Specifying and Querying Database Repairs using Logic Programs with Exceptions

Marcelo Arenas[1], **Leopoldo Bertossi**[1], and **Jan Chomicki**[2]

[1] Pontificia Universidad Catolica de Chile, Departamento de Ciencia de Computacion, Casilla 306, Santiago 22, Chile, {marenas,bertossi}@ing.puc.cl

[2] Monmouth University, Dept. of Computer Science, West Long Branch, NJ 07764, chomicki@monmouth.edu

Abstract Databases may be inconsistent with respect to a given set of integrity constraints. Nevertheless, most of the data may be consistent. In this paper we show how to specify consistent data and how to query a relational database in such a way that only consistent data is retrieved. The specification and queries are based on disjunctive extended logic programs with positive and negative exceptions that generalize those previously introduced by Kowalski and Sadri.

1 Introduction

Information repositories of today are often built on top of multiple heterogenous data sources. This fact has spurred the rethinking of the classical database problems like query evaluation and optimization [11,31], and integrity constraints. Data coming from multiple, autonomous sources is unlikely to satisfy given integrity constraints, even though each source separately may satisfy them. (Consider a person's address or name that may be different in different data sources.) The autonomy of different data sources makes it infeasible to correct global integrity violations. However, the constraints describe important semantic properties of the data and should not be cast aside. We propose to use them to augment the query answering process. In our approach, databases will provide not only regular query answers (computed without taking into account the integrity constraints) but, in addition, query answers that are *consistent* with the constraints.

The following example presents the basic intuitions behind the notion of consistent query answer.

Example 1. Consider a database subject to the following IC: $\forall x(P(x) \Rightarrow Q(x))$. The databases instance $\{P(a), P(b), Q(a), Q(c)\}$ (as a first order structure) violates this constraint. Now if the query asks for all x such that $P(x)$ is true, then only a is returned as an answer consistent with the integrity constraint. There is a repair in which $P(b)$ is not true, and thus b is not a consistent answer. □

Intuitively, if a query answer is consistent, then it is stable: regardless of how the integrity violation is fixed it remains an answer. On the other hand, a query answer which is inconsistent may be due to some transient error in the data and therefore is not very reliable. Therefore, the information about answer consistency serves as an important indication of data quality and reliability.

In [4] we formalized the notion of consistent query answer as an *answer true in every minimal repair.* Here, we introduce a modal operator $\mathcal{K}$ with the meaning that $\mathcal{K}\alpha$ is true in a relational database instance r if α is true in all minimal repairs of r. We show how to evaluate first-order queries in which the operator $\mathcal{K}$ occurs (we call them $\mathcal{K}$-queries for short). The basic insight is that the (minimal) repairs of a given instance r correspond to the e-answer sets of a logic program with exceptions Π^r. For this purpose, we introduce *disjunctive extended logic programs with exceptions* that generalize those introduced in [21]. Our programs contain strong and procedural negation, disjunctive heads, and both negative and positive exceptions to positive and negative defaults, resp. As the programs in [21], they can be transformed into disjunctive extended logic programs as presented in [18]. In consequence, any evaluation method for our logic programs with exceptions (e.g., by using an ad hoc evaluator[1], or by transforming them to disjunctive extended logic programs [18] and evaluating those) can be combined with some standard evaluation method for first-order queries (e.g., by translating them to relational algebra or SQL) to yield a method for evaluating $\mathcal{K}$-queries. Computational implementations for evaluating extended logic programs exist [9,26,14,15].

In [4] we proposed a different method for obtaining consistent query answers by transforming given queries using the techniques of semantic query optimization. Unfortunately, the scope of that method is quite limited: Queries cannot contain quantifiers or disjunction and the classes of integrity constraints allowed are also severely restricted. The present method does not have such limitations: It handles arbitrary first-order queries with the $\mathcal{K}$ operator and arbitrary universally quantified integrity constraints (and also some existential cases). On the other hand, it requires a much more powerful evaluation mechanism than the approach of [4], that relies on the standard evaluation mechanism for relational queries and can be implemented on top of any relational DBMS [8].

The plan of the paper is as follows. In section 2 we define the basic notions of our approach, including those of a database repair and consistent query answer. In section 3 we summarize the framework of logic programs with exceptions as presented in [21]. In section 4 we show several examples of the correspondence between database repairs and e-answer sets of logic programs with exceptions. In section 5 we present a general methodology to derive repair programs from sets of universally-quantified constraints. There we also introduce disjunctive extended logic programs with exceptions and their semantics based on e-answer sets. In section 6 we consider consistent query evaluation. In section 7 we consider possible extensions of the specification formalism and alternatives. In section 8 we briefly summarize related work. We conclude by outlining further work in section 9.

---

[1] We implemented in XSB [28], *EAnswers*, our own evaluator for disjunctive extended logic programs with exceptions.

2 Consistent Query Answers

We assume a fixed set, *IC*, of integrity constraints associated with a fixed relational database schema. We also have a fixed, possibly infinite, database domain D. A database instance is a structure over D, consisting of a collection of finite extensions for the predicates in the database schema. Built-in predicates may have an infinite, but fixed extension in each database instance. A database instance r is *consistent* if it satisfies *IC*, that is $r \vDash IC$. Otherwise, we say that r is *inconsistent*. We assume that *IC* is consistent in the sense that there is a database instance r that satisfies *IC*.

