Declarative Repairing Policies for Curated KBs

Yannis Roussakis
FORTH-ICS/Univ. of Crete
rousakis@ics.forth.gr

Giorgos Flouris
FORTH-ICS
fgeo@ics.forth.gr

Vassilis Christophides
FORTH-ICS/Univ. of Crete
christop@ics.forth.gr

ABSTRACT

Curated ontologies and semantic annotations are increasingly being used in e-science to reflect the current terminology and conceptualization of scientific domains. Such curated Knowledge Bases (KBs) are usually backended by relational databases using adequate schemas (generic or application/domain specific) and may 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 mostly enforced manually by the scientists acting as curators, we propose a *generic* and *personalized* repairing framework for assisting them in this arduous task. Our framework supports a variety of useful integrity constraints using Disjunctive Embedded Dependencies (DEDs) as well as complex curator preferences over interesting features of the resulting repairs (e.g., their size and type) that can capture diverse notions of *minimality* in repairs. 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.

1. INTRODUCTION

An increasing number of scientific communities 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 artefacts form a kind of curated *Knowledge Bases (KBs)* which are developed with a great deal of human effort by communities of scientists and are usually backended by relational database support using adequate schemas [21]. Curated KBs often have to satisfy various domain or application specific constraints [14, 18, 20] which are mostly enforced nowdays manually. For example, one may want to impose acyclicity of subsumption relations between classes or properties [20], primary/foreign key constraints [14], or cardinality constraints [18]. In this paper, we are interested in *declarative repairing* frameworks for assisting curators in the arduous task of repairing inconsistent KBs.

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 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 [2]) or by actually *repairing* KBs [9]. Clearly, the value of curated KBs lies in the quality of the encoded knowledge so they need to be frequently repaired.

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 [9]. 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, 6]. 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 [15]. In existing algorithms (e.g., [3, 4, 7]) 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 2, this makes the result of repairing sensitive to the evaluation order of the constraints, as well as their syntactic form. In a nutshell, in this paper we make the following contributions:

- In Sections 3 and 4 we propose 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) [8], 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 [5, 10, 12]. 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, 9]). As we will further detail in Section 7, *several repairing algorithms can be captured as special-cases of our framework*.

- In Section 5 we propose a novel, *globally-optimal (GO)*, algorithm, which successfully addresses the aforementioned issues by considering in an exhaustive manner all possible resolutions in order to identify globally optimal repairs (per the minimality preference). In contrast, existing algorithms (e.g., [3, 4, 7]), are *locally-optimal (LO)*, because they consider minimality only locally during the resolution of each inconsistency (by ignoring non-optimal resolution options) in a greedy fashion. We prove that the two algorithms do not give, in general, the same resulting KB repairs.

- In Subsection 5.2 we provide analytical complexity bounds for both GO and LO algorithms and show that the worst-case complexity is exponential for both algorithms. However, for typical curated KBs and for the typical error rates of 5-6% w.r.t. the original KB size (as reported in the literature [3, 7]), GO execution time does not exceed 5 minutes, thanks to a series of optimizations; we scaled our experiments up to an 8% error rate, in which case the GO execution time was around 13 minutes. Memory requirements did not exceed 200MB in all cases. On the other hand, the LO execution is typically 1-2 orders of magnitude faster than GO (see Section 6).

2. 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 [13, 21] of both schema and data of curated ontologies consists of atoms 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 [8]. 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 6.

$$c_1 : \forall u_1, u_2 C\_IsA(u_1, u_2) \rightarrow CS(u_1) \wedge CS(u_2)$$
$$c_2 : \forall u PS(u) \rightarrow \exists v_1, v_2 Domain(u, v_1) \wedge Range(u, v_2)$$
$$c_3 : \forall u_1, u_2 Range(u_1, u_2) \rightarrow PS(u_1) \wedge CS(u_2)$$
$$c_4 : \forall u_1, u_2 Domain(u_1, u_2) \rightarrow PS(u_1) \wedge CS(u_2)$$

Note that K violates c_1, c_2. The widely-used repairing strategy [4, 7] 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 choices are not independent, but may have unforeseen consequences. For example, if we choose to remove $PS(P)$

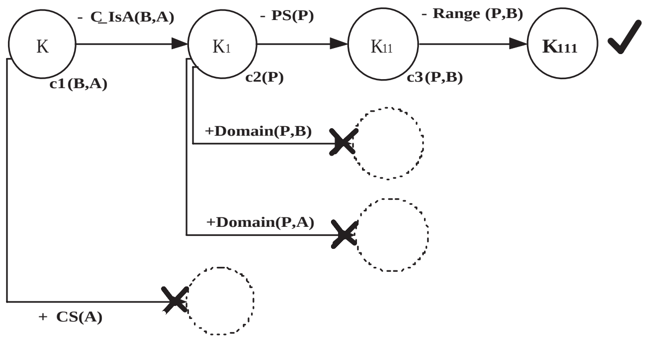

Figure 1: A LO Resolution Tree

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.

