1\}$ and the feature: $f : \Delta \times D \mapsto S$ such that $f(\delta, w) = 1$ iff $\mathcal{O}(w) \bullet \delta \in P(w)$, $f(\delta, w) = 0$ iff $\mathcal{O}(w) \bullet \delta \notin P(w)$. We define preference $P_1 = (S, Highest(S))$, and denote by $>$ the order defined by P_1. Then, $\delta > \delta'$ iff $f(\delta, w) = 1$ and $f(\delta', w) = 0$, i.e., $\delta > \delta'$ iff $\mathcal{O}(w) \bullet \delta \in P(w)$ and $\mathcal{O}(w) \bullet \delta' \notin P(w)$. Considering a prevention policy $>^I$, we will show that $PR^I(w) = P(w)$ for all $w \in D$. So, take some $w \in D$ and set $K = \mathcal{O}(w) \in \mathcal{K}$.
Consider some $K_1 \in P(w)$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$, so $K \bullet \delta_1 = K_1$. Then, the following hold for δ_1:

1. $K \bullet \delta_1 = K_1 \in P(w)$, so $K \bullet \delta_1 \vDash \mathcal{C}$.

2. Suppose that there is some $\delta' = \langle \delta'_a, \delta'_d \rangle$ such that $K \bullet \delta' \vDash \mathcal{C}$, $\delta'_a \subseteq \delta_{a1}$, $\delta'_d \subseteq \delta_{d1}$. Set $K_2 = K \bullet \delta'$ and $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle = \langle K_2 \setminus K, K \setminus K_2 \rangle$. Then, by Lemma 1 and our hypotheses, $K \bullet \delta_2 = K_2$ and $\delta_{a2} \subseteq \delta'_a \subseteq \delta_{a1}$, $\delta_{d2} \subseteq \delta'_d \subseteq \delta_{d1}$. Therefore, $K_2 \setminus K \subseteq K_1 \setminus K$ and $K \setminus K_2 \subseteq K \setminus K_1$, so by the hypotheses of the theorem, $K_1 = K_2$, i.e., $\delta_2 = \delta_1$, thus, $\delta' = \delta_1$.

3. Given that $K_1 = K \bullet \delta_1$ and $K_1 \in P(w)$, it follows that $f(\delta_1, w) = 1$. So, there is no δ' such that $\delta' > \delta_1$.

The above conditions show that $\delta_1 \in PRD^I(w)$, so $K_1 \in PR^I(w)$, thus $P(w) \subseteq PR^I(w)$.
Consider now some $K_1 \in PR^I(w)$. There is some $\delta = \langle \delta_a, \delta_d \rangle \in PRD^I(w)$ such that $K \bullet \delta = K_1$.

By definition 7, $K_1 \vDash \mathcal{C}$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$. By lemma 1, $K \bullet \delta_1 = K_1$, and $\delta_{a1} \subseteq \delta_a$, $\delta_{d1} \subseteq \delta_d$, so, by definition 7, $\delta_1 = \delta$.
Let's assume that $K_1 \notin P(w)$. This means that $f(\delta, w) = 0$.
By the definition of a prevention strategy, $P(w) \neq \emptyset$, so there is some $K' \in P(w)$. Setting $\delta' = \langle K' \setminus K, K \setminus K' \rangle$, it follows that $K \bullet \delta' = K'$ and $f(\delta', w) = 1$.
Given the above, it follows that $\delta' > \delta$ and $K \bullet \delta' = K' \vDash \mathcal{C}$, because $K' \in P(w)$. By definition 7, it follows that $\delta_a \subseteq \delta'_a$, $\delta_d \subseteq \delta'_d$. Given that $\delta_1 = \delta$, we get the following: $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$.
But then we have that $K' \in P(w)$, $K_1 \in \mathcal{K}$, $K_1 \vDash \mathcal{C}$ and $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$, so by our hypotheses, $K' = K_1$, i.e., $K_1 \in P(w)$, a contradiction.
This implies that $PR^I(w) \subseteq P(w)$, which completes the proof. $\square$

Proposition 9 Consider an operation $\mathcal{O} : D \mapsto \mathcal{K}$ and a prevention algorithm $P : D \mapsto 2^{\mathcal{K}} \setminus \emptyset$. Then P is an IB prevention algorithm iff there exists a family $\{R_w | w \in D\}$ of IB repair finding algorithms that implements P.

Proof. ($\Rightarrow$) P is an IB prevention strategy, so there is some prevention policy $>^I$ such that for all $w \in D$ $P(w) = PR^I(w)$.
Take any given IB repair finding algorithm R and set $R_w(K) = P(w)$ whenever $K = \mathcal{O}(w)$ and $R_w(K) = R(K)$ whenever $K \neq \mathcal{O}(w)$.
We need to show that R_w is an IB repair finding algorithm for all $w \in D$. First of all, given that R is a repair finding algorithm and P is a prevention algorithm, it follows that R_w is a repair finding algorithm.
Now take any $K \in \mathcal{K}$. If $K \neq \mathcal{O}(w)$ then the condition of proposition 2 holds, because $R_w(K) = R(K)$ and R is an IB repair finding algorithm. If $K = \mathcal{O}(w)$ then $R_w(K) = P(w)$ and P is an IB prevention algorithm, so, by proposition 2, it follows that the condition of proposition 2 holds again. Thus,R_w is an IB repair finding algorithm.
Finally, the family $\{R_w | w \in D\}$ implements P, by the definition of R_w.
($\Leftarrow$) Consider the set $S = \{0, 1\}$ and the feature: $f : \Delta \times D \mapsto S$ such that $f(\delta, w) = 1$ iff $\mathcal{O}(w) \bullet \delta \in RS_w(\mathcal{O}(w))$, $f(\delta, w) = 0$ iff $\mathcal{O}(w) \bullet \delta \notin R_w(\mathcal{O}(w))$. We define preference $P_1 = (S, Highest(S))$, and denote by $>$ the order defined by P_1. Then, $\delta > \delta'$ iff $f(\delta, w) = 1$ and $f(\delta', w) = 0$, i.e., $\delta > \delta'$ iff $\mathcal{O}(w) \bullet \delta \in R_w(\mathcal{O}(w))$ and $\mathcal{O}(w) \bullet \delta' \notin R_w(\mathcal{O}(w))$. Considering a prevention policy $>^I$, we will show that $PR^I(w) = P(w)$ for all $w \in D$. So, take some $w \in D$ and set $K = \mathcal{O}(w)$.
Consider some $K_1 \in P(w) = R_w(\mathcal{O}(w))$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$, so $K \bullet \delta_1 = K_1$. Then, the following hold for δ_1:

1. $K \bullet \delta_1 = K_1 \in P(w)$, so $K \bullet \delta_1 \vDash \mathcal{C}$.

2. Suppose that there is some $\delta' = \langle \delta'_a, \delta'_d \rangle$ such that $K \bullet \delta' \vDash \mathcal{C}$, $\delta'_a \subseteq \delta_{a1}$, $\delta'_d \subseteq \delta_{d1}$. Set $K_2 = K \bullet \delta'$ and $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle = \langle K_2 \setminus K, K \setminus K_2 \rangle$. Then, by Lemma 1 and our hypotheses, $K \bullet \delta_2 = K_2$ and $\delta_{a2} \subseteq \delta'_a \subseteq \delta_{a1}$, $\delta_{d2} \subseteq \delta'_d \subseteq \delta_{d1}$. Therefore, $K_2 \setminus K \subseteq K_1 \setminus K$ and $K \setminus K_2 \subseteq K \setminus K_1$, so by the fact that R_w is an IB repair finding algorithm and proposition 2, $K_1 = K_2$, i.e., $\delta_2 = \delta_1$, thus, $\delta' = \delta_1$.

3. Given that $K_1 = K \bullet \delta_1$, $P(w) = R_w(\mathcal{O}(w))$ and $K_1 \in P(w)$, it follows that $K \bullet \delta_1 \in R_w(\mathcal{O}(w))$, so $f(\delta_1, w) = 1$. So, there is no δ' such that $\delta' > \delta_1$.

The above conditions show that $\delta_1 \in PRD^I(w)$, so $K_1 \in PR^I(w)$, thus $P(w) \subseteq PR^I(w)$.
Consider now some $K_1 \in PR^I(w)$. There is some $\delta = \langle \delta_a, \delta_d \rangle \in PRD^I(w)$ such that $K \bullet \delta = K_1$.

By definition 7, $K_1 \vDash \mathcal{C}$. Set $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle = \langle K_1 \setminus K, K \setminus K_1 \rangle$. By lemma 1, $K \bullet \delta_1 = K_1$, and $\delta_{a1} \subseteq \delta_a$, $\delta_{d1} \subseteq \delta_d$, so, by definition 7, $\delta_1 = \delta$.
Let's assume that $K_1 = \mathcal{O}(w) \bullet \delta \notin P(w) = RS_w(\mathcal{O}(w))$. This means that $f(\delta, w) = 0$.
By the definition of a repair finding algorithm strategy, $R_w(\mathcal{O}(w)) \neq \emptyset$, so there is some $K' \in R_w(\mathcal{O}(w))$. Setting $\delta' = \langle K' \setminus K, K \setminus K' \rangle$, it follows that $K \bullet \delta' = K'$ and $f(\delta', w) = 1$.
Given the above, it follows that $\delta' > \delta$ and $K \bullet \delta' = K' \vDash \mathcal{C}$, because $K' \in R_w(\mathcal{O}(w))$. By definition 7, it follows that $\delta_a \subseteq \delta'_a$, $\delta_d \subseteq \delta'_d$. Given that $\delta_1 = \delta$, we get the following: $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$.
But then we have that $K' \in R_w(\mathcal{O}(w))$, $K_1 \in \mathcal{K}$, $K_1 \vDash \mathcal{C}$ and $K_1 \setminus K \subseteq K' \setminus K$ and $K \setminus K_1 \subseteq K \setminus K'$, so, given that R_w is an IB repair finding algorithm and theorem 2, $K' = K_1$, i.e., $K_1 \in R_w(\mathcal{O}(w))$, a contradiction. Thus, $K_1 = \mathcal{O}(w) \bullet \delta \in P(w) = R_w(\mathcal{O}(w))$.
This implies that $PR^I(w) \subseteq P(w)$, which completes the proof. □

A.3 Proofs of Theorems for Chapter 3

Firstly, we will introduce some basic symbols and terminology which will be used in the following proofs.

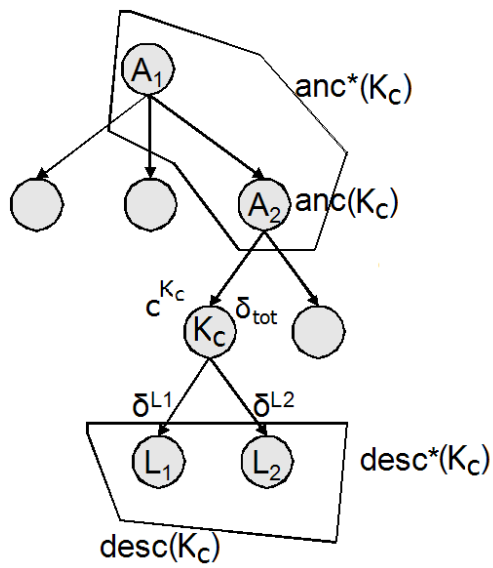

Figure A.1: Algorithms Terminology

For a node K_c of the tree, we denote by $desc^*(K_c)$, $anc^*(K_c)$ the set of all descendant and ancestor nodes of K_c respectively. We use $desc(K_c), anc(K_c)$ to denote the set of direct descendants/ancestors respectively. For example, in the resolution tree of Figure A.1 set $desc^*(K_c) = \{L_1, L_2\}$, $desc(K_c) = \{L_1, L_2\}$, $anc^*(K_c) = \{A_1, A_2\}$ and $anc(K_c) = \{A_2\}$. We denote by c^{K_c}(see Figure A.1) the constraint instance being repaired at node K_c. In leaf nodes, there does not exist any constraint to be violated thus we set $c^{K_c} = \emptyset$.

We denote by δ_{tot} the accumulated delta that has been produced so far (until this node), and by δ^{K_c} (or simply δ) the delta that was used in the (only) ancestor node of K_c (say $K_{c0} \in anc(K_c)$) to repair the constraint $c^{K_{c0}}$ (and is "added" to the corresponding δ_{tot} to create δ_{tot} for K_c (i.e., $\delta_{tot} = \delta_{tot} \uplus \delta^{K_c}$. In other words, δ^{K_c} is the delta that repaired constraint instance $c^{K_{c0}}$ and "led" to K_c. Moreover, given a node D, we denote $\delta^{K_c *}$ to be equivalent with delta

δ_{tot}, and we will use this symbolism for the next proofs as it is more convenient in order to be able to refer to the state of the KB (i.e., K_c) and the accumulated delta which led to this state (i.e., δ^{K_c*}). Similarly, δ^{L1} and δ^{L2} are the deltas which repaired the constraint instance c^{K_c} and created the KBs L_1, L_2 respectively (see Figure A.1). For a leaf node, say K_c, δ_{tot}(or δ^{K_c*}) corresponds to the delta reported as a repairing delta from this branch.