If r is inconsistent, its *repairs* are database instances (with respect to the same schema and domain) that, each of them, satisfy *IC* and differ from r by a minimal set (wrt to set inclusion) of inserted or deleted tuples.

Example 2. Consider the functional dependency $P(x,y,z) \land P(x,u,v) \Rightarrow y = u$, and the inconsistent database instance r with the table $P = \{(a,1,5), (a,2,6),(a,1,7)\}$. This instance has two possible repairs, r_1, r_2, with tables $P_1 = \{(a,1,5),(a,1,7)\}$ and $P_2 = \{(a,2,6)\}$, respectively. □

The syntax of first-order queries (termed *basic queries* here) is defined by the following grammar: $B \ ::= \ \mathit{Atom} \mid B \land B \mid \neg B \mid \exists x\ B$. The syntax of $\mathcal{K}$-queries is similarly defined: $A \ ::= \ \mathcal{K}B \mid A \land A \mid \neg A \mid \exists x.\ A$.

We say that a ground query $\mathcal{K}\alpha$ is true in an instance r wrt *IC*, in symbols $r \vDash \mathcal{K}\alpha$, if the query α is true in every repair r' of r wrt *IC*, in symbols $r' \vDash \alpha$. The semantics of the remaining constructs is the standard semantics of first-order logic. A tuple $\bar{t}$ is an answer to a $\mathcal{K}$-query $Q(\bar{x})$ in r if $r \vDash \mathcal{K}Q(\bar{t})$. A tuple $\bar{t}$ is a *consistent* answer to a basic query $Q(\bar{x})$ in r wrt *IC* if $r \vDash \mathcal{K}Q(\bar{t})$.

Example 3. (example 2 continued) The query $Q(x) : \exists y \exists z P(x,y,z)$ has a as a consistent answer: $r \models \mathcal{K}Q(a)$. On the other hand, the query $\exists x \exists z P(x,1,z)$ has no consistent answers, because $r_2 \not\models \exists x \exists z P(x,1,z)$. □

As we have seen, consistent answers to first-order queries are a special case of answers to $\mathcal{K}$-queries. Thus, the method to compute the latter presented in this paper is automatically applicable to computing the former.

3 Logic Programs with Exceptions

In [21], Kowalski and Sadri introduced *logic programs with exceptions* (LPe). These are programs with the syntax of an extended logic program (ELP) [17,18], that is, in them we may find rules with both logical (or strong) negation ($\neg$) and procedural negation (not) [25]. In these programs, rules with a positive literal in the head represent a sort of general defaults, whereas rules with a logically negated head represent exceptions.

Since general defaults and exceptions may contradict each other, the semantics of ELPs has to be changed in order capture the intuition that exceptions have higher priority than defaults. This solves the contradictions at the semantical level. In order to capture the new semantics at the procedural level, an LPe is transformed into a new ELP, with its usual semantics [22].

As indicated in [21] and shown in [17,18], the extended logic program can be further transformed into an equivalent normal logic program (without logical negation), with a stable model semantics [16].

Now we will give a brief account of logic programs with exceptions. An LPe consists of clauses of the form

$$L_0 \longleftarrow L_1, \ldots, L_k, \text{ not } L_{k+1}, \ldots, \text{ not } L_n, \tag{1}$$

where each L_i is a literal, that is, an atom or a logically negated atom, $\neg A$. *Rules* or defaults are clauses with a positive head. The *exceptions* are the clauses with a negative head.

Example 4. The following is a logic program with exceptions

$$\mathit{fly}(x) \longleftarrow \mathit{bird}(x) \tag{2}$$

$$\neg \mathit{fly}(y) \longleftarrow y = \mathit{emu} \tag{3}$$

As we can see, with the fact $\mathit{bird}(\mathit{emu})$, unless we assign to the program an appropriate semantics (or evaluate it in a proper way), we will obtain a contradiction. The semantics should sanction (3) as an exception to the default rule (2). □

The semantics of an LPe is obtained from the semantics for ELPs, by adding an extra condition that assigns higher priority to exceptions. Next we review the definition of the semantics for LPe's as presented in [21].

Let Lit be the set of ground literals, and Π be a program consisting of ground clauses of the form (1), but without *not*. The *answer set* of Π, denoted $as(\Pi)$, is the smallest subset S of Lit, such that: (a) For any clause $L_0 \longleftarrow L_1, \ldots, L_k$, if $L_1, \ldots, L_k \in S$, then also $L_0 \in S$. (b) If S contains a pair of complementary literals, then $S = \mathit{Lit}$. In this case, Π is said to be contradictory.

Now, consider a ground LPe, Π, with clauses as in (1). Let S be a subset of Lit. We define a new program, ${}^S\Pi$, obtained from Π and S as follows:

1. Delete every clause containing a condition *not* L, with $L \in S$.
2. Delete in the remaining clauses every condition *not* L if $L \notin S$.
3. Delete every clause having a positive conclusion L, with $\neg L \in S$.