2.1 Repair Strategies

In most repair finding frameworks (e.g., [3, 4, 7]), 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*. Figure 1 shows one application of this *locally-optimal (LO)* 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 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 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 LO 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

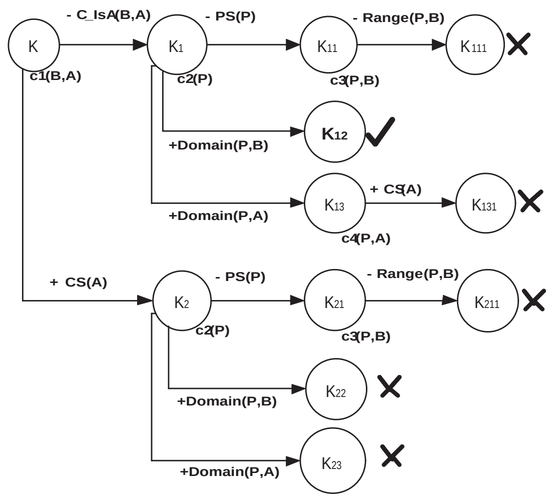

Figure 2: A GO Resolution Tree

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 LO strategy, we propose an alternative, *globally-optimal (GO)* resolution strategy. Under GO, none of the resolution options of a violation is rejected; instead, all resolution branches are considered (see Figure 2). 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, GO needs to compare all potential repairs in order to select only the preferred ones. Figure 2 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 2). Note that this is a different repair from the one returned by the LO strategy for the same KB and policy. Given that complete resolutions are considered, the returned repairs of a GO strategy are always the most preferred ones (unlike LO); in addition, as we will show later (Proposition 1), GO strategies are immune to changes in the constraints' evaluation order or syntax. However, GO strategies have to compute a larger resolution tree, so they are, on average, less efficient than LO strategies; note however that the worst-case complexity is the same in both cases (Section 5). 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).

2.2 Repair Preferences

Given the heterogeneity of curated KBs, different groups of curators should be able to apply (and experiment with) different repair policies. Being able to declaratively state repairing policies, calls 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 [12, 5, 10]. 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); such a preference would give, under a GO strategy, the repair $K_{111} = \{CS(B)\}$ (cf. Figure 2). 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., [3, 4, 7]) 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*. 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 7).

2.3 Wildcards

One problem with the policies following the GO strategy is that the number of 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.

3. FRAMEWORK

We consider the standard relational semantics; even though we don't commit ourselves to any particular schema, we will always assume some arbitrary, but predetermined finite set of relations $\mathcal{R}$ and infinite set of constants ($\mathcal{U}$). A *KB* K is any set of relational atoms of the form $R(\vec{A})$. The updates performed during repairs form a *delta* ($\langle\delta_a,\delta_d\rangle$), which essentially contains the sets of relational atoms to be added (δ_a) or removed (δ_d) from the KB. We denote by $\mathcal{K},\Delta$ the set of all KBs/deltas respectively. The *application* of a delta $\delta = \langle\delta_a,\delta_d\rangle$ upon a KB K is an operation $\bullet$ s.t. $K \bullet \delta = (K \cup \delta_a) \setminus \delta_d$. Deltas can be *composed* (denoted by $\uplus$) to produce a delta with a cumulative effect as follows: $\langle\delta_{a1},\delta_{d1}\rangle \uplus \langle\delta_{a2},\delta_{d2}\rangle = \langle\delta_{a1}\cup\delta_{a2},\delta_{d1}\cup\delta_{d2}\rangle$. Obviously, for any relational atom $Q(\vec{A})$ and KB K it holds that $K \vdash Q(\vec{A})$ iff $Q(\vec{A}) \in K$, and $K \vdash \neg Q(\vec{A})$ iff $K \nvdash Q(\vec{A})$ (i.e., iff $Q(\vec{A}) \notin K$).

We equip our framework with *wildcards* ($\varepsilon_1,\varepsilon_2,\ldots$), taken from an infinite set $\mathcal{E}$, which is disjoint from $\mathcal{U}$. A wildcard provides a compact representation of several constants and essentially introduces an *existential variable*. A *wildcard mapping* 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 $[[R(\varepsilon)]]^{\mu_1}$. 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). 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 2, KBs and deltas may feature wildcards. A K (δ) with wildcards corresponds to a set of KBs (deltas) as determined by some mapping μ. This set 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: $[[K]]^{\mu_K} \bullet [[\delta]]^{\mu_\delta} = [[(K\cup\delta_a)\setminus\delta_d]]^{\mu_K\cap\mu_\delta}$.

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 3.1), so we can control which wildcards are used, and the application semantics given previously is sufficient. Similarly, the composition operator $\uplus$ for deltas $\delta_1 = \langle\delta_{a1},\delta_{d1}\rangle$, $\delta_2 = \langle\delta_{a2},\delta_{d2}\rangle$ with wildcards is defined as follows: $[[\delta_1]]^{\mu_1} \uplus [[\delta_2]]^{\mu_2} = [[\delta]]^{\mu\cap\mu_2}$ where $\delta = \langle\delta_{a1}\cup\delta_{a2},\delta_{d1}\cup\delta_{d2}\rangle$.

3.1 Integrity Constraints

Integrity constraints are FOL formulas over relational atoms without wildcards. We restrict our attention to DEDs [8], i.e., formulas of the form: $\forall\vec{u}\bigwedge_{i=1,\ldots,n} P_i(\vec{u}) \rightarrow \bigvee_{i=1,\ldots,m} \exists\,\vec{v_i}\bigwedge j=1,\ldots,k_i$ $Q_{ij}(\vec{u},\vec{v_i})$. A KB K *satisfies* a constraint c (or more generally a set of constraints C) iff $K \vdash c$ ($K \vdash C$); K *violates* c (C) iff it does not satisfy c (C). 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 $\vec{A}$. As before, we assume that some arbitrary, but predetermined set of constraints C is given. A KB is called *consistent*, iff it satisfies all the defined integrity constraints; to guarantee the existence of consistent KBs, we require that the set of integrity constraints C is consistent in the standard logical sense.

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. 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 $j \in \{1,\ldots,k_i\}$

Considering the motivating example of Section 2, 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 neither #1 nor #2 above 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. Thus, the resolution of this violation can be made with one of the following deltas: $\delta_0 = \langle\{PS(P)\},\emptyset\rangle$, $\delta_x = \langle\emptyset,\{Domain(P,x)\}\rangle$ for all x. As explained in Section 2, wildcards can be used to compact the above set of deltas $\{\delta_x \,|\, \text{for all } x\}$, by using $\delta = \langle\emptyset,\{Domain(P,\varepsilon)\}\rangle$ for an unused wildcard ε.