Overusing notation, for two deltas $\delta_1 = \langle \delta_{a1}, \delta_{d1} \rangle$, $\delta_2 = \langle \delta_{a2}, \delta_{d2} \rangle$, we will write $\delta_1 \subseteq \delta_2$ iff $\delta_{a1} \subseteq \delta_{a2}$ and $\delta_{d1} \subseteq \delta_{d2}$. Note also, that based on the second bullet of Definitinion 7, repairing deltas which are minimal w.r.t. the subset relation are called *useful*. Similarly, $\delta_1 \subset \delta_2$ iff $\delta_1 \subseteq \delta_2$ and $\delta_1 \neq \delta_2$. The above can be extended for the case of deltas with wildcards as well. Given two deltas with wildcards $[[\delta_1]]^{\mu_1}$, $[[\delta_2]]^{\mu_2}$, we say that $[[\delta_1]]^{\mu_1} \subseteq [[\delta_2]]^{\mu_2}$ (equivalently $[[\delta_1]]^{\mu_1} \subset [[\delta_2]]^{\mu_2}$) iff $\delta_1 \subseteq \delta_2$ (equivalently $\delta_1 \subset \delta_2$) for all $\delta_1 \in [[\delta_1]]^{\mu_1}$, $\delta_2 \in [[\delta_1]]^{\mu_2}$. Also, for a delta $\delta = \langle \delta_a, \delta_d \rangle$ and a KB K, we write $K \vDash \delta$ iff $\delta_a \subseteq K$, $\delta_d \cap K = \emptyset$. Equivalently, $K \vDash \delta$ iff $K \bullet \delta = K$. This can easily be extended for the case of KBs and deltas with wildcards. We say that $[[K]]^{\mu_K} \vDash [[\delta]]^{\mu_\delta}$ iff $K \vDash \delta$ for all $K \in [[K]]^{\mu_K}$ and $\delta \in [[\delta]]^{\mu_\delta}$. Next and for ease of readability, we will write K, δ in the following proofs but we will imply KBs and deltas with wildcards.

Lemma 4 *Consider a KB K and a useful repairing delta $\delta = \langle \delta_a, \delta_d \rangle$. Then $\delta_a \cap K = \emptyset$, $\delta_d \subseteq K$, $\delta_a \cap \delta_d = \emptyset$.*

Proof. Suppose that $\delta_a \cap K \neq \emptyset$ and take $x \in \delta_a \cap K$. Set $\delta_0 = \langle \delta_{a0}, \delta_{d0} \rangle = \langle \delta_a \setminus \{x\}, \delta_d$. Obviously, $K \bullet \delta_0 = K \bullet \delta$, so δ_0 is a repairing delta and $\delta_0 \subset \delta$, so δ is not a useful repairing delta, a contradiction. So $\delta_a \cap K = \emptyset$.
For the second condition, similarly, suppose that $\delta_d \nsubseteq K$ and take $x \in \delta_d \setminus K$. Set $\delta_0 = \langle \delta_{a0}, \delta_{d0} \rangle = \langle \delta_a, \delta_d \setminus \{x\}$. Obviously, $K \bullet \delta_0 = K \bullet \delta$, so δ_0 is a repairing delta and $\delta_0 \subset \delta$, so δ is not a useful repairing delta, a contradiction. So $\delta_d \subseteq K$.
The third condition is an obvious consequence of the first two. □

In all the following lemmas, we assume a run of the information-based algorithm, creating a tree, consisting of nodes as explained in section 3.

Lemma 5 *Consider a KB K, a violation selection function $NextV$ and the corresponding repair algorithm. For any leaf L it holds that $\delta_a \cap K = \emptyset$, $\delta_d \subseteq K$, $\delta_a \cap \delta_d = \emptyset$ (where $\delta = \delta^{L*}$).*

Proof. In every node D we check whether $\delta_a^{D*} \cap \delta_d^{D*} = \emptyset$ ($isConfliciting(\delta^{D*}) = true$ in line 15 of IB_{opt} algorithm), and the entire branch is rejected if it is not so, the latter condition is true.
Now suppose that $\delta_a \cap K \neq \emptyset$, so there is some $x \in \delta_a \cap K$. Suppose that x was added in some node D_x. If D_x is the root node, then, given that $x \in K$, it would have been removed from δ_a (line 6 in the IB_{opt} algorithm). So D_x is not the root node. Set D the (only) direct ancestor of D_x. Since $x \in \delta_a^{D_x}$, it follows (line 6) that $x \notin K \bullet \delta^{D*}$ (otherwise it would be removed from $\delta_a^{D_x}$). Given that $x \in K$, it follows that $x \in \delta_d^{D*}$, so $x \in \delta_a^{D_x*} \cap \delta_d^{D_x*}$, a contradiction.
Using similar arguments, the second condition can be shown. □

Proposition 6 Consider a KB K and a violation selection function $NextV$. Suppose that there are two nodes, D, L, and a delta δ_{rej} for which:

- L is a leaf node, i.e., $desc(L) = \emptyset$.
- D is an ancestor of L, i.e., $D \in anc(L)$.

- δ_{rej} is a possible resolution for c^D, i.e., $\delta_{rej} \in Res(c^D, K)$.
- δ_{rej} was not selected in the branch that led to L, i.e., for the only node $D' \in anc(L) \cap desc_0(D)$ it holds that $\delta^{D'} \nsubseteq \delta_{rej}$.
- $K \bullet \delta^{L*} \vDash \delta_{rej}$

Then, there is a second leaf node, say $L' \neq L$, for which $\delta^{L'*} \subseteq \delta^{L*}$.

Proof. For ease of symbolism, set $\delta = \delta^{L*}$ and $\delta' = \delta^{L'*}$. By the algorithm and the fact that L is a leaf node, it follows that δ is a repairing delta, so $K_0 = K \bullet \delta$ is valid.
Set D_1 the (only) direct descendant of D for which $\delta^{D_1} = \delta_{rej}$. By our hypotheses, $D_1 \notin anc(L)$. Suppose that $r^{D_1} = c_1$. Then, $K_0 \vDash r_1$, so there is some $\delta_1 \in Res(c_1, K_0)$ such that $K_0 \vDash \delta_1$. So, we select the resolution through δ_1 and set D_2 the node for which $\delta^{D_2} \subseteq \delta_1$. Note here that we don't write $\delta^{D_2} = \delta_1$, because δ_1 may contain void changes. Also note that there is always at least one δ_1 with this property (if there are more than one, we arbitrarily select one) and there is always exactly one node D_2 for which $\delta^{D_2} \subseteq \delta_1$.
We repeat the same process for D_2, selecting the next violation, say $c_2 = c^{D_2}$ and choosing some δ_2 and a path as above, leading to a node D_3 for which $\delta^{D_3} \subseteq \delta_2$. Eventually, after, say, n steps, we will reach a leaf node, say D_n, for which $K \bullet \delta^{D_n*}$ is valid. We will show that by setting $L' = D_n$ it holds that $\delta' \subseteq \delta$, where $\delta' = \delta^{L'*} = \delta^{D_n*}$.
Take some $x \in \delta'_a$. It follows that $x \notin K$. Suppose that x was added in δ'_a at node D_x, i.e., $x \in \delta_a^{D_x}$. If $D_x \in anc(D)$ or $D_x = D$, then $D_x \in anc(L)$, so $x \in \delta_a$. If $D_x \in desc_0(D)$ then $D_x = D_1$ and $\delta^{D_x} \subseteq \delta_{rej}$, so $x \in \delta_{rej_a}$. Note also that $K \bullet \delta \vDash \delta_{rej}$ by our hypotheses and $x \notin K$, so $x \in \delta_a$. Finally, if $D_x \in desc(D)$ and $D_x \neq D_1$, then, by the process of generating δ' it follows that $x \in K_0 = K \bullet \delta$ and $x \notin K$. Thus, $x \in \delta_a$.
Now take some $x \in \delta'_d$. It follows that $x \in K$. Suppose that x was added in δ'_d at node D_x, i.e., $x \in \delta_d^{D_x}$. If $D_x \in anc(D)$ or $D_x = D$, then $D_x \in anc(L)$, so $x \in \delta_d$. If $D_x \in desc_0(D)$ then $D_x = D_1$ and $\delta^{D_x} \subseteq \delta_{rej}$, so $x \in \delta_{rej_d}$. Note also that $K \bullet \delta \vDash \delta_{rej}$ by our hypotheses and $x \in K$, so $x \in \delta_d$. Finally, if $D_x \in desc(D)$ and $D_x \neq D_1$, then, by the process of generating δ' it follows that $x \notin K_0 = K \bullet \delta$ and $x \in K$. Thus, $x \in \delta_d$.
We conclude that $\delta' \subseteq \delta$ and the proof is complete. □

Proposition 7 Consider a KB K and a violation selection function $NextV$. Suppose that there are two nodes, D, L, and a delta δ_{rej} for which:

- L is a leaf node, i.e., $desc(L) = \emptyset$.
- D is an ancestor of L, i.e., $D \in anc(L)$.
- δ_{rej} is a possible resolution for c^D, i.e., $\delta_{rej} \in Res(c^D, D)$.
- δ_{rej} was not selected in the branch that led to L, i.e., for the only node $D' \in anc(L) \cap desc_0(D)$ it holds that $\delta^{D'} \nsubseteq \delta_{rej}$.
- $\delta^{L*} \supseteq \delta_{rej}$

Then, there is a second leaf node, say $L' \neq L$, for which $\delta^{L'*} \subseteq \delta^{L*}$.

Proof. Given that $\delta^{L*} \supseteq \delta_{rej}$, it follows that $K \bullet \delta^{L*} \vDash \delta_{rej}$, so by proposition 6 the result follows. □

Corollary 1 *Consider a KB K and a violation selection function $NextV$. If a branch is pruned per the conditions in line 15 in IB_{opt} algorithm then this branch could not have led to any new useful repairing delta.*

Proof. If a branch is pruned under condition ($isConflicting(\delta^{D*}) = true$) then none of its leaves will correspond to a useful repairing delta (because the deltas are only increased in size and do not terminate – see also lemma 4). If a branch is pruned under the other condition ($nodeCheck(RepOpt, Pairs) == true$), then for every leaf and its corresponding delta δ, there is always another delta δ' for which $\delta' \subseteq \delta$. If $\delta' \subset \delta$, then δ is not useful; if $\delta' = \delta$, then at least one of the two should be kept, which is guaranteed by the mapping $Pairs$ which is used for such cases and to avoid losing useful deltas. □

Lemma 6 *Consider a KB K, a useful repairing delta $\delta_0 = \langle \delta_{a0}, \delta_{d0} \rangle$, and a violation selection function $NextV$. Then, there is some leaf node L in the recursive tree for which $\delta^{L*} = \delta_0$.*

Proof. First of all, per corollary 1 it follows that none of the conditions checked in line 15 of IB_{opt} algorithm will prune a branch leading to a useful repairing delta. Therefore, if the algorithm without these conditions leads to a leaf node L as in the theorem, it will also lead there with the conditions.
If K is valid, then the only useful repairing delta is $\delta_0 = \langle \emptyset, \emptyset \rangle$, and the proof is obvious. So, suppose that K is not valid and set $K_0 = K \bullet \delta_0$.
Set D_1 the root node of the tree, and suppose that $NextV(K) = r_1$. Then $r^{D_1} = r_1$. Also, $K_0 \vDash r_1$, so there is some $\delta_1 \in Res(r_1(\vec{x}), K_0)$ such that $K_0 \vDash \delta_1$. So, we select the resolution through δ_1 and set D_2 the node for which $\delta^{D_2} \subseteq \delta_1$. Note here that we don't write $\delta^{D_2} = \delta_1$, because δ_1 may contain void changes. Also note that there is always at least one δ_1 with this property (if there are more than one, we arbitrarily select one) and there is always exactly one node D_2 for which $\delta^{D_2} \subseteq \delta_1$.
We repeat the same process for D_2, selecting the next violation, say $r_2 = r^{D_2}$ and choosing some δ_2 and a path as above, leading to a node D_3 for which $\delta^{D_3} \subseteq \delta_2$. Eventually, after, say, n steps, we will reach a leaf node, say D_n, for which $K \bullet \delta^{D_n*}$ is valid. Set $\delta = \delta^{D_n*}$. We will show that $\delta = \delta_0$.
Take some $x \in \delta_a$. By the definition of δ, no void changes are considered, so $x \notin K$; also, by the process of generating δ it follows that $x \in K_0 = K \bullet \delta_0$. Thus, $x \in \delta_{a0}$.
Now take some $x \in \delta_d$. By the definition of δ, no void changes are considered, so $x \in K$; also, by the process of generating δ it follows that $x \notin K_0 = K \bullet \delta_0$. Thus, $x \in \delta_{d0}$.
We conclude that $\delta \subseteq \delta_0$. Suppose that $\delta \subset \delta_0$. Given that $K \bullet \delta$ is valid by our construction, this hypothesis implies that δ_0 is not relevant, a contradiction. Thus, $\delta = \delta_0$.
Therefore, for $L = D_n$, it follows that $\delta^{L*} = \delta = \delta_0$. □

Lemma 7 *Consider two leaf nodes L_1, L_2, returning deltas $\delta_1 \delta_2$ respectively. Suppose also that for all constraint instances c and all deltas $\delta \in Res(r, K)$ it holds that $K \bullet \delta_1 \vDash \delta$ iff $K \bullet \delta_2 \vDash \delta$. Then $K \bullet \delta_1 = K \bullet \delta_2$.*

Proof. Take some x for which there is a constraint instance c and $\delta \in Res(c, K)$ such that $x \in \delta_a$. Then, obviously by our assumptions, $x \in K \bullet \delta_1$ iff $x \in K \bullet \delta_2$; same argumentation if $x \in \delta_d$. Now take some x for which there is no constraint instance c and $\delta \in Res(c, K)$ such that $x \in \delta_a$ or $x \in \delta_d$. Then, by the construction of the algorithm, $x \notin \delta_{1a}$, $x \notin \delta_{1d}$, $x \notin \delta_{2a}$, $x \notin \delta_{2d}$. So, if $x \in K$ then $x \in K \bullet \delta_1$ and $x \in K \bullet \delta_2$, whereas if $x \notin K$ then $x \notin K \bullet \delta_1$ and $x \notin K \bullet \delta_2$. We conclude that $K \bullet \delta_1 = K \bullet \delta_2$. □