The resulting program ${}^S\Pi$ is a ground ELP without *not* . In consequence, $as({}^S\Pi)$ is defined. We say that a set S of ground literals is an *e-answer set* for Π, the original LPe, if $S = as({}^S\Pi)$. It can be shown [21,22] that e-answer sets are in correspondence with answer sets of extended logic programs. This can be established by transforming the original LPe into an ELP with the answer set semantics. Among other transformation rules, a positive default rule of the form $P(\bar{t}) \longleftarrow C$, if there is a negative exception clause of the form $\neg P(\bar{t}') \longleftarrow B$, is transformed into the clause

$$[P(\bar{t}) \longleftarrow C, \mathit{not}\ \neg P(\bar{t}')]\theta, \tag{4}$$

where θ is mgu of $\bar{t}$ and $\bar{t}'$.

In this way we obtain an ELP in which the original exceptions remain as ordinary clauses of the new ELP, but are no longer exceptions in the sense that they have already generated extra conditions in the bodies of the corresponding rules. In this way, contradictions are avoided. The resulting extended logic program Π_t can be evaluated as any extended logic program [22].

4 Specifying Database Repairs

Our approach here consists in the direct specification of the database repairs in a logic programming formalism. We expect the database repairs to correspond to the intended models of the program.

In general, given a predicate P, for a table of a database instance r, that participates in some of the ICs, we will introduce a new predicate P' that should be the repaired version of predicate P, that is the one that contains the tuples corresponding to P in a repair of the original database.

Example 5. Consider the functional dependency *FD*: $P(x,y) \wedge P(x,z) \Rightarrow y = z$, and the inconsistent database instance $r = \{P(a,b), P(a,c), P(c,a)\}$.

We introduce a new predicate P'. By default, P' contains what is contained in P. This generates the (default) *rule*:

$$P'(x,y) \longleftarrow P(x,y). \tag{5}$$

Now, FD is logically equivalent to $P(x,y) \wedge y \neq z \Rightarrow \neg P(x,z)$. From it we obtain the *exception*:

$$\neg P'(u,v) \longleftarrow P'(u,z), \neg\, v = z. \tag{6}$$

Clauses (5) and (6), together with the facts, $P(a,b), P(a,c), P(c,a)$, and the clauses for equality (treated as exceptions, because they are not defeasible), $\neg\ x = y \leftarrow \mathit{not}\ x = y$ and $x = x$, constitute a logic program with exceptions, Π^r, that specifies the repaired predicate P'. Notice that there may be contradictions between clauses (5) and (6) if they are treated as usual.

Running *EAnswers*, our e-answer sets generator and checker, on this program, we obtain as only e-answer sets

$$\begin{aligned}
S_1 = \{&\neg P'(a,b), P'(a,c), P'(c,a), P(a,b), P(a,c), P(c,a), a = a, b = b, c = c,\\
&\neg a = b, \neg b = a, \neg a = c, \neg c = a, \neg b = c, \neg c = b, \neg P'(a,a), \neg P'(c,b),\\
&\neg P'(c,c)\}\\
S_2 = \{&P'(a,b), \neg P'(a,c), P'(c,a), P(a,b), P(a,c), P(c,a), a = a, b = b, c = c,\\
&\neg a = b, \neg b = a, \neg a = c, \neg c = a, \neg b = c, \neg c = b, \neg P'(a,a), \neg P'(c,b),\\
&\neg P'(c,c)\}
\end{aligned}$$

Considering only the P' atoms in each of these e-answer sets, we obtain $\{P'(a,c), P'(c,a)\}$ and $\{P'(a,b), P'(c,a)\}$, corresponding to the two expected database repairs, namely $r_1 = \{P(a,c), P(c,a)\}$ and $r_2 = \{P(a,b), P(c,a)\}$, resp.

Example 6. Consider now the inclusion dependency *ID*: $P(x) \Rightarrow Q(x)$, and the inconsistent database instance $r = \{P(a)\}$. As in the previous example, in order to specify the database repairs, we introduce new predicates P', Q' and the following four (default) *rules* expressing that P' and Q' contain exactly what P and Q contain, resp.: $P'(x) \leftarrow P(x)$; $Q'(x) \leftarrow Q(x)$; $\neg P'(x) \leftarrow$ *not* $P(x)$; and $\neg Q'(x) \leftarrow$ *not* $Q(x)$.

Now, *ID* generates two clauses: $Q'(x) \leftarrow P'(x)$ and $\neg P'(u) \leftarrow \neg Q'(u)$. The first one is a *rule*[2] and the second one, an *exception*. These clauses plus the fact $P(a)$ constitute the repair logic program with exceptions, Π^r. In this case, *EAnswers* gives us the following e-answer sets: $S_1 = \{\neg P'(a), \neg Q'(a), P(a)\}$ and $S_2 = \{P'(a), Q'(a), P(a)\}$, corresponding to the two expected repairs, namely the empty set and $\{P(a), Q(a)\}$. □

There are cases where an integrity constraint can not be fully captured by the logic programs with exceptions as we presented them. The reason is that they do not generate any exception clause. Although those constraints may not be of much interest in practice, we still are in position to accommodate them in a natural way by making the ICs generate *positive exceptions* to rules with *negative* heads. This is illustrated in the following example.