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 an atom $P(a_1,\ldots,a_n)$ 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 an atom 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 $a \in \mu$, or $a \notin \mu$; 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 $K_0 = K \cup \{Domain(P,\varepsilon)\}$ for $\mu = \{A,B\}$, where K is as defined in the motivating example. 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 atom 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: $\delta_{NV} = \langle\emptyset,\emptyset\rangle$, $\mu_{NV} = \{B\}$, $\delta_{V1} = \langle\emptyset,\{Domain(P,\varepsilon)\}\rangle$, $\delta_{V2} = \langle\{CS(\varepsilon)\},\emptyset\rangle$, $\mu_V = \{A\}$. In the following, the deltas that can be used to resolve the violation of a certain instance $c(\vec{A})$ in a KB K will be called the *resolution set* of $c(\vec{A})$ w.r.t. K and denoted by $Res(c(\vec{A}),K)$. The same symbol will be used 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, $Res(c_4(P,A),[[K]]) = \{\ [[\delta_{NV}]]^{\mu_{NV}}, [[\delta_{V1}]]^{\mu_V}, [[\delta_{V2}]]^{\mu_V}\ \}$.

4. DECLARATIVE REPAIRING POLICIES

We have seen in Section 2 that repairing policies essentially reflect the notion of minimality [1] a curator wants to impose to the constraint resolution choices. For example, a curator may want to apply the policy: "I want a minimum number of updates during repair; in case of a tie, prefer those repairs that perform most deletions". To 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 (LO/GO), 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).

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. For example, to specify the aforementioned repairing policy we need to define the "number of updates" in a delta $\delta = \langle\delta_a,\delta_d\rangle$ as a feature: $f_{size}(\delta) = f_{additions}(\delta) + f_{deletions}(\delta)$ where $f_{additions}(\delta) = |\delta_a|$, $f_{deletions}(\delta) = |\delta_d|$.

To formalize the notion of declarative repairing policies, we rely on qualitative preference models proposed for relational databases [5, 10, 12]. 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 [10]. 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 [10]. 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 [10] 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 [10]. 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 $\delta_{10} > \delta_{20}$ for all $\delta_{10} \in [[\delta_1]]$, $\delta_{20} \in [[\delta_2]]$.

Besides the order $>_P$, a repairing policy must also specify the strategy (GO/LO) to follow for the filtering of the non-preferred repairing options. In the following, we will use the symbol $>^I_P$ ($>^V_P$) to denote the GO (LO) 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 (GO/LO).

For the LO 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 LO 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 LO 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:

DEF. 1. *Consider a LO repairing policy $>^V_P$, 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 $K_i \vdash C$ 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 $>^V_P$ iff there is some preferred repairing sequence SEQ of K for $>^V_P$ which terminates after n steps and $K' = K_n$.*

Let us now consider the preference expression described earlier $P = Min(size)\&Max(deletions)$, the KB K of our motivating example (see Section 2), and the corresponding order that defines the repairing policy $>^V_P$. If we choose to repair $c_1(B,A)$ first (i.e., $NextV(K) = c_1(B,A)$), we would take $K_1 = K$, $K_2 = K_1 \bullet$

Table 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

$\langle\emptyset, \{C\_IsA(B, A)\}\rangle$ as this is the only delta in $Res(c_1(B, A), K)$ with size 1 and 1 deletion (cf. Figure 1 and Table 1). 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 $K_4 = K_3 \bullet \langle\emptyset, \{Range(P, B)\}\rangle$ (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 1 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 GO strategy, a KB K' is a preferred repair for K iff it is consistent and there is some delta which is *globally* optimal per $>^I_P$, and whose application upon K leads to K'. Unlike LO 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 GO strategy, there is a subtle issue deserving further clarification. If we restrict ourselves only to optimality defined by the preference expression (per $>^I_P$) we may not always obtain *useful repairs*. For instance, consider the resolution tree of Figure 2 and the preference $Min(deletions)$. 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, 6]. This issue does not arise in LO 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.

DEF. 2. *Consider a GO repairing policy $>^I_P$ and some KB K. A delta $\delta = \langle\delta_a, \delta_d\rangle$ is a* preferred repairing delta *of K for $>^I$ iff:*

1. *$K \bullet \delta \vdash C$*
2. *The δ is useful, i.e., there is no $\delta' = \langle\delta'_a, \delta'_d\rangle$ such that $K \bullet \delta' \vdash C$, $\delta'_a \subseteq \delta_a$, $\delta'_d \subseteq \delta_d$ 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 $>^I_P$ iff there is some preferred repairing delta δ of K for $>^I_P$ such that $K' = K \bullet \delta$.*

Consider again the KB K of Section 2, the preference P = $Min(size)$ & $Max(deletions)$ and the corresponding order that defines the repairing policy $>^I_P$. Then, δ_{12} is a preferred repairing delta (cf. Figure 2 and Table 1). 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 2, 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).

Table 2: Selecting the Preferred Repairing Delta of K

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

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

4.1 Formal Properties

In Section 2 it was shown that when the LO strategy is employed, the syntax of the constraints affects the preferred repairs. This is not true when the GO strategy is employed:

PROP. 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 GO (or LO) strategy. Formally, a repair finding algorithm R will be called *GO-expressible* (*LO-expressible*) iff there is some repairing policy $>^I$ ($>^V$) such that for all KBs K it holds that $K' \in R(K)$ iff K' is a preferred repair for K per $>^I$ ($>^V$). The following propositions describe the aforementioned characterization and prove the generality of our framework:

PROP. 2. *A repair finding algorithm R is GO-expressible iff for all KBs 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 $K'_r = K_r$.*

PROP. 3. *A repair finding algorithm R is LO-expressible iff $R(K) = \{K\}$ when $K \vdash C$ 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 a GO-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 $K \nvdash C$.*

The condition of Proposition 2 is quite general and implies that a repair finding algorithm is GO-expressible iff it returns useful repairs. Similarly, the condition of Proposition 3 captures the recursive and “memory-less” character of LO 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 GO/LO strategies, not by the use of preferences. Therefore, preferences form an extremely powerful tool for modeling repair approaches; in Section 7, we will review existing repairing frameworks that are reducible to our framework.