Lemma 8 *Consider a KB K and a violation selection function $NextV$. Then, if a delta δ is returned by the algorithm then it is a useful repairing delta.*

Proof. By the construction of the algorithm, δ is repairing. Suppose that δ is non-useful and is returned from a leaf node, say L. Then there is some relevant repairing delta, say δ', for which $\delta' \subset \delta$. Per lemma 6, δ' is returned, via a (different) leaf node, say L' ($L' \neq L$).
Let us initially suppose that for all constraint instances c and all $\delta_c \in Res(c, K_1)$ it holds that $K_1 \vDash \delta_c$ iff $K_2 \vDash \delta_c$. Then, by lemma 7 it follows that $K_1 = K_2$. Given that $\delta' \subset \delta$, there is some x such that $x \in \delta_a \setminus \delta'_a$ or $x \in \delta_d \setminus \delta'_d$. If $x \in \delta_a \setminus \delta'_a$, then $x \in K_2 = K_1$, so, given that $x \notin \delta_a$, it follows that $x \in K$, a contradiction because $x \in \delta'_a$. Similarly, we reach a contradiction if $x \in \delta_d \setminus \delta'_d$. We conclude that there is some r and some $\delta_c \in Res(c, K_1)$ for which $K_1 \vDash \delta_c$ and $K_2 \nvDash \delta_c$ or $K_1 \nvDash \delta_c$ and $K_2 \vDash \delta_c$.
If the former holds, then there is some $\delta'_c \in Res(c', K_2)$ such that $K_2 \vDash \delta'_c$ (because K_2 is valid). Similarly, if the latter holds, then there is some $\delta'_c \in Res(c, K_1)$ such that $K_1 \vDash \delta'_c$. We conclude that there are two deltas, δ_c, δ'_c such that $\delta_c \neq \delta'_c$, $\delta_c, \delta'_c \in Res(c', K_2)$ and $K_1 \vDash \delta_c$, $K_2 \vDash \delta'_c$ (or vice-versa – perform a renaming of δ_c, δ'_c if necessary).
Take some $x \in \delta'_{ra}$. If $x \in \delta'_a$, then $x \in \delta_a$, so $x \in K_1$. If $x \notin \delta'_a$, then, given that $x \in K_2$, it follows that $x \in K$ and $x \notin \delta'_d$, so $x \notin \delta_d$, so $x \in K_1$. Now take some $x \in \delta'_{cd}$. If $x \in \delta'_d$ then $x \in \delta_d$, so $x \notin K_1$. If $x \notin \delta'_d$, then, given that $x \notin K_2$, it follows that $x \notin K$ and $x \notin \delta'_a$, so $x \notin \delta_a$, so $x \notin K_1$. We conclude that $K_1 \vDash \delta'_c$, a contradiction. □

Corollary 2 *Consider a KB K and a violation selection function $NextV$. Then, a delta will be returned by the algorithm iff it is a useful repairing delta.*

Proof. Per lemma 6 all useful repairing deltas will be returned. Per the construction of the algorithm, all deltas returned will be repairing and per lemma 8, if a delta is returned, then it is useful. □

Corollary 3 *Consider a KB K and two different violation selection functions $NextV_1, NextV_2$. Then, a delta δ, and the corresponding repair $K_1 = K \bullet \delta$ will be returned by the algorithm using $NextV_1$ iff it is returned by the algorithm using $NextV_2$.*

Proof. By corollary 2, a delta δ will be returned by the algorithm using $NextV_1$ iff it is a useful repairing delta. Similarly, using the same corollary δ will be returned by the algorithm using $NextV_2$ iff it is a relevant repairing delta. We conclude that δ will be returned by the algorithm using $NextV_1$ iff δ will be returned by the algorithm using $NextV_2$. □
Note. As a result, any repair finding algorithm which is affected the evaluation order i.e., a different violation selection function returns different useful repairing deltas, will not produce them correctly. We found cases of constraints and KBs where the evaluation order affects the created repairing deltas when we incorporate replacements in our framework (see section 2.3). As a result we couldn't guarantee the correctness of our algorithm and this was the reason we do not support them in our framework.

Lemma 9 *Consider a KB $K \in \mathcal{K}$, a constraint instance $c(\vec{x})$ s.t. $K \nvDash (\vec{x})$ and a $\delta_\in = \langle \delta_{a\in}, \delta_{d\in} \rangle \Delta$. Delta δ is a useful repairing delta of K w.r.t. $\{c(\vec{x})\}$ iff $\delta \in Res(c(\vec{x}), K)$.*

Proof. Let $c(\vec{x}) = P(\vec{x}) \rightarrow \vee_{i=1,\ldots,n} \exists \vec{y_i} Q_i(\vec{x}, \vec{y_i})$ s.t. $P(\vec{x}) = P_1(\vec{x}) \wedge P_2(\vec{x}) \wedge \ldots \wedge P_k(\vec{x})$ for some $k \geq 1$ and $Q_i(\vec{x}, \vec{y_i}) = Q_{i_1}(\vec{x}, \vec{y_i}) \wedge Q_{i_2}(\vec{x}, \vec{y_i}) \wedge \ldots \wedge Q_{i_m}(\vec{x}, \vec{y_i})$ for some $m \geq 0$ depending on i. The DNF form of $c(\vec{x})$ is:

$\neg P_1(\vec{x}) \vee \neg P_2(\vec{x}) \vee \ldots \neg P_k(\vec{x}) \vee \exists \vec{y_1} H_1(\vec{x}, \vec{y_1}) \vee \exists \vec{y_2} H_2(\vec{x}, \vec{y_2}) \vee \ldots \vee \exists \vec{y_n} H_n(\vec{x}, \vec{y_n})$.
As a result, any repairing delta of this rule must contain at least one of the elements of the disjunction which appear in $c(\vec{x})$. So, a delta will be a relevant repairing delta iff it is of the form: $\langle \emptyset, \{P_j(\vec{x})\}\rangle$, $1 \leq j \leq k\}$ or $\langle \{Q_{i1}(\vec{x}, \vec{y}), Q_{i2}(\vec{x}, \vec{y}), \ldots, Q_{ik_i}(\vec{x}, \vec{yi})\}, \emptyset\rangle$, where $\vec{y}$ is a tuple of constants and for $\vec{x}, \vec{z}$ the equality atoms $H_{i,j}(\vec{x}, \vec{z})$ for $k_i < j \leq m_i$ hold where $1 \leq i \leq n\}$. The result follows from Definition 5. □

Lemma 10 *Consider a KB $K \in \mathcal{K}$, and a set of constraint instances $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ such that $K \nvDash c$, for all $c \in \mathcal{C}$. A delta $\delta = \langle \delta_a, \delta_d \rangle$ will be a usefull repairing delta of K iff $\exists \delta_i \in Res(c_i, K)$, $\forall c_i \in \mathcal{C}$ such that :*

1. *$\forall x \in \delta_{ia}$, it exclusively holds ($x \in K$ and $x \notin \delta_d$) or $x \in \delta_a$.*
2. *$\forall x \in \delta_{id}$, it exclusively holds ($x \notin K$ and $x \notin \delta_a$) or $x \in \delta_d$.*

Proof. ($\rightarrow$) Given that δ is a useful repairing delta of K, it holds that $K \bullet \delta \vDash \mathcal{C}$. Moreover, $K \bullet \delta \vDash c_i$, $\forall c_i \in \mathcal{C}$. From Lemma 9, there exists $\delta_i \in Res(c_i, K)$ such that $K \bullet \delta_i \vDash c_i$. Thus, any tuple x which belongs in δ_{ia}, will also belongs in $K \bullet \delta$. We notice that condition #1 is satisfied. On the other hand, any tuple x which belongs in δ_{id}, will not belong in $K \bullet \delta$. We also notice that #2 is satisfied.
($\leftarrow$) Given that there exists a delta $\delta_i \in Res(c_i, K)$, then from Lemma 9 it is a useful repairing delta. We can easily induce that for any set of constraints $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ such that $K \nvDash c$, for all $c \in \mathcal{C}$, if there exists a $\delta_i \in Res(c_i, K)$, then there exists a delta $\delta = \langle \delta_a, \delta_d \rangle$ such that $K \bullet \delta \vDash \mathcal{C}$. □

Proposition 4 Consider a KB K, a constraint selection function $NextV$ and an information based repairing policy $>^I$. Then, function *repair* returns exactly the preferred repairs of K with respect to $>^I$.

Proof. When the employed policy is an information based, then function IB (or IB_{opt}) is called. Based on Definition 7, a KB K' is a preferred reapir of K iff there is some preferred repairing delta δ of K with respect to $>^I$ such that $K' = K \bullet \delta$. Based on the same definition, there are three conditions such that if they are satisfied by a delta δ, then δ is a preferred repairing delta. The first two conditions must be satisfied by deltas produced from IB(or IB_{opt}). Condition #3 of Definition 7 is enforced in line 4 of *repair* function. Based on the condition in line 1 of both IB and IB_{opt} algorithms, they create resolution trees whose leaves represent repairs K' (i.e., $K' \vDash \mathcal{C}$) of the initial K and the branches represent the corresponding repairing deltas δ' (i.e. $K \bullet \delta' = K'$). As a result, the condition #1 of Definition 7 holds. Based on Corollary 2, we know that any delta will be returned by the algorithm iff it is a useful repairing delta. Thus the condition #2 of Definition 7 holds as well. Back in *repair* function, we can consider all the returned useful repairing deltas as tuples with the repsective feature values as atributes. The problem of finding the most preferred repairing delta w.r.t. a complex preference expression referring to specific attributes, has been addressed in the field of Preference Theory. Thus, starting from preferences upon features, we move to preferences upon deltas, which are the prefered deltas w.r.t. our preferences. As a result, in line 4, the Algorithm 1 will sort the repairing deltas w.r.t. the defined preferences and the application of those preferred deltas over the initial K will successfuly create the set of preferred repairs PR.

We also have to prove that there not exists any delta which satisfies these conditions(i.e., preferred repairing delta) and is not returned by IB and IB_{opt} algorithms. Based on Lemma 10,

we cannot construct any useful repairing delta of a set of constraints without considering the resolution sets of these constraints. Moreover functions *clean*, *isConflicting* prune, a priori, branches which would lead to non useful deltas or non repairing because they latter do not reach to a consistent KB.

Generalizing the above, there are no preferred repairing deltas which are not returned by algorithms IB and IB_{opt}. □

Proposition 5 Consider a KB K, a constraint selection function $NextV$ and a violation based repairing policy $>^V$. Then, function *repair* returns exactly the preferred repairs of K with respect to $>^V$.

Proof. When the employed policy is a violation based, then function VB is called. Based on Definition 6, a KB K' is a preferred repair of K for $>^V$ iff there is some preferred repairing sequence SEQ of K for $>^V$ which terminates after n steps and $K' = K_n$. First of all, the check in line 6 guarantees that we reject circular sequences because we guarantee that we will not move to a KB's state which has been revisited before resulting to a non finite sequence which is definitely non preferred. Next we will show that all the sequences produced by our algorithm, are preferred as well. Setting $K_1 = K$, the algorithm begins from an initial KB K_1. If K_1 is inconsistent ($K_1 \nvDash \mathcal{C}$)), we find the next violated constraint instance using $NextV$. Again, using preferences upon specific features, we keep only the preferred repairing deltas of this constraint w.r.t. $>^V$ and the ordering is applied among deltas which belong in the resolution set of the constraint itself. Each preferred repairing delta is applied upon the current KB and we recursively call VB until we have a valid KB (i.e., a preferred repair). Note that this process describes the construction process of a preferred repairing sequence (i.e., condition #2 in Definition 6) and creates a resolution tree whose branches represent preferred repairing sequences which terminate. Each node is a KB instance and the leaf node is the preferred repair and also an element of PR. Moreover, by the definition of preferred repairing sequence, it is impossible to find a preferred repairing sequence which is not returned by VB function. □

A.4 Proofs for emulating other Repair Strategies

A.4.1 Strategies applicable in Relationals KBs

Relational KBs are described by a first-order logic predicate language. It is defined upon a schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$ s.t.:

- $\mathcal{U}$ is the possibly infinite database domain such that $null \in \mathcal{U}$.
- $\mathcal{R}$ is a fixed set of database predicates (also called relations), where each relation $\mathcal{R}$ has a finite, ordered set of attributes A_R. The attribute in position i of predicate $R \in \mathcal{R}$ is denoted by $R[i]$. For $A \subseteq A_R$, $R[A]$ denotes predicate R projected on the attributes in A.
- B is a fixed set of built-in predicates, like comparison predicates.

The general form of the integrity constraints is:

$$\forall \vec{u}(\wedge_{i=0,\ldots,n} B_i(\vec{u}) \rightarrow \vee_{i=1,\ldots,n} \exists \vec{v_i} H_i(\vec{u}, \vec{v_i}) \vee \phi$$

where:

- Every B_i, H_j predicate belongs in $\mathcal{R}$.
- Every constant value c which may appear in the constraint, must belong in $\mathcal{U}$.
- Expressions contained in ϕ must use fixed constants from $\mathcal{U}$ or variables and the fixed build in predicates contained in $\mathcal{B}$ without using universal ($\forall$) and existential ($\exists$) quantifiers.

We notice that the integrity contstraints used in relational databases are also DED constraints.

Subset Repairs

Subset repairs are presented in [1]. Given a potentialy inconsistent KB K, this strategy creates repairs (i.e., repaired KBs) K' which are maximal consistent sub-instances of K. This means that, given a set of constraints $\mathcal{C}$, K' is a subset repair of K iff $K' \vDash \mathcal{C}$ and there is no instance K'' s.t.: $K'' \vDash \mathcal{C}$ and $K' \subset K'' \subseteq K$. Based on the form of constraints in $\mathcal{C}$ (i.e., DEDs), there will always exist at least one subset repair, thus this strategy creates always at least one repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$R_{ss} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_{ss}(K) = \{K_i | K_i \text{ is a subset repair of } K\}.$$

The fact that we prefer repairs which are sub instances of the initial KB, means that we prefer to resolve violations by deleting, rather than adding tuples, so we want repairs with as few additions of tuples as possible. In fact, this strategy creates repairs by deleting tuples. Equivalently, using repairing deltas to model the changes that lead from an inconsistent KB K to a consistent KB K', this algorithm creates repairing deltas which contain only deletions of tuples($\delta = \langle \delta_a, \delta_d \rangle$ with $\delta_a = \emptyset$).