Example 7. Consider the IC: $x = a \Rightarrow P(x)$, that forces $P(a)$ to hold[3]. In this case, no exception clause is generated. Now, the logic program, that does not have any facts, consists of the definition of equality plus the clauses

$$P'(x) \longleftarrow P(x), \tag{7}$$
$$P'(x) \longleftarrow x = a, \tag{8}$$
$$\neg P'(x) \longleftarrow \textit{not } P'(x). \tag{9}$$

The third rule corresponds to the closed world assumption for P'. If we treat (7) and (8) as defaults and (9) as an exception, it is easy to check that this program has $\{P'(a), a = a\}$ as an e-answer set, corresponding to the expected repair, but also has $\{\neg P'(a), a = a\}$ as an e-answer set, which does not correspond to a database repair. The reason is that, from the corresponding ground program, the IC (8) is removed.

Nevertheless, as pointed out in [22], it is possible to extend the logic programs with exceptions in such a way that they include positive exceptions, in this case, (8), to a negative (default) rule, in this case, (9). We have to change the transformation rule 3. in section 3 used to check e-answer sets as follows: *3'. Delete every clause having a negative conclusion* $\neg L$, *with* $L \in S$. With it, $\{P'(a), a = a\}$ is still an e-answer set, but $\{\neg P'(a), a = a\}$ is not.

[2] Later on, we will see that we can (and usually need to) treat it as positive exception.

[3] ICs of this form, that generate knowledge about specific values, had already been discarded as problematic and non interesting in [4,8].

5 General Approach

So far, we have presented some particular examples of ICs. We have also shown that sometimes negative exceptions are needed, in other occasions positive exceptions are needed.

We have also considered single ICs. The most natural way of dealing with multiple integrity constraints consists of repeating the construction of the repair logic program for every constraint separately and taking the union of the results. Unfortunately, this does not work in general. However, under the additional assumption that the constraints are *closed under resolution* [25] this approach does work. In addition, as we will see in the next example, the techniques we have used so far do not give an account of simple examples of ICs.

Example 8. Consider the set of integrity constraints $IC = \{\neg P(x) \vee Q(x), \neg P(x) \vee \neg Q(x),\ \neg Q(x) \vee R(x),\ \neg Q(x) \vee \neg R(x), \neg R(x) \vee P(x),\ \neg R(x) \vee \neg P(x)\}$, and the database instance $\{P(a), Q(a), R(a)\}$. This set of ICs is equivalent to the formula $\neg P(x) \wedge \neg Q(x) \wedge \neg R(x)$. In this latter form, the IC can be treated with the techniques used in previous examples, but not in its original version, where the information about $\neg P(x) \wedge \neg Q(x) \wedge \neg R(x)$ is hidden. In this case, the only expected repair is the empty database instance, that is $\{\neg P(a), \neg Q(a), \neg R(a)\}$.

A natural candidate to be a repair program contains the persistence defaults $P'(x) \leftarrow P(x)$, $\neg P'(x) \leftarrow \mathit{not}\ P(x)$, *etc.*, plus both positive and negative exceptions (that block the negative and positive defaults, resp.) derived from IC. For example, for the first constraint the exceptions are $\neg P'(x) \leftarrow \neg Q'(x)$ and $Q'(x) \leftarrow P'(x)$. In this case, we do not obtain $S = \{P(a),\ Q(a),\ R(a),\ \neg P'(a),\ \neg Q'(a),\ \neg R'(a)\}$ as an e-answer, that corresponds to the only repair. The reason is that, with S, the persistence defaults cannot be used and then, there are no facts to apply the exceptions, without getting the empty set of primed literals as the only minimal model. A way of connecting persistence defaults and exceptions is missing. For this purpose, we will introduce *triggering exceptions* below. □

In this section we will present a general framework for generating repair programs with exceptions that can handle multiple constraints in the the so-called "standard format" (see below). The resulting programs will have both negative and positive exceptions, strong and procedural negations, and disjunctions of literals in the heads of some of the clauses; that is, they will be disjunctive extended logic programs [18] with exceptions.

As in [4], we consider a set of integrity constraints, IC, written in the standard format

$$\bigvee_{i=1}^{n} P_i(\bar{x}_i) \vee \bigvee_{i=1}^{m} \neg Q_i(\bar{y}_i) \vee \varphi, \tag{10}$$

where φ is a formula containing only built-in predicates, and there is an implicit universal quantification in front.

In order to specify the repairs of the database, r, by means of a logic program with exceptions, Π^r, we introduce a new predicate P' for each database predicate P, and replace the Ps by the P's in (10).

a. Persistence Defaults For each base predicate P, introduce the persistence defaults:

$$P'(\bar{x}) \longleftarrow P(\bar{x}) \tag{11}$$

$$\neg P'(\bar{x}) \longleftarrow \mathit{not}\ P(\bar{x}). \tag{12}$$

The rules of type (11) will be subject to negative exceptions, and the rules of type (12) will be subject to positive exceptions.

b. Stabilizing Exceptions From each IC (10) we generate for each negative literal $\neg Q_{i_0}$ in it, the *negative exception clause*:

$$\neg Q'_{i_0}(\bar{y}_{i_0}) \ \longleftarrow \ \bigwedge_{i=1}^{n} \neg P'_i(\bar{x}_i), \bigwedge_{i \neq i_0} Q'_i(\bar{y}_i), \overline{\varphi}, \tag{13}$$

where $\overline{\varphi}$ is a formula that is logically equivalent to the logical negation of φ.