4.2 The Issue of Modifications

Apart from additions and deletions, one would 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 $K_v = \{P(a, b, c), P(a', b, c')\}$. 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 $K_{v1} = \{P(a', b, c')\}$, $K_{v2} = \{P(a, b, c)\}$ 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 $K_{v3} = \{P(a', b, c), P(a', b, c')\}$, $K_{v4} = \{P(a, b, c), P(a, b, c')\}$ respectively.

The option of tuple modifications has been already proposed in the literature, e.g., in [7]. It could be easily incorporated as additional options in the resolution set of constraints for both the GO and the LO strategy, without affecting the rest of our formal model. Nevertheless, tuple modifications have not been incorporated in our framework because they cause several problems.

First, when modifications are considered, the repair process exhibits undesirable properties, like dependence on the resolution order of the integrity constraints and dependence on the syntax, rather than the semantics, of the constraints. Moreover, it raises various philosophical issues: it can be argued that modifying part of a tuple is actually the same operation as modifying a tuple altogether. Similarly, it is not clear whether modifying a tuple is one operation or if it consists of a deletion followed by an addition. Furthermore, it is not clear whether modification should be "local", i.e., affecting the tuple that causes the problem only, or "global", i.e., affecting all appearances of the modified constant. In particular, for the database context, constants represent values, so it often makes sense to replace said constant only locally. On the other hand, in the ontological context, a constant may be a URI representing a schema construct (e.g., a class A); in this case, deciding to equate two classes should be a global operation, i.e., causing all appearances of class A to be replaced by the replacing class. Finally, the option of modifying tuples introduces additional resolution options per violated constraints, thereby increasing the search space.

These problems have not been considered in previous work using tuple modifications (e.g., [7]); nevertheless, we consider them to be important issues that need to be resolved before incorporating modifications in the framework; we plan to address these problems as part of future work.

5. ALGORITHMS

In this section we give the two algorithms implementing GO and LO 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, $\vdash$, $Res(c(\vec{A}), K_c)$, $\uplus$, $\bullet$ $>$, etc) should be read under their extended semantics. We then sketch possible optimizations of the GO algorithm and provide complexity bounds for both algorithms.

LO (Algorithm 1) takes as input the current KB K_c (initially $K_c = K$). First, LO selects the next constraint violation to consider (using $NextV$) and then spawns a new recursive branch (call to LO) for the *preferred resolutions* per $>$ (set RD). The recursion stops when K_c is consistent. The current KB K_c in a leaf node is a preferred repair and is added to the final result (PR).

GO (Algorithm 2) takes as input the current KB K_c (initially $K_c = K$) 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 $>$. Then, the next violation is selected (here $NextV$ is not necessary) and a new recursive branch is spawned for *all* possible resolutions. Note that 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. The recursion stops when K_c is consistent, but, unlike LO, the current KB K_c in a leaf node is not necessarily a preferred repair. To determine whether it is a preferred repair or not, we add the corresponding delta (that has been preserved in δ_{tot}) to a collection (RD), and use an extra step (implemented in the calling function), which filters RD to find the preferred repairing deltas.

Algorithm 1 $LO(K_c)$
```
1: if $K_c \nvdash C$ then
2:     Find $Res(c(\vec{A}), K_c)$, where $c(\vec{A}) = NextV(K_c)$
3:     $RD = \{$preferred (per $>$) deltas in $Res(c(\vec{A}), K_c)\}$
4:     for all $\delta$ in $RD$ do
5:         $LO(K_c \bullet \delta)$
6:     end for
7: else
8:     $PR = PR \cup \{K_c\}$
9: end if
```

Algorithm 2 $GO(K_c, \delta_{tot})$
```
1: if $K_c \nvdash C$ then
2:     Find $Res(c(\vec{A}), K_c)$ for a violated constraint instance $c(\vec{A})$
3:     for all $\delta \in Res(c(\vec{A}), K_c)$ do
4:         $GO(K_c \bullet \delta, \delta_{tot} \uplus \delta)$
5:     end for
6: else
7:     $RD = RD \cup \{\delta_{tot}\}$
8: end if
```

As implied by Definition 1, the preferred repairs returned by the LO (Algorithm 1) are sensitive to the constraint resolution order ($NextV$). For GO (Algorithm 2), no matter which evaluation order is considered, the computed preferred repairs will be the correct ones, per Definition 2; however, we have found that for performance purposes it makes sense to consider first the constraints that have the fewest resolution options. Due to space limitations formal proofs regarding correctness of GO and LO algorithms are omitted.