Concluding, we call a repair K' a subset repair of K iff:

- $K' \vDash \mathcal{C}$.
- $K' \subseteq K$. (sub-instance)
- $\nexists K''$ s.t. $K'' \vDash \mathcal{C}$ and $K' \subset K'' \subseteq K$ or $K' = K \bullet \delta'$ and δ' is minimal w.r.t. the deletions it applies (maximal repair or minimal repairing delta).

The term minimal refers to the subset relation. I.e., a repairing delta $\delta = \langle \delta_a, \delta_d \rangle$ will be minimal against a repairing repairing delta $\delta' = \langle \delta'_a, \delta'_d \rangle$ iff $\delta_a \subseteq \delta'_a$ and $\delta_d \subseteq \delta'_d$.
We should mention that it is not clearly mentioned in [1] whether this kind of preference is applied; a) upon repairing deltas per constraint, or b) upon repairing deltas which repair the set of inconsistencies (i.e., information or violation based approach). It seems more intuitively correct to follow an information based approach creating the entire resolution tree and finally check whether the produced repairing deltas lead to subset repairs.

Proposition 10 *Subset repair strategy [1] used in relational KBs defines an information based repair policy $>^I$.*

Proof. We will prove that the the information based repair policy $>^I$ gives the same repairs as preferred with R_{ss} (i.e., $PR^I(K) = R_{ss}(K)$) for all $K \in \mathcal{K}$, where $>$ is a preference based repair order, upon possible repairing deltas defined by the preference $P = Min(f_{additions})$. Preference P denotes that deltas which add the lowest number of tuples are preferred.

Consider a repair $K' \in PR^I(K)$. Then there is some $\delta' \in PRD^I(K)$ s.t. $K' = K \bullet \delta'$. Based on the form of the constraints in $\mathcal{C}$: $\forall \vec{A} \bigwedge_{i=1,\ldots,n} P_i(\vec{A}) \rightarrow \vee_{j=1,\ldots,m} \exists \vec{B}_j Q_j(\vec{A}, \vec{B}_j)$and preference P, we prefer repairing deltas which do not add any tuples to K'. As the body of the constraints in $\mathcal{C}$ is not empty, we will always have the option to delete at least one tuple. Thus, there will always exist some $\delta' = \langle \delta'_a, \delta'_d \rangle$ s.t. $\delta'_a = \emptyset$ and K' will be a subset of K (i.e., $K' \subset K$). Also, for any preferred repairing delta $\delta' = \langle \emptyset, \delta'_d \rangle$, δ'_d is a minimal subset (condition 2 in definition 7) thus, $K' = K \bullet \delta'$ will be maximal subset of K. We notice that K' is a subset repair thus, $K' \in R_{ss}(K)$.

Consider a subset repair $K' \in R_{ss}(K)$. Set $\delta' = \langle K' \setminus K, K \setminus K' \rangle$. Setting $\delta'_a = K' \setminus K$, it holds $\delta'_a = \emptyset$ as $K' \subseteq K$ and $f_{additions}(\delta') = 0$. Moreover:

1. δ' is a repairing delta as $K' \vDash \mathcal{C}$.
2. $\nexists\ \delta''$: δ'' ¿ δ' as $f_{additions}(\delta') = 0$ and $f_{additions}(\delta) \geq 0\ \forall \delta \in \Delta$.
3. Due to Lemma 1, $\forall \delta'' = \langle \delta''_a, \delta''_d \rangle \in \Delta$ such that $K \bullet \delta'' = K'$, it holds that $\delta'_a \subseteq \delta''_a$, $\delta'_d \subseteq \delta''_d$.

Based on the above and Definition 7 we conclude that $\delta' \in PRD^I(K)$ and $K' \in PR^I(K)$. Eventualy, it holds $PR^I(K) = R_{ss}(K)$ for any $K \in \mathcal{K}$. $\square$

Symmetric Difference Repairs

Symmetric Difference (or $\oplus$) repairs are presented in [1]. Given a potentialy inconsistent KB K, this strategy creates repairs K' (i.e., repaired KBs) such that the set of changes which lead from K to K' is minimal. This means that, given a set of constraints $\mathcal{C}$, K' is a $\oplus$-repair of K iff $K' \vDash \mathcal{C}$ and there is no instance K'' s.t.: $K'' \vDash \mathcal{C}$ and $K \oplus K'' \subset K \oplus K'$. Based on the form of constraints in $\mathcal{C}$, there will always exist at least one $\oplus$-repair thus, this strategy creates always at least one repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$R_{sd} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_{sd}(K) = \{K_i | K_i \text{ is a } \oplus\text{-repair of } K\}.$$

Using repairing deltas to model the changes that lead from an inconsistent KB K to a consistent KB K', this strategy prefers repairing deltas which are minimal (per the subset relation). I.e., a repairing delta $\delta = \langle \delta_a, \delta_d \rangle$ will be minimal against a repairing repairing delta $\delta' = \langle \delta'_a, \delta'_d \rangle$ iff $\delta_a \subseteq \delta'_a$ and $\delta_d \subseteq \delta'_d$.

Concluding, we call a repair K' as a $\oplus$-repair of K iff:

- $K' \vDash \mathcal{C}$.
- $\nexists K''$ s.t. $K'' \vDash \mathcal{C}$ and $K \oplus K'' \subset K \oplus K'$ or $K' = K \bullet \delta'$ and δ' is minimal

The basic idea behind this strategy is that we prefer repairing deltas which contain only the changes needed to repair all the violated constraints starting from a potentialy inconsistent KB K, to a consistent KB K'. As such repairing deltas do not contain changes which are "useless" (i.e., they don't repair any violation) we could say that these deltas are called *useful* in our framework. Moreover, all such repairing deltas (and repairs) are equaly prefered. In simple words, this strategy creates the entire resolution tree whose leaves will *all* be $\oplus$-repairs.

Proposition 11 *Symmetric Difference repair strategy [1] used in relational KBs defines an information based repair policy* $>^I$.

Proof. We will prove that the the information based repair policy $>^I$ gives the same repairs as preferred with $R_{sd}(K)$ (i.e., $PR^I(K) = R_{sd}(K)$) for all $K \in \mathcal{K}$, where $>$ is a preference based repair order, upon possible repairing deltas defined by the preference $P : -\epsilon$(i.e., the *empty* preference). Preference P denotes that all possible repairing deltas are equaly prefered.

Consider a repair $K' \in PR^I(K)$. Then there is some $\delta' \in PRD^I(K)$ s.t. $K' = K \bullet \delta'$. Preference P denotes that set $PRD^I(K)$ will contain all possible repairing deltas of K. Any preferred repairing delta δ', will be also also useful due to condition 2 in definition 7. As a result, $K' = K \bullet \delta'$ is a $\oplus$-repair and $K' \in R_{sd}(K)$.

Consider a $\oplus$-repair $K' \in R_{sd}(K)$. It means that there not exists $K'' \vDash \mathcal{C}$ and $K \oplus K'' \subset K \oplus K'$. Equivalently, setting $\delta' = K \oplus K'$, there not exists $\delta'' = K \oplus K''$ such that $\delta'' \subset \delta'$. Thus, δ' is a useful repairing delta (without expressing any preference). So, by the definition 7 and definition of symmetric difference repairs in [1], $\delta' \in PRD^I(K)$ and $K' = K \bullet \delta' \in PR^I(K)$.
Eventualy, it holds $PR^{\mathcal{PBRS}}(K) = RS_{ss}^{\mathcal{SDB}}(K)$ for any $K \in \mathcal{K}$. $\square$

Cardinality Repairs

Cardinality (or C) repairs are presented in [1]. Given a potentialy inconsistent KB K, this strategy creates repairs K' by applying the minimum number of repairing changes(additions or deletions) to K. This means that, given a set of constraints $\mathcal{C}$, K' is a C-repair of K iff $K' \vDash \mathcal{C}$ and there is no instance K'' s.t.: $K'' \vDash \mathcal{C}$ and $|K \oplus K''| < |K \oplus K'|$. Based on the form of constraints in $\mathcal{C}$(i.e. DEDs), there will always exist at least one C-repair thus, this strategy creates always at least one repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$R_c : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_c(K) = \{K_i | K_i \text{ is a } C\text{-repair of } K\}.$$

Using repairing deltas to model the changes that lead from an inconsistent KB K to a consistent KB K' or to show their symmetric difference $K \oplus K'$, this strategy prefers repairing deltas which contain the lowest number of additions and deltions of tuples.
Concluding, we call a repair K' as a C-repair of K iff:

- $K' \vDash \mathcal{C}$.
- $\nexists K''$ s.t. $K'' \vDash \mathcal{C}$ and $|K \oplus K''| < |K \oplus K'|$ or $K' = K \bullet \delta'$ and δ' contains the minimum number of changes.

The basic idea behind this strategy is that we prefer repairing deltas with the lowest size. As a result, this strategy must firstly create the entire resolution tree and then it prefers only those with the minimum size (i.e., information based approach).

Proposition 12 *Cardinality repair (C-repair) strategy [1] used in relational KBs defines an information based repair policy $>^I$.*

Proof. We will prove that the the information based repair policy $>^I$ gives the same repairs as preferred with $R_c(K)$ (i.e., $PR^I(K) = R_c(K)$) for all $K \in \mathcal{K}$, where $>$ is a preference based repair order, upon possible repairing deltas defined by the preference $P = Min(f_{size})$. Preference P denotes that deltas which contain the minimum number of tuples are preferred.

Consider a repair $K' \in PR^I(K)$. Then there is some $\delta' \in PRD^I(K)$ s.t. $K' = K \bullet \delta'$. Based on

the preference P, we prefer repairing deltas which apply the minimum number of changes to the result. Formally, if $\delta' = \langle \delta'_a, \delta'_d \rangle$, then $\nexists \delta'' = \langle \delta''_a, \delta''_d \rangle$ s.t. $K \bullet \delta'' \vDash \mathcal{C}$ and $|\delta''_a| + |\delta''_d| < |\delta'_a| + |\delta'_d|$. Thus, $K' = K \bullet \delta'$ is a C-repair and $K' \in R_c(K)$.

Consider a C-repair $K' \in R_c(K)$. It means that there not exists $K'' \vDash \mathcal{C}$ and $|K \oplus K''| < |K \oplus K'|$. Setting $\delta'' = K \oplus K''$ and $\delta' = K \oplus K'$ it holds that $|\delta''_a| + |\delta''_d| > |\delta'_a| + |\delta'_d|$, $\delta' \in PRD^I(K)$ and $K' = K \bullet \delta' \in PR^I(K)$.
Eventualy, it holds $PR^I(K) = RS_s(K)$ for any $K \in \mathcal{K}$. $\square$

Component Cardinality Repairs

Component Cardinality (or CC) repairs are presented in [1]. Given a potentialy inconsistent KB K, this strategy creates repairs K' considering their characteristic sequence w.r.t. K. As defined in [1], it is the sequence with coordinates the cardinalities $|P^K \oplus P^{K'}|$, as P varies over the relation symbols of the database schema. Given a repair K' of K, this strategy examines the characteristic sequence of $|P^K \oplus P^{K'}|$ and prefers it to have as few elements as possible. This means that, given a set of constraints $\mathcal{C}$, K' is a CC-repair of K iff $K' \vDash \mathcal{C}$ and there is no instance K'' s.t. $|P^K \oplus P^{K''}| < |P^K \oplus P^{K'}|$. As a result, given two repairs K', K'' of K ($K' \vDash \mathcal{C}_{DB}$, $K'' \vDash \mathcal{C}$), we say that $|K \oplus K'| <_{CC} |K \oplus K''|$ iff $|P^K \oplus P^{K'}| < |P^K \oplus P^{K''}|$ (see [1] for a more detailed description). Based on the form of constraints in $\mathcal{C}$(i.e. DEDs), there will always exist at least one CC-repair thus, this strategy creates always at least one repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$R_{cc} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_{cc}(K) = \{K_i | K_i \text{ is a } CC\text{-repair of } K\}.$$

Concluding, we call a repair K' as a CC-repair of K iff:

- $K' \vDash \mathcal{C}$.
- $\nexists K''$ s.t. $K'' \vDash \mathcal{C}$ and $|K \oplus K''| <_{CC} |K \oplus K'|$.

The basic idea behind this strategy is that for each repairing delta we keep how many times each predicate appears either as an added, or as a deleted tuple. Considering all predicates to have the same importance, we prefer deltas which contain the lowest appearance of at least one predicate. If there are more than one repairing deltas where a predicate appears the same times, then if one of them dominates them for another predicate, it is more preferable (i.e., pareto order upon the occuarances of all predicate atoms). As a result, this strategy must firstly create the entire resolution tree and then it prefers only those which have at least one minimal appearance of predicates (i.e., information based approach).

Proposition 13 *Component Cardinality repair (CC-repair) strategy [1] used in relational KBs defines an information based repair policy $>^I$.*

Proof. We will prove that the the information based repair policy $>^I$ gives the same repairs as preferred with $R_{cc}(K)$ (i.e., $PR^I(K) = RS_{cc}(K)$) for all $K \in \mathcal{K}$, where $>$ is a preference based repair order, upon possible repairing deltas defined by the preference P which is defined as the pareto accumulation of preferences $P_i = Min(f^i_{pr\_appearance})$. Feature $f^i_{pr\_appearance}$ is defined upon the set of predicates $\{R_i\}$ and counts the number of appearances of a certain relational $R_i \in \mathcal{R}$ in the delta (i.e., $\mathcal{P}_\rangle = \{R_i\}$). It denotes that deltas which have the least occurrences for all the different predicate atoms of K are preferred.