Similarly, for each positive literal P_{i_1} in (10), generate the *positive exception clause*:

$$P'_{i_1}(\bar{x}_{i_1}) \ \longleftarrow \ \bigwedge_{i \neq i_1} \neg P'_i(\bar{x}_i), \bigwedge_{i=1}^{m} Q'_i(\bar{y}_i), \overline{\varphi}. \tag{14}$$

These exceptions may override the persistence stated in the defaults above. Their role is to make the ICs be satisfied by the new predicates. Nevertheless, with only these exceptions we are not in position to ensure that the changes that the original predicates should be subject to in order to restore consistency are propagated to the new predicates. We need special exceptions to trigger the first changes, from the P_is to the P'_js; next, the stabilizing transactions propagate all the required changes.

c. Triggering Exceptions From (10), produce the *disjunctive exception clause*:

$$\bigvee_{i=1}^{n} P'_i(\bar{x}_i) \vee \bigvee_{i=1}^{m} \neg Q'_i(\bar{y}_i) \ \longleftarrow \ \bigwedge_{i=1}^{n} \mathit{not}\ P_i(\bar{x}_i), \bigwedge_{i=1}^{m} Q_i(\bar{y}_i), \overline{\varphi}, \tag{15}$$

Finally, we add *facts* corresponding to the original database and rules for the built-ins if necessary. We call a program Π^r constructed as shown above a "(disjunctive extended) repair logic program with exceptions for the database instance r".

It is not difficult to extend the semantics, transformation and results presented in [21] for our repair programs. All we need to do is realize that positive defaults are blocked by negative conclusions, and negative defaults, by positive conclusions. Notice that clauses like (15) will be exceptions, then we do not need to worry about the way disjunctive conclusions obtained from them are blocked by exceptions.

For example, the rule 3. in the e-answer set semantics in section 3 should be changed to: *3". Delete every (positive) default having a positive conclusion*

L, with $\neg L \in S$*; and every (negative) default having a negative conclusion* $\neg L$*, with* $L \in S$. In this way we obtain an extended e-answer semantics: having applied pruning rules 1., 2. (of section 3) and 3"., we are left with a ground disjunctive logic program without *not* . If the candidate set S is one of the minimal models of this program [18], then we say that S is an *e-answer set*.

With respect to the transformation to obtain an extended disjunctive logic program (without exceptions), we do not need to change anything wrt what we had before, except, as suggested by (4), qualify the negative defaults (12) as follows: $\neg P'(\bar{x}) \longleftarrow not\ P(\bar{x}), not\ P'(\bar{x})$.

It can be proved that the e-answer sets of the disjunctive extended repair program with exceptions correspond, via the positive primed predicates P_i', to the repairs of r. Notice that the non primed parts of the e-answer sets (in terms of the original P predicates) coincide with the original database. This is because there is no rule in the repair program to modify the original predicates.

Example 9. (example 6 continued) With the new treatment, the resulting repair program has the stabilizing exceptions $Q'(x) \leftarrow P'(x)$, $\neg P'(x) \leftarrow \neg Q'(x)$, the triggering exception $\neg P'(x) \vee Q'(x) \leftarrow P(x), not\ Q(x)$; and the default rules $P'(x) \leftarrow P(x)$, $\neg P'(x) \leftarrow not\ P(x)$, $Q'(x) \leftarrow Q(x)$, $\neg Q'(x) \leftarrow not\ Q(x)$. With the original instance (facts) $P(a)$, the e-answer sets are $\{\neg P'(a), \neg Q'(a), P(a)\}$ and $\{P'(a), Q'(a), P(a)\}$, that correspond to the two expected database repairs.

Example 10. (example 8 continued) Apart from all the defaults and exceptions already introduced, it is necessary to introduce the disjunctive triggering exceptions. For example, for the first two constraints in *IC*, they are $\neg P'(x) \vee Q'(x) \leftarrow P(x),\ not\ Q(x)$ and $\neg P'(x) \vee \neg Q'(x) \leftarrow P(x),\ Q(x)$. In this case, the only e-answer set is $\{P(a), Q(a), R(a), \neg P'(a), \neg Q'(a), \neg R'(a)\}$, that corresponds to the only database repair.

Theorem 1. *For a set of domain independent binary integrity constraints IC of the form (10) and a database instance* r*, there is a one to one correspondence between the e-answers sets of the repair program* Π^r *and the repairs of* r*.* □

By a *domain independent constraint* [30] we understand that the constraint can be checked wrt satisfaction by looking to the finite active domain (plus possibly to the constants mentioned in the ICs). If we allow non domain independent ICs, then all repairs will be obtained as e-answer sets, but it may happen that an e-answer set of the program, even being minimal and satisfying IC, is not a repair in the sense that it assigns an infinite extension to some database predicates. This is the case of the IC $\forall x\ (Q(a) \vee P(x))$ on the database with empty tables (but still with an infinite domain D). According to the definition of repair, the only repair is $\{Q(a)\}$, but the other e-answer set that can be obtained from the program and makes P true of all elements in D is not a repair (because the extension of P is infinite).