5.1 Optimizations

Despite the exhaustive nature of the GO 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 $Min(size)$; 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 the requirement for useful repairs in Definition 2 (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 2 and the corresponding Figure 2. 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

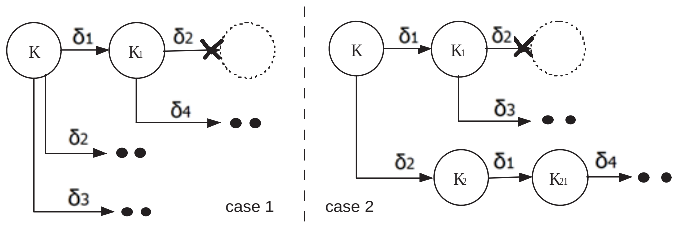

Figure 3: Examples of Tree Pruning

useful (cf δ_{23}). Thus, K_{131} could not have been a preferred repair, regardless of preference. This observation can be generalized:

PROP. 4. *Consider a KB K, a repairing policy $>^I$ and a leaf node L in the resolution tree created by GO. Suppose that the input of L was $K_L, \delta_L = \langle \delta_{aL}, \delta_{dL} \rangle$. Then the following are equivalent:*

1. *There is a node D (with input K_D, δ_D) in the same branch as L, such that $c(\vec{A})$ was resolved in D, and $\delta_1 = \langle \delta_{1a}, \delta_{1d} \rangle$, $\delta_2 = \langle \delta_{2a}, \delta_{2d} \rangle \in Res(c(\vec{A}), K_D)$ such that $\delta_{1a}, \delta_{2a} \subseteq K_L$, $\delta_{1d} \cap K_L = \delta_{2d} \cap K_L = \emptyset$.*
2. *There is another leaf node, say L' (with input $K_{L'}, \delta_{L'} = \langle \delta_{aL'}, \delta_{dL'} \rangle$) for which $\delta_{aL} \subseteq \delta_{aL'}$, $\delta_{dL} \subseteq \delta_{dL'}$.*

Proposition 4 can be used in several ways to optimize Algorithm 2. Firstly, assume that the first bullet of Proposition 4 holds and consider δ_L. Then, there is some node L' as described in the second bullet of Proposition 4. If $\delta_L \neq \delta_{L'}$, it follows that δ_L is not useful and must be rejected per Definition 2; if $\delta_L = \delta_{L'}$, then there is no need to keep both of δ_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 this check in the 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 know that all the leaf nodes that will result from this node will satisfy the first condition of Proposition 4. This gives us the option to prune entire branches as depicted in Figure 3. In 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, 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 further 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 2) when considering potentially preferred deltas (leafs). Namely, if the first bullet of Proposition 4 does not hold, then the delta is certainly useful, so we can avoid this check.

It should be noted at this point that, due to the notion of useful deltas (which does not exist in the LO strategy), and the related optimizations, the resolution tree created by the greedy (LO) strategy is not always a subtree of the one created by the exhaustive one (GO).

Another optimization is related to consistency checking. It exploits the fact that one consistency check can identify several violations, whereas only one violation is resolved in each recursive node; based on this, subsequent consistency checks are performed only after the initially detected violations have been resolved. In subsequent checks, any detected violations did not exist in the original KB, so they were introduced by the resolution of previously considered violations; thus, we can reduce the number of integrity constraints that need to checked to those that previous resolutions could violate.

Table 3: 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)

5.2 Complexity Analysis

The complexity of GO and LO repair finding algorithms is determined by the size of the corresponding resolution trees (say NOD) that need to be constructed. In our subsequent analysis we assume a fixed number of constraints in C, the largest of which has size N_r; the number of universally (resp. existentially) quantified variables in a constraint are at most N_x (resp. N_y); the original KB has size N_K and contains c constants.

Table 3 illustrates the complexity bounds for the height (H) and width (W) of the resolution tree for various forms of DEDs in C (cf. Figure 10). Readers are referred to [1, 8] for a detailed classification of DEDs expressivity. 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. Our analysis for the height relies on the existence of a loop detection procedure. For the GO algorithm, this is embedded in the check for useful deltas (Proposition 4), because deltas produced by loops are always non-useful; for the LO algorithm, loop detection need to identify that the same K_c is produced twice in the tree and thus reject the looping branch.

In this context the size of the resolution tree NOD in the worst case will be $O(W^H)$ (see Table 4). Obviously, since the total number of tree nodes is finite, the algorithm will always terminate. It should be stressed that the same results hold, in the worst-case, for both GO and LO 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 LO) is reduced to a chain, possibly with a polynomial number of nodes (e.g., for full TGDs and the preference $Min(additions)$).

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 4 contains the complexity results for $T_\&$ and the result found in [11] 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

Table 4: 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_{GO}	$O((N_K + H)^{N_r}) + O(W \cdot (N_K + H))$
T_{LO}	$O((N_K + H)^{N_r}) + O(W \cdot (N_K + H)) + T_{\otimes}(W,k)$
NOD	$O(W^H)$
T_{Repair}	$max\{NOD \cdot T_{GO} + T_{\otimes}(W^H,k), NOD \cdot T_{LO}\}$

larger than $T_{\&}$, we will use $T_{\otimes}$ as the final comparison cost.

In both GO and LO 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 LO 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 4 (T_{GO}, T_{LO}). The total cost (T_{Repair}) is computed by multiplying NOD with T_{GO} or T_{LO} costs. Note that for the GO 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. Wildcards will essentially reduce the average size of the resolution tree (NOD) when constraints with existential quantifiers exist, as expected by Table 3. Similarly, note that the optimizations described in the previous subsection do not affect the reported worst-case complexity, but reduce the average tree size, by pruning it in several cases.

6. EXPERIMENTAL EVALUATION

To experimentally evaluate the performance of our GO and LO repair finding algorithm, we relied on synthetically generated KBs and errors, and studied the impact of critical parameters (e.g., number and type of violated constraints, preference expressions) on the execution time and memory requirements of the algorithm, as well as on the quality (and number) of the obtained repairs.