Consider a repair $K' \in PR^I(K)$. Then there is some $\delta' \in PRD^I(K)$ s.t. $K' = K \bullet \delta'$. Based on the preference P, we prefer repairing deltas which have the least occurrences for all the different predicate atoms of K. Using pareto among the occurrences for each predicate, means that they are all of the same importance for us. This is exactly the characteristic sequence of the repairing delta. Thus, $K' = K \bullet \delta'$ will be a CC-repair and $K' \in R_{cc}(K)$.

Consider a CC-repair $K' \in R_{cc}(K)$. It means that there does not exist K'' s.t. $K'' \models \mathcal{C}$ and $|K \oplus K''| <_{CC} |K \oplus K'|$. Setting $\delta'' = K \oplus K''$ and $\delta' = K \oplus K'$ it holds that $|\delta''| >_{CC} |\delta'|$. Thus, the characteristic sequence of K' will have less size than the characteristic sequence of K''. As a result, at least one predicate atom, will appear in δ' less times than in δ'' and it would be more preferable w.r.t. preference P. So, by definition 7, $\delta' \in PRD^I(K)$ and $K' = K \bullet \delta' \in PR^I(K)$. Eventualy, it holds $PR^I(K) = R_{cc}(K)$ for any $K \in \mathcal{K}$. $\square$

Repair using Null

This repair strategy is presented in [7]. Given a potentialy inconsistent KB K, and a set of constraints $\mathcal{C}$ (i.e., DEDs: $\forall \vec{A} \bigwedge_{i=1,\ldots,n} P_i(\vec{A}) \rightarrow \vee_{j=1,\ldots,m} \exists \vec{B}_j Q_j(\vec{A}, \vec{B}_j)$) this strategy repairs each violated constraint by:

- removing one of the tuples which belong in the body of the constraint (i.e., $P_i(\vec{A})$)
- adding one of the tuples which belong in the head of the constraint and do not have one existentialy quantified attribute(i.e., $Q_i(\vec{x})$)
- adding one of the tuples which belong in the head of the constraint and have at least one existentialy quantified attribute, by using the constant value "$null$" for the existential attributes (i.e., $Q_i(\vec{x}, \vec{y})$, where $\vec{y} = \langle null, null, \ldots, null \rangle$)

We will call a KB K' which is created by this strategy process as RU_{null} repair of K. In order to create repairs of a KB K using this strategy, we follow a recursive process s.t.:

1. find a violated constraint
2. apply upon the current instance of the KB one of the previously defined solutions (based on the constraint's syntactic form)
3. for each applied solution, if K inconsistent then repeat from step 1; else $RU_{null}(K) = K$

The process terminates when we have no more violated constraints (i.e., a consistent KB K'). Based on the form of constraints in $\mathcal{C}$ (i.e., DEDs), every constraint can be corrected with at least one of the previous options, thus this strategy can produce at least one RU_{null} repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$R_{null} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_{null}(K) = \{K_i | K_i \in RU_{null}(K)\}.$$

Taking a closer look at the recursive process followed in this strategy, we notice that it examines each violation constraint separately from the others, repairs it using specific choices, and continues. As a result, a violation based approach is followed, because the approach considers only the preferable resolutions per constraint violation.

Proposition 14 *Given a violation selection function $NextV$, repairing using null strategy [7] used in relational KBs defines a violation based repair policy $>^V$.*

Proof. We will prove that the violation based repair policy $>^V$ gives the same repairs as preferred with $R_{null}(K)$ (i.e., $PR^V(K) = R_{null}(K)$) for all $K \in \mathcal{K}$ where $>$ is a preference based repair order, upon possible repairing deltas, which is defined using the composite preference $P_{null} = Max(f_{deletions}) \otimes (Min(f_{deletions})\ \&\ (P \otimes Max(f_{const\_additions})))$, where $P = Min(f^1_{pr\_additions}) \otimes \ldots \otimes Min(f^n_{pr\_additions})$, where each feature $f^i_{pr\_additions}$(see Appendix B), counts the number of appearances of a certain relational atom R_i in the δ_a set of a delta. (i.e., $\mathcal{P}_{\rangle} = \{R_i\}$). Feature $f_{const\_additions}$ is defined upon the set of constant values $\{null\}$. The composite preference P_{null} denotes that we equaly prefer repairing deltas which remove the highest number of tuples or remove the lowest number of tuples. We should mention that the preferences $(P \otimes Max(f_{const\_additions}))$ which are composed via prioritized denote that for every relation R_i which appears in the head of the violated constraint, we prefer those deltas which add tuples in those relations with as many *null* values as possible. We will show that given a KB K, any repair of K will belong in $PR^V(K)$ iff it belongs in $R_{null}(K)$. Thus, it holds $PR^V = R_{null}(K)$.

Consider a repair $K' \in PR^V(K)$. Then there is some $K_{n_{SEQ}}$ which belongs to a preferred repairing sequence of K w.r.t. the preferred repairing policy $>^V$ and terminates after n_{SEQ} steps. Based on definition 6 and using a violation selection function $NextV(K)$ for each K_{i+1} element of the repairing sequence, it holds $K_{i+1} = K_i \bullet \delta_i$ where δ_i is a preferable repairing delta w.r.t. $>$. Based on the form of the constraints in $\mathcal{C}$: $\forall \vec{A} \bigwedge_{i=1,\ldots,n} P_i(\vec{A}) \rightarrow \vee_{j=1,\ldots,m} \exists \vec{B_j} Q_j(\vec{A}, \vec{B_j})$and preference P_{null}, we prefer repairing deltas δ_i which remove tuples from K_i (i.e., $K_{i+1} \subset K_{i+1}$) or add tuples which a) have no existential quantified attributes, or b) have *null* values in their existentialy quantified attributes. As the body of the constraints in $\mathcal{C}$ is not empty, we will always have the option to delete at least one tuple. Thus, there will always exist some $\delta_i = \langle \delta_{a_i}, \delta_{d_i} \rangle$ s.t. $\delta_{a_i} = \emptyset$ and $\delta_{d_i} = \{P_i(\vec{x})\}$. Also, if the head of the constraint, does not have predicate atom(s) with existential variables then the repairing deltas $\delta_i = \langle \delta_{a_i}, \delta_{d_i} \rangle$ s.t. $\delta_{d_i} = \emptyset$, $\delta_{a_i} = \{P_i(\vec{x})\}$ are equally prefered. Last if the head of the constraint has predicate atom(s) with existential variables then the preferable repairing deltas, have the form $\delta_i = \langle \delta_{a_i}, \delta_{d_i} \rangle$ s.t $\delta_{d_i} = \emptyset$, $\delta_{a_i} = \{Q_i(\vec{x}, \vec{y})\}$ and $\vec{y} = \langle null, null, \ldots, null \rangle$. We notice that we prefer the reparing deltas which do exactly the same repairing changes with the repairing deltas greedily selected in step 2 of the process used in repairing using null strategy. Moreover, as this strategy uses the violation selection function $NextV$, then we can create a $K_{n_{SEQ}}$ repair which will be a RU_{null} repair.

Consider a RU_{null} repair $K' \in R_{null}(K)$. As described, there exists a recursive process which starts from an initial KB K, follows the steps 1-3 and leads to the creation of a K'. Based on this strategy, for each violated constraint, the repairing option which removes tuples from the head of the constraint, corresponds to a preference of highest deletions of tuples. Based on the form of DEDs, the deltas which apply deletion will have the form: $\langle \emptyset, \{P_i(\vec{x})\} \rangle$. The repairing choice which add tuples with as many null values in the position of the existentialy quantified values, for each relational R_i which appears in the head of the constraint must be expressed using a preference which counts the appearance of each R_i along with the appearance of the *null* values it can take. As a result, each call of the recursive process (which repairs a single constraint instance) will prefer the same repairing deltas with the ordering $>$ in violation based repairing policy $>^V$. Starting from the same initial KB K, using the violation selection function $NextV(K)$ in step 1 and preferring the same repairing deltas per constraint the RU_{null} repair of K will belong in $PR^V(K)$, as we will be able to create a preferred repairing sequence of K.

Eventualy, it holds $PR^V(K) = R_{null}(K)$ for any $K' \in \mathcal{K}$. $\square$

Repair Inclusion Dependencies using Null

This repair strategy is presented in [6]. They also use a repair strategy for the repair of Functional Dependencies but as they use replacements, it is not emulated within our framework. Given a potentialy inconsistent KB K, and a set of constraints $\mathcal{C}$ (i.e., IND: $\forall \vec{A} P(\vec{A}) \rightarrow \exists \vec{C} Q(\vec{B}, \vec{C})$, $\vec{B} \subseteq \vec{A}$) this strategy repairs each violated constraint by inserting the tuple which belongs in the head of the constraint. Moreover, the special value *null* is given to its existentialy quantified attributes.

We will call a KB K' which is created by this strategy process as $RIND_{null}$ repair of K. In order to create repairs of a KB K using this strategy, we follow a recursive process s.t.:

1. find a violated constraint
2. apply upon the current instance of the KB the previously defined solution
3. for each applied solution, if K inconsistent then repeat from step 1; else $RIND_{null}(K) = K$

The process terminates when we have no more violated constraints (i.e., a consistent KB K'). Based on the form of constraints in $\mathcal{C}$ (i.e., INDs), every constraint can be corrected with at least one of the previous options, thus this strategy can produce at least one $RIND_{null}$ repair. It is clear that we can define a repair finding algorithm upon this policy as:

$$RI_{null} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } RI_{null}(K) = \{K_i | K_i \in RIND_{null}(K)\}.$$

Taking a closer look at the recursive process followed in this strategy, we notice that it examines each violation constraint separately from the others, repairs it using a specific choice, and continues. As a result, a violation based approach is followed, because the approach considers only the preferable resolution per constraint violation.

Proposition 15 *Given a violation selection function $NextV$, repairing IND using null strategy [6] used in relational KBs defines a violation based repair policy $>^V$.*

Proof. We will prove that the violation based repair policy $>^V$ gives the same repairs as preferred with $RI_{null}(K)$ (i.e., $PR^V(K) = RI_{null}(K)$) for all $K \in \mathcal{K}$ where $>$ is a preference based repair order, upon possible repairing deltas, which is defined using the preference $Max(f_{const\_additions})))$, where $f_{const\_additions}$ is defined upon the set of constant values $\{null\}$. This preference denotes that we prefer the repairing deltas wich add tuples with as many *null* values as possible. We will show that given a KB K, any repair of K will belong in $PR^V(K)$ iff it belongs in $RI_{null}(K)$. Thus, it holds $PR^V = RI_{null}(K)$.

Consider a repair $K' \in PR^V(K)$. Then there is some $K_{n_{SEQ}}$ which belongs to a preferred repairing sequence of K w.r.t. the preferred repairing policy $>^V$ and terminates after n_{SEQ} steps. Based on definition 6 and using a violation selection function $NextV(K)$ for each K_{i+1} element of the repairing sequence, it holds $K_{i+1} = K_i \bullet \delta_i$ where δ_i is a preferable repairing delta w.r.t. $>$. Based on the form of the constraints in $\mathcal{C}$: $\forall \vec{A} P(\vec{A}) \rightarrow \exists \vec{C} Q(\vec{B}, \vec{C})$, $\vec{B} \subseteq \vec{A}$ and the expressed preference, we prefer a single repairing delta δ which adds the tuple with *null* values in its existentialy quantified attributes. This deltas has the form $\delta = \langle \delta_a, \delta_d \rangle$ s.t $\delta_d = \emptyset$, $\delta_a = \{Q(\vec{x}, \vec{y})\}$ and $\vec{y} = \langle null, null, \ldots, null \rangle$. We notice that we prefer the reparing delta which

does exactly the same repairing changes with the repairing delta greedily selected in step 2 of the process used in repairing inclusion dependencies using null strategy. Moreover, as this strategy uses the violation selection function $NextV$, then we can create a $K_{n_{SEQ}}$ repair which will be a $RIND_{null}$ repair.

Consider a $RIND_{null}$ repair $K' \in RI_{null}(K)$. As described, there exists a recursive process which starts from an initial KB K, follows the steps 1-3 and leads to the creation of a K'. Based on this strategy, for each violated constraint, the repairing choice which adds the tuple tuples with as many null values in the position of the existentialy quantified values, obviously corresponds to the preference which denotes the addition of tuples with the maximum number of attributes which take null values. As a result, each call of the recursive process (which repairs a single constraint instance) will prefer the same repairing deltas with the ordering $>$ in violation based repairing policy $>^V$. Starting from the same initial KB K, using the violation selection function $NextV(K)$ in step 1 and preferring the same repairing deltas per constraint the $RIND_{null}$ repair of K will belong in $PR^V(K)$, as we will be able to create a preferred repairing sequence of K.
Eventualy, it holds $PR^V(K) = RI_{null}(K)$ for any $K' \in \mathcal{K}$. $\square$

A.4.2 Repair strategy used in RONDO Framework

RONDO is a framework which, among others, supports the merging operator upon KBs in the context of relational schemas, XML schemas, and SQL views. The merging operation could create a potentialy inconsistent result. Thus, RONDO defines a repair finding algorithm R over a specific set of constraints which allows it to repair the invalidities that appear in the merged schema.