All these problems are avoided assuming the the domain independence of the set IC.

By a *binary integrity constraint* we mean an IC like (10) where at most two literals L_i appear (but extra built-ins are allowed). This covers most cases of interesting ICs.

6 Evaluating $\mathcal{K}$-queries

The results in section 5 provide the underpinning of a general method of evaluating $\mathcal{K}$-queries. Assume r is a database instance and the set of integrity constraints IC is given. We show how to evaluate queries of the form $\beta \equiv \mathcal{K}\alpha$ where α is a basic query. First, from α we obtain a stratified logic program P_α (this is a standard construction) in terms of the new, primed predicates. One of the predicate symbols, Q_α, of P_α is designated as the query predicate: its extension is the answer to α in r. Second, determine all the e-answers sets $S_1, \ldots, S_k$ of the logic program with exceptions $P_\alpha \cup \Pi^r$. Third, compute the intersection $r_\beta = \bigcap_{1 \leq i \leq k} S_i/Q_\alpha$, where S_i/Q_α is the extension of Q_α in S_i. The set of tuples r_β is the set of answers to β in r. To obtain query answers to general $\mathcal{K}$-queries the above method needs to be combined with some method of evaluating first-order queries. For example, safe-range first-order queries [1] can be translated to relational algebra. The same approach can be used for $\mathcal{K}$ queries with the subqueries of the form $\mathcal{K}\alpha$ replaced by new relation symbols. Then when the resulting relational algebra query is evaluated and the need arises to materialize one of the new relations, the above method can be used to accomplish that goal.

7 Extensions and Alternatives

The approach proposed in section 5 also works for classes of ternary ICs, like a transitivity constraint, $P(x,y) \wedge P(y,z) \rightarrow P(x,z)$.

Nevertheless, there are rather artificial examples of ternary ICs for which the repair program does provide repairs (it can be proved that the methodology is always sound), but some repairs are missed.

Example 11. Consider the DB instance $r = \{P(a), Q(a), R(a)\}$ and the following set of integrity constraints $IC = \{\neg P(x) \vee \neg Q(x) \vee \neg R(x), \neg P(x) \vee \neg Q(x) \vee R(x), \neg P(x) \vee Q(x) \vee \neg R(x), P(x) \vee \neg Q(x) \vee \neg R(x), \neg P(x) \vee Q(x) \vee R(x), P(x) \vee \neg Q(x) \vee R(x), P(x) \vee Q(x) \vee \neg R(x)\}$. In this case, the repair program obtained by applying the methodology introduced in section 5 contains the usual persistence defaults plus triggering exceptions, e.g.

$$\neg P'(x) \vee \neg Q'(x) \vee \neg R'(x) \longleftarrow P(x), Q(x), R(x), \tag{16}$$

and stabilizing exceptions, e.g.

$$\neg P'(x) \longleftarrow Q'(x), R'(x), \quad \neg R'(x) \longleftarrow \neg P'(x), Q'(x), \tag{17}$$

etc. In this case we do not obtain any stable models, that is, the only repair, the empty instance, is missed. □

The methodology can be extended to cover more general ICs in standard format, like ternary constraints. Nevertheless, in order to simplify the presentation, we decided to present it as in section 5. The other reason for proceeding this way has to do with efficiency. Actually, in order to make it work for an arbitrary number of literals in the ICs, we have to add to the stabilizing constraints like (13), new clauses with disjunctive heads; one for each combination of two literals in the head, one for each combination of three literals in the head, etc. Unfortunately all this makes the length of the program grow exponentially.

Example 12. (example 11 continued) We keep all the rules appearing in example 11, e.g. (16), (17), but we add the new stabilizing exceptions

$$\neg P'(x) \vee \neg Q'(x) \longleftarrow R'(x), \tag{18}$$

etc. In this case we obtain as the only e-answer set, the empty instance, that is, the only repair. □

Finally, there is an alternative way of specifying database repairs by means of disjunctive logic programs with exceptions. A detailed analysis is left for future work. Here we just show it by means of example 11.

Example 13. (example 11 continued) The persistence and triggering rules are as in example 11, but the stabilizing exceptions are changed by introducing procedural negation in the bodies, obtaining, e.g.

$$\neg P'(x) \longleftarrow \mathit{not}\ \neg Q'(x), \mathit{not}\ \neg R'(x), \quad \neg R'(x) \longleftarrow \mathit{not}\ P'(x), \mathit{not}\ \neg Q'(x),$$

instead of (17), resp. In this case we obtain as the only e-answer set the only repair, that is, the empty instance.

Now we do obtain the right result in comparison to example 11 and the program is much shorter than the program in example 12, actually, the length of the program is linear in the size of the set of integrity constraints plus the database. □

The methodology presented in section 5 can be applied to ICs that are not in standard format. The most interesting case corresponds to a referential IC.