To cover a more representative variety of DEDs, we extended the set of constraints presented in Section 2 (c_1-c_4) as follows:

c_5 : $\forall u_1, u_2 C\_IsA(u_1, u_2) \wedge C\_IsA(u_2, u_1) \rightarrow \perp$

c_6 : $\forall u_1, u_2, u_3 Domain(u_1, u_2) \wedge Domain(u_1, u_3) \rightarrow \perp$

c_7 : $\forall u_1, u_2, u_3 Range(u_1, u_2) \wedge Range(u_1, u_3) \rightarrow \perp$

c_8 : $\forall u_1, u_2, u_3, u_4 PI(u_1, u_2, u_3) \wedge Domain(u_3, u_4) \rightarrow CIns(u_1, u_4)$

c_9 : $\forall u_1, u_2, u_3, u_4 PI(u_1, u_2, u_3) \wedge Range(u_3, u_4) \rightarrow CIns(u_2, u_4)$

In the above constraints, $PI(x, y, z)$ denotes that the pair (x, y) is a direct or transitive instance of property z, and $CIns(x, y)$ denotes that x is a direct or transitive instance of class y. Constraint c_5 requires that all subsumption relationships are acyclic while c_6, c_7 ensure that properties have a unique domain and range. Constraints c_8, c_9 concern ontology data and require that the subject and object of a property instance must be of a type compatible with the corresponding property's domain and range.

For our experiments, we created synthetic RDF/S KBs featuring different structural characteristics using PowerGen [22][1]. In particular, we created one KB privileging classes (Class Centric – CC) and one KB privileging properties (Property Centric – PC). CC has 80 classes and 40 properties, whereas PC has 40 classes and 120 properties. The maximum subsumption depth in both KBs is 5. CC and PC represent typical RDF/S ontologies on the Semantic Web [22]. We enriched these KBs with instances, by adding 15 instances per class and 1-11 instances per property in both cases, creating the KBs CC with Data (CCD) and PC with Data (PCD) respectively.

[1]http://139.91.183.30:9090/RDF/PoweRGen

All the above KBs are consistent, so we developed an *error insertion algorithm (EIA)* to make them inconsistent. The parameters of EIA are the number of errors to introduce and the DED constraints to be violated. We applied EIA on CC and PC to violate the ontology schema constraints (c_1-c_7) and on PCD and CCD to violate the ontology data constraints (c_8, c_9). The errors to be inserted were uniformly distributed over the constraints to be violated, and were created by adding/removing some fact that would invalidate the selected constraint. Note that the resolution of c_8,c_9 could potentially invalidate c_1-c_7, so our analysis below takes into account the *actual* errors produced, not the initial errors introduced by the EIA. 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. Thus, the investigated error margins are similar to those used in other works (e.g., [3, 7]), even though the error generation procedure is different.

All the experiments were contacted on a quad core CPU machine at 2.40GHz with 3GB of memory running Ubuntu 10.04. To smoothen the effects of the randomized EIA algorithm, each experiment was run 40 times, and the averages were taken.

6.1 GO Algorithm Evaluation

Per the complexity analysis of Table 4, the performance of the GO 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 GO strategy. Then, we study the time and memory requirements of the GO 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 [10].

As we can see in Figure 4 the size of the resolution tree grows exponentially as the number of introduced errors increase. This is because 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. This explains why the four KBs of our testbed give resolution trees of different size for the same number of errors. For CC and PC KBs, which violate the schema constraints $c_1 - c_7$ only, the tree's fan-out is 2, so one would expect the corresponding curves to coincide, which is not the case. Similarly, CCD and PCD, where the fan-out ranges from 2 to 3 (due to the constraints c_8, c_9), were expected to give larger resolution trees (this is indeed the case up to 17 errors). Both discrepancies are explained by the fact that a different amount of pruning is achieved in each case through the use of Proposition 4.

As we can see in Figure 5, the execution time scales linearly with the size of the corresponding resolution trees in all cases, confirming our complexity results of Table 4. We note also that the inclinations are similar, indicating a similar execution time per node of the tree in the four KBs. The CC line has a slightly higher execution time per node, because of the wildcards that appear in CC and PC execution; unlike PC however, in CC the wildcards span over more objects, because CC contains more classes, causing the increased

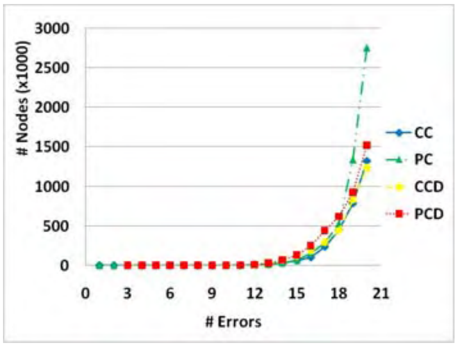

Figure 4: Size of the Resolution Tree for GO

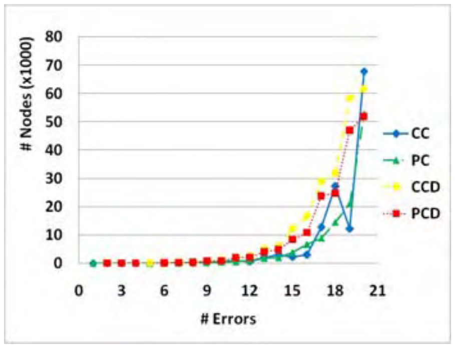

Figure 6: Size of the Resolution Tree for LO

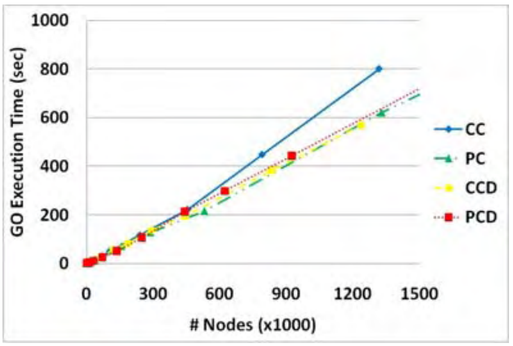

Figure 5: GO Execution Time

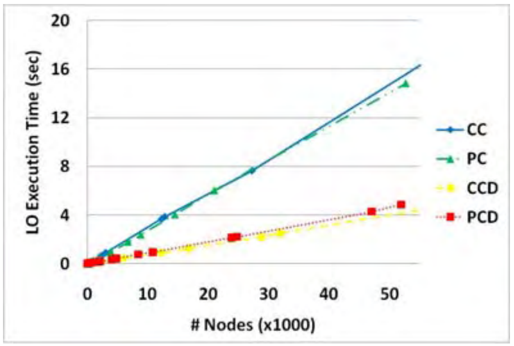

Figure 7: LO Execution Time

inclination.