Knowledge contained in schemas, which are supported in RONDO framework, is encoded using directed labeled graphs (V, A). There are two types of nodes in the graph and each type denotes whether the node refers to an attribute of a relation, or a relation itself. More specificaly, having nodes with no incoming edges, we understand that they refer to relations, else they refer to attributes. Furthermore, knowledge is presented using "IsA" hierarchies and as a result, the graph has the form of a dag (directed acyclic graph). Every arc has a label which can take a value $l \in L = \{00, 0+, 0-, +0, ++, +-, -0, -+, --\}$. We formalize this representation method using a single predicate namely $SuccOf = \{f, c, l\}$ where a) $SuccOf[1] \in V$, $SuccOf[2] \in V$, and $SuccOf[3] \in L$.

Based on the above, we can define a mapping between a labeled graph in RONDO, with a relational KB as: Consider a labeled graph $G = (V, A)$ in RONDO framework and the corresponding relational KB $K \subseteq L$. It holds: $\forall e \in Aedge$ from v_1 to v_2, labeled with a value $l \in L$, $\exists$ tuple $SuccOf(v_1, v_2, l) \in K$.

The set $\mathcal{C}$ of integrity constraints which must be satisfied contains only one constraint which has the form:

$$\forall f, c, c', l, l' SuccOf(f, c, l) \wedge SuccOf(f, c', l') \rightarrow c = c'$$

which means that we allow edges starting from the same node, but not edges which lead to the same node. Equivalently, each child, can have only one father. We notice that the validity constraint used in this model, is a DED constraint and more specificaly it denotes a functional dependency (FD).

This repair strategy is presented in [29]. Consider a potentialy inconsistent KB K, which represents a labeled directed graph in RONDO framework, the set of constraints $\mathcal{C}$ and the tuple

$\vec{P_{RF}} = \langle --, -+, +-, ++, -0, +0, 0-, 0+, 00 \rangle$. For simplicity, we will say $P_{RF}[i]$ in order to get the label which is in position i of the tuple. We define an ordering $>$ over labels considering their respective position in $\vec{P_{RF}}$ as:
$\forall l_1, l_2 \in L : l_1 > l_2$ iff $\exists i, j$ s.t. $P_{RF}[i] = l_1$, $P_{RF}[j] = l_2$ and $i > j$.

Consider a constraint violation for the instance: $r(\langle f_1, c_1, c_2, l_1, l_2 \rangle)$. The constraint can be repaired by either a) removing the tuple $SuccOf(f_1, c_1, l_1)$ or b) removing the tuple $SuccOf(f_1, c_2, l_2)$. This strategy selects which tuple to remove by examining the label of each tuple. Here

- $SuccOf(f_1, c_1, l_1)$ will be removed iff $l_1 < l_2$.
- $SuccOf(f_1, c_2, l_2)$ will be removed iff $l_2 < l_1$.

We will call a KB K' which is created by this strategy as RU_{rondo} repair of K.In order to create the repairs of a KB K using this strategy, we follow a recursive process as follows:

1. find a violated constraint.
2. remove the tuple using the above selection mechanism to correct the violation.
3. for each applied solution, if K inconsistent then repeat from step 1; else $RU_{rondo}(K) = K$.

The process terminates when we have no more violated constraints (i.e., a consistent KB K'). Based on the form of constraints in $\mathcal{C}$ (i.e., DEDs), every constraint can be corrected with at l east one of the previous options, thus this strategy can produce at least one RU_{rondo} repair. As a result, we can define a repair strategy over the previously defined setting about relational databases as:

$$R_{rondo} : \mathcal{K} \mapsto 2^{\mathcal{K}} \setminus \emptyset \text{ and } R_{rondo}(K) = \{K_i | K_i \text{ is a } RU_{rondo} \text{ repair of } K\}.$$

Taking a closer look at the recursive process followed in this strategy, we notice that it examines each violation constraint separately from the others, repairs it using specific choices, and continues. As a result, a violation based approach is followed, because the approach considers only the preferable resolutions per constraint violation.

Proposition 16 *Given a violation selection function $NextV$, the repair strategy used in RONDO framework [29], defines a violation based repair policy $>^V$.*

Proof. We will prove that the the violation based repair policy $>^V$ gives the same repairs as preferred with $R_{rondo}(K)$ (i.e., $PR^V(K) = R_{rondo}(K)$) for all $K \in \mathcal{K}$, where $>$ is a preference based repair order, upon possible repairing deltas, which is defined as the prioritized accumulation of preferences:

- $P_1 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{oo\}$.
- $P_2 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{o+\}$.
- $P_3 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{o-\}$.
- $P_4 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{+o\}$.
- $P_5 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{-o\}$.
- $P_6 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{++\}$.

- $P_7 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{+-\}$.
- $P_8 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{-+\}$.
- $P_9 = Max(f_{const\_deletions})$, where feature $f_{const\_deletions}$ is defined over the set $\mathcal{CV} = \{--\}$.

It holds $P : -P_9 \ \& \ P_8 \ \& \ P_7 \ \& \ \ldots \& \ P_1$ and if we create the corresponding tuple $\vec{P_{ac}}$ w.r.t. the prioritized accumulation we have $\vec{P} = \langle --, -+, +-, ++, -o, +o, o-, o+, oo \rangle$ where $P_{ac}[i]$ denotes the label which is in position i. Preference P denotes that given two tuples $SuccOf(f, c, l)$, $SuccOf(f', c', l')$, we are based on the next ordering over labels, to decide which tuple to remove:
$\forall l, l' \in L : l > l'$ iff $\exists i, j$ s.t. $P_{ac}[i] = l$, $P_{RF}[j] = l'$ and $i < j$.
In this point we should mention that preference P, defines the same ordering over labels as the ordering created in RONDO framework. As a result,

- $SuccOf(f, c, l)$ will be removed iff $l > l'$.
- $SuccOf(f', c', l')$ will be removed iff $l' > l$.

Next, we will show that given a KB K, any repair of K will belong in $PR^V(K)$ iff it belongs in $R_{rondo}(K)$. Thus, it holds $PR^V(K) = R_{rondo}(K)$.

Consider a repair $K' \in PR^V(K)$. Then there is some $K_{n_{SEQ}}$ which belongs to a preferred repairing sequence of K w.r.t. the preferred repairing policy $>^V$ and terminates after n_{SEQ} steps. Based on definition 6 and using a violation selection function $NextV(K)$ for each K_{i+1} element of the repairing sequence, it holds $K_{i+1} = K_i \bullet \delta_i$ where δ_i is a preferable repairing delta w.r.t. $>$. Consider a constraint instance violation $c(\langle f_1, c_1, c_2, l_1, l_2 \rangle)$. The possible repairing deltas (which belong in $Res(r(\langle f_1, c_1, c_2, l_1, l_2 \rangle), K)$ are $\delta_1 = \langle \emptyset, \{SuccOf(f_1, c_1, l_1)\} \rangle$ and $\delta_2 = \langle \emptyset, \{SuccOf(f_1, c_2, l_2)\} \rangle$. Based on preference P, $\delta_1 > \delta_2$ iff $l_2 > l_1$. We notice that we prefer the reparing deltas which do exactly the same repairing changes with the repairing deltas greedily selected in step 2 of the process which is used in RONDO's repair strategy. Moreover, as this strategy uses the violation selection function $NextV$, then we can create a $K_{n_{SEQ}}$ repair which will be a RU_{rondo} repair.

Consider a RU_{rondo} repair $K' \in R_{rondo}(K)$. As described, there exists a recursive process which starts from an initial KB K, follows the steps 1-3 and leads to the creation of a K'. Based on this strategy, for each violated constraint instance $r(\langle f_1, c_1, c_2, l_1, l_2 \rangle)$ and among its repairing solutions which denote to a) remove $SuccOf(f_1, c_1, l_1)$ or b) remove $SuccOf(f_1, c_2, l_2)$, we prefer to remove the tuple whose label's value belongs in the minimum index of P_{RF}. This preference upon labels and their positions corresponds to a prioritized (as P_{RF} is ordered) accumulation of preferences, where each preference P_i denotes the removal of tuples with the maximum number of attributes which take the value of a label $P_{RF}[i]$. We notice that this prioritized accumulation is equivalent to preference P we previoudly defined. As a result, each call of the recursive process (which repairs a single constraint instance) will prefer the same repairing deltas with the ordering $>$ denoted by the policy $>^V$. Starting from the same initial KB K, using violation selection function $NextV(K)$ in step 1 and preferring the same repairing deltas per constraint the RU_{rondo} repair of K will belong in $PR^{\mathcal{PBRS}}$, as we will be able to create a preferred repairing sequence of K.
Eventualy, it holds $PR^V = R_{rondo}(K)$ for any $K' \in \mathcal{K}$. $\square$

A.5 Proofs for emulating other Prevention Strategies

Before showing the connection between the evolution approach of [25] and our work, we will recast the theory of [25] using our terminology. In [25], a language L is used, which is the same as in our case, and a set of validity constraints $\mathcal{C}$, which are DEDs (as in this work). A KB K is a set containing positive ground facts, i.e., $K \in \mathcal{K}$, as in our case. An update U is a set of positive and negative ground facts, where the positive ground facts correspond to additions, and the negative ones correspond to deletions. Equivalently, we will assume that an update is a delta (i.e., $U = \langle U_a, U_d \rangle \in \Delta$), where U_a corresponds to the additions (positive ground facts) and U_d corresponds to the deletions (negative ground facts, which are turned into positive ones before being included in U_d). In [25], an operation called "raw application" is defined, and denoted by $+$; this operation corresponds to the application operation in our terminology, i.e., $\bullet$. The evolution operator defined in [25] (denoted by $\bullet$ there), will be denoted using the symbol $\circ$, to avoid confusion. Note that $\circ$ returns a single KB, not a set of KBs. The operation $\circ$ is based on a certain "change-generating ordering", i.e., an ordering upon updates which satisfies certain properties (see [25]).

The theorem below will show that the evolution operator $\circ$ is indeed an IB prevention strategy. However, this is true only when the original KB is consistent to begin with, and the update is *feasible* [25], i.e., it does not pose conflicting requirements upon the evolution operator (e.g., explicitly or implicitly requesting the addition and deletion of the same fact). This analysis leads to the following theorem:

Proposition 17 *Consider a language L and a set of DED constraints $\mathcal{C}$, as in [25]. Consider also a change-generating order $>_C$ and the corresponding evolution operator $\circ$. Set $D = \{\langle K, U \rangle | K \vDash \mathcal{C}, U \in \Delta,\ U : \textit{feasible}\ \}$ and $\mathcal{O} : D \mapsto \mathcal{K}$ such that $\mathcal{O}(\langle K, U \rangle) = K \bullet U$. Consider the extended setting $\mathcal{GS} = \langle L, \mathcal{C}, \mathcal{O} \rangle$. Then the function $PS^{\mathcal{GS}} : D \mapsto 2^{\mathcal{K}} \setminus \emptyset$, such that $PS^{\mathcal{GS}}(\langle K, U \rangle) = \{K \circ U\}$ is an IB prevention strategy.*

Proof. By the definition of $\circ$ it follows that an algorithm P is well-defined. Furthermore, by the Principle of Validity, it follows that for all $w \in D$, $K \in P(w)$ it holds that $K \vDash \mathcal{C}$. Therefore, P is a prevention algorithm.
To show that P is an IB prevention algorithm, we will use Proposition 2. So, take some $w = \langle K, U \rangle \in D$ such that $U = \langle U_a, U_d \rangle$. Set $K_U = \mathcal{O}(K, U) = K \bullet U = (K \cup U_a) \setminus U_d$. Take also some $K_r \in P(w)$, $K'_r \in \mathcal{K}$ for which $K'_r \vDash \mathcal{C}$ and $K'_r \setminus K_U \subseteq K_r \setminus K_U$, $K_U \setminus K'_r \subseteq K_U \setminus K_r$. We need to show that $K'_r = K_r$.
By our hypotheses, $\langle K, U \rangle \in D$, U is feasible, so $U_a \cap U_d = \emptyset$. Also, $K_r \in P(w)$, so $K_r = K \circ U$, so, by the definition of $\circ$, $U_a \subseteq K_r$, $U_d \cap K_r = \emptyset$.
Based on the above facts, we will show the following:

1. $K'_r \setminus K \subseteq K_r \setminus K$
   Consider some ground fact $x \in K'_r \setminus K$. Then $x \in K'_r, x \notin K$. Assuming that $x \in U_a$, then $x \in K_r$, so $x \in K_r \setminus K$. Assuming that $x \notin U_a$, then $x \notin K_U$, so $x \in K'_r \setminus K_U \subseteq K_r \setminus K_U$, so $x \in K_r$, so $x \in K_r \setminus K$. Thus, in any case, $x \in K_r \setminus K$, so $K'_r \setminus K \subseteq K_r \setminus K$.

2. $K \setminus K'_r \subseteq K \setminus K_r$
   Consider some ground fact $x \in K \setminus K'_r$. Then $x \in K, x \notin K'_r$. Assuming that $x \in U_d$, then by the fact that $K_r \cap U_d = \emptyset$ it follows that $x \notin K_r$, so $x \in K \setminus K_r$. Assuming that $x \notin U_d$, then $x \in K \bullet U = K_U$, so $x \in K_U \setminus K'_r \subseteq K_U \setminus K_r$, so $x \notin K_r$, so $x \in K \setminus K_r$. Thus, in any case, $x \in K \setminus K_r$, so $K \setminus K'_r \subseteq K \setminus K_r$.

3. $U_a \subseteq K'_r$
   Consider some ground fact $x \in U_a$. Then $x \in K_r$ and $x \in K \bullet U = K_U$. Thus, $x \notin K_U \setminus K_r \supseteq K_U \setminus K'_r$, so $x \notin K_U \setminus K'_r$. Given that $x \in K_U$, it follows that $x \in K'_r$, so $U_a \subseteq K'_r$.