Example 14. Consider a referential constraint *RIC*: $P(x) \longrightarrow \exists y\ R(x,y)$, and the inconsistent database instance $r = \{P(a), P(b), R(b,a)\}$. For things to work properly, we need to assume that there is an underlying database domain $D = \{a, b\}$. The repair program has the persistence default rules $P'(x) \leftarrow P(x)$; $\neg P'(x) \leftarrow \mathit{not}\ P(x)$; $R'(x,y) \leftarrow R(x,y)$; and $\neg R'(x,y) \leftarrow \mathit{not}\ R(x,y)$. In addition, it has the stabilizing exceptions

$$\neg P'(x) \longleftarrow \neg aux1(x), \neg R'(x, null), \tag{19}$$
$$aux1(x) \longleftarrow R'(x,y),$$

$$R'(x, null) \longleftarrow P'(x), not\ aux1(x); \tag{20}$$

and the triggering exception $\neg P'(x) \vee R'(x, null) \leftarrow P(x), not\ aux2(x)$, with $aux2(x) \leftarrow R(x, y)$.

The variables in this program range over D, that is, they do not take the value *null*. This is the reason for the last literal in clause (19). The last literal in clause (20) is necessary to insert a null value only when it is needed; this clause relies on the fact that variables range over D only. Instantiating variables on D only[4], the only two e-answer sets are the expected ones, namely delete $P(a)$ or insert $R(a, null)$.

8 Related Work

The semantics underlying our notion of consistent query answers comes from the area of *belief revision*. More precisely, such answers are the same as the answers obtained from a relational database *revised* with integrity constraints. Our notion of minimal change is equivalent to that of Satoh [29]. There are, however, several important differences between our work and past work on belief revision. Typically, belief revision considers revising a propositional theory with a propositional formula [20,12]. In our case we revise a single first-order structure (a database instance) with a first-order formula, often of a very restricted kind (e.g., functional dependencies). The program Π^r may be viewed as a representation of the revised database. However, the revised database is not explicitly constructed. Instead the program Π^r is used to answer queries to the revised database.

Bry [7] was, to our knowledge, the first author to consider the notion of consistent query answer in inconsistent databases. He defined consistent query answers using provability in minimal logic. The proposed inference method is nonmonotonic, but fails to capture minimal change. Moreover, Bry's approach is entirely proof-theoretic and does not provide a computational mechanism to obtain consistent query answers, except in the propositional case.

It has been widely recognized that in database integration the integrated data may be inconsistent with the integrity constraints. A typical (theoretical) solution is to augment the data model to represent disjunctive information. [2,5,10,24]. There are several important differences between the above approaches and ours. First, they rely on the construction of a single (disjunctive) instance and the deletion of conflicting tuples. Second, they usually handle severely restricted classes of integrity constraints and queries.

Gertz [19] describes techniques and algorithms for computing repairs of single constraint violations. The issue of query answering in the presence of an inconsistency is not addressed in his work.

[4] A simple way to enforce this at the object level is to introduce the predicate D in the clauses, to force variables to take values in D only, excluding the null value.

9 Conclusions and Further Work

There are other logic programming formalisms that allow some forms of reasoning in the presence of inconsistency that could be used in principle for our task of reasoning about database repairs. For example, the *symmetric logic programs* introduced in [27] might work. They also allow monotonic negation and their semantics is also based on a modification of the stable model semantics. Nevertheless, these programs have an operator for representing assumptions, that can be used to qualify the literals in the body of a clause.

We know that the logic programs with exceptions that specify the database repairs eventually lead to disjunctive logic programs with a stable model semantics. It would be interesting to examine in a more detailed manner what kind of programs we eventually obtain, depending on the kind of integrity constraints; the fact that the repair programs follow a rather fixed format, in particular, most of it has to do with the information in the original database instance; and the fact that we can express the *closed world assumption* for every database predicate. Depending on the peculiarities of the resulting program, specialized computation methods could be used.

In particular, it is interesting to investigate if our repair programs can be eventually transformed into (non disjunctive) normal programs. Cases where this can be achieved have been investigated in [6,23]. The idea is to push in turn all but one of the disjuncts in the head of a clause to the body preceded by a procedural negation *not* . As studied in [13] this reduction is not always possible[5]. Actually, example 10 cannot be treated in this way, by transforming, e.g. its first disjunctive clause, into the two clauses $p' \leftarrow \mathit{not}\ p, \mathit{not}\ q, \mathit{not}\ q'$ and $q' \leftarrow \mathit{not}\ p, \mathit{not}\ q, \mathit{not}\ p'$; disjunctive heads are needed.

We are currently exploring the possibilities of existing implementations of stable model semantics for normal logic programs in combination with database management systems to compute consistent answers. Actually, once the repair program is transformed into a disjunctive extended program with the stable model semantics, we can benefit from implementations like DLV [14,15]. We have successfully ran our examples on DLV. To make it efficient in real database applications, it would be necessary to have the possibilities of handling non ground queries and computing the *consistent core* of the database, that is, the intersection of the repairs (or stable models). In [3], it is described how to take advantage of core computations to obtain consistent answers to aggregate queries in inconsistent databases.