The memory requirements of our GO 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_2 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[2] 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 GO experiments required more than 200MB of memory.

6.2 LO Algorithm Evaluation

For the evaluation of the LO algorithm we used the preference expression $Min(additions)$. 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 LO results with the ones presented for GO in the previous subsection.

As we see in Figure 6, the size of the LO resolution tree grows exponentially with the number of violations despite LO algorithm's greedy nature. However, the actual number of nodes is significantly smaller than in the GO 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 ($c_1 - c_4$, c_8, c_9) by 1. As a result, the constructed resolution tree of the GO algorithm is about 2 orders of magnitude larger than the one created by the LO. Comparing Figures 4, 6, we note that the relative order of the curves for the different KBs is different than the GO case. This is because the pruning performed depends on the actual constraints violated, which is different per KB; the same fact explains the reduced nodes for CC when the number of errors is 19.

As with the GO case, Figure 7 shows that the execution time scales linearly with the tree size. The execution time for the LO algorithm, for the same KB and number of errors, is 2 orders of magnitude smaller than the time required for the GO 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 LO case. This is because the GO algorithm (unlike the LO) includes the optimization checks described after Proposition 4. Note however that this extra cost in the GO algorithm is more than compensated by the reduction in the number of nodes caused by these optimizations. The different inclinations of the curves are due to the different (average) cost for determining whether the KB is consistent depending on the type of constraints (c_8, c_9 are faster). This effect is not visible in the GO case, because the cost per node is dominated by the optimization checks.

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

6.3 Quality of LO Repairs

In this subsection, we evaluate the quality of the repairs returned by the LO algorithm. More precisely, we measure the number of globally optimal repairs (returned by GO) which are not returned by the LO (*false negatives*, denoted by $GO \setminus LO$ in Figures 8, 9),

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

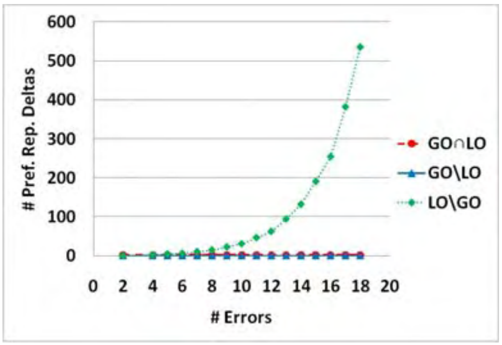

Figure 8: LO vs GO Repairs in CCD for Min(additions)

and the number of preferred repairs returned by LO which are not globally optimal (*false positives*, denoted by $LO \setminus GO$ in said figures). 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. Using the preference $Min(additions)$ GO and LO return exactly the same results for CC and PC. However, when CCD and PCD are considered, LO contains several non-globally optimal results (false positives). For the CCD in particular (see Figure 8), we observe that GO returns just one repair, whereas LO returns this repair (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, LO will also consider the option to delete a domain or range in each case, which generates several non-optimal repairs (false positives).

If we change the preference into $Max(additions)$ then similar results hold: GO and LO return the same results for PC and CC, but different ones for CCD, PCD. Considering CCD again (Figure 9), GO and LO return a completely disjoint set of deltas. In particular, LO returns a single repair in this case, which is not returned by GO (i.e., one false positive). On the other hand, GO (unlike LO) returns several more repairs (false negatives) whose number grows exponentially with the number of violations (albeit in a less smooth way than in Figure 8). The reason is similar as above: LO 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 optimal repairs.

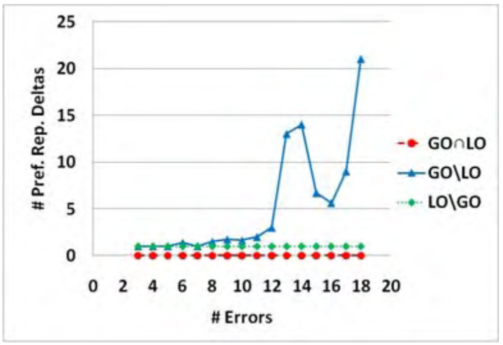

Figure 9: LO vs GO Repairs in CCD for Max(additions)

7. RELATED WORK

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 [8]. Note that the task of determining whether a repair is preferred is different than actually computing a preferred repair, which is the problem we deal with in this paper. Four variations of minimality have been proposed in [1], all of which can be captured in our framework, and fit under the GO 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 repairing KB should be a subset of the original, containing the least possible (per the $\subseteq$ relation) updates (i.e., deletions). This minimality notion can be captured using the preference $Min(additions)$ over the feature $f_{additions}$ defined in Section 4. The second is *symmetric difference repairs*, which require that the repairing 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 $Min(size)$ over the feature f_{size} defined in Section 4. 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 defining a set of features f_i, each one counting the number of appearances of a certain relational R_i in the delta. Each of those features defines a preference of the form $P_i = Min(f_i)$, which should be combined using pareto to get the final composite preference (i.e., $P_1 \otimes \ldots \otimes P_n$).