4. $U_d \cap K'_r = \emptyset$
   Consider some ground fact $x \in U_d$. Then $x \notin K_r$ and $x \notin K \bullet U = K_U$. Thus, $x \notin K_r \setminus K_U \supseteq K'_r \setminus K_U$, so $x \notin K'_r \setminus K_U$. Given that $x \notin K_U$ it follows that $x \notin K'_r$, so $U_d \cap K'_r = \emptyset$.

Using the monotonicity property of $>_C$ (see [25]) and the facts #1, #2 above, it follows that $\langle K_r \setminus K, K \setminus K_r \rangle \leq_C \langle K'_r \setminus K, K \setminus K'_r \rangle$. Moreover, by our hypotheses, the facts #3, #4 above and the Principle of Minimal Change that is satisfied by $\circ$, it follows that $\langle K'_r \setminus K, K \setminus K'_r \rangle \leq_C \langle K_r \setminus K, K \setminus K_r \rangle$. The above two facts, combined with the delta antisymmetry property of $>_C$ (see [25]) implies that $\langle K_r \setminus K, K \setminus K_r \rangle = \langle K'_r \setminus K, K \setminus K'_r \rangle$, i.e., $K_r \setminus K = K'_r \setminus K$, $K \setminus K_r = K'_r \setminus K$. Thus, $K_r = K'_r$ and the proof is complete. $\square$

Appendix B

Features

In this part, we discriminate three categories of features; a) generic features which consider a delta quantitively w.r.t. the number of ground facts it contains, b) features which examine the appearance of specific predicates in the delta, c) features which examine the appearance of specific values in ground facts contained in the delta.

Definition 11 *Consider a KB K and a $\delta = \langle \delta_a, \delta_d \rangle$. We define:*

- $f_{additions}(\delta) = |\delta_a|$. *It denotes the number of additions.*
- $f_{deletions}(\delta) = |\delta_d|$. *It denotes the number of deletions.*
- $f_{size}(\delta) = f_{additions}(\delta) + f_{deletions}(\delta)$. *It denotes the size of a δ (i.e., the number of additions plus the number of deletions).*

Note. For example if $K = \{B_1(a),\ B_2(a,b),\ B_3(c)\}$ and $\delta = \langle \{H_1(a,b),\ H_2(b,d)\}, \{B_1(a),\ B_2(a,b),\ B_3(c)\}, \rangle$, $f_{additions}(\delta) = 3$ $f_{deletions}(\delta) = 4$ and $f_{size}(\delta) = 7$.

Definition 12 *Given a set PGF of positive ground facts and a set $\mathcal{P}$ of predicate atoms. We define an* operation $\sqcap$ *upon these two sets such that: $PGF \sqcap P = \{Q(\vec{x}) | Q \in P$ and $Q(\vec{x}) \in PGFs\}$.*

Definition 13 *Consider a KB K, a $\delta = \langle \delta_a, \delta_d \rangle$ and a set of predicates $\mathcal{P}$. We define:*

- $f_{pr\_additions}(\delta) = |\delta_a \sqcap \mathcal{P}|$. *It denotes the number of additions of ground facts which have a predicate from set $\mathcal{P}$.*
- $f_{pr\_deletions}(\delta) = |\delta_d \sqcap \mathcal{P}|$. *It denotes the number of deletions of ground facts which have a predicate from set $\mathcal{P}$.*
- $f_{pr\_appearance}(\delta) = f_{pr\_additions}(\delta) + f_{pr\_deletions}(\delta)$. *It denotes the number of ground facts appearing in the delta which have a predicate from set $\mathcal{P}$.*

Note. For example if $K = \{B_1(a),\ B_2(a,b),\ B_3(c)\}$ and $\delta = \langle \{H_1(a,b),\ H_2(b,d)\}, \{B_1(a),\ B_2(a,b),\ B_3(c)\} \rangle$ and $\mathcal{P} = B_1, B_3$, $f_{pr\_additions}(\delta) = 0$, $f_{pr\_deletions}(\delta) = 2$ and $f_{pr\_appearance}(\delta) = 2$.

Definition 14 *Consider a vector $\vec{x}$ (e.g., $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$) and a set $\mathcal{S}$ of constant values. We define the function $Occurences : X \times \mathcal{S} \mapsto \mathbb{N}$, where X is the set of all vectors and $\mathbb{N}$ the set of naturals. It holds $Occurences(\vec{x}, \mathcal{S}) = |\{x_i | x_i \in \mathcal{S}\}|$.*

Definition 15 *Consider a KB K, a $\delta = \langle \delta_a, \delta_d \rangle$ and a set of constant values $\mathcal{CV}$. We define:*

- $f_{const\_additions}(\delta) = \sum_{H(\vec{x}) \in \delta_a} Occurences(\vec{x}, \mathcal{CV})$.
- $f_{const\_deletions}(\delta) = \sum_{B(\vec{x}) \in \delta_a} Occurences(\vec{x}, \mathcal{CV})$.
- $f_{const\_appearance}(\delta) = f_{const\_additions} + f_{const\_deletions}$

Note. For example, consider a KB $K = \{B(a)\}$, a set of values $\mathcal{CV} = \{a, b\}$ and a $\delta = \langle \{H(a, a)\}, \emptyset \rangle$. Then, $f_{const\_additions}(\delta) = 2$, $f_{const\_deletions}(\delta) = 0$, $f_{const\_appearance}(\delta) = 0$.

Appendix C

Algorithms

In this chapter, we present the optimized algorithms called IB_{prun}, IB_{opt}, VB_{opt}. We also present the algorihtms for all the used assisting functions.

Algorithm 4: $isConflicting([[\delta_{tot}]])$

```
A = [[δ_a]]
D = [[δ_d]]
if A ∩ D ≠ ∅ then
    return true
else
    return false
end if
```

Algorithm 5: $clean([[\delta_{tot}]])$

```
set D = ∅
for all δ ∈ [[δ_tot]]^μ do
    if isConflicting(δ) = true then
        D = D ∪ {δ}
    end if
end for
[[δ_tot]]^μ = [[δ_tot]]^μ \ D
```

C.1 Optimizations for IB Algorithm

Function *nodeCheck* (Algorithm 6) checks if the subtree under a current node will lead to non useful deltas with respect to Proposition 7. First of all, we must check if the current delta $[[\delta_{cur}]]$ contains at least one delta which was not selected for the resolution of a previous violation(line 4). To achieve this, we use a map called *RepOpt* in which we store, for each violation resolution, the delta which was selected along with the rest deltas of the resolution set (line 9 in IB_{opt} algorithm). Next we examine the alternative resolutions, $[[\delta_r]]$ of the constraints whose selected repairing deltas created the current delta $[[\delta_{tot}]]$. If delta $[[\delta_{tot}]]$ contains one of those, then we find the appropriate selected $[[\delta]]$, $[[\delta_{ch}]]$ using map *RepOpt*. If one of the deltas $[[\delta_r]]$ is not

examined yet (i.e., its position in resolution set is greater than the position of $[[\delta_{ch}]]$), then based on Proposition 7, it will lead to non usefull deltas and function returns true(case 1 in figure 3.2). However, we should mention here that we should store the pair $([[\delta_{ch}]], [[\delta_r)]]$, because it is possible in another branch to see the opposite pair as in case 2 of figure 3.2. In that case we don't want to prune that branch. The final is ensured by the use of map $Pairs$.

Algorithm 6: $nodeCheck(RepOpt, Pairs, [[\delta_{tot}]])$

```
found = false
for all ([[δ_ch]], Δ_rem) ∈ RepOpt do
   for all [[δ_r]] ∈ Δ_rem do
      if [[δ_r]] ⊆ [[δ_tot]] then
         if pos(δ_r) > pos(δ_ch) then
            Pairs = Pairs ∪ {([[δ_ch]], [[δ_r)]]}
            return true
         else
            if ([[δ_r]], [[δ_ch]]) ∉ Pairs then
               return true
            end if
         end if
      end if
   end for
end for
return false
```

In function $leafCheck$ (Algorithm 7), we follow a similar process. The purpose of this function is to determine whether we have a certain useful delta, or a potentialy useful (i.e., one extra check is required in IB_{opt} algorithm). When we reach a valid KB (i.e., a leaf node), we examine the *non selected* repairing deltas for the constraints which where repaired in the running branch. For each of those deltas, we check the fifth condition of Proposition 6 which is translated by the implication test of line 6. If the latter condition holds, then we must check again the map $Pairs$ for the existence of the reversed pair (as in $nodeCheck$ function) because if such pair exist, then the repairing delta $[[\delta_{tot}]]$ is a potential useful delta. In this case, after the insertion of $[[\delta_{tot}]]$ into RD (line 21 in Algorithm 13) we must perform the check to ensure that RD contains only useful deltas (line 23). Otherwise, this check is avoided.

For the IB_{prun} algorithm, we created two assisting functions; $leafCheck$ performs the aforementioned checks, described in Proposition 6, at the level of leaves and $nodeCheck$ which performs the aforementioned checks, described in Proposition 7, at the level of internal nodes. Function $nodeCheck$(aglorithm 6) is called in line 10 of the IB_{prun} algorithm and is used to determine whether a current node will lead to a subtree with leaves which respond to non useful deltas, thus it can be pruned. It takes as input a set (i.e., $RepOpt$) which contains, for each node, the delta which was selected along with the rest deltas (from the resolution set) which was not selected. It also takes as input a set of delta pairs (i.e., $Pairs$) which is used to avoid the pruning of branches of case 2 in figure 3.2. Function $leafCheck$(aglorithm 7) is called in line 20 of the IB_{prun} and is used to determine which created repairing deltas are certain useful deltas so we can avoid the respective check which was applied for every produced repairing delta in the initial IB algorithm. The input of these function are the sets $RepOpt$, $Pairs$ and the current KB $[[K_c]]$ which is valid. The latter argument is required for the last condition of Proposition 6,

Algorithm 7: $leafCheck(RepOpt, Pairs, [[K_c]])$

```
for all ([[δ_ch]], Δ_rem) ∈ RepOpt do
    if found = true then
        break
    end if
    for all [[δ_r]] ∈ Δ_rem do
        if K_c ⊨ [[δ_r]] then
            if ([[δ_r]], [[δ_ch]]) ∈ Pairs then
                return true
            end if
        end if
    end for
end for
return false
```

Algorithm 8: $IB_{prun}([[K_c]], [[\delta_{tot}]])$

```
if [[K_c]] ⊭ C then
    Find Res(c(A⃗), [[K_c]]) for a violated constraint instance c(A⃗)
    for all [[δ]] ∈ Res(c(A⃗), [[K_c]]) do
        [[δ_tmp]] = [[δ_tot]]
        [[δ_tot]] = [[δ_tot]] ⊎ [[δ]]
        clean([[δ_tot]])
        if isConflicting([[δ_tot]]) = false then
            Δ_rem = Res(c(A⃗), [[K_c]]) \ [[δ]]
            RepOpt = RepOpt ∪ {([[δ]], Δ_rem)}
            found = nodeCheck(RepOpt, Pairs)
            if found = false then
                IB([[K_c]] • [[δ]], [[δ_tot]])
            end if
        end if
        if isConflicting([[δ_tot]]) = true or found = true then
            [[δ_tot]] = [[δ_tmp]]
        end if
    end for
else
    potUs = leafCheck(RepOpt, Pairs, [[K_c]])
    RD = RD ∪ {[[δ_tot]]}
    if potUs = true then
        RD = RD \ {[[δ]] | [[δ]] is a non useful repairing delta}
    end if
end if
```

where the subset check $\delta_L \subseteq \delta'_L$ is transformed to $K \bullet \delta'_L \vDash \delta_L$ and $[[K_c]] = [[K]] \bullet [[\delta_L]]$.

C.2 General Purpose Optimizations

Algorithm 9: $repair_{opt}(>^P, [[K]])$

1: $Errors = \langle c_1(\vec{A}),\ c_2(\vec{A}),\ \ldots,\ c_n(\vec{A}\rangle,\ [[K_c]] \nvDash c_i(\vec{A})$
2: **if** $P = I$ **then**
3: $\quad RD = \emptyset$
4: $\quad IB_{opt}([[K]], \langle\emptyset, \emptyset\rangle, c_1(\vec{A}))$
5: $\quad PR = \{[[K]] \bullet [[\delta]] \mid [[\delta]]$: preferred (per $>^I$) deltas in $RD\}$
6: **else**
7: $\quad PR = \emptyset$
8: $\quad VB_{opt}([[K]], c_1(\vec{A}))$
9: **end if**
10: **return** PR

Function $fetchNextError$ (Algorithm 10) returns the next violated constraint instance from tuple $Errors$. Notice that the condition in line 2 is used to ensure that violations which are potentialy repaired are ingored (due to interdependencies).

Algorithm 10: $fetchNextError(c_i(\vec{A}), [[K_c]])$

1: **for all** $(c_{i+1}(\vec{A}) \in Errors$ **do**
2: $\quad$ **if** $[[K_c]] \nvDash c_{i+1}(\vec{A})$ **then**
3: $\quad\quad$ **return** $c_{i+1}(\vec{A})$
4: $\quad$ **end if**
5: **end for**
6: **return** $null$

Algorithm 11: $findNextError([[K_c]], \mathcal{C}, [[\delta]])$

1: $Viols = \langle\rangle$
2: **for all** $Q(\vec{A}) \in \delta_a$ **do**
3: $\quad$ **if** $\exists c_i \in \mathcal{C},\ c_i(\vec{A})$ s.t. $c_i(\vec{A}) = P(\vec{A}) \rightarrow Q(\vec{A})$ and $P(\vec{A}) \notin [[K_c]]$ **then**
4: $\quad\quad Viols = Viols \parallel \langle c_i(\vec{A})\rangle$
5: $\quad$ **end if**
6: **end for**
7: **for all** $P(\vec{A}) \in \delta_d$ **do**
8: $\quad$ **if** $\exists c_i \in \mathcal{C},\ c_i(\vec{A})$ s.t. $c_i(\vec{A}) = P(\vec{A}) \rightarrow Q(\vec{A})$ and $Q(\vec{A}) \in [[K_c]]$ **then**
9: $\quad\quad Viols = Viols \parallel \langle c_i(\vec{A})\rangle$
10: $\quad$ **end if**
11: **end for**
12: **return** $Viols$

Function $findNextError$ (Algorithm 11) examines only the constraints which may be violated by taking into account the changes a repairing delta enforced (lines 3, 8). If a change

violates one or more constraints (condition in lines 3, 8 is true) then the corresponding violated instances are appended into the tuple Viols (lines 4, 9) which in turn is returned by the function. Thus, the required input for this function is the current KB, the set of constraints and a repairing delta. As we mentioned, the output of this function is a tuple with the violated constraint instances.