The treatment of existential constraints, like referential integrity constraints (see example 14), requires further analysis and implementation. Actually, DLV has also been useful in treating referential constraints. With it, it is possible, with a suitable representation of ICs, to impose preferences

[5] For example, the disjunctive logic program $p \vee p \leftarrow$, has $\{p\}$ as a stable model, whereas the transformed program $p \leftarrow \mathit{not}\ p$ does not have $\{p\}$ as a stable model.

on possible repairs, for example, the alternative of repairing the database by introducing null values can be blocked if this is considered undesirable or less desirable than the cascade elimination of tuples. Further experiments and implementations with DLV are also left for future work.

Acknowledgment: This Work has been supported by FONDECYT grants (1980945, 1990089 and 1000593) and NSF grant INT-9901877/CONICYT Grant 1998-02-083. The authors are grateful to Bob Kowalski and Jorge Lobo for illuminating discussions; and to Georg Gottlob and Nicola Leone for very useful information on disjunctive logic programs. We are also grateful to Francisco Orchard for his support with DLV.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. S. Agarwal, A.M. Keller, G. Wiederhold, and K. Saraswat. Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In *IEEE International Conference on Data Engineering*, 1995.
3. M. Arenas, L. Bertossi, and J. Chomicki. Scalar Aggregation in FD-Inconsistent Databases. Submitted to ICDT'01.
4. M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *Proc. ACM Symposium on Principles of Database Systems (ACM PODS'99, Philadelphia)*, pages 68–79, 1999.
5. C. Baral, S. Kraus, J. Minker, and V.S. Subrahmanian. Combining Knowledge Bases Consisting of First-Order Theories. *Computational Intelligence*, 8:45–71, 1992.
6. R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.
7. F. Bry. Query Answering in Information Systems with Integrity Constraints. In *IFIP WG 11.5 Working Conference on Integrity and Control in Information Systems*. Chapman &Hall, 1997.
8. A. Celle and L. Bertossi. Querying Inconsistent Databases: Algorithms and Implementation. Submitted, 2000.
9. W. Chen and D. S. Warren. Computation of Stabe Models and its Integration with Logical Query Processing. *IEEE Transactions on Data and Knowledege Engineering*, 8(5):742–757, 1996.
10. Phan Minh Dung. Integrating Data from Possibly Inconsistent Databases. In *International Conference on Cooperative Information Systems*, Brussels, Belgium, 1996.
11. O.M. Duschka, M.R. Genesereth, and A.Y. Levy. Recursive Query Plans for Data Integration. *Journal of Logic Programming*, 2000.
12. T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57(2-3):227–270, 1992.
13. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactons on Database Systems*, 22(3):364–418, 1997.

14. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The Knowledge Representation System DVL: Progress Report, Comparisons, and Benchmarks. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, KR98, Trento, Italy, June 1998*. Morgan Kaufman, 1998.
15. F. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In *Proceedings of the 5rd Logic Programming and Non-Monotonic Reasoning Conference, LPNMR99*. LNAI, Springer-Verlag, El Paso, Texas, December 2–4, 1999.
16. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *Proc. Fifth International Conference and Symposium on Logic Programming*, volume 2, pages 1070–1080. MIT Press, 1988.
17. M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In *Proc. Seventh International Logic Programming Conference*, pages 579–597. MIT Press, 1990.
18. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
19. M. Gertz. *Diagnosis and Repair of Constraint Violations in Database Systems*. PhD thesis, Universität Hannover, 1996.
20. H. Katsuno and A. O. Mendelzon. A Unified View of Propositional Knowledge Base Updates. In *International Joint Conference on Artificial Intelligence*, 1989.
21. R. Kowalski and F. Sadri. Logic Programs with Exceptions. *New Generation Computing*, 9:387–400, 1991.
22. R. Kowalski, F. Sadri, and F. Toni. The Role Of Abduction in Logic Programming. In *Handbbok of Logic in Artificial Intelligence and Logic Programming, Vol. 5*, pages 235–266. Oxford University Press, 1998.
23. N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, 1997.
24. J. Lin and A. O. Mendelzon. Merging Databases under Constraints. *International Journal of Cooperative Information Systems*, 7(1):55–76, 1996.
25. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
26. I. Niemela and P. Simons. Smodels - An Implementation of the Stable and Well-founded Semantics for Normal Logic Programs. In *Proc. 4th Intenational Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer. Lecture Notes in Artificial Intelligence 1265, 1997.
27. S. Pimentel and W. Rodi. Belief Revision and Paraconsistency in a Logic Programming Framework. In *Proc. Conf. Logic Programming and Nonmonotonic Reasoning*, pages 228–242, 1991.
28. K. Sagonas, T. Swift, and D.S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. ACM SIGMOD*, 1994.
29. K. Satoh. Nonmonotonic Reasoning by Minimal Belief Revision. In *International Conference on Fifth Generation Computer Systems*, pages 455–462, 1988.
30. J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I*. Computer Science Press, 1988.
31. J. D. Ullman. Information Integration Using Logical Views. In *International Conference on Database Theory*. Springer-Verlag, 1997.