Other works such as [3] deal with the actual problem of automatically finding the repairs. The framework of [3] only considers Functional and Inclusion Dependencies (FD/IND); in [7], this work has been extended to support Conditional Functional Dependencies (CFD) as well (see Figure 10). Each violated constraint is resolved in a predetermined manner, depending on its kind. FD and CFD constraints (e.g., $\forall x, y, z P(z, x) \wedge P(z, y) \rightarrow x = y$) are resolved through tuple modifications (in effect replacing x with y in the corresponding tuples or vice-versa). IND constraints (e.g., $\forall x P(x) \rightarrow \exists y Q(x, y)$) are resolved by adding a tuple of the form $Q(x, null)$; note that *null* is used to avoid considering all possible assignments for y. As resolution is made independently for each constraint, this framework implements a LO 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 Subsection 4.2), we cannot capture the repair policy used in [3, 7] for FDs and CFDs. We can only capture the repair policy for INDs, as follows: first we need to overrule all deletions, through the preference $P_1 = Min(deletions)$. This preference must be combined, using $\&$, with a preference that would prefer the additions that use *null* for the existential quantifiers, over the rest of the additions. To do so, we find the constraint instance that is being considered (which is an IND, so it has the form $P(\vec{A}) \rightarrow \exists \vec{v_i} Q(\vec{A}, \vec{v_i})$) using $NextV$; then, we define a feature that scans the delta and counts the atoms of the form $Q(\vec{A}, \vec{null})$ that have to added. Formally: $f_{null}(\delta) = |\{Q(\vec{A}, \vec{null}) \in \delta_a$, where $NextV(K)$ is of the form $P(\vec{A}) \rightarrow \exists \vec{v_i} Q(\vec{A}, \vec{v_i})\}|$. The final, composite preference is $P = Min(deletions)\&Max(null)$.

The use of *null* in order to avoid producing a large number of repairs has also been employed in the framework of [4] which is ca-

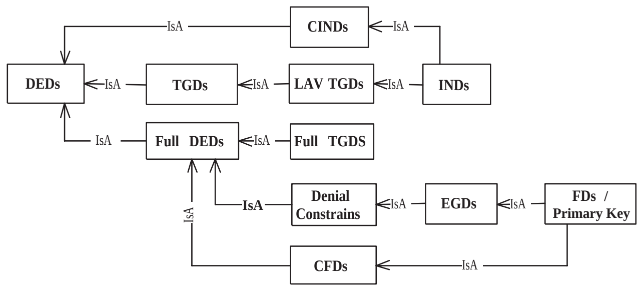

Figure 10: Types of DED Constraints

pable 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 LO 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 feature is necessary to consider other DED types as well. Due to lack of space we omit the details of the corresponding preference.

Another repair finding algorithm appears in Rondo [17], 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 $Max(f_l)$, where f_l is a function counting the number of tuples whose label is l in δ. We combine these preferences using $\&$ to form the final preference as follows: $P_{--} \& P_{-+} \& P_{+-} \& P_{++} \& P_{-0} \& P_{+0} \& P_{0-} \& P_{0+} \& P_{00}$.

Several works have been also proposed for repairing ontologies (e.g., PROMPT [19], Chimaera [16]). These works only address violations of constraints which can be expressed in the underlying knowledge representation formalism and essentially correspond to a subset of DEDs (e.g., functional and inclusion dependencies). Unlike our work, only violation detection is done automatically, whereas violation resolution and repair must be done manually.

8. CONCLUSIONS

We proposed a *declarative repairing* framework for assisting curators in identifying a consistent KB that would preserve as much knowledge as possible from a given KB that violates a number of integrity constraints. Our framework (a) is *generic*, since it captures a wide class of integrity constraints using DEDs; (b) is *customizable*, even at run-time, according to the requirements of different curation settings using qualitative preferences over repairs' features; (c) is *flexible*, since it supports both greedy (LO) and exhaustive (GO) repair finding strategies; (d) is *expressive*, given that most existing algorithms proposed in the literature can be expressed as special cases of our framework using adequate repair policies. We believe that one of the major contributions of our work is the distinction between locally (LO) and globally (GO) optimal repairs w.r.t. a preference, which opened the way to devise a repair finding algorithm (GO) which is immune to changes in the constraints' syntax and evaluation order.

As a future work, we could further consider the issue of tuple modifications, addressing the problems described in Subsection 4.2. Given that the algorithm's complexity is inherently exponential, we also need to devise heuristics that would improve its efficiency. In the same direction, an additional topic 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. 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.

9. REFERENCES

[1] F. N. Afrati and P. G. Kolaitis. Repair checking in inconsistent databases: algorithms and complexity. In *ICDT-09*, 2009.
[2] L. E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 2006.
[3] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *ACM SIGMOD*, 2005.
[4] L. Bravo and L. Bertossi. Semantically correct query answers in the presence of null values. In *EDBT-06*. 2006.
[5] J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems (TODS)*, 28(4), 2003.
[6] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. *Inconsistency Tolerance*, 2005.
[7] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB-07*, 2007.
[8] A. Deutsch. Fol modeling of integrity constraints (dependencies). In *Encyclopedia of Database Systems*. 2009.
[9] W. Fan. Dependencies revisited for improving data quality.
[10] P. Georgiadis, I. Kapantaidakis, V. Christophides, E. M. Nguer, and N. Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE-08*, 2008.
[11] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB Journal*, 2007.
[12] W. Kießling. Foundations of preferences in database systems. In *VLDB-02*, 2002.
[13] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A formal approach for rdf/s ontology evolution. In *ECAI*, 2008.
[14] G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for rdf. In *EDBT-08*, 2008.
[15] M. Lenzerini. Data integration: A theoretical perspective. In *PODS-02*, 2002.
[16] D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *KR*, 2000.
[17] S. Melnik. *Generic Model Management: Concepts and Algorithms*. Springer, 2004.
[18] B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *WWW-07*, 2007.
[19] N. F. Noy and M. A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI*, 2000.
[20] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *ISWC-05*, 2005.
[21] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *ISWC-05*, 2005.
[22] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On graph features of semantic web schemas. *IEEE TKDE*, 20(5), 2008.