The optimized algorithms IB_{opt}, VB_{opt} (Algorithms 13, 12) take as an extra input a violated constraint instance(taken from $Errors$). We notice that both algorithms iterate over the violated constraints recursively using function $fetchNextErrors$ leading to a significant reduction of diagnosis calls. However there are two main points we should take into account using this repair process. The first is that, due to interdependencies, one resolution may repair other violations as well. For that purpose, function $fetchNextError$ (Algorithm 10) ensures that we will not repair a violation which was repaired. The second is that a repair solution may introduce other inconsistencies, thus we have to diagnose the KB after the resolution of all the inconsistencies found in $Errors$ tuple. At this point we use function $findNextError$ (Algorithm 11) which examines only the constraints which may be violated by the enforced changes by running repairing delta. As a result, re-diagnosing the KB is much faster in the average case. If such inconsistencies are found, they are stored in $NewErrors$ tuple (line 15 in Algorithm 13, line 11 in Algorithm 12) updating tuple $Errors$. Next, the above process is repeated recursively, until we have no more inconsistencies to resolve.

Algorithm 12: $VB_{opt}([[K_c]], c_i(\vec{A})$

1: Find $Res(c_i(\vec{A}), [[K_c]])$
2: $RD = \{$preferred (per $>$) deltas in $Res(c(\vec{A}), [[K_c]])\}$
3: **for all** $[[\delta]]$ in RD **do**
4: $\quad [[K_c]] = [[K_c]] \bullet [[\delta]]$
5: $\quad$ **if** $K_c \notin KB$ **then**
6: $\quad\quad push(KB, [[K_c]])$
7: $\quad\quad error = fetchNextError(c_i(\vec{A}), [[K_c]] \bullet [[\delta]])$
8: $\quad\quad$ **if** $error \neq null$ **then**
9: $\quad\quad\quad VB([[K_c]], c_{i+1}(\vec{A}))$
10: $\quad\quad$ **else**
11: $\quad\quad\quad NewErrors = findNextError([[K_c]]\cdot\mathcal{C}, \langle[[K_c]]\backslash[[K]]\cdot[[K]]\backslash[[K_c]]\rangle)$
12: $\quad\quad\quad$ **if** $NewErrors \neq \emptyset$ **then**
13: $\quad\quad\quad\quad Errors = Errors \parallel NewErrors$
14: $\quad\quad\quad\quad VB([[K_c]], c'_1(\vec{A}))$
15: $\quad\quad\quad\quad$ Remove $c_i(\vec{A})$, $c_{i+1}(\vec{A})$, ..., $c'_n(\vec{A})$
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad PR = PR \cup \{[[K_c]]\}$
18: $\quad\quad\quad\quad pop(KB)$
19: $\quad\quad\quad$ **end if**
20: $\quad\quad$ **end if**
21: $\quad$ **end if**
22: **end for**

Algorithm 13: $IB_{opt}([[K_c]], [[\delta_{tot}]], c_i(\vec{A}))$

```
1:  Find Res(c_i(A⃗), [[K_c]])
2:  for all [[δ]] ∈ Res(c(A⃗), [[K_c]]) do
3:     [[δ_tmp]] = [[δ_tot]]
4:     [[δ_tot]] = [[δ_tot]] ⊎ [[δ]]
5:     clean([[δ_tot]])
6:     if isConflicting([[δ_tot]]) = false then
7:        Δ_rem = Res(c(A⃗), [[K_c]]) \ [[δ]]
8:        RepOpt = RepOpt ∪ {([[δ]], Δ_rem)}
9:        found = nodeCheck(RepOpt, Pairs)
10:       if found = false then
11:          error = fetchNextError(c_i(A⃗), [[K_c]] • [[δ]])
12:          if error ≠ null then
13:             IB_opt([[K_c]] • [[δ]], [[δ_tot]], error)
14:          else
15:             NewErrors = findNextError([[K_c]]·C, [[δ_tot]])
16:             if NewErrors ≠ ∅ then
17:                Errors = Errors ∥ NewErrors
18:                IB_opt([[K_c]] • [[δ]], [[δ_tot]], c'_1(A⃗))
19:                Remove c_i(A⃗), c_{i+1}(A⃗), ..., c'_n(A⃗)
20:             else
21:                potUs = leafCheck(RepOpt, Pairs, [[K_c]])
22:                RD = RD ∪ {[[δ_tot]]}
23:                if potUs = true then
24:                   RD = RD \ {[[δ]] | [[δ]] is a non useful repairing delta}
25:                end if
26:             end if
27:          end if
28:       end if
29:    end if
30:    if isConflicting([[δ_tot]]) = true or found = true then
31:       [[δ_tot]] = [[δ_tmp]]
32:    end if
33: end for
```

Appendix D

Consistency Model

D.1 Constraints Involving Typing Predicates

As we mentioned, SWKM incorporates a typing mechanism which characterizes the type of objects with respect to the triples they participate. Next we present the constraints which involve the typing predicates presented in Table 4.1.1. Any violation of the next constraints will create a typing error into the SWKM. In our work, we assume that the user has inserted correctly typed objects and we are confident that our approach does not create typing errrors as well.

- The next constraints quarantee that each object is of one type only. Any violation of the next rules throws a typing error which is not considered in our implementation.
  $\forall x\ \neg isCl(x) \vee \neg isCl(x)$
  $\forall x\ \neg isCl(x) \vee \neg isIn(x)$
  $\forall x\ \neg isCl(x) \vee \neg isMC(x)$
  $\forall x\ \neg isCl(x) \vee \neg isMP(x)$
  $\forall x\ \neg isCl(x) \vee x \neq rdfs : Literal$
  $\forall x\ \neg isPr(x) \vee \neg isIn(x)$
  $\forall x\ \neg isPr(x) \vee \neg isMC(x)$
  $\forall x\ \neg isPr(x) \vee \neg isMP(x)$
  $\forall x\ \neg isPr(x) \vee x \neq rdfs : Literal$
  $\forall x\ \neg isIn(x) \vee \neg isMC(x)$
  $\forall x\ \neg isIn(x) \vee \neg isMP(x)$
  $\forall x\ \neg isIn(x) \vee x \neq rdfs : Literal$
  $\forall x\ \neg isMC(x) \vee \neg isMP(x)$
  $\forall x\ \neg isMC(x) \vee x \neq rdfs : Literal$
  $\forall x\ \neg isMp(x) \vee x \neq rdfs : Literal$

- The next constraints refer to some basic objects.
  $\forall x\ isCl(rdfs : Resource)$
  $\forall x\ isMC(rdfs : Class)$
  $\forall x\ isMP(rdf : Property)$
  $\forall x\ \neg isCl(rdfs : Literal)$
  $\forall x\ \neg isMC(rdfs : Literal)$
  $\forall x\ \neg isCP(rdfs : Literal)$

- Both the domain and the range of a property must be classes or metaclasses or metaproperties; the range can also be the special class $rdfs : Literal$. This is not true for the domain of a property.
  $\forall x, y\ Domain(x, y) \rightarrow isPr(x)$
  $\forall x, y\ Domain(x, y) \rightarrow isCl(y) \vee isMC(y) \vee isMP(y)$
  $\forall x, y\ Range(x, y) \rightarrow isPr(x)$
  $\forall x, y\ Range(x, y) \rightarrow isCl(y) \vee isMC(y) \vee isMP(y) \vee y = rdfs : Literal$

- There may exist subsumption relationship ($rdfs : subClassOf$) between classes, metaclasses, or metaproperties. The subsumption relationship can be used for inference, using its transitivity property. Moreover, the basic objects $rdfs : Resource$, $rdfs : Class$, $rdf : Property$ cannot have ancestors in the subsumption hierarchies.
  $\forall x, y\ Cl\_IsA(x, y) \rightarrow x \neq rdfs : Resource \wedge x \neq rdfs : Class \wedge x \neq rdf : Property$
  $\forall x, y\ Cl\_IsA(x, y) \rightarrow (isCl(x) \wedge isCl(y)) \vee (isMC(x) \wedge isMC(y)) \vee (isMP(x) \wedge isMP(y))$
  $\forall x, y\ Cl\_IsA(x, y) \rightarrow Cl\_IsA_{inf}(x, y)$
  $\forall x, y, z\ Cl\_IsA(x, y) \wedge Cl\_IsA(y, z) \rightarrow Cl\_IsA_{inf}(x, z)$

- The $instance\_of$ relation ($rdf : type$) may be applied between a class instance and a class, or between a class and a metaclass, or between a property and a metaproperty. In every different case a typing error is thrown. If an instance is classified under a class (or metaclass, or metaproperty), then it is automatically classified under all the superclasses of the class (or metaclass, or metaproperty) as well, through inference.
  $\forall x, y\ Type(x, y) \rightarrow (isIn(x) \wedge isCl(y)) \vee (isCl(x) \wedge isMC(y)) \vee (isPr(x) \wedge isMP(y)) \vee (isMC(x) \wedge y = rdfs : Class) \vee (isMP(x) \wedge y = rdfs : Class)$
  $\forall x, y\ Type(x, y) \rightarrow Type_{inf}(x, y)$
  $\forall x, y, z\ Type(x, y) \wedge Cl\_IsA_{inf}(y, z) \rightarrow Type_{inf}(x, z)$
  $\forall x, y, z\ P_I nst(x, y, z) \rightarrow isPr(z)$

D.2 Resolution In Practise

We made the following assumptions with respect to diagnosis and repair. As a result the created resolution sets for some constraints are different than the ones expected in theory and shown in Table 4.1.2.

As. 1 When we have to add or remove a predicate which is translated into a specific triple, we add or remove the respective triple.

As. 2 RDF/S KBs which have to be validated and repaired do not contain typing errors because of the restrictions of the underlying model. Due to the model restrictions, whenever we try to change the type of an object, we will get a typing error from the underlying model. We don't attempt to circumvent this; instead, we make the assumption that changing the type of an object is not allowed. Therefore, we don't attempt to resolve this error, but propagate it, in effect cancelling the current branch and ignoring the alternative resolution (repair) that caused the error.

As. 3 The repairing change which denotes the insertion of an inferred predicate, is implemented by inserting the respective explicit. If it wasn't for this assumption, we would have to add a (maybe infinite) number of explicit predicates in order to create the inferred one.

As. 4 If we have to delete an object of a specific type (eg., a class, metaclass, metaproperty etc.) we must also remove all the triples which use it as a subject or predicate or object. This is done because typing predicates are not associated with particular triples; nevertheless, our rules imply that whenever an object must be deleted, we are forced to remove from the model all triples that involve it.

As. 5 We do not allow the insertion of new URIs/Literals during repairs.

As. 6 We do check constraints which refer to property instances explicitly whether they hold for the respective inferred predicates as well. We do not check rules which refer to implicit property instances, because, if they hold for the explicit predicates, they will also hold for the inferred predicates as well due to the transitivity of the subclass of relation.

As. 7. As we mentioned in As3, we chose to add the respective explicit predicate of the inferred predicate which appears in the head of the rule. However it is possible, relations represented with predicates P_Inst to be redundant (in redundant KBs). We remind that in redundant RDF/S KBs, all the inferred relations (subsumption and typeof) are expressed explicitly via the respective triples. This may cause in rare cases a non- deterministic repair process when inserting such explicit predicates. To avoid this case, the diagnosis process guarantees that we firstly repair violations of this rule referring to non- redundant and explicitly defined subproperty of or property instance relations. When all such violations have been resolved, any violation caused by an implicit and/or redundant subproperty of or property instance will have been resolved as well (as a side-effect).

As already mentioned, a constraint violation is resolved by applying one the repairing deltas which belong in the resolution set of the constraint. Next we redefine the resolution sets for the constraints presented in Table 4.1.2

- c_1, c_2: The corresponding resolution sets remains unchanged.
- c_3, c_4: Based on As. 5 and for the delta $[[\delta]]^{\mu}$, μ can take as value every class, metaclass, metaproperty which exist in the KB and $rdfs : Class$, $rdf : Property$, $rdfs : Resource$. Moreover, when we chose the solution to delete a property (i.e., delta $\delta = \langle \emptyset, \{isPr(P)\}\rangle$), we also remove all the triples which use the respective property as subject, predicate or object.
- c_5: For efficiency purposes, to repair such violations, we decompose the inferred IsA relationships to explicit. As a result, the resolution set will contain all the explicit IsA relations whose removal will repair the invalidity and it will be of the form: $Res(c_5(\langle x, y_1, y_2, \ldots, y_n\rangle), K)$ $= \{\langle \emptyset, \{Cl\_IsA(x, y_1)\}\rangle, \{\langle \emptyset, \{Cl\_IsA(y_1, y_2)\}\rangle, \ldots, \{\langle \emptyset, \{Cl\_IsA(y_n, x)\}\rangle$.
- c_6, c_7: Due to As. 3, the repairing delta $\delta = \langle \{Type_{inf}(y_1, y)\}, \emptyset\rangle$ is transformed into $\delta' = \langle \{Type_{inf}(y_1, y)\}, \emptyset\rangle$ and the rest repairing deltas of the resolution set remain as are.